Vérification des propriétés structurelles et non fonctionnelles d'assemblages de composants UML2.0

Mourad Kmimech*, Mohamed Tahar Bhiri*
Mohamed Graiet*, Philippe Aniorté**

*MIRACL, ISIMS, BP 1030, Sfax 3018, TUNISIE
mkmimech@gmail.com
tahar_bhiri@yahoo.fr
mohamed.graiet@imag.fr
** LIUPPA, I.U.T. de Bayonne / Pays Basque – 2 Allée du Parc de Montaury Anglet, France
philippe.aniorte@iutbayonne.univ-pau.fr

Résumé. L'approche par composant vise la réutilisation par un assemblage aisé et cohérent des composants. Un assemblage cohérent de composants exige la vérification des propriétés liées à la cohérence d'interface, sémantique, de synchronisation et non fonctionnelle. Nous visons la vérification des propriétés structurelles et non fonctionnelles sur un assemblage de composants UML2.0 dotés des propriétés non fonctionnelles (performance, fiabilité, sécurité, sûreté de fonctionnement, etc.) décrites dans un langage de type CQML. Notre approche basée sur des contrats d'assemblage établis entre les composants serveurs et les composants clients, plaide en faveur de l'utilisation de l'ADL Acme/Armani comme machine de vérification d'un assemblage de composants UML2.0/CQML.

1 Introduction

L'approche par composants vise la réutilisation par assemblage aisé et cohérent des composants. Vis-à-vis de l'approche par objets, cette approche déplace la complexité d'un graphe de classes (hiérarchie de classes, redéfinition de méthodes et relation client) vers des points de connexion entre les composants en distinguant de façon nette deux types de composants : composant serveur et composant client. Afin de vérifier la cohérence -absence de contradiction- d'un assemblage de composants, une approche contractuelle basée sur des contrats d'assemblage établis entre les composants serveurs et les composants clients est préconisée. Celle-ci distingue quatre niveaux de contrats d'assemblage (Beugnard et al., 1999), (Beugnard, 2005) : contrats syntaxiques (ou structurels), contrats sémantiques, contrats de synchronisation et contrats de qualité de services (Propriétés Non-Fonctionnelles : PNF). Cette approche contractuelle inter-composants est perçue comme un prolongement à la conception par contrats (Design by Contracts) célèbre dans le monde OO et supportée par divers langages comme Eiffel (Meyer, 1992), OCL (OMG, 2004) et JML (Leavens et al., 2000).

CQML (Aagedal, 2001) couplé à UML2.0 permet la description des PNF d'un assemblage de composants UML2.0 sans pouvoir les analyser. Dans cet article, nous proposons une ap-

proche qui permet la vérification des PNF d'un assemblage de composants UML2.0 (OMG, 2004) dotés des PNF. Pour y parvenir, nous proposons d'utiliser l'ADL Acme (Garlan et al., 2000) (Garlan et Schmerl, 2006a) comme une machine de vérification des assemblages de composants UML dotés des PNF décrites dans un langage de type CQML (Aagedal, 2001). Le choix de cet ADL est défendu par sa richesse et sa possibilité de typage qui permet de définir des types d'éléments architecturaux (Composants, Connecteur, Port, Rôle, etc.) et des propriétés spécifiques à ces éléments. De plus, l'ADL Acme est couplé à un langage de prédicats assez puissant appelé Armani (Garlan et al., 2000), (Garlan et al., 2001). Ce dernier permet de spécifier des contraintes sous formes des invariants et des heuristiques. En outre, Acme/Armani est supporté par l'environnement AcmeStudio (ABLE, 2009). Ce dernier permet d'analyser et d'évaluer les contraintes spécifiées en Armani.

L'article est structuré de la manière suivante. La section 2 examine les travaux relatifs à la vérification d'assemblages de composants logiciels. La section 3 présente les formalismes utilisés par l'approche préconisée, à savoir le modèle de composants UML2.0, le langage de modélisation des propriétés non fonctionnelles CQML et le langage de description d'architecture Acme/Armani. La section 4 présente notre style Acme/Armani "*SCUML*" (Kmimech et al., 2009b), (Kmimech et al., 2009a) permettant la formalisation en Acme/Armani d'un assemblage de composants UML2.0 dotés des PNF décrites en CQML. La section 5 présente la vérification de la cohérence d'assemblages de composants UML2.0/CQML en Acme/Armani. La section 6 présente une étude de cas d'un système CaméraVidéo (Blair et Stefani, 1998) décrit par un assemblage de composants UML2.0 dotés des PNF. Enfin, la section 7 présente une conclusion et des perspectives envisageables de ce travail. L'annexe donne la traduction de la description UML2.0/CQML du système CaméraVidéo sous forme d'un système Acme/Armani en passant par notre style "*SCUML*".

2 Travaux relatifs

Tout d'abord, nous présentons une vision contractuelle permettant de vérifier la cohérence d'un assemblage de composants en distinguant quatre niveaux : contrats syntaxiques, contrats sémantiques, contrats de synchronisation et contrats de Qualité de Services (QdS). Ensuite, nous abordons la vérification statique d'assemblages de composants en se limitant aux contrats syntaxiques et QdS. Enfin, nous proposons notre approche de vérification d'assemblages de composants décrits par le modèle de composants semi-formel UML2.0.

2.1 Classification des contrats

Afin de formaliser les relations conceptuelles fortes (client et héritage) entre les classes, Bertrand Meyer a introduit le paradigme de la conception par contrat (Design by Contract) (Meyer, 1992), (Meyer, 1997). En effet, son langage Eiffel supporte d'une façon native la conception par contrat. Notons au passage que la conception par contrat est une application pratique des travaux de Hoare liés à la spécification pré/post des programmes (Hoare, 1969). En Eiffel, les prédicats **require** (précondition), **ensure** (postcondition) et **invariant** (invariant) permettent de décrire un contrat dit contrat client entre l'objet client d'une méthode (ou routine en Eiffel) et l'objet serveur (ou fournisseur) qui implante cette méthode. Le développement de la notion de composant (Szyperski, 2002) a offert une opportunité d'appliquer cette vision

contractuelle afin de vérifier la cohérence d'un assemblage de composants. En effet, les travaux décrits dans (Beugnard et al., 1999), (Beugnard, 2005) proposent une classification des contrats selon 4 niveaux (voir Figure 1). Cette classification est considérée comme un prolongement des propositions de Bertrand Meyer sur la conception par contrat.

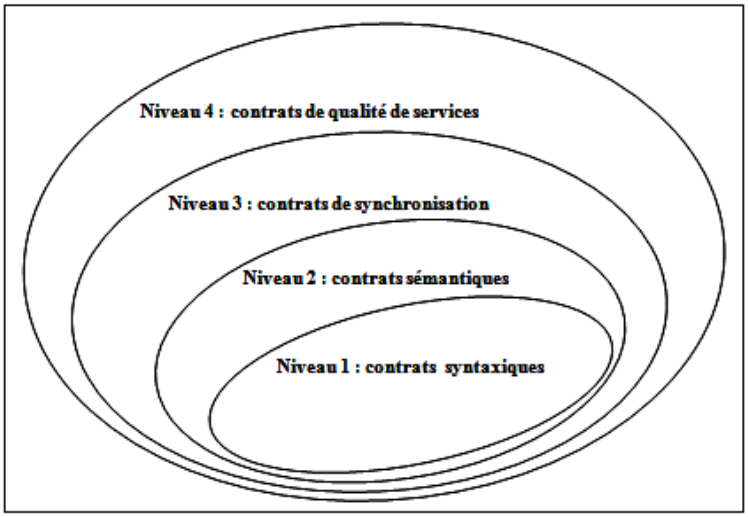

FIG. 1 – *Niveaux de contrats pour les composants*

2.1.1 Les contrats syntaxiques

Les contrats syntaxiques permettent de vérifier la conformité entre les signatures des opérations des interfaces. La signature d'une opération peut comporter les éléments suivants :

- nature de l'opération : opération de construction, consultation ou modification,
- paramètres formels : pour chaque paramètre, trois informations à prendre en considération à savoir son type, sa position et sa nature logique (in, out et in/out),
- exceptions levées.

