Développement basé composant

Une approche centrée composition

Nicolas Belloir — Jean-Michel Bruel

LIUPPA
Université de Pau et des Pays de l'Adour
BP 1155
F-64013 Pau Cedex
{Nicolas.Belloir, Jean-Michel.Bruel}@univ-pau.fr

RÉSUMÉ. *La complexité croissante des SI plaide pour un développement à base de composants. Notre point de vue est que la composition est le point crucial de ce type de développement. Par composition nous entendons la prise en compte de la sémantique de la composition et non plus la simple intégration de composants. Nous présentons dans cet article une démarche centrée sur la composition pour le développement des applications basées composant, couvrant une partie du cycle de développement. La composition est traitée par une relation spécifique basée sur la relation Tout-Partie, permettant de spécifier des propriétés formellement définies et de contraindre les relations structurelles et dynamiques entre composants. La mise en œuvre de cette relation de composition est assurée par une plate-forme de composition dédiée que nous avons développée. Celle-ci comprend un profil UML implémentant la relation de composition et un environnement de déploiement assurant le respect des propriétés de composition.*

ABSTRACT. *The growing complexity of IS pleads for a development of these systems based on components. Our point of view is that composition is the critical point of this kind of development. By composition, we mean semantic integration of components. We present a composition centered approach for component-based development. Composition is treated by a specific relation based on Whole-Part relationship. This relationship can be characterized by formally specified properties and allows to constraint the structural and dynamic relations between components. The realization is realized by a composition framework, consisting in a UML profile that implements the composition relationship and a deployement environment insuring the respect of the composition properties.*

MOTS-CLÉS : *CBSE, composition, UML, MDE.*

KEYWORDS: *CBSE, composition, UML, MDE.*

RSTI - ISI – 10/2005. Méthodes de développement des SI, pages 59 à 80

1. Introduction

Les systèmes d'information modernes se doivent d'être réactifs, fiables, performants, et évolutifs afin de s'adapter aux nouvelles technologies de l'entreprise et aux nouveaux besoins de ses usagers. Dans ce cadre, le développement à base de composant (CBD[1]) apporte une réponse élégante. En plus d'être réutilisables, les composants facilitent le développement de systèmes logiciels plus flexibles, risquant moins de devenir prématurément obsolètes (Bachmann *et al.*, 2000). La construction d'une application consiste donc, non plus en un développement intégral et complet, mais en un assemblage de briques de bases réutilisables.

Les standards actuels (EJB, CCM ou .Net) font interagir des composants qui ont été conçus pour cela. Dans le cas contraire, les composants sont adaptés, ou insérés au sein d'adaptateurs (*i.e.*, *container*, ou *enveloppeur*[2] ou *code glu*) les rendant compatibles entre eux, au prix d'actions complexes. Une des raisons à cela est que les efforts de recherche ont plus porté sur le concept de composant que sur celui de composition d'une part, et d'autre part, plus sur le niveau logiciel (programmation, paradigme) que sur le niveau conceptuel (modèles et démarches). Les techniques d'ingénierie logicielle basées composant[1], sont encore insuffisantes et demandent de plus amples efforts de recherche (High Confidence Software and Systems Coordinating Group, 2001). Dans ce cadre, la formalisation de la composition est particulièrement critique et les techniques de modélisation pêchent par manque de constructions de modélisation dédiées et bien formalisées. En particulier, UML, qui est un standard *de facto* tant au niveau industriel qu'au niveau académique, n'offrait jusqu'à ce jour que peu de moyens permettant de modéliser les concepts de la CBSE. Malgré la dernière évolution du langage, la version 2.0, de nombreuses incohérences sont encore présentes dans la sémantique des concepts propres à la CBSE.

Dans ce cadre, nous proposons une démarche de développement centrée composition. Pour cela, nous décrivons les problèmes liés à la composition à la section 2 puis présentons en section 3 un modèle de composition basé sur la relation Tout-Partie. Ce modèle propose une modification du métamodèle UML permettant la définition de propriétés de composition au niveau conceptuel. Nous détaillons ce que notre modèle apporte aux démarches de développement centrées composants. Nous présentons plus particulièrement à la section 4 une plate-forme de composition supportant le modèle de composition, et décrivons son utilisation.

---

1. Nous utiliserons dans ce document les acronymes anglais CBD (*Component-Based Development*) et CBSE (*Component-Based Software Engineering*) à la place des équivalents français DBC et ILBC peu connus et rarement employés.
2. *Wrapper* en anglais

2. Composition de composants dans le processus de développement

2.1. *La composition des composants*

Il convient de différencier l'intégration de composants (on parle aussi d'assemblage) de la composition. (Stafford *et al.*, 2002) définit l'intégration comme le lien syntaxique entre deux composants et la composition comme la possibilité de spécifier des propriétés à un tout basé sur les propriétés de ses parties et les relations entre elles ; la composition intègre donc l'aspect sémantique d'un assemblage. Tous les assemblages de composants réalisés par intégration ne sont pas viables en exécution. Combiner les dimensions architecturales et sémantiques d'un assemblage est alors nécessaire, dans le but de déterminer quelles combinaisons de composants sont valides.

Nous distinguons deux types de relation de composition. La composition *horizontale* représente l'échange de services entre composants de même granularité. Il s'agit de la relation la plus couramment utilisée pour assembler des composants. La composition *verticale* concerne la réalisation d'un assemblage entre un composant de granularité faible (*sous-composant*) et un composant de granularité plus forte (*composé*). On parle alors d'un composant composé de sous-composants, ou de composition hiérarchique (comme dans Fractal (Bruneton *et al.*, 2002) par exemple).

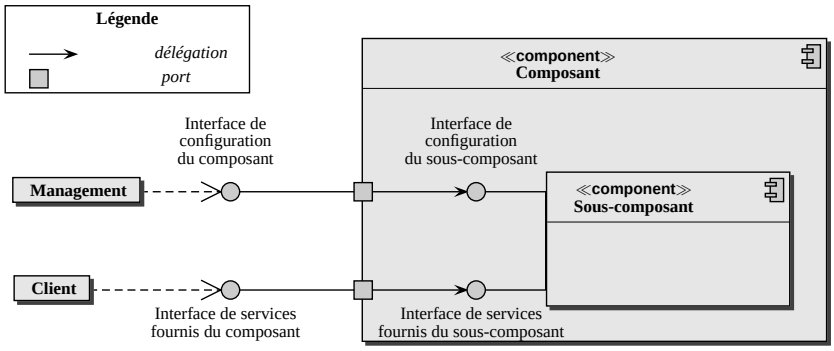

Figure 1. *Organisation canonique d'un composé en regard d'un de ses sous-composants en UML 2.0*

Le composé s'appuie sur les services fournis par un ou plusieurs sous-composants pour proposer à son tour des services plus complexes (voir figure 1). Lors de son déploiement, le composé est configuré afin d'être adapté à son nouvel environnement. La configuration de ses sous-composants peut être nécessaire mais elle est alors à la charge directe du composé. Le *client* du composé interagit avec lui au moyen de son interface de services fournis[3]. Les services décrits, peuvent être implémentés, soit

3. Notons qu'UML 2.0 n'offre pas de moyen prédéfini pour différencier les interfaces de services fournis des interfaces de configuration. C'est au concepteur de définir le moyen qu'il

directement par le composé, soit par un des sous-composants. Ils peuvent également être le fruit d'une conjugaison entre les fonctionnalités du composé et celles d'un ou de plusieurs sous-composants.

