Le contrat d'évolution d'architectures : un outil pour le maintien de propriétés non fonctionnelles[1]

Régis Fleurquin, Chouki Tibermacine et Salah Sadou

Laboratoire Valoria
Université de Bretagne Sud
Campus de Tohannic
F-56000 Vannes Cedex
(Chouki.Tibermacine,Regis.Fleurquin,Salah.Sadou)@univ-ubs.fr

RÉSUMÉ. *Tout logiciel doit évoluer pour répondre aux exigences changeantes de ses utilisateurs et aux modifications de son environnement. Ces changements, souvent imprévisibles, réalisés par un tiers et dans l'urgence, mènent parfois le logiciel vers un état que ses créateurs n'auraient pas souhaité. Nous présentons dans cet article un cadre pour une évolution contrôlée d'applications à base de composants. Ce contrôle garantit la préservation de propriétés architecturales et par là, de certaines propriétés non fonctionnelles.*

ABSTRACT. *Software systems should evolve in order to respond to changing client requirements and their evolving environments. These changes, often unforeseen, done by tiers and in urgency, lead sometimes the software to a state initially undesired by its creators. We present, in this paper, a framework to preserve the consistency of component-based applications during their evolution. This evolution consistency is achieved by maintaining some architectural strategies and thus, preserving the corresponding non-functional properties indicated as important.*

MOTS-CLÉS : *Maintenance, Architecture logicielle, Contrat d'évolution, Propriétés non fonctionnelles, Composants logiciels.*

KEYWORDS: *Maintenance, Software Architecture, Evolution Contract, Non-Functional Properties, Software Components.*

1. Le travail présenté dans cet article bénéficie du soutien de la région de Bretagne (contrat n° 20046839)

1. Introduction

Une caractéristique intrinsèque d'un logiciel, représentant une activité du monde réel, est la nécessité qu'il a d'évoluer pour satisfaire de nouvelles exigences. La première loi de Lehman, issue de constatations sur le terrain, stipule qu'un logiciel doit nécessairement évoluer faute de quoi il devient progressivement inutile [LEH 85]. Bien qu'ancienne cette loi ne s'est jamais démentie. La réactivité, toujours plus grande, exigée des applications informatiques, supports de processus métiers évoluant eux-mêmes de plus en plus vite, a même accru au fil des ans la portée de cette loi. La maintenance est, plus que jamais, une activité incontournable. Cette activité coûte cependant de plus en plus cher. Estimée dans les années 80 et 90 à environ 50 à 60 % [LIE 81, MCK 84] des coûts associés aux logiciels ; de récentes études évaluent désormais ce coût entre 80 et 90 % [ERL 00, SEA 03]. La maintenance, trop longtemps perçue comme une activité peu valorisante et de moindre intérêt, doit, à l'instar de l'activité de développement se doter de méthodes, de techniques et d'outils efficaces.

Parmi les activités de maintenance, la compréhension de l'architecture de l'application, avant évolution, et la vérification de sa non régression fonctionnelle et non fonctionnelle, après évolution, sont de loin les plus coûteuses. La première de ces activités, à elle seule, compte pour plus de 50 % du temps de maintenance [BEN 96]. Elle est d'autant plus facile que la documentation fournie avec le logiciel est de qualité. Cette documentation doit, en particulier, être complète, à jour et non ambiguë. La seconde de ces activités vérifie d'une part, l'existence après modification de la propriété ou du nouveau service recherché et d'autre part, que les autres propriétés et/ou services n'ont pas été altérés. Cette vérification peut se faire soit a posteriori, une fois la modification entérinée en constatant in vivo ses effets, par exemple au travers de tests de non régression ; soit a priori, au moment où l'on détaille la modification que l'on souhaite entreprendre, par exemple en alertant des conséquences de celle-ci. La non régression a posteriori de traits fonctionnels est de loin la mieux maîtrisée car la plus simple. Bien que complémentaires, il est cependant connu que les techniques préventives, favorisant la détection au plus tôt des problèmes, sont moins coûteuses que les techniques correctives.