Les incohérences détectées sont liées principalement à l'incompatibilité de types (type mismatch) en tenant compte des possibilités de typage offertes par le langage de description d'architectures utilisé. De même, nous pouvons étendre ce niveau en incluant les contrats structurels. De tels contrats expriment des contraintes liées aux règles de composition structurelle qui varient en fonction du modèle de composants traité. Par exemple, dans un assemblage UML2.0, un connecteur d'assemblage doit établir un lien entre une interface offerte et une interface requise de deux composants différents.

2.1.2 Les contrats sémantiques

La sémantique d'une opération offerte/requise figurant au sein d'une interface offerte/requise peut être décrite en utilisant la conception par contrat : précondition, postcondition et invariant. Une telle sémantique peut être exprimée en utilisant un langage de contraintes de type OCL (Warmer et Kleppe, 2003).

2.1.3 Les contrats de synchronisation

Les contrats de synchronisation s'intéressent à l'enchaînement des opérations acceptées et/ou demandées (Samek, 2005). Ces contrats peuvent être décrits en utilisant des formalismes à base d'algèbres de processus, IOLTS (Input Output Labeled Transition Systems) et PSM (Protocol State Machine).

2.1.4 Les contrats de qualité de services

Les propriétés non fonctionnelles (Taylor et al., 2009), (Chung et al., 1999) (Performance, Sûreté, Disponibilité, Fiabilité, etc.) sont classiquement considérées après avoir réalisé le logiciel. Des travaux de recherche visent la prise en compte des PNF -appelées aussi qualitatives ou QoS- dès la phase de conception architecturale (Frolund et Koistinen, 1998). Taylor et Al. (Taylor et al., 2009) proposent un guide méthodologique favorisant l'obtention d'un assemblage de composants ayant les PNF suivantes : Efficiency, Complexity, Scalability, Adaptability et dependability. Le travail décrit dans (Chung et al., 1995) propose une approche guidée par les PNF permettant de comparer plusieurs architectures logicielles (AL) à bases des composants alternatives à une application donnée : KWIC (Key Word In Context). Les AL considérées dans ce travail sont : Shared Data, Abstract Data Type, Implicit Invocation et Pipe & Filter. Les PNF utilisées comme critères de sélection sont : Modifiability, Space Performance, Time Performance et Reusability.

Les contrats de qualité de services permettent de décrire les propriétés non fonctionnelles souhaitées ou offertes par une opération, une interface ou un composant. Sachant qu'une propriété non fonctionnelle (PNF) d'une entité logicielle est une contrainte liée à l'implémentation et la présentation de ses fonctionnalités (Taylor et van der Hoek, 2007). Parmi les PNF, nous citons : performance, sûreté, disponibilité, fiabilité, complexité, réutilisabilité, extensibilité, etc. Plusieurs formalismes de description des PNF sont proposés tels que : CQML (Aagedal, 2001), un profil UML pour CQML (Aagedal et Ecklund, 2002), un profil UML pour la qualité de services (OMG, 2008).

Dans la suite, nous nous limitons à l'étude des différentes approches permettant de vérifier les contrats syntaxiques et QdS.

2.2 Vérification des contrats syntaxiques

Les contrats syntaxiques englobant les propriétés liées à la compatibilité des signatures des opérations (offertes et requises) et les propriétés structurelles limitant les connexions entre les composants. Ces contrats dépendent des possibilités de typage (types prédéfinis, constructeurs de types simples, constructeurs de types composés, redéfinition de méthode et surcharge) et des règles de composition du modèle de composants traité. Les modèles de composants qui

	Modèles de composants	**Outils de vérification**	**Commentaires**
Contrats syntaxiques	UML2.0/OCL, Fractal/CCLJ, Acme/Armani	Évaluateur des prédicats logiques	OCL est plutôt adapté au monde OO
Contrats de QdS	AADL/Propriété,	Plug-in OSATE logiques	AADL vise des PNF spécifiques : sécurité et sûreté
	Acme/Armani	Évaluateur des prédicats	Le concept Property d'Acme

TAB. 1 – *Description et vérification des contrats syntaxiques et de QdS*

supportent des langages de contraintes peuvent spécifier les contrats syntaxiques notamment les propriétés structurelles comme des propriétés invariantes. Celles-ci sont vérifiées à l'aide d'un évaluateur de prédicats logiques. Parmi ces modèles, nous citons UML2.0 (OMG, 2004), Fractal (Bruneton et al., 2004) et Acme (Garlan et al., 2000) dotés respectivement d'un langage de contraintes OCL (OMG, 2004), CCLJ (Collet et al., 2005) et Armani (Garlan et al., 2001). Mais contrairement à CCLJ et en particulier à Armani, OCL n'est pas dédié à exprimer des contraintes sur des modèles à composants. Il est plutôt conçu pour spécifier des contraintes sur des modèles orientés objets.

2.3 Vérification des contrats de qualité de services

Rares sont les modèles de composants qui offrent des mécanismes permettant de décrire les PNF et les contrats de QdS. Le modèle de composants AADL (SAE, 2004) introduit la notion de propriété. A chaque composant, nous pouvons associer des propriétés et leur donner des valeurs. Le plug-in OSATE (SAE, 2008) permet l'analyse de propriétés spécifiques telles que : niveaux de sécurité et de sûreté. Le modèle de composants Acme offre des facilités permettant la description des PNF en utilisant notamment le concept Property. Également, Armani couplé à Acme permet de spécifier les contrats de QdS d'un assemblage de composants Acme. Hormis l'évaluateur des prédicatfs Armani faisant partie intégrante de la plate-forme AcmeStudio, Acme ne supporte pas d'outils spécifiques d'analyse des PNF.

2.4 Approche proposée

La table 1 récapitule les possibiliés des modèles de composants examinés vis-à-vis de la description et vérification des contrats applicatifs : syntaxiques et de qualité de services.

Le modèle de composants semi-formel UML2.0 ne permet pas de décrire tous les aspects d'une application à base de composants. Il a besoin d'autres formalismes intégrables dans UML2.0 comme CQML (voir 3.2) afin de spécifier les aspects non fonctionnels.

Notre approche de vérification de la cohérence d'assemblages de composants UML2.0 dotés des PNF décrites en CQML consiste à traduire d'UML2.0/CQML vers Acme/Armani. Ceci favorise la **continuité** entre le modèle de composants semi-formel source (UML2.0/CQML) et le modèle de composants formel cible (Acme/Armani). Ainsi, on se démarque de la plupart des travaux existants (Lanoix et Souquières, 2008), (Lanoix et al., 2008), (Mouakher et al., 2008) utilisant des techniques et des outils généraux tels que B (Abrial, 2005) et CSP (Hoare, 1985).

3 Les modèles et langages retenus

3.1 Le modèle de composants UML2.0

UML 2.0 (OMG, 2004) apporte vis-à-vis d'UML1.x des nouveaux concepts permettant de modéliser un assemblage de composants. En effet, le modèle de composants UML2.0 (OMG, 2004) introduit la notion de composant ainsi que la notion de connecteur. L'introduction de ces concepts, ainsi que ceux de port ou encore la distinction entre interfaces offertes et requises fournit une palette d'éléments de notation intéressante pour la modélisation des architectures logicielles à base des composants. En plus, UML2.0 supporte un langage de contraintes OCL (OMG, 2004). Ce dernier permet de spécifier les contrats syntaxiques notamment les propriétés structurelles comme des propriétés invariantes. Celles-ci sont vérifiées à l'aide d'un évaluateur de prédicats logiques. Mais OCL n'est pas dédié à exprimer des contraintes sur des modèles à composants. Il est plutôt conçu pour spécifier des contraintes sur des modèles orientés objets en utilisant les clauses **pre**, **post** et **inv**. Tandis que le langage Armani (voir section 3.3) couplé au modèle de composants Acme permet de décrire des propriétés architecturelles sous formes d'invariants ou heuristiques attachées à divers éléments architecturaux (composant, port, connecteur, role, style et configuration). Contrairement à OCL, Armani offre un ensemble d'opérations de manipulation spécifiques aux architectures logicielles : fonctions de type, fonctions de graphe, fonctions de propriété et fonctions d'ensemble. En outre, le modèle de composants UML2.0 ne permet pas de décrire tous les aspects d'une application à base de composants. Il a besoin d'autres formalismes intégrables dans UML2.0 comme CQML (voir section 3.2) afin de spécifier les propriétés non fonctionnelles.