Le risque d'augmentation du couplage entre les composants est une des conséquences négatives de la composition verticale, mais dépend de la manière dont le lien de composition est implémenté. Cependant, cette approche permet la réduction de la complexité des applications, en organisant l'architecture en grands groupes de composants. D'autre part, le remplacement d'un sous-composant par un autre n'aura d'impact direct que sur le composé dont il fait partie.

Au niveau conceptuel, la prise en compte de la composition permet de bien spécifier les relations entre les composants, sur des points tels que les dépendances entre composants. Cela est particulièrement important, par exemple pour les préoccupations d'adaptation des composants (McKinley *et al.*, 2004). La spécification des compositions est encore plus vitale dans le cadre des composants boîte noire, puisque dans ces cas, on ne peut pas en général les adapter. Il est alors nécessaire de déterminer les propriétés des compositions pour améliorer la compatibilité des composants lors de leur exécution (Stafford *et al.*, 2002). Cette préoccupation est récurrente en CBSE (Garlan *et al.*, 1995; ARTIST, 2003; Bosch *et al.*, 2003).

D'un point de vue composition logicielle, un certain nombre de modèles de composants (Koala (Ommering, 2002) et Fractal (Bruneton *et al.*, 2002)) et de langages formels de compositions (PICCOLA (Achermann, 2002) et C2 (Ivers *et al.*, 2002)) prennent en compte la composition verticale. On trouve plusieurs travaux visant à améliorer la sémantique de la composition au niveau conceptuel. L'utilisation du patron *Composite* (Gamma *et al.*, 1995) par exemple permet d'organiser les entités à modéliser sous forme hiérarchique dans laquelle entités et compositions d'entités sont traitées uniformément. L'utilisation de contrats (Meyer, 1997) pour spécifier les interactions entre composants est une approche désormais reconnue. (Beugnard *et al.*, 1999) définit quatre niveaux de contrats permettant d'exprimer différents niveaux de sémantique. Le projet ACCORD[4] définit les principes de construction d'une architecture logicielle fondée sur les concepts de composants logiciels et de contrats.

La communauté des méthodes formelles s'est intéressée également à la composition (Kinitry, 2002; Moschoyiannis *et al.*, 2003). Les approches formelles offrent une sémantique précise mais sont encore peu accessibles. A l'inverse, les notations telles qu'UML sont souvent décriées par les scientifiques à cause de leur « manque » de sémantique. Elles proposent cependant un formalisme aisément accessible, connu des industriels, qui a prouvé son utilité dans le développement logiciel (Harel *et al.*, 2004). Le couplage des deux approches apporte une amélioration à l'activité de spécification. Des approches proposent de coupler une méthode formelle avec une notation gra-

---

utilise pour identifier ces interfaces, en utilisant une note par exemple ou bien une convention de notation (les interfaces de services fournis à gauche du composant, celles de services requis à droite et celles de configuration au-dessus du composant).

4. http ://www.infres.enst.fr/projets/accord/

phique pour pallier ce problème (Bruel, 1996; Dupuy, 2000; Ledang, 2002; Petit *et al.*, 2003). Notre cheminement est inverse. Nous nous proposons d'améliorer la sémantique des approches semi-formelles en contraignant leur syntaxe, à l'aide de langages comme OCL. UML est l'approche semi-formelle la plus utilisée dans la CBSE. Ce langage de modélisation utilise d'ailleurs un langage formel, OCL, pour décrire sa sémantique. Nous appliquons notre travail portant sur l'amélioration de la sémantique de la relation de composition au niveau conceptuel à UML.

2.2. *La composition dans les méthodes de développement basées composant*

Quand on parle de méthode de développement basée composant, il convient de distinguer deux processus complémentaires : le processus de développement *POUR* la réutilisation et le processus de développement *PAR* la réutilisation. Nos travaux, même s'ils traitent une problématique transversale, se situent plutôt dans le processus PAR réutilisation, c'est à dire la construction d'application par composition de composants réutilisables.

Dans ce type de processus, on retrouve le même manque de considération pour la composition des composants. Les méthodes focalisent plus sur les composants eux-mêmes ou sur les relations entre composants, mais la plupart du temps sans traiter la sémantique de ces relations, comme par exemple dans (Silaghi *et al.*, 2003). D'ailleurs, elles utilisent souvent UML comme support de modélisation (Hassine *et al.*, 2005), avec les limites sémantiques qu'on lui connaît (Bruel *et al.*, 2003). Dans (Cheesman *et al.*, 2001), la composition est juste évoquée au niveau de la phase d'assemblage et la méthode ne donne aucune aide précise sur le sujet. Dans Symphony (Hassine *et al.*, 2005), la composition verticale est traitée à travers le concept d'*agrégation* de composants, proche de la relation de *Containment* d'OPEN (Firesmith *et al.*, 1998). Elle se limite cependant à une seule vision de la composition.

Les lacunes en termes de prise en compte de la sémantique de la composition se retrouvent à la fois sur les support de modélisation, mais aussi sur les méthodes. C'est pourquoi nos travaux portent fondamentalement sur la définition d'un modèle de composition, mais sans négliger sa mise en œuvre à travers les processus de développement utilisés en CBD, comme nous le montrons dans les sections suivantes.

3. Un modèle de composition

Notre équipe a présenté des travaux portant sur l'amélioration de la sémantique de la composition et de l'agrégation en UML, dans le cadre de la modélisation objet (Barbier *et al.*, 2003). Pour cela, la relation Tout-Partie a été utilisée comme base théorique. Ces travaux ont abouti à une proposition de modification du métamodèle UML et à la définition de propriétés de composition et à leur formalisation en OCL. Nous avons utilisé ces résultats comme point de départ de nos travaux, puis appliqué une démarche similaire à la composition conceptuelle des composants.

3.1. *Différence entre objet et composant*

Chessman et Daniels dans (Cheesman *et al.*, 2001) définissent les composants comme une extension de la technologie orientée objet (OO). A ce titre, un composant est vu comme une unité logicielle structurée selon les principes objets suivants : unification des données et des fonctions, encapsulation et identité unique. Un composant étend ces principes et se différencie d'un objet par les points suivants :

– il fournit une représentation de sa spécification distincte de son implémentation à travers son interface ;

– il possède généralement des moyens de communication plus étendus que les objets (ces derniers se limitent généralement aux mécanismes de message) ;

– il a souvent des capacités de persistance là où un objet est limité à son état local ;

– il est généralement de granularité plus importante qu'un objet et propose des actions complexes *via* ses interfaces.

Le tableau 1 résume les différences entre objets et composants.

La question de savoir si une classe seule peut être un composant est souvent posée. Nous soutenons que, si une classe est livrée avec ses interfaces de services requis et fournis, alors elle peut être considérée comme un composant de granularité faible. La définition précédente présente un point de vue intéressant : il s'agit du fait qu'un composant possède souvent des capacités de persistance. Ce point est en contradiction avec Szyperski qui dit qu'un composant "has no (externally) observable state". Ces deux approches sont souvent débattues dans la communauté composant. L'approche de Szyperski est une vue dans laquelle un composant ne peut être distingué des autres copies de ce composant. L'approche dans laquelle un composant a des capacités de persistance implique le contraire. En effet, à partir du moment où un composant peut stocker des informations, il peut être distingué des autres copies de ce composant. Cette approche nous semble plus ouverte et plus intéressante car elle permet d'utiliser la technologie composant pour un plus grand nombre de cas. Nous adhérons donc à la vue de Chessman et Daniels sur ce point.