Dans cet article, nous nous intéressons aux problèmes posés par les deux activités de maintenance précédentes dans le cadre de la technologie des composants. Nous mettons notamment en valeur l'intérêt de contraindre les évolutions possibles d'une architecture dans le but d'une part, de garantir la présence d'une documentation de conception de qualité et d'autre part, de prévenir la disparition de certaines propriétés non fonctionnelles. Dans la première section, nous soulignons sur un exemple de maintenance, quelques uns des problèmes induits par une documentation de qualité insuffisante et par un processus de mise à jour somme toute trop libéral. Nous décrivons ensuite l'approche que nous proposons pour remédier à ce type de problème. Dans la quatrième section, nous présentons une implantation de cette approche s'appuyant sur le langage OCL et sur un outil de vérification de contraintes. Dans la dernière section, avant la conclusion, nous discutons des travaux connexes.

2. Illustration de la problématique

La figure 1 présente l'architecture d'un système de contrôle d'accès à un bâtiment (cas d'un musée). Cette architecture est un cas particulier du patron *pipe & filter* : le système reçoit en entrée des données permettant l'authentification d'un usager. Après identification, ces données sont envoyées au composant `Contrôleur d'accès`. Celui-ci ajoute à ce flux d'autres données (l'heure d'entrée, la galerie dans le musée, etc.). Puis, il passe le tout au composant responsable de l'archivage local (composant `Archiveur`). Ces données sont ensuite transmises (composant `Emetteur`), via le réseau externe à un serveur central d'archivage de l'entreprise qui s'occupe de la sécurité du musée. A travers le choix d'une architecture de type *pipe & filter*, les développeurs ont cherché à respecter les exigences de maintenabilité formulées dans le document de spécification de ce système. En effet, ce style architectural garantit un couplage faible entre des composants. Un composant n'est lié qu'à un seul composant par son interface requise et un seul par son interface fournie. Ces développeurs n'ont cependant pas pris la peine de documenter les raisons de ce choix. A ce stade, une information importante pour la compréhension de la structure du système est alors définitivement perdue.

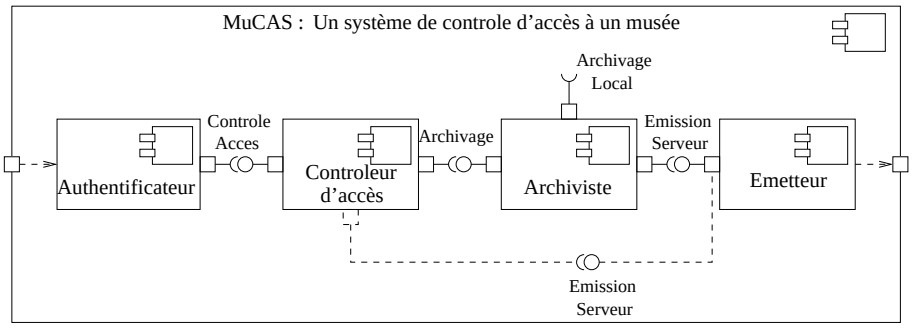

Figure 1 – Architecture simplifiée d'un système de contrôle d'accès à un musée

Supposons maintenant que l'archivage local de certaines informations (voir composant `Archiveur`) est désormais inutile et qu'il faille transmettre ces informations directement au serveur central. Les personnes en charge de cette modification peuvent décider, après consultation des documents de conception, de créer un lien direct entre le composant `Contrôleur d'accès` et le composant `Emetteur`. Le composant `Contrôleur d'accès` se retrouve, alors, avec deux liens : le premier avec le composant `Archiveur`, pour le flux non affecté par la modification ; le second avec le composant `Emetteur` pour les données à transmettre directement (voir les lignes en pointillé sur la figure 1). Cette modification, réalisée en toute bonne foi, fait perdre à l'application les bénéfices d'une structure en *pipe* et, par conséquent, abaisse son niveau de maintenabilité. Une propriété non fonctionnelle, qui ne devait pas être altérée, est mise à mal par le simple fait de remettre en cause un choix d'architecture dont la présence n'était pas innocente.