3.2 Le langage de spécification des propriétés non fonctionnelles CQML

Rares sont les modèles de composants qui offrent des mécanismes permettant de décrire les PNF et les contrats de qualité de services. Le langage CQML Aagedal (Aagedal, 2001) est un langage permettant d'exprimer des PNF des différents modèles à composants. Jan Oyvind Aagedal (Aagedal, 2001) a développé un argumentaire intéressant en faveur de CQML. Il a établi 25 exigences souhaitées dans un langage d'expression de propriétés de qualité de services pour les composants logiciels. Parmi ces exigences, nous citons : Generality, UMLintegration, Separation. Vis-à-vis de ces exigences CQML est mieux noté que QML, QDL et QuO. En effet, CQML est générique, sépare l'aspect qualitatif de l'aspect fonctionnel et intégrable à UML. CQML est principalement basé sur les trois concepts suivants :

- Le concept caractéristique est la construction de base d'une spécification CQML. Cette caractéristique (dimension dans QML (Frolund et Koistinen, 1998)) représente un aspect non fonctionnel tel que performance, fiabilité, disponibilité, etc. Chaque caractéristique possède un nom et un domaine. Le domaine est constitué d'une direction (increasing ou decreasing), d'un ensemble de valeurs possibles (numeric, set ou enum) et peut avoir une unité. Les caractéristiques CQML peuvent être paramétrées. Les paramètres admis peuvent être des opérations, des classes ou des interfaces aux sens d'UML. En outre, une caractéristique CQML peut inclure dans sa définition un invariant exprimé à l'aide des prédicats OCL comme elle peut avoir aussi une clause "Values" exprimant la formule de calcul de la valeur de la qualité.

- Le concept qualité permet de spécifier une catégorie de qualité d'un service ou d'un ensemble de services. Toute qualité CQML est identifiée par un nom et un ensemble de sous-qualités et peut avoir des paramètres. Chaque sous-qualité est définie par une contrainte exprimée en OCL. Cette contrainte présente une restriction du domaine d'une caractéristique non fonctionnelle.
- Le concept profile permet d'attacher à chaque composant ses propriétés non fonctionnelles. CQML sépare une propriété offerte d'une autre requise :
  - le mot-clé "provides" indique que toutes les qualités qui suivent ce mot sont de type qualité offerte.
  - le mot-clé "uses" indique que toutes les qualités qui suivent ce mot sont de type qualité requise.

La Figure 2 issue de (Aagedal, 2001) donne le méta-modèle de CQML. Les méta-classes ayant des noms en gras correspondent aux concepts fournis par CQML. Tandis que les associations qui portent des noms en gras correspondent aux mots-clés fournis par CQML.

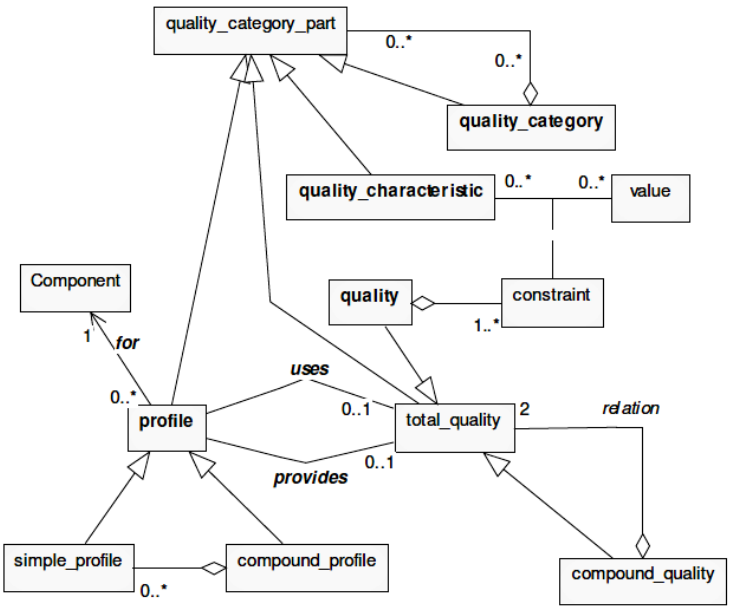

FIG. 2 – *Méta-modèle de CQML*

Par exemple, la propriété non fonctionnelle temps de réponse peut être définie en CQML comme suit :
Quality_characteristic *TempsDeReponse* { **domain : decreasing numeric real** msec ;}

Le domaine de variation de la caractéristique *TempsDeReponse* est de type (**numeric real**). Le spécifieur peut attribuer à sa guise une unité à la caractéristique définie : ici l'unité msec. La direction de variation d'une caractéristique peut être soit **increasing**, soit **decreasing**. Pour la caractéristique *TempsDeReponse*, la direction est naturellement **decreasing**.

A partir de la caractéristique *TempsDeReponse*, on peut définir plusieurs qualités. Par exemple, les deux qualités *efficace* et *treseffícace* sont définies comme suit :
Quality *efficace* {*TempsDeReponse* <= 30}
Quality *treseffícace* {*TempsDeReponse* <= 10}

L'expression *TempsDeReponse* <= 30 est dite contrainte simple en CQML. Elle comporte : une caractéristique définie précédemment, un opérateur relationnel OCL et une constante.

Dans ce travail nous avons retenu un sous-ensemble des constructions CQML. Les constructions liées aux grandeurs statistiques telles que **maximum**, **minium**, **range**, **mean**, **variance** et les concepts relatifs au modèle de calcul de CQML tels que **Operation**, **EventSequence**, **TraceableEntity**, **Location**, **Context**, **Flow** ne sont pas pris en compte. En fait, ces possibilités de CQML favorisent l'écriture des contraintes OCL vérifiables lors de **l'exécution** d'un assemblage de composants. Tandis que notre objectif est de vérifier **statiquement** la cohérence d'un assemblage de composants dès les phases d'analyse architecturale.

Bien que CQML apparu comme le langage le plus complet pour la spécification des PNF des composants logiciels, son utilisation reste limitée à cause du manque de sémantique. En effet, le niveau sémantique de CQML est exprimé sans bases formelles, ceci ne permet pas une vérification formelle des contrats non fonctionnels lors de la spécification de l'application. De plus, CQML est aussi peu implanté dans des outils permettant la vérification des contrats non fonctionnels lors de l'assemblage de composants.

Notons au passage que le langage CQML+ (Röttger et Zschaler, 2003) considéré comme une extension à CQML apporte un nouveau concept appelé **Resource** permettant de définir des PNF relatives aux ressources matérielles attendues et/ou exigées par les composants : **Resource_CPU**, **Resource_Memory** et **Resource_Network**. Nous n'avons pas retenu le concept **Resource** de CQML+ car il concerne plutôt les phases d'implémentation d'une architecture logicielle.

3.3 Le langage de description d'architecture Acme/Armani

Dans ce travail, nous utilisons l'ADL Acme (Garlan et al., 2000) comme une machine de vérification des assemblages de composants UML/CQML. En effet, l'ADL Acme (Garlan et al., 2000) offre des concepts architecturaux structuraux tels que component, connector, role, port, representation, system et style. En outre, il fournit un langage de prédicats assez puissant appelé Armani (Garlan et al., 2001) avec des fonctions appropriées au domaine d'architecture logicielle. Le langage Armani permet de décrire des propriétés architecturales sous formes d'invariant ou d'heuristique attachées à n'importe quel élément architectural (component, family, system, connector, etc.). De telles propriétés sont exécutables au sein de l'environnement AcmeStudio (ABLE, 2009). De même, l'ADL Acme supporte la notion de type. On peut définir des types d'éléments architecturaux (type de composant, de connecteur, de rôle, de port et de style). En plus, Acme offre des facilités permettant la description des PNF en utilisant notamment le concept Property. Egalement, Armani couplé à Acme permet de spécifier les PNF d'un assemblage de composants Acme. Hormis l'évaluateur des prédicats Armani faisant

partie intégrante de la plate-forme AcmeStudio, Acme ne supporte pas d'outils spécifiques d'analyse des PNF. La Figure 3 illustre la définition d'un style Pipe-Filter (*PipeFilterFam*). Ce style ou cette famille définit deux types de composants (*FilterT* et *UnixFilterT*), un type de connecteur (*PipeT*) et une contrainte qui stipule que tout connecteur doit être de type `PipeT`. Celle-ci est formulée en Armani comme suit :
rule type_pipet=**invariant forall** c in self.Connectors|HasType(c,PipeT) ;