Propriété	**Objet**	**Component**
Interface bien définie	Possible	Obligatoire
Capacité à être déployé	Non	Oui
Communication	Invocation de méthode	Invocation de méthode, de service, envoie de signal ...
Capacité de persistance	Etat interne	Possible
Granularité	Simple	Variée

Tableau 1. *Résumé des différences entre un objet et un composant*

3.2. *Modèle de composition basé sur la relation Tout-Partie*

Propriété	Dimension architecturale	Dimension dynamique
Asymétrie au niveau instance	X	
Antisymétrie au niveau type	X	
Encapsulation	X	
Partageabilité	X	
Séparabilité		X
Mutabilité		X
Dépendance de cycles de vie		X

Tableau 2. *Classification des propriétés de composition retenues selon les dimensions architecturale et dynamique*

Nous avons effectué une revue systématique (Belloir, 2004) des propriétés identifiées par Barbier *et al.* (2003). Pour chacune d'elles, nous avons étudié son applicabilité et sa pertinence dans le monde orienté composant, puis en avons étudié les interactions. Le tableau 2 présente les propriétés retenues et leur classification selon leur impact sur la dimension architecturale (modélisation d'un système comme un ensemble de composants interconnectés) ou la dimension dynamique (modélisation d'un système pour supporter les changements dynamiques d'un système déployé) de la composition. Les propriétés ci-dessous sont celles que nous n'avons pas retenues :

– la nature binaire de la relation, car elle est évidente dans le cadre de l'approche composant ;

– les propriétés émergentes et résultantes, car nous les considérons comme des heuristiques de modélisation ;

– la transitivité, car elle est d'une part définie de manière axiomatique dans les travaux de Lesniewski et, d'autre part, incompatible au typage ;

– la configurationalité, car d'une part elle induit des relations ternaires, ce qui est incompatible avec la nature binaire de la relation Tout-Partie, et d'autre part, elle est mal formalisée dans (Barbier *et al.*, 2003).

En UML 1.x, la relation de composition est modélisée à travers les relations de *composition* et d'*agrégation*. Dans (Barbier *et al.*, 2003), ces deux relations sont décrites suivant les propriétés issues de la relation Tout-Partie qui les caractérisaient le mieux, *i.e.* la non partageabilité et l'encapsulation pour la *composition* et la partageabilité et la non encapsulation pour l'*agrégation*. L'*agrégation* traduit une relation de couplage (dépendance). La *composition* traduit un lien plus fort (dépendances de vie par exemple). L'agrégation et la composition sont habituellement décrites comme suffisantes pour caractériser les différentes relations de composition verticale. Nous pensons que ces deux relations sont trop éloignées l'une de l'autre et qu'il existe de la place pour définir d'autres types de relation de composition et offrir ainsi une plus

grande latitude au concepteur. Prenons l'exemple, tiré de (Belloir, 2004), d'une machine à café, comme illustré à la figure 2 en (1)[5]. Celle-ci est composée d'un module de fabrication de boisson et d'un monnayeur. Elle fournit également un mécanisme de porte-monnaie électronique, qu'on peut approvisionner avec le monnayeur. Cet exemple montre la combinaison de 3 relations de composition : COFFEEMACHINE - *Coiner*, COFFEEMACHINE - *DrinkMaker* et E-MONEY - *Coiner*[6]. La partie *(1)* représente le modèle de composants de l'application.

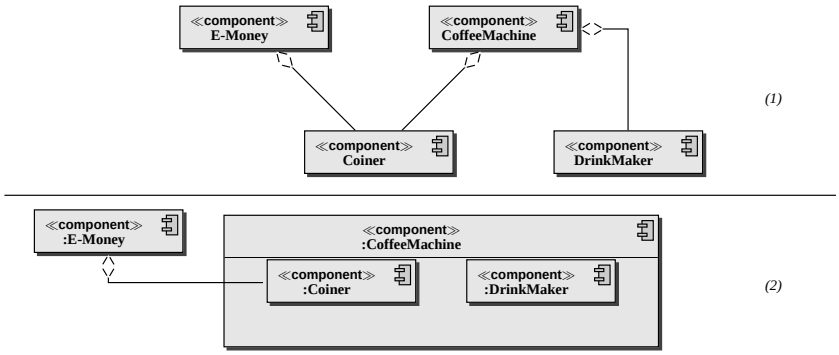

Figure 2. *Relation de composition non exprimable par* agrégation *ou* composition

Admettons que l'on veuille spécifier la propriété d'encapsulation sur COFFEEMACHINE - *Coiner* ainsi que la propriété de partage global (en *(2)*), ce qui n'est pas possible avec la *composition* ni avec l'*agrégation*. Cela signifie que l'instance de *Coiner* est encapsulée à l'intérieur du composant *CoffeeMachine*. Mais cela implique que l'instance de *E-Money* puisse accéder également à l'instance de *Coiner*. D'un point de vue implémentation, l'encapsulation impose que l'instance de *CoffeeMachine* instancie *Coiner*. Il faut donc que *E-Money* ait une référence vers l'instance de *Coiner* pour l'utiliser. Ce cas est intéressant puisque, alors qu'il est réalisable techniquement (en utilisant Fractal, par exemple), il viole la définition classique de l'encapsulation et pose la question de son application stricte. Par définition, elle implique que le sous-composant ne soit accessible par aucun autre composé. Or, la spécification de la non partageabilité globale et locale est équivalente (Belloir, 2004). Il y a donc redondance. Nous considérons donc l'encapsulation comme une spécification architecturale signifiant la contenance physique mais décorrélée de son lien avec la partageabilité. Le couplage de l'encapsulation et du partage global par exemple signifie que, bien qu'encapsulé physiquement au sein du composé initial, le sous-composé peut rendre directement des services à un composé d'un autre type.

5. Nous utilisons comme symbole le losange en pointillé pour représenter une relation de composition. Ainsi, nous évitons le problème de l'incohérence existant entre la notion de *composition* et celle d'*agrégation* dans UML

6. Par convention, nous écrivons en petites majuscules le composant Tout.

Cet exemple montre que les simples relations de *composition* et d'*agrégation* sont insuffisantes pour couvrir les différentes possibilités offertes par la combinaison des propriétés de partage global et d'encapsulation. Nous proposons donc la définition de quatre relations, couvrant la dimension architecturale de la composition. Le tableau 3 résume les différents types de la relation en fonction des propriétés qui les caractérisent. Nous invitons le lecteur à consulter nos travaux antérieurs pour avoir une définition précise de chacune des propriétés. Nous rappelons uniquement la signification de la séparabilité qui traduit la capacité d'un composant partie à être séparé de son tout, de la mutabilité qui traduit la capacité d'un composant partie à changer de type sans modifier intrinsèquement la relation elle-même (remplacement d'un composant *pile* par un autre composant *pile* de type différent par exemple), et la dépendance de cycle de vie qui décrit le lien entre un composant tout et son composant partie en termes de créations et destructions des composants.