Le type de problème illustré ici a plusieurs origines. Tout d'abord, les modifications faites à un logiciel sont souvent réalisées longtemps après la version précédente et généralement par des tiers ; les raisons des choix architecturaux étant rarement explicitées, elles sont alors perdues pour les phases de maintenance suivantes. Même dans le cas où ces raisons sont détaillées, le code est souvent le seul artefact modifié et parcouru sans considération pour les documentations associées. Un réflexe acquis par la plupart des développeurs est donc de n'accorder du crédit, malheureusement avec raison, qu'au seul code source. Quoi qu'il en soit, rien n'empêche au final un développeur de passer outre un choix architectural bien qu'informé des conséquences de cet acte. Résoudre ce type de problème passe nécessairement par deux choses. Premièrement, il doit y avoir obligation de documenter les raisons des choix architecturaux faits lors du développement et de la maintenance. Cette obligation garantit l'existence d'une documentation, remise à jour si nécessaire, utilisable lors de la phase de compréhension d'une architecture. Deuxièmement, le respect de ces choix doit être vérifié lors de chaque mise à jour du code. Cette vérification assure la non régression des propriétés visées par ces choix. L'approche que nous allons décrire respecte ces deux aspects.

3. Notre approche : le contrat d'évolution

Il est admis que ce ne sont pas les fonctions attendues qui déterminent l'architecture d'une application ou d'un composant, mais bien les exigences non fonctionnelles [BAS 03]. C'est d'ailleurs le point de vue de certaines méthodes de développement d'architectures, comme la méthode ADD [BAS 01] du *Software Engineering Institute*. La connaissance des liens unissant propriétés non fonctionnelles et choix architecturaux est donc du plus grand intérêt pour les personnes en charge de la maintenance et ce à double titre. Premièrement, si l'architecture est bâtie sur la base d'une recherche de certaines propriétés non fonctionnelles, 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 propriétés non fonctionnelles 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 non fonctionnelles. Sur les fils des 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).

Nous proposons donc d'expliciter les liens unissant les spécifications non fonctionnelles 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 à même de lier ces choix à des énoncés de spécifications non fonctionnelles. Le choix d'un langage formel garantira non seulement la non ambiguïté des descriptions, mais permettra également l'automatisation de certaines opérations. Il sera par exemple possible d'alerter, à chaque "pas" d'évolution, le développeur sur les conséquences éventuelles des modifications qu'il applique à une architecture. On se retrouve alors dans un système bouclé. Ce type de système augmente grandement les chances d'aboutir à une solution remplissant les nouvelles exigences, tout en préservant les propriétés non fonctionnelles qui ne devaient pas être altérées.

Dans ce processus, nous ne devons pas interdire un "pas" d'évolution. On signale simplement la tentative de rupture d'un choix architectural, dont on précise les conséquences. A charge pour le développeur, en toute connaissance, de maintenir ou non sa modification. Il se peut, en effet, qu'un choix architectural soit remplacé par un autre sans remise en cause des propriétés non fonctionnelles. Car il existe souvent plusieurs architectures capables de garantir le respect d'une même propriété non fonctionnelle. De plus, pour réaliser une modification, on peut être amené à invalider un choix pour le restaurer plus tard dans un contexte changé. On peut faire ici le parallèle avec un invariant de classe qui peut être non valide dans le corps d'une méthode mais garanti finalement en sortie de méthode. La seule condition bloquante pour le développeur est d'indiquer, en fin d'évolution, pour tout choix ayant été remis en cause lors du processus de maintenance, le ou les nouveaux choix qui lui sont substitués. Cette condition assure qu'aucune propriété non fonctionnelle ne reste pendante ; c'est-à-dire sans éléments architecturaux ayant pour objet son obtention. Le non respect de cette condition implique, de facto, l'obligation pour le développeur de modifier la spécification non fonctionnelle. Ainsi, lors d'une évolution, de nouveaux choix architecturaux peuvent compléter, amender ou remplacer les anciens choix.

Un choix architectural est perçu dans cette approche 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 non fonctionnelles, constituent le **contrat d'évolution** d'un composant. Nous parlons de contrat car il documente les droits et devoirs de deux parties : le développeur de la précédente version du composant qui s'engage à garantir les propriétés non fonctionnelles, sous réserve du respect, par le développeur de la nouvelle version, des contraintes architecturales que le premier avait établies. Le contrat d'évolution est élaboré lors du développement de la première version d'un composant. Des contraintes apparaissent à chaque stade du développement dans lequel un choix architectural motivé est fait. Le contrat est donc construit progressivement et enrichi au fil du projet. Certaines contraintes peuvent même être héritées d'un plan qualité logiciel (contraintes de projet) ou d'un manuel qualité (contraintes d'entreprise) et donc, émerger avant même le démarrage du développement logiciel. Par la suite, dans le respect de la condition de blocage, ce contrat pourra à son tour être modifié. A charge pour les développeurs, informés

des conséquences de leurs actes, de garantir à leur tour, l'obtention des propriétés non fonctionnelles sur la base de nouvelles contraintes qu'ils ont pu établir.