Une contrainte en Armani peut avoir une étiquette (ici type_pipet). Le prédicat qui suit le mot **invariant** exprime la propriété invariante. L'ensemble *self.Connectors* est un ensemble prédéfini. Il contient tous les connecteurs définis au sein de la configuration (construction system en Acme) qui dérive da la famille ou style *PipeFilterFam*. Sachant que *HasType* est une fonction booléenne prédéfinie d'Armani, **forall** est le quantificateur universel ($\forall$) et **in** est l'opérateur d'appartenance ensembliste ($\in$).

```
Family PipeFilterFam = {
Component Type FilterT = {
                Ports { stdin; stdout; };
                Property throughput : int;
        };
Component Type UnixFilterT extends FilterT with {
        Port stderr;
        Property implementationFile : String;
        };
Connector Type PipeT = {
        Roles { source; sink; };
        Property bufferSize : int;
        };
Property Type StringMsgFormatT = Record [ size:int; msg:String; ];
rule type_pipet=invariant forall c in self.Connectors | HasType(c, PipeT);
}
```

FIG. 3 – *Définition d'un style Pipe-Filter en Acme*

4 Formalisation d'assemblages de composants UML2.0/CQML en Acme/Armani

Dans cette partie, nous proposons un style Acme/Armani appelé *SCUML* (Kmimech et al., 2009b), (Kmimech et al., 2009a) permettant de représenter en Acme/Armani les principaux concepts issus du modèle de composants UML2.0 tels que : interface offerte, interface requise, composant et connecteur d'assemblage (voir Figure 4). Pour y parvenir, nous avons utilisé avec profit les possibilités de typage offertes par Acme : property type, port type, component type, connector type, role type, set, record, sequence.

Les règles de cohérence relatives au modèle de composants UML 2.0 sont modélisées par des propriétés invariantes en utilisant le concept invariant d'Acme. De telles règles de cohérence permettent de vérifier des propriétés structurelles telles que :

- tous les ports d'un composant UML2.0 doivent être des interfaces UML2.0

```
Family SCUML = {
//**formalisation des concepts liés au modèle de composants UML2.0 **//
//une interface requise est modélisée par un type de port Acme
   port type RequiredInterface = {
          property services: set {string};
   }
//une interface offerte est modélisée par un type de port Acme

  port type ProvidedInterface = {
          property services: set {string};
   }

//un composant UML2.0 est modélisé par un type de composant Acme
   component type ComponentUML ={
//tous les ports d'un composant UML2.0 doivent être des interfaces UML2.0
     rule RCcomposant1= invariant forall p:port in self.ports|
         declaresType (p, ProvidedInterface) or
         declaresType (p, RequiredInterface);
       }

//un connecteur d'assemblage UML2.0 est un connecteur binaire
   connector type ConnectorUML = {
     rule RCconnecteur1  = invariant size (self.roles) == 2;
     role client = { //ce rôle est relié à une seule interface requise
       rule RCconnecteur2 = invariant size (self.attachedPorts) == 1 and
          forall p:Port in   self.attachedPorts |
              declaresType (p, RequiredInterface);
       };
     role serveur = {//ce rôle est relié à une seule interface offerte
       rule RCconnecteur3 = invariant size (self.attachedPorts) == 1 and
          forall p:Port in   self.attachedPorts |
              declaresType (p, ProvidedInterface);
      };
}}
```

FIG. 4 – *Description du style* SCUML *en Acme/Armani*

- un connecteur d'assemblage UML2.0 est un connecteur binaire

Acme/Armani fournit des types prédéfinis (int, float, boolean et string), des constructeurs de types simples (enum) et structurés (record, sequence et set). Nous avons utilisé avec profit ces possibilités de typage offertes par Acme/Armani afin d'enrichir notre style *SCUML* par trois nouveaux types de propriétés (Caracteristique, Qualite et Profile) permettant la formalisation des PNF des composants UML2.0 en Acme/Armani. Ces trois types de propriété sont inspirés du langage CQML.

4.1 Formalisation du concept "Caracteristique"

La caractéristique de qualité est la construction de base de toute spécification CQML. Cette caractéristique représente un aspect non fonctionnel tels que la performance, la fiabilité, la disponibilité, etc. On peut formaliser une caractéristique de qualité par une propriété Acme/Armani. Cette propriété doit être de type enregistrement (record) composé de deux champs :

- Nom : qui représente le nom de la caractéristique (Performance, Disponibilité,etc). Ce champ peut être modélisé par une propriété de type chaîne de caractères.
- Domaine : qui représente le domaine de la caractéristique. Ce champ peut être modélisé par un enregistrement composé de trois champs :
  - Direction : qui modélise la direction (increasing ou decreasing) de la caractéristique. Ce champ doit être de type énuméré (enum {increasing, decreasing}).
  - DomDefinition : qui modélise l'ensemble de valeurs possibles de la caractéristique. Ce champ est de type chaîne de cacartères.
  - Unite : qui modélise l'unité de la caractéristique si elle existe. Il est de type chaîne de cacartères.

La Figure 5 illustre la formalisation en Acme/Armani du concept Caracteristique par un type de propriété. Ce nouveau type de propriété représente un type de base de toute caractéristique non fonctionnelle.

```
property type Caracteristique = record
[Nom : string;
Domaine : record
[ direction: enum {increasing, decreasing};
domDefinition: string};
unite: string; ];
];
```

FIG. 5 – *Formalisation du concept "Caracteristique" en Acme/Armani*

4.2 Formalisation du concept "qualité"

Une qualité CQML spécifie un ensemble de PNF proposées par un composant. Chaque PNF représente une restriction du domaine d'une caractéristique de qualité. Cette restriction est généralement de la forme suivante : "CaracteristiqueNF Operateur Valeur" avec CaracteristiqueNF correspond à une caractéristique non fonctionnelle, Operateur est un simple opérateur de comparaison (<, <=, =, > ou >=) et Valeur correspond à la valeur permettant de restreindre le domaine de la caractéristique. Une qualité peut avoir un nom. On peut formaliser une qualité par une propriété Acme. Cette propriété doit être de type enregistrement composé de deux champs :

- Nom : qui représente le nom de la qualité (GoodAvailability, MediumAvailability, etc.). Ce champ peut être modélisé par une chaîne de caractères.
- SetPNF : qui correspond à l'ensemble des PNF. Ce champ peut être modélisé par un ensemble d'éléments (PNF) de type enregistrement composé de trois champs :
  - CaracteristiqueNF : qui modélise la caractéristique d'une PNF de cette qualité. Ce champ doit être de type "Caracteristique",

- Operateur : qui modélise l'opérateur de la contrainte appliquée sur cette caractéristique. Ce champ doit être de type enum {<, <=, =, >, >= },
- Valeur : qui modélise la valeur permettant de restreindre la caractéristique de cette PNF. Ce champ doit être de type réel puisque le type réel regroupe tous les types numériques.

```
property type Qualite = record [
Nom : string ;
SetPNF : set { PNF };
];
property type PNF =record
[CaracteristiqueNF : Caracteristique;
Operateur : OperComparaison;
Valeur : float; ];
property type OperComparaison= enum
{Inferieur, InferieurOuEgal, Superieur, SuperieurOuEgal, Egal};
```

FIG. 6 – *Formalisation du concept "Qualite" en Acme/Armani*

La Figure 6 illustre la formalisation en Acme/Armani du type propriété "Qualite" qui représente un type de base des qualités formalisées en Acme/Armani.

4.3 Formalisation du concept "profile"

Un composant peut avoir plusieurs qualités qui peuvent être requises et/ou offertes. De la même façon que CQML, nous avons proposé de regrouper les qualités de chaque composant dans un profile. Ce profile peut être modélisé en Acme/Armani par une propriété de type enregistrement composé de deux champs :

- QualitesExigees : qui représente l'ensemble des qualités exigées par un composant. Ce champ doit être de type ensemble de propriétés de type "Qualite",
- QualitesFournies : qui représente l'ensemble des qualités fournies par un composant. Ce champ doit être de type ensemble de propriétés de type "Qualite".