	Strong Composition		**Lightweight Composition**		**Strong Aggregation**		**Lightweight Aggregation**	
	stat.	*dyn.*	*stat.*	*dyn.*	*stat.*	*dyn.*	*stat.*	*dyn.*
Encapsulation	oui	oui	oui	oui	non	non	non	non
Partageabilité	non	non	oui	oui	non	non	oui	oui
Séparabilité / mutabilité.	non	oui	non	oui	non	oui	non	oui
Dépendance de cycles de vie	1 cas	4 cas	1 cas	4 cas	9 cas	9 cas	9 cas	9 cas

Tableau 3. *Les différents types de la relation Tout-Partie*

Notre modèle recoupe un certain nombre de travaux portant sur la caractérisation de la composition. Par exemple, la classification de la composition décrite dans (Odell, 1994) est effectuée par la combinaison des 3 critères suivants : la *configuration*, que l'on peut comparer à notre propriété d'encapsulation, la propriété d'*homeoreous*, qui représente une similitude de type entre le tout et la partie et l'*invariance* qui peut être rapprochée de l'inséparabilité. La notion de partageabilité d'une partie vis-à-vis d'autre tout, qui est une composante forte de notre modèle, n'est pas traitée dans le modèle d'Odell. Elle apporte une vue globale de la composition par rapport à son environnement. D'autre part, nous fixons les problèmes de typage avec les propriétés d'asymétrie au niveau des instances et d'antisymétrie au niveau des types. En revanche, on retrouve dans les deux premières classes de composition d'Odell la composition forte statique, et la composition forte dynamique.

3.3. *Application à UML*

Notre travail a porté sur UML 1.x. Nous avons cependant cherché à ce que l'effort nécessaire au passage de nos travaux de UML 1.x à UML 2.0 soit le moins important possible. Pour ce faire, nous avons fait apparaître certains concepts de la version 2.0

dans notre approche. D'autre part, les aspects traitant de la composition entre composants dans UML 2.0 sont encore limitatifs et sans sémantique particulière (Bruel *et al.*, 2003). Nous ne retenons donc pas les concepts tels que les structures composites pour traduire la relation de composition et exprimons nos relations de composition à la manière de l'*agrégation* et de la *composition* dans UML 1.x. Pour ces raisons nous avons choisi de proposer une modification du métamodèle UML spécifique à notre approche et en même temps facilement intégrable dans les versions ultérieures d'UML. Nous avons défini un nouveau métatype abstrait modélisant la relation de composition entre composants. Ce choix spécialise une notion de conception propre à un type de *classifier* particulier. En ce sens, il complexifie le langage UML au détriment de sa généricité. Cependant, cette branche de modèle pourra à terme être insérée dans une version stable d'UML 2.0, en tant que généralisation d'une forme de relation. La prise en compte d'une relation de composition spécifique aux composants est à notre avis indispensable pour supporter l'utilisation de plus en plus importante de ce paradigme.

La figure 3 montre le métamodèle que nous proposons. Les métaclasses ajoutées sont en gris foncé. Les liens ajoutés sont en double épaisseur. Nous avons défini la métaclasse abstraite *Component Relationship*. Elle matérialise toute forme de relation entre composants. Nous matérialisons respectivement les compositions verticales et les compositions horizontales par les deux métaclasses *Vertical Component Relationship* et *Horizontal Component Relationship*. Ces deux métaclasses sont disjointes et implémentent la métaclasse *Component Relationship*.

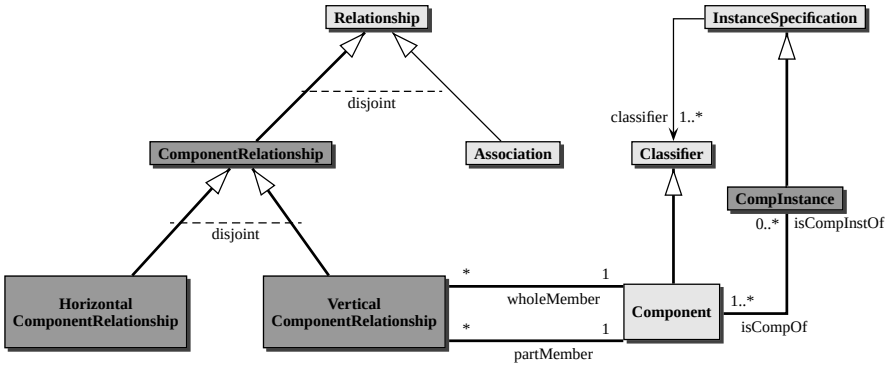

Figure 3. *Ajout des métaclasses modélisant la relation de composition*

La caractérisation de la notion de composant Tout ou Partie se fait sur le *rôle* que joue un composant dans la relation traduit par les liens d'association entre la métaclasse *component* et la métaclasse *Vertical Component Relationship*. Enfin, nous avons spécialisé la métaclasse *InstanceSpecification* en *CompInstance* afin de traduire le concept d'instance de composant. Nous revenons ainsi à un concept qui existait en UML 1.x et qui a disparu dans UML 2.0. Nous justifions ce choix par notre optique de spécialisation orientée composition de composants. Les propriétés de composition

identifiées dans la section précédente caractérisent la *Vertical Component Relationship*. Nous incluons les propriétés paramétrables en tant qu'attributs de celle-ci (voir la figure 4).

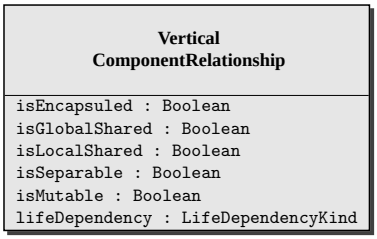

Figure 4. *Détail de la métaclasse* VerticalComponentRelationship

Les quatre types spécifiques de relation Tout-Partie sont insérés dans le métamodèle sous la forme de quatre métaclasses, chacune héritant de *Vertical Component Relationship*. La figure 5 représente cette portion du métamodèle. Nous avons traduit les propriétés sous forme de stéréotypes qui indiquent le positionnement à une valeur spécifique des métaattributs. Cela permet de spécialiser aisément les métaclasses en fonction des stéréotypes. Toutes les relations implémentent les propriétés primaires : nature binaire, asymétrie au niveau des instances et antisymétrie au niveau des types. Elles sont définies sur la relation *Vertical Component Relationship*. Les sous-types de relations de composition en héritent.