Ces contraintes architecturales sont bien plus qu'un autre formalisme pour les commentaires classiques de modèles de conception ou de code. Tout d'abord, à l'instar des pré et post-conditions de langages comme EIFFEL, leur format évaluable autorise de nouveaux usages et modifie le cadre méthodologique de la maintenance. Enfin, certaines contraintes n'ont de sens que dans le cadre évolutif. Il suffit de considérer une contrainte comme : la complexité d'un composant ne doit pas augmenter. Cette contrainte, associée à la propriété de maintenabilité, est une pure condition de maintenance. Elle n'aurait aucun sens dans un commentaire classique. Bien évidemment, à l'extrême, les contraintes peuvent devenir aussi nombreuses que les lignes de code. Il faut donc trouver des niveaux de granularité architecturaux influençant significativement une propriété particulière d'interface. Il est nécessaire d'user avec intelligence et parcimonie de ces contraintes et faire preuve de la même hygiène que lors de l'écriture des commentaires classiques : ni trop, ni trop peu.

4. Une implantation du contrat d'évolution

Notre approche s'appuie sur un langage formel pour décrire le contrat d'évolution et sur un outil pour son évaluation. Nous allons décrire le langage dans une première section et l'outil dans une seconde.

4.1. *L'expression du contrat d'évolution*

Un contrat d'évolution est composé d'une part, d'un ensemble de contraintes architecturales et d'autre part, de la liste des liens unissant ces contraintes avec des énoncés de spécification non fonctionnelle. Nous allons décrire le langage de description de contraintes. Nous présenterons, ensuite, le mécanisme d'association.

4.1.1. *Le langage d'expression de contraintes*

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 ou des conventions de codage doivent pouvoir être écrites. Face à cette diversité, nous avons opté pour une solution à deux niveaux. Le premier niveau s'appuie sur un langage facile à appréhender, largement répandu et standardisé par l'OMG, à savoir OCL [OMG 03]. Le second niveau prend la forme d'un méta-modèle. Ce méta-modèle (voir figure 2) nous permet d'adapter OCL, sans changement de syntaxe, à l'écriture de contraintes sur des architectures à base de composants. Ce second niveau permet également, par simple amendement du méta-modèle, d'étendre comme bon nous semble la liste des opérateurs disponibles sans pour cela avoir besoin de toucher à la syntaxe du langage de

premier niveau (donc à la syntaxe OCL). Cette structure à deux niveaux est le gage de l'extensibilité du pouvoir d'expression de notre formalisme.

Pour comprendre comment cette structure à deux niveaux s'articule, il est nécessaire de revenir sur le mode d'expression habituel des contraintes OCL dans un diagramme de classes. Dans ce type de diagramme, OCL s'utilise le plus souvent pour spécifier des invariants de classe, des pré/post conditions d'opération, des contraintes de cycle entre associations, etc. Ces contraintes restreignent le nombre des diagrammes d'objets valides instanciables depuis un diagramme de classes. Elles pallient à 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, une opération ou une association présente sur ce diagramme. Voici deux exemples de contrainte OCL.

```
context ArticleLMO inv:
self.taille <= 13
context a: ArticleLMO inv:
a.ecritPar->size() >= 1
```

Dans les deux cas, le contexte est une classe (`ArticleLMO`). Les deux contraintes sont écrites selon le point de vue d'une instance quelconque du contexte (ici un objet instance de la classe `ArticleLMO`). C'est l'approche adoptée pour toute contrainte 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 méta-modèle. Un méta-modèle expose les concepts d'un langage et les liens qu'ils entretiennent entre eux. Il décrit une grammaire abstraite. Une contrainte ayant pour contexte une méta-classe va, de ce fait, limiter la puissance d'expression des règles de production de cette grammaire et donc le nombre des phrases (i.e. modèles) dérivables. Certaines structures de phrase sont écartées. Si ce méta-modèle décrit la grammaire d'un langage de description d'architectures, une contrainte exprime que seules certaines architectures (i.e. modèles) sont dérivables (i.e. instanciables) dans ce langage. Le langage est 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 méta-classe `composant`. Cette contrainte est exactement du type de celle que nous souhaitons pouvoir exprimer. Malheureusement sa portée est globale. Elle s'applique à tout com-