La Figure 7 illustre la formalisation en Acme/Armani du concept profile par un type de propriété.

```
property type Profile = record [
QualitesExigees : set { Qualite };
QualitesFournies : set { Qualite };
];
```

FIG. 7 – *Formalisation du concept profile en Acme/Armani*

5 Vérification de la cohérence d'assemblages de composants UML2.0/CQML en Acme/Armani

Rappelons que Armani est un langage de prédicats assez puissant. Ce langage couplé à Acme permet de décrire des propriétés architecturales sous formes d'invariant ou d'heuristique attachées à divers éléments architecturaux : composant, port, connecteur, style et configuration. Acme/Armani est supporté par un environement de développement appelé AcmeStudio (ABLE, 2009). Celui-ci implémente un outil d'analyse et de vérification des contraintes Armani : évaluateur des prédicats logiques. L'outil de vérification et d'analyse est directement intégré dans l'environement AcmeStudio. Ce qui permet à l'outil de fournir des messages d'erreurs de haut niveau d'abstraction en relation avec la conception d'architecture étudiée. De cette manière, l'architecte logiciel peut comprendre l'erreur et modifier sa conception. Enfin, Acme/Armani s'accompagne également d'une librairie d'API écrites en Java à l'intention des développeurs d'outils.

Notre style *SCUML* (voir Figure 4) décrit précédemment propose une structure préétablie permettant de traduire en Acme/Armani un assemblage de composants UML2.0 dotés des PNF. Afin de vérifier la cohérence non fonctionnelle d'un assemblage de composants UML2.0/CQML, nous proposons d'étendre le style *SCUML* par des contrats Armani permettant de vérifier la cohérence non fonctionnelle d'un assemblage de composants UML2.0. Mais la vérification non fonctionnelle exige la vérification fonctionnelle (ici on se limite à la vérification syntaxique). Pour y parvenir, nous proposons des contrats syntaxiques permettant de s'assurer de la cohérence syntaxique d'un assemblage de composants UML2.0.

5.1 Contrats syntaxiques

Plusieurs règles de cohérence relatives à un assemblage de composants UML2.0 sont modélisées par des propriétés invariantes. Ces règles sont définies au niveau M2 (niveau style ou familly). De telles règles de cohérence permettent de vérifier des propriétés structurelles génériques telles que :

- " composants_admis " : cette règle stipule que seuls les composants de type ComponentUML sont admis dans un assemblage de composants UML2.0, (voir figure 8),
- " connecteurs_admis " : cette règle stipule que seuls les connecteurs d'assemblage de type AssemblageUML sont admis dans un assemblage de composants UML2.0, (voir figure 8),
- " appelant_appele " : cette règle stipule que l'appelant et l'appelé doivent être différents dans un assemblage de composants. C'est-à-dire que chaque connecteur d'assemblage est binaire d'une part et d'autre part les attachements se font entre une interface requise et interface offerte, (voir figure 8),
- " interface_requise_satisfaite " : cette règle stipule que chaque interface requise doit être satisfaite, (voir figure 8).

Les propriétés architecturales décrites dans la Figure 8 sont vérifiées sur des configurations (*system* en Acme/Armani) qui dérivent directement ou indirectement de notre style *SCUML*.

```
rule composants_admis=invariant forall c:Component in
self.COMPONENTS|declaresType(c,ComponentUML) ;

rule connecteurs_admis=invariant forall con:Connector in
self.CONNECTORS|declaresType(con,ConnectorUML);

rule appelant_appele=invariant forall c:Component in
self.COMPONENTS|forall p1:Port in c.PORTS|forall p2:Port in c.PORTS|
declaresType(p1,ProvidedInterface) and declaresType(p2,RequiredInterface)
-> (forall con:Connector in self.CONNECTORS|forall r1:Role in
con.ROLES|forall r2:Role in con.ROLES|!(attached(p1,r1)and
attached(p2,r2)) );

rule interface_requise_satisfaite=invariant forall c:Component in
self.COMPONENTS|forall p:Port in c.PORTS|declaresType(p,RequiredInterface)
-> (exists con:Connector in self.CONNECTORS|exists r:Role in
con.ROLES|attached(p,r));
```

FIG. 8 – *Formalisation des règles de cohérence relatives à un assemblage de composants UML2.0 en Acme/Armani.*

Une propriété non respectée signalée par un prédicat (expression booléenne qui suit le mot réservé **invariant**) évalué à Faux traduit forcément une erreur structurelle architecturale qui devrait être corrigée par l'architecte.

5.2 Contrats non fonctionnels

Afin de vérifier la cohérence non fonctionnelle (ou de qualité) d'un assemblage de composants UML2.0, nous proposons le contrat non fonctionnel "*CQuality*" défini d'une façon informelle comme suit :

- "*CQuality*" : toutes les qualités exigées par un composant doivent être assurées par les composants connectés à ce dernier. Une Qualité requise QRequise est assurée par un composant C si et seulement si ce dernier propose une qualité offerte QOfferte répondant à la qualité requise QRequise. Une qualité QRequise est satisfaite par une qualité offerte QOfferte si et seulement si toutes les PNF formant la qualité QRequise sont assurées par celles formant la qualité QOfferte.

La Figure 9 montre une formalisation de cette contrainte par un invariant Armani. Sachant que PNF1 et PNF2 représentent respectivement une contrainte requise et fournie. Notre contrat *"CQuality"* garantit que la contrainte fournie implique la contrainte requise en tenant compte des propriétés usuelles des opérateurs relationnels (<, <=, > et >=). Nous avons utilisé avec profit les ensembles prédéfinis offerts par Armani tels que **Components** et **Properties** désignant respectivement l'ensemble des composants de la configuration courante qui dérive de notre style *SCUML* (self.Components) et l'ensemble des propriétés attachées à un composant appartenant à self.Components. En outre, nous avons utilisé judicieusement les deux quantificateurs d'Armani **forall** ($\forall$) et **exists** ($\exists$).

```
rule CQuality = invariant
//lignes A1, B1 et C1 traitent les qualités exigées par chaque composant
//considéré : Comp1
 forall Comp1: ComposantUML in self.Components|                    //A1
     forall prof1:Profile in  Comp1.properties|                    //B1
        forall QoSExigee:Qualite in prof1.QualitesExigees|         //C1
//lignes A2, B2 et C2 ont pour objectif de vérifier s'il existe un
//composant Comp2 attaché à Comp1 offrant la qualité requise par Comp1
   exists Comp2: ComposantUML in {select C: ComposantUML in
self.Components|  connected (Comp1, C) } |                        /A2
   exists prof2:Profile in Comp2.properties|                      //B2
         exists QoSFournie:Qualite in prof2.QualitesFournies|     //C2
//toutes les PNF numériques de  la qualité  exigée sont assurées par
//celles de la qualité fournie
  ( forall PNF1 :PNFNum in QoSExigee.SetPNFNum|
      exists PNF2 :PNFNum in QoSFournie.SetPNFNum|
       PNF1.CaracteristiqueNF == PNF2.CaracteristiqueNF and
       ((PNF1.Operateur == SuperieurOuEgal and
        (PNF2.Operateur==SuperieurOuEgal or PNF2.Operateur==Superieur)->
         PNF2.Valeur >= PNF1.Valeur) and (PNF1.Operateur == Superieur and
        (PNF2.Operateur==SuperieurOuEgal or PNF2.Operateur==Superieur)->
         PNF2.Valeur > PNF1.Valeur) and PNF1.Operateur == InferieurOuEgal
         and (PNF2.Operateur==InferieurOuEgal or
         PNF2.Operateur==Inferieur)-> PNF2.Valeur <= PNF1.Valeur) and
        (PNF1.Operateur == Inferieur and (PNF2.Operateur==InferieurOuEgal
         or PNF2.Operateur==Inferieur)-> PNF2.Valeur < PNF1.Valeur)) and
         ((PNF2.Operateur==Egal and PNF1.operateur==Egal)->
          PNF2.Valeur==PNF1.Valeur))
  and
 //toutes les PNF ordinaires de  la qualité  exigée sont assurées par
// celles de la qualité fournie
  ( forall PNF1 :PNFOrdi in QoSExigee.SetPNFOrdi|
     exists PNF2 :PNFOrdi in QoSFournie.SetPNFOrdi|
     PNF1.CaracteristiqueNF  == PNF2.CaracteristiqueNF and
     PNF1.Operateur==PNF2.Operateur
// la vérification des valeurs  des caractéristiques ordinaires peuvent
// être spécifiées au niveau du système
)  ;
```

FIG. 9 – *Formalisation du contrat non fonctionnel en Acme/Armani.*

6 Etude de cas : système Caméra Vidéo

Le système Caméra Vidéo numérique (VideoCamera) (Blair et Stefani, 1998) est composé essentiellement par trois éléments : une Caméra qui capture des séquences vidéos, une Mémoire qui lui permet d'enregistrer ces séquences et d'un VideoPlayer permettant la présentation des séquences déjà enregistrées.