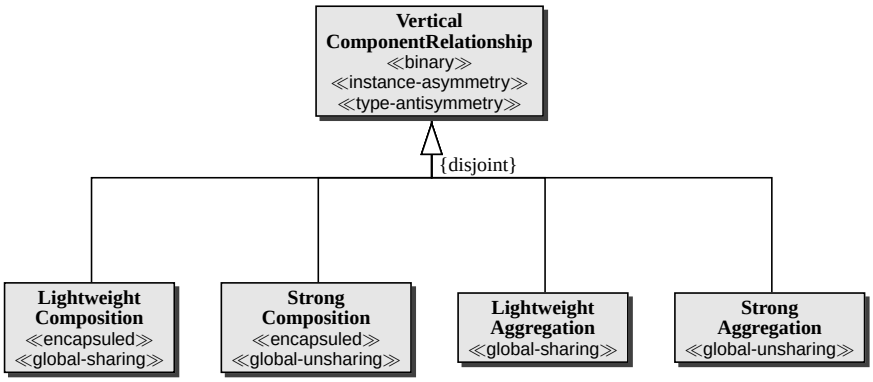

Figure 5. *Ajout des métaclasses spécialisant les différents types de relations*

(Barbier *et al.*, 2003) a défini des règles OCL traduisant les propriétés de la relation Tout-Partie, en s'appuyant sur le métaschéma proposé. Globalement, la sémantique des propriétés de composition que nous avons retenues est la même que celle des propriétés initiales. Aussi les règles OCL que nous associons à notre métamodèle ne sont que des adaptations des règles définies initialement. Seul le contexte est différent. Nous renvoyons le lecteur à (Belloir, 2004) pour plus de détails,

mais à titre d'illustration, voici une règle traduisant l'irréflexivité, nécessaire à la spécification de la propriété d'asymétrie au niveau instance :

```
context VerticalComponentRelationship inv Irreflexivite :
    wholeMember.compInstance->forAll(w | w.oclIsKindOf(
     partMember) implies not w.part->includes(w))

    partMember.compInstance->forAll(p | p.oclIsKindOf(
     wholeMember) implies not p.whole->includes(p))
```

3.4. *Utilisation du modèle de composition*

Nous allons maintenant donner une brève explication sur l'application de notre modèle de composition dans les méthodes de développement orientées composants. De nombreuses études ont été réalisées pour comparer ces méthodes. Nous renvoyons le lecteur à une étude récente (Hassine *et al.*, 2005). Le constat le plus souvent réalisé, et en tout cas celui qui nous intéresse plus particulièrement, est le manque de support pour l'analyse et la validation des assemblages de composants, tant au niveau structurel que comportemental. Pour nous, cela vient d'un manque de précision au niveau des relations de composition. Notre approche n'a pas consisté à développer une nouvelle méthode de développement, mais à améliorer, par la formalisation et l'outillage, la prise en compte du problème essentiel de la composition. Il nous faut néanmoins situer l'utilisation de ce modèle, et nous avons choisi de parler du cycle de développement orienté composants utilisé avec UML (Cheesman *et al.*, 2001) et d'une approche plus récente (Hassine *et al.*, 2005).

Les démarches de développement centrées composants manquent en général de support à la vérification de la conformité des implémentations aux spécifications. Par exemple, dans l'approche de Cheesman & Daniels (*UML Components*), la partie qui traite plus particulièrement de l'assemblage ne fait qu'une page, de même que celle qui traite du déploiement en environnement d'exécution (Cheesman *et al.*, 2001, p. 162-164). L'apport de notre modèle dans cette démarche consiste justement à fournir les supports à ces activités, pourtant cruciales. En renforçant la spécification des composants et de leur architecture, au travers des relations de compositions structurelles et comportementales, nous aidons le développeur dans cette phase du cycle de vie (cf. figure 6 – partie en gras).

Notre modèle traite clairement de la gestion des liens de compositions entre composants depuis la conception détaillée jusque l'implémentation. Dans le cadre de la démarche Symphony, il permet donc de traiter l'analyse structurelle et dynamique. Dans cette démarche la dichotomie entre classe *Maître* et *Partie* sert à mettre en évidence les aspects structuraux. Néanmoins la différenciation se fait au niveau des classes et non des composants eux-même. En fait, le concept Symphony qui se rapproche le plus de notre modèle est celui d'*agrégation de composants*, ou encore *composants imbriqués*.

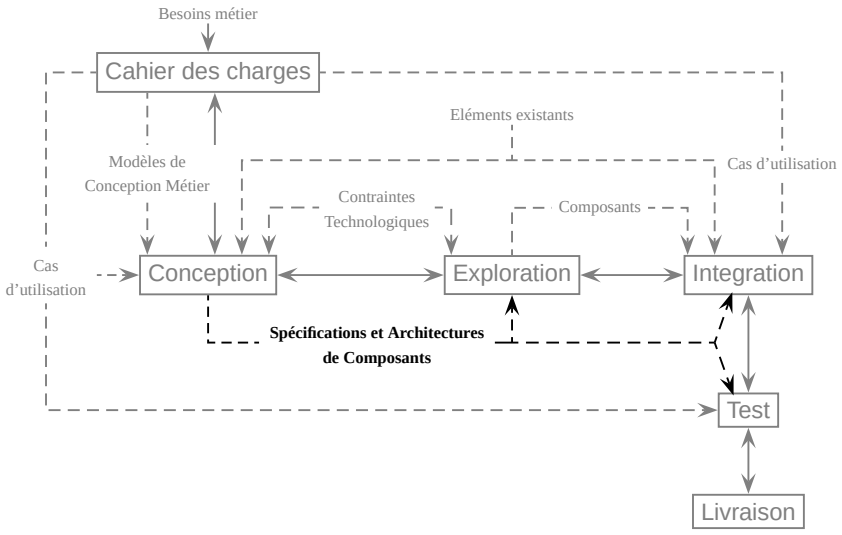

Figure 6. *Le cycle de développement "UML Components"*

Il est utilisé lorsque qu'un composant n'est pas réutilisable dans le contexte de la nouvelle application. En ce sens, on peut considérer cette approche comme la conception d'un *container*. Sa caractérisation selon nos propriétés de composition est la suivante. La relation est non partageable et encapsulée. Il s'agit donc d'une *Strong Composition*. Cette caractérisation semble confirmée par l'utilisation d'un losange noir (composition en UML) dans l'exemple donné par Hassine *et al.* (2005). Le terme agrégation peut donc être considéré comme un abus de langage puisqu'il s'agit plus vraisemblablement d'une composition.

Dans la section suivante, nous présentons une plate-forme centrée composition qui concrétise les arguments que nous venons d'énoncer et permet de montrer la faisabilité et l'utilité de notre modèle.

4. Une plate-forme centrée composition

Dans cette section, nous présentons notre plate-forme centrée composition qui a été créée suite à deux constatations. D'une part, la prise en compte du concept de composition est important à tous les niveaux du cycle de vie d'une application. D'autre part, un des problèmes lié à l'introduction de nouveaux concepts au niveau des modèles est que ces concepts ont rarement une traduction directe en tant que paradigmes de programmation. L'ingénieur chargé de la réalisation des modèles doit donc choisir arbitrairement un moyen d'implémenter le concept avec les paradigmes existants.

Notre plate-forme répond à ces contraintes. Elle se compose d'un profil UML[7] intégrant notre modèle de composition, et d'un environnement de déploiement de composants intégrant le paradigme de la relation de composition définie précédemment.

4.1. *Profil de composition WPCP*

Un profil UML (OMG, 1999) permet d'adapter la sémantique d'UML sans en changer le métamodèle, à l'aide de mécanismes d'extension. Ces mécanismes sont les notions de *stéréotype*, de *valeur marquée* et de *contrainte*. Nous utilisons ces mécanismes pour implémenter le modèle de composition. Nous avons développé notre profil avec le module *Profile Builder* d'*Objecteering*. A notre connaissance, il s'agit du seul logiciel offrant une extension permettant le développement de profils UML.