posant et non à un composant particulier comme nous souhaitons le faire. 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. Sur le plan syntaxique, nous imposons que tout contexte introduise un identificateur. Cet identificateur doit être, de plus, le nom d'une instance particulière de la méta-classe citée dans le contexte. Sur le plan sémantique, nous interprétons la contrainte avec le sens qu'elle aurait dans le contexte d'une méta-classe mais en limitant sa portée à l'instance citée dans le contexte.

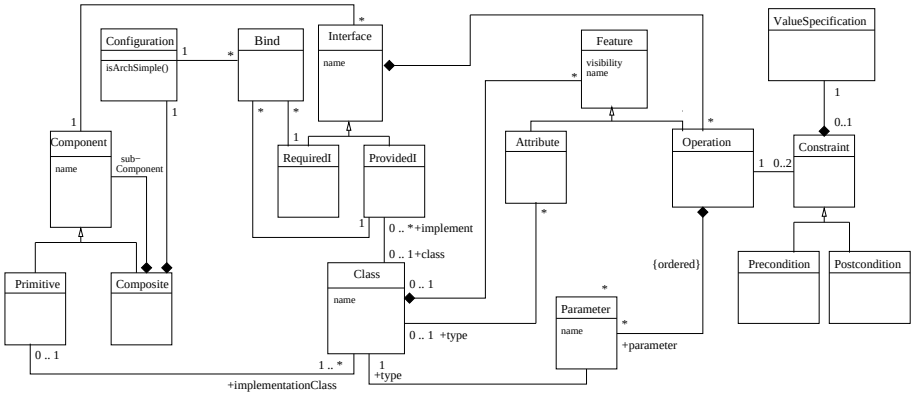

Figure 2 – Méta-modèle pour les architectures d'applications à base de composants.

Selon ce principe, la contrainte qui suit, appliquée à notre méta-modèle (voir figure 2), stipule que le composant primitif de nom `ControleurAcces` doit être lié à un et un seul composant par le biais de son interface requise `Archivage` :

```
context ControleurAcces:Primitive inv:
ControleurAcces.interface.select(i : Interface |
(i.isTypeOf(RequiredI)) and (i.name = 'Archivage')).bind->size() = 1
```

Ainsi, moyennant une modification minime de la syntaxe et de la sémantique d'OCL nous arrivons, en usant d'une structure langagière bicéphale OCL/Méta-modèle, à décrire des contraintes faisant état d'un mécanisme d'introspection. Un composant est à même d'exprimer une contrainte portant sur sa propre structure. A titre d'exemple, la condition de couplage faible que l'on souhaitait voir respectée par l'application présentée à la section 2 s'exprime de la manière suivante :

```
context muCAS: Composite inv:
(muCAS.configuration.bind.requiredI.component=(muCAS.subComponent))
and
(muCAS.configuration.bind.ProvidedI.component=(muCAS.subComponent))
```

La première partie de cette contrainte exprime le fait que tout sous-composant doit avoir un et un seul *bind* via son interface requise. La seconde fait de même pour les interfaces fournies.

En ajoutant des méta-classes ou des opérations dans le méta-modèle on peut étendre le langage d'expression de contraintes. Ainsi, pour faciliter l'écriture de certaines contraintes, nous avons introduit des opérateurs de la théorie des graphes (`isSimple()`, `isConnexe()`, etc.) dans la méta-classe `Configuration`.

4.1.2. *Le mécanisme d'association*

Le mécanisme d'association doit permettre de lier une contrainte architecturale à un énoncé de spécification non fonctionnelle. Une contrainte doit pouvoir être liée à plusieurs énoncés (au moins un) et réciproquement un énoncé peut se voir affecter plusieurs contraintes (au moins une). Dans l'état actuel de nos travaux le mécanisme proposé est encore rudimentaire. Sa formalisation est dépendante du choix d'un langage d'expression de propriétés non fonctionnelles ; en particulier de la structure de ce langage. Or, ce type de langage est un domaine encore ouvert et en devenir, objet de nombreux travaux.