6.1 Spécification informelle du système Caméra Vidéo

Pour mieux comprendre le fonctionnement de ce système, nous allons essayer de détailler les trois composants qui le constituent :

- Le premier composant appelé Camera, propose une interface appelée Memorization. Cette interface comporte deux services (Stored et Stoped).

- Le second composant appelé Memory, propose à son tour deux interfaces. La première exige les services proposés par l'interface Memorization du composant Camera. La seconde, appelée VideoStream, fournit les quatre services (Play, Forward, Rewind et Stop).
- Le troisième composant appelé VideoPlayer exige une interface VideoStream et fournit une autre interface appelée VideoPresented. Cette interface définit le service Presented.

Pour garantir le bon fonctionnement de ce système, on décide de vérifier quelques propriétés non fonctionnelles de ces composants. Les propriétés traitées dans ce système sont : la fiabilité, la disponibilité et quelques propriétés de performance telles que le débit et le temps de réponse. La spécification suivante illustre la signification détaillée de ces PNF.

- Fiabilité (Reliability) : la probabilité qu'un composant soit en état de fonctionnement (sans panne). Cette propriété assure la continuité du service ;
- Disponibilité (Availability) : la probabilité qu'un composant soit en état de marche à un instant donné. Cette propriété assure que le service soit prêt à l'emploi ;
- Débit : taux de transfert de données (ici image) par unité de temps ;
- Temps de réponse : temps entre l'activation et la réponse d'un service ;

Le composant Camera offre un taux de disponibilité supérieur à 90% et un niveau de fiabilité supérieur à 80%. Le composant Memory exige un taux de disponibilité supérieur à 85%, un niveau de fiabilité supérieur à 70%. En outre, ce composant offre une performance acceptable (c'est-à-dire le temps de réponse du service Play est inférieur ou égal à 20 msec et le taux de transfert des données lors de l'utilisation du service Play est supérieur ou égal à 25 image/sec). Le composant VideoPlayer exige une bonne performance (c'est-à-dire le temps de réponse du service Play est inférieur ou égal à 15 msec et le taux de transfert des données lors de l'utilisation du service Play est supérieur ou égal à 30 image/sec).

6.2 Modélisation de l'architecture du système de Caméra Vidéo en UML2.0/CQML

La Figure 10 présente une description architecturale du système " CameraVideo " en utilisant un assemblage de composants UML2.0. Cet assemblage est accompagné d'une description CQML (voir Figure 11) des différentes PNF proposées par les composants constituant cet assemblage.

6.3 Vérification de l'assemblage en Acme/Armani

Un assemblage de composants UML2.0/CQML est modélisé par un système Acme/Armani qui dérive du style *SCUML*. Ceci permet de vérifier les règles de cohérence ainsi que les contraintes syntaxiques et non fonctionnelles spécifiées au niveau du style "*SCUML*" sur l'assemblage de composants UML2.0 décrit en Acme/Armani.

L'annexe donne la traduction de la description UML2.0/CQML du système CameraVideo sous forme d'un système Acme/Armani en passant par le style "*SCUML*".

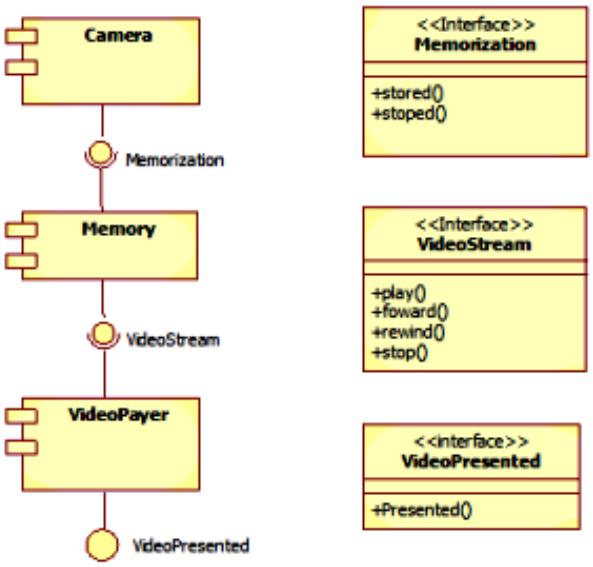

FIG. 10 – *Description architecturale en UML2.0 du système CameraVideo.*

La contrainte non fonctionnelle "*CQuality*" spécifiée au niveau de notre style *SCUML* et exécutée au sein de la configuration *CameraVideo* dérivant de *SCUML* est évaluée à Faux (voir Figure 12). Ceci traduit forcément une incohérence qualitative (ou non fonctionnelle) dans l'assemblage des composants UML2.0/CQML de l'application Caméra Vidéo proposée précédemment. L'architecte est invité à localiser et corriger l'erreur architecturale en tenant compte du message fourni par la plate-forme AcmeStudio (ABLE, 2009) et précisément son outil d'analyse et de vérification des contraintes Armani.

7 Conclusion

Dans ce travail, nous avons apporté une contribution liée à la détection de non cohérence des propriétés non fonctionnelles (PNF) d'un assemblage des composants UML2.0. Pour y parvenir, nous avons retenu le langage CQML afin de spécifier des PNF d'un assemblage des composants UML2.0. Nous avons utilisé avec profit les possibilités offertes par Acme/Armani telles que constructeurs de types simples et structurés, concepts architecturaux structuraux, fonctions d'architecture appropriées, fonctions d'analyse et invariant afin de formaliser des assemblages de composants UML2.0 dotés des PNF décrites dans un langage de type CQML.
Nous avons regroupé au sein d'un style Acme/Armani appelé *SCUML* la formalisation des concepts structuraux issus du modèle de composants UML2.0 et des concepts non fonctionnels venant du langage CQML. Les concepts structuraux traités sont : interface offerte, interface requise, composant et connecteur d'assemblage. Tandis que les concepts non fonctionnels formalisés sont : caractéristique de qualité, qualité et profile. De plus, notre style *SCUML* propose

```
quality_characteristic Fiabilite {
    domain : increasing numeric real [0..100]  %;  }
quality FiabiliteAcceptable  {
    Fiabilite  >= 70  ; };
quality FiabiliteBonne {
    Fiabilite >= 80  ; };

quality_characteristic Disponibilite {
    domain : increasing numeric real [0..100]  % ;  }
quality DispBonne {
    Disponibilite  >= 85 ; };
quality DispTresBonne {
    Disponibilite  >= 90 ; };

quality_characteristic TempsDeReponse {
        domain : decreasing numeric msec;  }
quality_characteristic TauxDeTransfert {
        domain : increasing numeric Integer image/sec ; }
quality PerformanceAcceptable  {
    TempsDeReponse  <= 20  ;
    TauxDeTransfert >= 25  ; };
quality PerformanceBonne {
    TempsDeReponse  <= 15  ;
    TauxDeTransfert >= 30  ; } ;
profile QoSCamera for Camera {
    provides DispTresBonne   and
             FiabiliteBonne  ; }
profile QoSMemory for Memory {
    uses  DispBonne           and
          FiabiliteAcceptable ;
      provides PerformanceAcceptable  ;     }

profile QoSVideoPlayer for VideoPlayer {
    uses    Performance Bonne ; }
```

FIG. 11 – *Description en CQML des PNF des composants du système CameraVideo.*

un contrat de qualité "*CQuality*" exprimé sous forme d'une propriété invariante. Ce contrat permet de vérifier la cohérence non fonctionnelle d'un assemblage de composants UML2.0 dotés des PNF.