4.1.1. *Description de WPCP*

Nous utilisons deux types de diagramme de classes spécifiques. Le premier, que nous appelons DTC (Diagramme de Types de Composants), modélise l'architecture d'un point de vue types. Le second, que nous appelons DIC (Diagramme d'Instances de Composants), instancie le DTC et donc représente l'architecture d'un point de vue instances. Les notions de composant et d'instance de composant sont exprimées par les stéréotypes héritant de la métaclasse *Class* : *« WPCPComponent »* et *« WPCPCompInstance »*. Les stéréotypes *« VCR »*, *« SCOMP »*, *« LCOMP »*, *« SAGG »* et *« LAGG »* sont des spécialisations de la métaclasse *Relationship*. Enfin, les stéréotypes *« Whole »* et *« Part »* caractérisent les extrémités de la relation. Les valeurs marquées représentent les métaattributs du métamodèle. Dans notre modèle, seules la métaclasse *VerticalComponentRelationship* et les métaclasses héritant de celle-ci, possèdent des attributs. Les propriétés de composition sont exprimées sous forme de valeurs marquées.

Nous avons développé des commandes permettant de vérifier la sémantique des modèles réalisés en fonction des propriétés de composition. *Objecteering* ne permet pas d'implémenter de règles OCL. Cependant, il est possible de définir des contraintes dans le *langage J*, qui est le langage propriétaire d'*Objecteering*. Nous avons donc traduit les contraintes OCL dans ce langage. L'exemple de code en *langage J* ci-dessous vérifie que la relation est binaire.

```
boolean Association::checkRolesOfRelation () {
// Check relationship size
if (ConnectionAssociationEnd.size != 2) {
    StdOut.write("ERROR on ", Name, " relationship – A
     Composition relationship must be binary : ", NL);
    return false;}
return true; }
```

7. Nommé Whole Part Composition Profil (WPCP)

4.1.2. *Utilisation de WPCP*

Nous illustrons ici l'utilisation de notre profil avec l'étude de cas de la machine à café. Le DTC modélise quatre composants : *CoffeeMachine*, *Coiner*, *DrinkMaker* et *E-Money*. Nous créons quatre classes que nous spécialisons avec le stéréotype *« WPCPComponent »*. Nous définissons ensuite les relations entre ces composants en créant une association pour chaque paire de composants. A l'aide de la fenêtre de propriété de l'association, il faut sélectionner le stéréotype correspondant à la relation de composition voulue. La figure 7 montre une capture d'écran du modèle généré. Une fois le modèle spécifié, le concepteur exécutera une commande qui teste la validité du modèle vis-à-vis du modèle de composition.

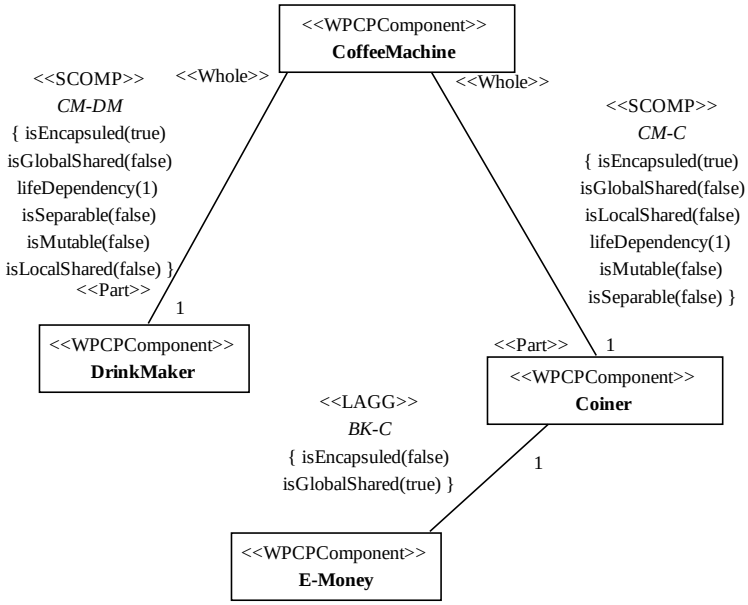

Figure 7. DTC *généré avec le profil* WPCP

Une fois le DTC créé et vérifié, le concepteur spécifie l'application en termes d'instances de composants, et vérifie que ce modèle est bien cohérent avec le DTC. En premier lieu, il convient de créer les instances de composants. Le nombre d'instances d'un même type de composant dépend des relations qui ont été spécifiées. Par exemple, la relation entre *CoffeeMachine* et *Coiner* est de type *SCOMP*. Cela implique les propriétés d'encapsulation et d'exclusion globale. Or, le composant *E-Money* est également en relation avec *Coiner*. La non partageabilité globale dans ce cas implique l'existence de deux instances de *Coiner* : une (*Coiner1 :Coiner*) en relation avec

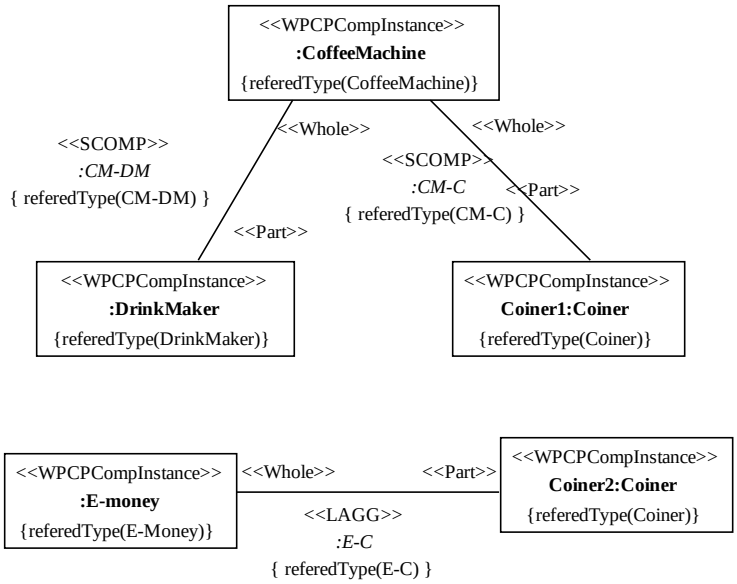

Figure 8. DIC *spécifié avec le profil* WPCP

l'instance de *CoffeeMachine*, l'autre (*Coiner2 :Coiner*) en relation avec l'instance de *E-Money*. Les instances créées, les relations sont spécifiées. Par exemple, une relation de type *SCOMP* est créée entre *:DrinkMaker* et *:CoffeeMachine*. La figure 8 montre une capture d'écran du modèle généré.

Lorsque le DIC est terminé, il reste à vérifier sa cohérence avec les éléments du DTC correspondant (l'outil Objecteering ne faisant pas cette vérification « à la volée »). La deuxième étape de la vérification consiste à évaluer la cohérence du DIC vis-à-vis des propriétés de composition du DTC. Toutes ces vérifications sont réalisées à l'aide de scripts réalisés en *langage J*.

4.2. *Environnement de composition WPCE*

Dans la pratique, la conformité des logiciels développés par rapport aux modèles de conception correspondants n'est malheureusement pas assurée systématiquement. Une des raisons à cela est que les environnements de déploiement n'offrent pas les moyens d'exprimer la totalité des concepts spécifiés au niveau modèle. Pour pallier cela, nous avons développé (Belloir *et al.*, 2004) un environnement de pilotage de

composants Java, nommé WPCE[8], basé sur les postulats énoncés dans la section précédente d'une part, et sur notre modèle de composition d'autre part.