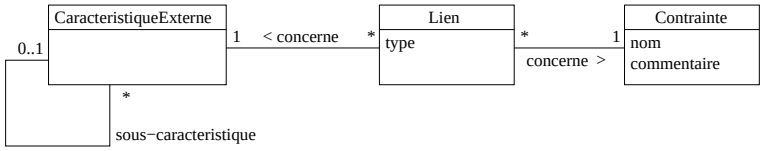

Figure 3 – Structure du mécanisme d'association.

Le mécanisme que nous proposons respecte la structure décrite par la figure 3. Dans son esprit, elle s'inspire de la norme qualité ISO 9126 [ISO 01]. Un lien associe une contrainte à une caractéristique qualité externe. Ces caractéristiques sont organisées sous la forme d'une forêt. Cette forêt compte 6 arbres dont les racines sont pour l'instant les 6 caractéristiques externes de plus haut niveau énoncées par la norme : maintenabilité, portabilité, fiabilité, rendement, capacité fonctionnelle, facilité d'utilisation. Le deuxième niveau de chaque arbre contient les sous-caractéristiques externes détaillées par la norme. Chaque contrainte a un nom et, éventuellement, un commentaire. Un lien unit une contrainte à une caractéristique. Chaque lien est typé. Pour l'instant nous n'usons que de deux types de lien : `contribute` et `limit`.

Concrètement, une contrainte annonce dans un en-tête ses liens avec les différentes caractéristiques qualité. Cet en-tête prend la forme d'une suite de commentaires respectant le format suivant :

```
(<caractéristique>,<type de lien>):<description>
```

A titre d'exemple, la contrainte décrite précédemment doit avoir pour en-tête le commentaire structuré suivant :

```
-- (maintainability,contribute) : Architecture en pipe
context muCAS: Composite
...
```

Dans la contrainte ci-dessus, le lien concerne l'attribut maintenabilité. Le type du lien est `contribute`. Il indique que la contrainte favorise l'obtention de la caractéristique concernée.

4.2. *Support logiciel pour l'interprétation des contrats d'évolution*

Nous avons développé `ACE` (*Architectural Contract Evaluator*), un outil permettant la rédaction puis la vérification d'un contrat d'évolution. Pour cela, il exploite les éléments suivants :

– Le méta-modèle de composants sous la forme d'un fichier XMI ;

– Le composant avant évolution : une archive contenant entre autres la description de l'architecture du composant (sous la forme d'un document XML) et son contrat d'évolution.

– La description de l'architecture du composant après évolution, sous forme d'un document XML.

Le fichier contenant le méta-modèle est utilisé par `ACE` pour contrôler le bien fondé des navigations exprimées dans les contraintes. Fournir notre méta-modèle en paramètre, nous permet de prendre en compte différents méta-modèles ou bien de le faire évoluer sans impact sur l'outil.

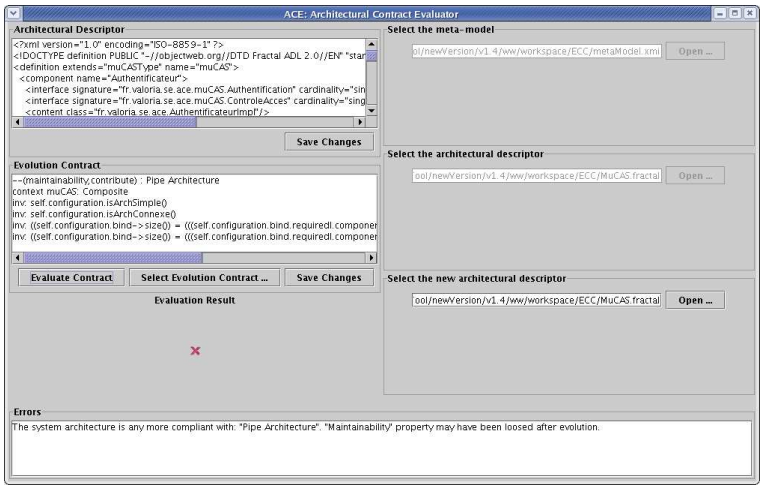

Figure 4 – ACE : outil d'évaluation de contrats d'évolution.