Dans cet article, nous avons traité uniquement la vérification des propriétés structurelles et non fonctionnelles d'assemblages de composants UML2.0/CQML en passant par l'ADL formel Acme/Armani. Dans (Kmimech, 2010), nous proposons une approche de vérification des contrats de synchronisation d'assemblages de composants UML2.0 en passant par l'ADL formel Wright (Allen, 1997). Etant donné, la proximité des concepts entre Acme et Wright, l'écriture de passerelles entre ces deux ADL ne devrait pas poser des problèmes.

Quant aux perspectives immédiates de ce travail, nous pourrions envisager les prolongements suivants :

- Automatiser la transformation d'UML2.0/CQML vers Acme/Armani en utilisant une approche de type IDM (Bézivin, 2004). Ceci nécessite l'élaboration d'un méta-modèle partiel UML2.0, méta-modèle partiel CQML, méta-modèle Acme/Armani et expression des règles de traduction UML2.0/CQML vers Acme/Armani en utilisant un langage de transformation de modèles tels que ATL (Jouault et Kurtev, 2005). Une telle auto-

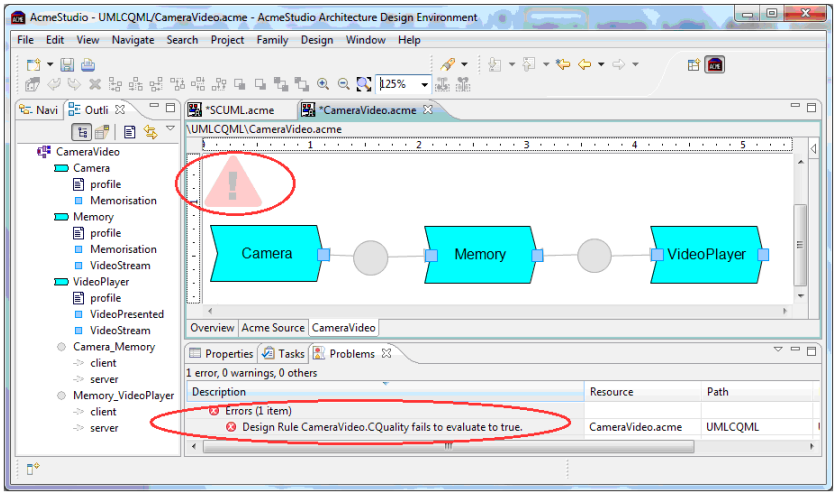

FIG. 12 – *Représentation graphique du système CameraVideo en Acme/Armani.*

matisation devrait valider les règles de traduction intuitives proposées dans ce travail d'UML2.0/CQML vers Acme/Armani en adoptant une approche de tests syntaxiques (Xanthakis et al., 1999).

- Déduire les PNF globales d'un assembl age de composants UML2.0 à partir des PNF locales attachées à chaque composant. En effet, la traduction d'UML2.0/CQML vers Acme/Armani rend possible l'utilisation des outils d'analyse externes (Garlan et Schmerl, 2006b) associés à l'environnement AcmeStudio.

CQML attache des PNF aux composants. Ainsi, le contrat non fonctionnel *CQuality* (voir 5.2) proposé dans cet article est associé aux composants. Nous comptons dans un futur proche élargir notre approche contractuelle afin d'associer des contrats aux interfaces. Ceci permettrait de couvrir le cas où l'on a plusieurs interfaces d'un composant requérant connectées à des composants fournisseurs différents. em En outre, il reste à intégrer dans le contrat *CQuality* des combinaisons correctes telles que >= (côté requis) et = (côté fourni) susceptibles de générer des faux négatifs.

Références

Aagedal, J. O. (2001). *Quality of Service Support in Development of Distributed Systems*. Phd thesis, University of Oslo.

Aagedal, J. O. et E. F. Ecklund, Jr. (2002). Modelling qos: Towards a uml profile. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, UML '02, London, UK, UK, pp. 275–289. Springer-Verlag.

ABLE (2009). (http://www-2.cs.cmu.edu/ acme/AcmeStudio/AcmeStudio.html). Technical report, Carnegie Mellon university ABLE Group, Acme Studio development environment.

Abrial, J.-R. (2005). *The B-book - assigning programs to meanings.* Cambridge University Press.

Allen, R. (1997). *A Formal Approach to Software Architecture*. Ph. D. thesis, School of Computer Science.

Beugnard, A. (2005). *Contribution à l'assemblage d'entités logicielles*. Habilitation à diriger des recherches, Université de Bretagne Sud.

Beugnard, A., J.-M. Jézéquel, N. Plouze, et D. Watkins (1999). Making components contract aware. *IEEE Computer 32*(7), 38–45.

Blair, G. et J.-B. Stefani (1998). *Open Distributed Processing and Multimedia. Addison Wesley Longman Ltd.*

Bruneton, E., T. Coupaye, et J.-B. Stefani (2004). The fractal component model. http ://fractal.objectweb.org/specification/.

Bézivin, J. (2004). Sur les principes de base de l'ingénierie des modèles. *L'OBJET 10*(4), 145–157.

Chung, L., B. Nixon, et E. Yu (1995). Using Non-Functional Requirements to Systematically Select Among Alternatives in Architectural Design. In *1st International Workshop on Architectures for Software Systems*, Seattle, pp. 31–43.

Chung, L., B. A. Nixon, E. Yu, et J. Mylopoulos (1999). *Non-Functional Requirements in Software Engineering*. The Kluwer international series in software engineering. Kluwer Academic Publishers Group, Dordrecht, Netherlands.

Collet, P., R. Rousseau, T. Coupaye, et N. Rivierre (2005). A contracting system for hierarchical components. In *CBSE'05*, pp. 187–202.

Frolund, S. et J. Koistinen (1998). QML : A language for quality of service specification. Technical Report HPL-98-10, Hewlett-Packard Software Technology Laboratory.

Garlan, D., R. T. Monroe, et D. Wile (2000). Acme : Architectural description of component-based systems. In G. T. Leavens et M. Sitaraman (Eds.), *Foundations of Component-Based Systems*, pp. 47–68. Cambridge University Press.

Garlan, D., R. T. Monroe, et D. Wile (2001). Capturing software architecture design expertise with armani. Technical Report CMU-CS-98-163, Carnegie Mellon University School of Computer Science.

Garlan, D. et B. Schmerl (2006a). Architecture-driven modelling and analysis. In T. Cant (Ed.), *Eleventh Australian Workshop on Safety Critical Systems and Software (SCS 2006)*, Volume 69 of *CRPIT*, Melbourne, Australia, pp. 3–17. ACS.

Garlan, D. et B. Schmerl (2006b). Architecture-driven modelling and analysis. safety critical systems and software. In *Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems*, Melbourne, Australia.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM 12*(10), 576–580.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Jouault, F. et I. Kurtev (2005). Transforming models with atl. In *Proceedings of the Model Transformations in Practice Workshop at MoDELS 2005*, Montego Bay, Jamaica.

Kmimech, M. (2010). *Vérification d'assemblages de composants logiciels : Application aux modèles de composants UML2.0 et Ugatze*. Ph. D. thesis, l'Université de Pau et des Pays de l'Adour.

Kmimech, M., M. T. Bhiri, et P. Aniorté (2009a). Checking component assembly in acme : an approach applied on uml2.0 components model. In *ICSEA*, Number 494-499, Porto, Portugal.

Kmimech, M., M. T. Bhiri, M. Graiet, et P. Aniorté (2009b). Vérification d'assemblage de composants uml2.0 à l'aide d'acme. In *In workshop LMO/SafeModels*, Nancy, France.

Lanoix, A., S. Colin, et J. Souquières (2008). Développement formel par composants assemblage et vérification à l'aide de b. *Technique et Science Informatiques 27*(8), 1007–1032.

Lanoix, A. et J. Souquières (2008). Trustworthy assembly of components using the b refinement. *e-Informatica 2*(1), 9–28.

Leavens, G. T., K. R. M. Leino, E. Poll, C. Ruby, et B. Jacobs (2000). Jml : notations and tools supporting detailed design in java. In *IN OOPSLA 2000 COMPANION*, pp. 105–106. ACM.