Un environnement spécifique à la composition permet une vérification *a priori* du respect des modèles. En effet, si le modèle de composition spécifie qu'une composition possède telle propriété, et que l'environnement de composition permet de l'implémenter directement au niveau assemblage, le respect de ces propriétés est assuré, par définition, par l'environnement. A la différence d'une approche purement formelle, où il faudrait établir la preuve du respect des propriétés, nous posons comme hypothèse que l'environnement suffit à garantir le respect des propriétés spécifiées.

4.2.1. *L'environnement JMX*

WPCE est basé sur la technologie JMX (Java Management eXtension) (Sun Microsystems Inc., 2001), développée par *Sun Microsystems*, et administrant des composants Java, appelés Managed Beans (MBeans). L'architecture de JMX permet de connecter un navigateur Web au serveur de MBeans : le serveur émet des pages HMTL à partir desquelles il est possible d'administrer les MBeans, en accédant notamment à leurs interfaces de gestion. Le service de relation de JMX définit les classes permettant de construire des relations entre composants MBeans, et centralise toutes les opérations sur ces relations JMX afin de maintenir leur cohérence. Les opérations de ces relations peuvent être rendues alors accessibles par un navigateur Web comme n'importe quel autre MBean.

4.2.2. *Environnement de composition Java pour un développement rigoureux*

Notre proposition a consisté à transformer un composant classique en MBean, et de modifier le service de relations de JMX afin qu'il propose la spécification des propriétés de composition. L'architecture conçue permet d'implémenter rapidement et facilement des sous-types de la relation Tout-Partie. La relation « *WholePart* » est dotée par essence des propriétés primaires, tandis que les propriétés secondaires sont assignées à ses sous-types. Par simplification, nous utilisons les noms abrégés de ces sous-types. L'implémentation proposée suit un modèle similaire pour spécialiser ces relations. Les propriétés sont traduites par des interfaces. Les classes implémentant les sous-types de relation Tout-Partie étendent la classe abstraite *WholePart* et héritent ainsi de toutes les propriétés de la relation. Cela se traduit par le code suivant pour la classe *SCOMP* :

```
public class SCOMP extends WholePart implements
 SCOMPMBean {
    public SCOMP(...) throws JMException {
        super(...);
}}
```

8. pour Whole-Part Composition Environment

Le code est extrêmement simple et il est proposé ici dans son intégralité. Seuls les paramètres des constructeurs ont été omis. Il est également similaire pour les quatre sous-types. Les différenciations entre les sous-types se situent dans l'interface MBean que ces relations implémentent. Celle-ci est composée des interfaces des propriétés secondaires que la relation doit satisfaire. Ainsi seules les propriétés secondaires intrinsèques de ces relations leurs sont transmises. Les interfaces des sous-types de la relation Tout-Partie sont définies ainsi dans le code suivant :

```
public interface SCOMPMBean extends Encapsulation,
                                    GlobalUnsharing,
                                    LifetimeDependency
                                      {
}
```

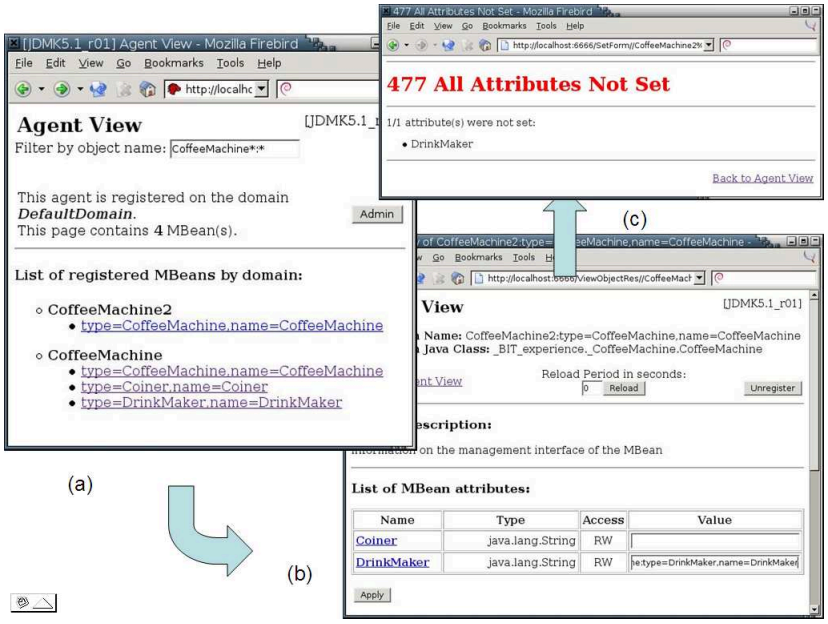

Figure 9. *Vérification d'une propriété de non-partageabilité*

4.2.3. *Exemple d'utilisation*

Afin d'illustrer l'utilisation de notre environnement, déployons notre étude de cas sur WPCE. Pour montrer des fonctionnalités de notre environnement, nous choisissons ici de rendre la relation COFFEEMACHINE - *Coiner* partageable globalement et non encapsulée par le composant *CoffeeMachine*. Selon la nature des propriétés, l'environnement effectue une vérification positive ou négative : soit l'environnement assure

lui-même la propriété de la relation (par implémentation), et dans ce cas on est sûr qu'elle est vérifiée ; soit l'environnement vérifie que la propriété de la relation est cohérente avec les relations déjà définies et interdit cette relation si une incohérence est détectée. Nous illustrons une violation de la non partageabilité entre *DrinkMaker* et *CoffeeMachine*. Dans la figure 9, on voit en (a) qu'il existe deux composants *CoffeeMachine*. L'un des deux est en relation avec un composant *Coiner* et un composant *DrinkMaker*. Le second n'est en relation avec aucun sous-composant. En cliquant sur ce second, on se retrouve sur sa fenêtre d'administration (en (b)) et on va lui spécifier de créer une relation entre le composant *DrinkMaker* utilisé par le premier composant *CoffeeMachine* et lui. L'environnement détecte la violation de la non partageabilité et lève une exception en (c).

5. Conclusion

Les systèmes logiciels à objets et à composants sont une solution élégante et opportune pour la réalisation de systèmes d'information modulables et évolutifs. Dans ce cadre, le concept de composition est fondamental puisqu'il caractérise les règles d'assemblage entre les composants, mais souvent considéré trop tard dans le cycle de développement des applications. Dans ce cadre, nous avons présenté une approche centrée sur la prise en compte de la composition le long du cycle de vie d'une application. Notre approche s'appuie sur un métamodèle UML faisant explicitement apparaître une relation de composition verticale dédiée aux composants ainsi que sur des propriétés de composition formalisées en OCL. Nous avons montré la faisabilité de cette approche en réalisant une plate-forme de composition. Celle-ci est basée sur deux outils : un profil UML pour le niveau conceptuel et une plate-forme de composition pour le niveau logiciel. Les deux outils implémentent tous deux les propriétés de composition sous formes de contraintes et garantissent le respect de ces contraintes au niveau du modèle et de l'application.