`ACE`, dont on peut voir l'interface dans la figure 4, s'utilise de la manière suivante : en phase de rédaction du contrat d'évolution, lors de la création du composant, le fichier représentant son architecture est exploité pour vérifier que les identifiants, apparaissant dans les contraintes, correspondent bien à des éléments de l'architecture.

Le contrat est, d'abord, évalué sur l'architecture initiale. Ceci nous permet de s'assurer que la première version du composant valide son propre contrat. Une fois cette version validée, les évaluations suivantes se font sur les versions présentées en tant que des évolutions du composant.

`ACE` s'appuie sur le compilateur `OCLCompiler` [DRE 02] pour la génération de l'arbre syntaxique abstrait (AST) des contraintes. Dans sa version actuelle, `ACE` supporte des descripteurs d'architecture décrits avec l'ADL `Fractal` [BRU 04]. Les spécificités de l'ADL `Fractal`, et leurs liens avec notre méta-modèle, sont complètement encapsulés dans une classe dite d'adaptation. Le passage à tout autre modèle, conforme à notre méta-modèle, consisterait en l'écriture de la classe d'adaptation correspondante.

5. Travaux connexes

Notre travail met à profit les résultats de divers domaines de recherche, notamment des études sur les langages de description d'architectures (ADL) et celles sur l'expression de patrons de conception. Toutefois, nous avons choisi de ne présenter, dans cet article, que les travaux ayant clairement affiché comme objectif la préservation de propriétés structurelles d'un système lors de son évolution.

Dans [KLA 97], Klarlund et al. ont proposé un nouveau langage de contraintes : CDL (*Category Description Language*). Ce langage, basé sur la logique du premier ordre sur les arbres d'analyse, permet d'expliciter, de manière formelle, des invariants architecturaux sur un système. Ils appellent une catégorie un ensemble nommé de contraintes et un style CDL, un ensemble de catégories. Une fois ces styles définis par l'architecte du système, un développeur d'application sélectionne les catégories qui l'intéressent et annote les éléments de conception concernés avec les noms de ces catégories. Par la suite, une vérification est faite, par un environnement dédié, pour déterminer si la conception satisfait toutes les catégories. Pour le même objectif, Wuyts propose le langage SOUL [WUY 98] : une solution basée sur la logique du premier ordre pour décrire les propriétés que doit respecter un programme orienté objets.

Dans les deux systèmes ci-dessus, on retrouve, comme pour notre approche, la notion de contraintes. Toutefois, trois éléments nous séparent : i) leur approche a pour cible les systèmes à base d'objets, alors que la notre vise les systèmes à base de composants ; ii) pour l'expression des contraintes, ils ont fait le choix d'un langage ad-hoc, alors que nous avons fait le choix de réutiliser un langage d'expression de contraintes existant (OCL) ; iii) nous associons à chaque contrainte ses objectifs, c'est-à-dire, un lien avec les propriétés non fonctionnelles cibles alors qu'eux, ne le font pas.

Avec le système `CoffeeStrainer` [BOK 99], Bokowski propose une approche plus pragmatique pour définir des contraintes structurelles d'applications Java. Il utilise comme langage, Java lui même. Les contraintes sont définies sous forme de méthodes dont les valeurs de retour sont des booléens. Ces méthodes reçoivent en para-

mètre des noeuds de l'arbre syntaxique des classes à vérifier (`Field, Assignment`, etc). Elles sont insérées dans des commentaires spéciaux à l'intérieur d'interfaces vides de marquage. Une classe doit implanter ces interfaces pour préserver les propriétés représentées par les contraintes. `CoffeeStrainer` se chargera de générer le nécessaire à partir des commentaires.

Contrairement aux approches précédentes, la cible est ici, un langage de programmation particulier. Le moyen d'expression est ce même langage. Cette approche a l'avantage de ne pas créer un langage ad-hoc pour l'expression des contraintes, mais a l'inconvénient de n'être utilisable que pour ce langage. Contrairement à notre approche, dont la cible est l'architecture des applications, donc les relations inter-composants ou inter-classes, l'approche `CoffeeStrainer` vise les contraintes intra-classe. La relation n'est inter-classes que dans les cas d'héritage. Cette approche est complémentaire à la nôtre.