Meyer, B. (1992). Applying "design by contract". *IEEE Computer 25*(10), 40–51.

Meyer, B. (1997). *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall.

Mouakher, I., J. Souquières, et F. Alexandre (2008). Diagnostic et correction d'erreurs de spécifications application à l'assemblage de composants. *L'OBJET 14 (4) 4*, 11–42.

OMG (2004). Uml 2.0 superstructure specification. *October 02*.

OMG (2008). Uml tm profile for modeling quality of service and fault tolerance characteristics and mechanisms sepcification. *Avril 11*.

Röttger, S. et S. Zschaler (2003). Cqml+ : Enhancements to cqml. *In J.-M.Bruel, editor, Proc. 1st Int'lWorkshop on Quality of Service in Component-Based Software Engineering Toulouse, France*, 43–56.

SAE (2004). The sae architecture analysis and design language (aadl) a standard for engineering performance critical systems. *2006 IEEE Conference on ComputerAided Control Systems Design*, 1206–1211.

SAE (2008). Sae aadl information. http ://www.aadl.info.

Samek, J. (2005). *Employing OCL for specifying behavior compliance*. Master's thesis, Charles University, Prague, Faculty of Mathematics and Physis, Prague,.

Szyperski, C. A. (2002). *Component software - beyond object-oriented programming*. Addison-Wesley-Longman.

Taylor, R. N., N. Medvidovic, et E. M. Dashofy (2009). *Software Architecture : Foundations, Theory and Practice*. Addison-Wesley.

Taylor, R. N. et A. van der Hoek (2007). Software design and architecture the once and future focus of software engineering. In *2007 Future of Software Engineering*, FOSE '07, Washington, DC, USA, pp. 226–243. IEEE Computer Society.

Warmer, J. et A. Kleppe (2003). *The Object Constraint Language : Getting Your Models Ready for MDA* (2 ed.). Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc.

Xanthakis, S., P. Regnier, et C. Karapoulios (1999). *Le test des logiciels*. Hermès science Publication : Etudes Et Logiciels Informatiques.

Annexe

Cette annexe donne la traduction de la description UML2.0/CQML du système CameraVideo sous forme d'un système Acme/Armani en passant par notre style *SCUML*.

```
    import families/SCUML.acme ;
System CameraVideo : SCUML = new SCUML extended with {

    //*****************formalisation du composant Camera*****************
Component Camera : ComponentUML = new ComponentUML extended with {
//************partie fonctionnelle**********************************
Port Memorisation : ProvidedInterface = new ProvidedInterface extended with {
property services = {"Stored()","Stoped()"} ;
}
//************partie non fonctionnelle*****************************
property QoSCamera : Profile = [
//spécifiquation de l'ensemble des qualités exigées par le composant
QualitesExigees= { } ;
//spécification de l'ensemble des qualités fournies par le composant
QualitesFournies= {[ Nom ="DispTresBonne" ;
SetSousQualite ={
// PNF1 : Disponibilité >= 90 %[ CaracteristiqueNF =[Nom = "Disponibilité" ;
Domaine = [ direction = increasing ; dom = numeric_real ; Operateur = SuperieurOuEgale ;
Valeur = 90 ; ] } ;]
, //fin de la qualité DispTresBonne [ Nom ="FiabiliteBonne" ;
SetSousQualite ={
// PNF1 : Fiabilité >= 80 % [ CaracteristiqueNF = [Nom = "Fiabilité" ;
Domaine = [ direction = increasing ;
dom = numeric_real ;
unite ="%" ; ] ;
] ;
Operateur = SuperieurOuEgale ;
Valeur = 80 ; ] } ;]
//fin de la qualité FiabiliteBonne
} ;//fin de l'ensemble des qualités offertes par le composant] ;
//fin du profile
} ;//fin du composant Camera
// **************formalisation du Composant Memory*******************
Component Memory : ComponentUML = new ComponentUML extended with {
//****************partie fonctionnelle*******************************
Port Memorisation : RequiredInterface = new RequiredInterface extended with {
property services = {"Stored()","Stoped()"} ;
}
Port VideoStream : ProvidedInterface = new ProvidedInterface extended with {
property services = {"Play()","Forward()","Rewind()","Stop()"} ;
}
//****************partie non fonctionnele*****************************
```

property QoSMemory : Profile = [
//spécifiquation de l'ensemble des qualités exigées par le composant
QualitesExigees= {[Nom ="DispBonne" ;
SetSousQualite ={
// PNF : Disponibilité >= 85 % [CaracteristiqueNF =[Nom = "Disponibilité" ;
Domaine = [direction = increasing ;
dom = numeric_real ;
unite ="'" ;] ;
Operateur = SuperieurOuEgale ;
Valeur = 85 ;] } ;], //fin de la qualité DispBonne [
Nom ="FiabiliteAcceptable" ;
SetSousQualite ={
// PNF : Fiabilité >= 70 %[
CaracteristiqueNF =[
Nom = "Fiabilité" ;
Domaine = [direction = increasing ;
dom = numeric_real ;
unite ="%" ;] ;
Operateur = SuperieurOuEgale ;
Valeur = 70 ;] } ;] //fin de la qualité FiabiliteAcceptable
} ;
//spécification de l'ensemble des qualités fournies par le composant
**********////////////
QualitesFournies= {[Nom ="PerformanceAcceptable" ;
SetSousQualite ={
// PNF1 : TempsDeReponse <= 20 msec[CaracteristiqueNF =[Nom = "TempsDeReponse" ;
Domaine = [direction = decreasing ;
dom = numeric_integer ;
unite ="msec" ;] ;
Operateur = InferieurOuEgale ;
Valeur = 20 ;],
// PNF2 : TauxDeTransfert >= 25 image/sec [CaracteristiqueNF = [Nom = "TauxDeTransfert" ;
Domaine = [direction = increasing ;
dom = numeric_integer ;
unite ="image/sec" ;] ;
Valeur = "" ;
Invar = "" ;] ;
Operateur = SuperieurOuEgale ;
Valeur = 25 ;]
} ;]
} ;]
} ;//fin de l'ensemble des qualités offertes par le composant] ;//fin du profile
}
// **********formalisation du Composant VideoPlayer**********

Vérification des propriétés structurelles et non fonctionnelles d'assemblages de composants UML2.0

```
Component VideoPlayer : ComponentUML = new ComponentUML extended with {
//****************partie fonctionnelle*******************************
Port VideoStream : RequiredInterface = new RequiredInterface extended with {
property services = {"Play()","Forward()","Rewind()","Stop()"} ;
}
//************formalisation de l'interface offerte VideoPresented***
Port VideoPresented : ProvidedInterface = new ProvidedInterface extended with {
property services = {"Presented()"} ;
}
property QoSVideoPlayer : Profile = [
//spécifiquation de l'ensemble des qualités requises
QualitesExigees = { [ Nom ="PerformanceBonne" ;
SetSousQualite ={
// PNF1 : TempsDeReponse <= 15 msec [ CaracteristiqueNF = [Nom = "TempsDeReponse" ;
Domaine = [ direction = decreasing ;
dom = numeric_integer ;
unite ="msec" ; ] ;
] ;
Operateur = InferieurOuEgale ;
Valeur = 15 ; ],
// PNF2 : TauxDeTransfert >= 30 image/sec [ CaracteristiqueNF = [Nom = "TauxDeTrans-
fert" ;
Domaine = [ direction = increasing ;
dom = numeric_integer ;
unite ="image/sec" ; ] ;
] ;
Operateur = SuperieurOuEgale ;
Valeur = 30 ; ]
} ;

    //spécification de l'ensemble des qualités offertes
QualitesFournies= { } ;] ;//fin du profile
} ;
//**********formalisation du Connecteur d'assemblage*****************
Connector Camera_Memory : ConnectorUML = new ConnectorUML extended with
{ }
Connector Memory_VideoPlayer : ConnectorUML = new ConnectorUML extended with
{ }
Attachment Camera.Memorisation to Camera_Memory.serveur ;
Attachment Memory.Memorisation to Camera_Memory.client ;
Attachment Memory.VideoStream to Memory_VideoPlayer.serveur ;
Attachment VideoPlayer.VideoStream to Memory_VideoPlayer.client ;
}// Fin du système CameraVideo
```