Notre travail se situe dans la dynamique actuelle qui accorde de plus en plus d'importance aux modèles dans le développement des applications. Cela implique l'utilisation de langages graphiques dans lesquels les éléments de notation ont une signification claire et précise. Dans ce cadre, nous avons travaillé à améliorer significativement le pouvoir d'expression de la composition dans UML et contribué à améliorer sa « sémantique », c'est-à-dire à contraindre sa syntaxe (Harel *et al.*, 2004).

Nous explorons deux perspectives issues de ces travaux. La première consiste à intégrer dans notre approche l'utilisation du modèle de composant Fractal (Bruneton *et al.*, 2002). Celui-ci prend en effet en compte l'aspect hiérarchique des compositions. D'autre part, il est bâti sur des propriétés implicites dont certaines sont relativement proches de nos propriétés de composition. D'autre part, nous cherchons à automatiser notre méthode de composition en utilisant la transformation de modèles. L'idée est de générer à partir des diagrammes conceptuels indépendants de tout support technologique, le squelette du code de l'application dans l'environnement de déploiement.

Remerciements

Les auteurs tiennent à remercier les membres de l'équipe AOC du LIUPPA, et plus particulièrement Fabien Romeo et Franck Barbier pour leur participation à ces travaux.

6. Bibliographie

Achermann F., Forms, Agents and Channels : Defining Composition Abstraction with Style, PhD Thesis, University of Berne ; Institute of Computer Science and Applied Mathematics, Berne, January, 2002.

ARTIST, Roadmap : Component-based Design and Integration Platforms, Technical report, ARTIST Project IST-2001-34820, May, 2003.

Bachmann F., Bass L., Buhman C., Comella-Dorda S., Long F., Robert J., Seacord R., Wallnau K., Volume II : Technical Concepts of Component-Based Software Engineering, 2nd Edition, Technical Report n° CMU/SEI-2000-TR-008, Carnegie Mellon Software Engineering Institute, May, 2000.

Barbier F., Henderson-Sellers B., Parc-Lacayrelle A. L., Bruel J.-M., « Formalization of the Whole-Part Relationship in the Unified Modeling Language », *IEEE Transactions on Software Engineering*, vol. 29, n° 5, May, 2003, p. 459-470.

Belloir N., Composition conceptuelle basée sur la relation Tout-Partie, PhD Thesis, Université de Pau et des Pays de l'Adour, Pau, December, 2004.

Belloir N., Romeo F., Bruel J.-M., « Whole-Part based Composition Approach : a Case Study », M. L. Ivica Crnkovic (ed.), *Proceedings of the 30th Euromicro Conference on Component-Based Software Engineering*, IEEE Computer Society Press, Rennes, France, September, 2004, p. 66-73.

Beugnard A., Jézéquel J.-M., Plouzeau N., Watkins D., « Making Components Contract Aware », *IEEE Computer*, vol. 32, n° 7, July, 1999, p. 38-45.

Bosch J., Szyperski C., Weck W., WS5. The Eighth International Workshop on Component-Oriented Programming (WCOP 2003), Workshop report, WCOP Co-Organizers, 2003.

Bruel J.-M., FUZE : un environnement intégré pour l'analyse formelle de logiciels distribués temps réels, PhD Thesis, Université Paul Sabatier - Toulouse III, Toulouse, France, December, 1996.

Bruel J.-M., Ober I., « The new UML 2.0 Component Model : Critical View », E. Grosspietsch, K. Klöckner (eds), *Proceedings of the Work in Progress Session at the 29th Euromicro Conference*, September, 2003.

Bruneton E., Coupaye T., Stefani J. B., « Recursive and Dynamic Software Composition with Sharing », *Seventh International Workshop on Component-Oriented Programming (WCOP02)*, Malaga, Spain, June 10, 2002.

Cheesman J., Daniels J., *UML Components – A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.

Dupuy S., Couplage des notations semi-formelles et formelles pour la spécification des systèmes d'information, PhD Thesis, University of Grenoble I - LSR/IMAG, Grenoble, September, 2000.

Firesmith D., Henderson-Sellers B., Graham I., *Open Modeling Language (OML) - Reference Manual*, Cambridge University Press, 1998.

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns : Element of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

Garlan D., Allen R., Ockerbloom J., « Architectural Mismatch, or, Why it's hard to build systems out of existing parts », *Proceedings of the 17th International Conference on Software Engineering*, Seattle, Washington, april, 1995, p. 179-185.

Harel D., Rumpe B., « Meaningful Modeling : What's the Semantics of "Semantics" ? », *IEEE Computer*, vol. 37, n° 10, October, 2004, p. 64-72.

Hassine I., Rieu D., Bounaas F., Jausseran E., Front A., Giraudin J.-P., « Méthodes de développement centrées composants », M. Oussalah (ed.), *Ingénierie des composants : concepts, techniques et outils*, Vuibert, 2005.

High Confidence Software and Systems Coordinating Group, High Confidence Software and Systems Research Needs, Technical report, Interagency Working Group on Information Technology Research and Development, january, 2001.

Ivers J., Sinha N., Wallnau K., A Basis for Composition Language CL, Technical Report n° CMU/SEI-2002-TN-026, Carnegie Mellon, Software Engineering Institute, 2002.

Kinitry J. R., « Semantic Component Composition », *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, October, 6-8, 2002.

Ledang H., Traduction systématique de spécifications UML en B, PhD Thesis, University of Nancy 2 - LORIA, Nancy, November, 2002.

McKinley P. K., Sadjadi S. M., Kasten E. P., Cheng B. H. C., « Composing Adaptive Software », *IEEE Transactions on Software Engineering*, vol. 37, n° 7, July, 2004, p. 56-64.

Meyer B., *Object-Oriented Software Construction*, Prentice Hall, Second Edition, 1997.

Moschoyiannis S., Shields M. W., « Component-Based Design : Towards Guided Composition », *Proceedings of the Third International Conference on Application of Concurrency to System Design (ACSD'03)*, Guimarães, Portugal, June 18-20, 2003.

Odell J. J., « Six Different Kinds of Composition », *Journal of Object-Oriented Programming*, vol. 7, n° 8, 1994, p. 10-15.

OMG, White Paper on the Profile mechanism - version 1.0, OMG document ad/99-04-07, Object Management Group, April, 1999.

Ommering R. V., « The Koala Component Model », I. Crnkovic, M. Larsson (eds), *Building reliable component-based software systems*, Artech House Publishers, Boston, 2002, p. 223-236.

Petit D., Mariano G., Poirriez V., « Génération de composant à partir de spécifications B », *Proceedings of the conférence on Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'2003)*, Rennes, France, January 15-17, 2003. http ://www.irisa.fr/manifestations/2003/AFADL03/.

Silaghi R., Strohmeier A., « Integrating CBSE, SoC, MDA, and AOP in a software Development Method », *Proceedings of the 7th International Enterprise Distributed Object Computing Conference*, IEEE Computer Society, Brisbane, Australia, 2003.

Stafford J. A., Wallnau K., « Component Composition and Integration », I. Crnkovic, M. Larsson (eds), *Building reliable component-based software systems*, Artech House Publishers, Boston, 2002, p. 179-191.

Sun Microsystems Inc., « Java™ Management Extensions (JMX) Specification 1.1 », 2001. http ://java.sun.com/products/JavaManagement/.