6. Conclusion

Le contrat d'évolution évite la remise en cause, de manière inconsciente, des propriétés importantes d'un composant lors d'une évolution. Toute décision concernant l'architecture du composant, devient explicite et vérifiable. Cela assure une meilleure cohérence entre les différentes versions d'un composant.

Sur le plan conceptuel, nous prévoyons d'étudier l'impact de la composition sur le contrat d'évolution. Pour l'instant, le concepteur doit, de lui-même, déduire cet impact. Nous prévoyons également de formaliser plus avant notre mécanisme d'association.

Sur le plan de l'outillage nous envisageons deux prolongements possibles :

1) Pour l'instant, l'outil `ACE` ne prend en compte que des descripteurs d'architecture définis avec le modèle `Fractal`. Mais, il a été conçu pour intégrer facilement d'autres modèles de composants. Nous confrontons le méta-modèle UML 2.0 avec notre méta-modèle. Nous espérons, à la fin de ce travail, pouvoir proposer un méta-modèle suffisamment générique pour supporter tous les modèles de composants du marché.

2) L'intégration de notre outil de vérification du contrat d'évolution dans un AGL permettrait de guider, de manière continue, l'auteur de l'évolution. Après chaque modification de l'architecture, une évaluation du contrat serait réalisée en arrière plan. L'auteur de l'évolution serait, donc, prévenu de la violation du contrat avant d'aller plus avant dans ses modifications.

Pour plus d'information sur notre travail ou pour télécharger l'outil `ACE`, le lecteur est invité à visiter l'adresse suivante :
http ://www-valoria.univ-ubs.fr/Composants/se/current/Cell.

7. Bibliographie

[BAS 01] BASS L., KLEIN M., BACHMANN F., « Quality Attribute Design Primitives and the Attribute Driven Design Method », *Proceedings of the 4th International Workshop on Product Family Engineering*, Bilbao, Spain, octobre2001.

[BAS 03] BASS L., CLEMENTS P., KAZMAN R., *Software Architecture in Practice, 2nd Edition*, Addison-Wesley, avril2003.

[BEN 96] BENNETT K., « Software evolution : past, present and future », *Information and Software Technology*, vol. 38, n° 11, 1996, p. 671–732.

[BOK 99] BOKOWSKY B., « CoffeeStrainer : Statically-Checked Constraints on the Definition and Use of Types in Java », *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, Toulouse, France, septembre1999, Springer-Verlag, p. 355-374.

[BRU 04] BRUNETON E., « Fractal ADL Tutorial, Version 1.2 », http ://fractal.objectweb.org/tutorials/adl/, mars2004.

[DRE 02] DRESDEN T. U., « OCL Compiler web site », http ://dresden-ocl.sourceforge.net/, 2002.

[ERL 00] ERLIKH L., « Leveraging Legacy System Dollars for E-Business », *IEEE IT Professional*, vol. 2, n° 3, 2000.

[ISO 01] ISO, « Software Engineering - Product quality - Part 1 : Quality model », International Organization for Standardization web site. ISO/IEC 9126-1. http ://www.iso.org, juin2001.

[KLA 97] KLARLUND N., KOISTINEN J., SCHWARTZBACH M. I., « Formal Design Constraints », *Theory and Practice of Object Systems*, , mars1997, p. 1–11, John Wiley & Sons, Inc.

[LEH 85] LEHMAN M., BELADY L., *Program Evolution : Process of Software Change*, London : Academic Press, 1985.

[LIE 81] LIENTZ B. P., SWANSON E. B., « Problems in Application Software Maintenance », *Communiactions of the ACM*, vol. 24, n° 11, 1981, p. 763–769.

[MCK 84] MCKEE J., « Maintenance as Function of Design », *Proceedings of AFIPS National Computer Conference*, Reston, Virginia, USA, 1984, p. 187–193.

[OMG 03] OMG, « UML 2.0 OCL Final Adopted Specification », Object Management Group web site : http ://www.omg.org/docs/ptc/03-10-14.pdf, octobre2003.

[SEA 03] 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, 1 édition, janvier2003.

[WUY 98] WUYTS R., « Declarative Reasoning about the Structure of Object-Oriented Systems », *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS-USA)*, Santa Barbara, California, USA, août1998, IEEE Computer Society, p. 112–124.