Implémentation d'un modèle UML de composition hiérarchique

Nicolas Belloir – Jean-Michel Bruel – Eric Cariou

Université de Pau et des pays de l'Adour
LIUPPA
BP 1155, F-64013 Pau Cedex
{belloir,bruel,cariou}@univ-pau.fr

RÉSUMÉ. *Nous décrivons ici une approche visant à se concentrer sur la composition hiérarchique lors de la conception et du développement des applications basées composants. Nous rappelons le modèle de composition sur lequel nous nous appuyons qui est basé sur l'expression de propriétés de composition bien identifiées. Nous montrons ensuite comment nous avons implémenté un mécanisme de contrôle de ces propriétés de composition dans Julia, une implémentation du modèle de composant Fractal.*

ABSTRACT. *We describe an approach focussing on hierarchical composition of components during both design and development levels. We present an overview of our UML composition model which is based on well-defined composition properties. Thus we present how we have implemented a control mechanism of theses properties into Julia, an implementation of the Fractal component model.*

MOTS-CLÉS : *UML, composition hiérarchique, composants, Fractal, Julia.*

KEYWORDS: *UML, hierarchical composition, components, Fractal, Julia.*

1. Introduction

En ingénierie logicielle basée composant[1], il convient de différencier l'intégration de composants (on parle aussi d'assemblage) de la composition. L'intégration peut être définie comme le lien syntaxique entre deux composants et la composition comme la possibilité de spécifier à un tout des propriétés basées sur les propriétés de ses parties et les relations entre elles [STA 02] ; la composition intègre donc l'aspect sémantique d'un assemblage. Malheureusement, UML qui est le langage de modélisation *de facto* utilisé en CBSE, n'offre pas un support bien défini pour exprimer les relations de composition [BRU 03], et de ce fait nécessite de préciser sa sémantique [HAR 04]. Il convient de différencier deux types de composition : la composition *horizontale* et la composition *verticale* [GOG 80] (encore appelée composition *hiérarchique*). La composition horizontale traite de la relation de composition entre deux entités de même niveau hiérarchique, comme sur une relation de type client-serveur classique. Nous nous intéressons dans cet article au cas de la composition hiérarchique qui représente la construction d'un composant de forte granularité avec un ou plusieurs composants de faible granularité appelés sous-composants. Dans ce contexte, nous avons défini dans des travaux antérieurs [BEL 05] un modèle de composition hiérarchique proposant une spécialisation des notions UML classiques que sont les relations d'*agrégation* et de *composition*. Ce modèle est basé sur l'expression et la combinaison de propriétés de composition bien identifiées, telle que par exemple la *partageabilité*. Celle-ci, lorsqu'elle est spécifiée sur une relation de composition entre un sous-composant *sca* et un composant de plus forte granularité *ca*, permet de spécifier le fait que *sca* peut être partagé par d'autres composants que *ca*.

D'un autre côté, une des problématiques toujours ouvertes en ingénierie logicielle concerne la question de l'implémentation/traduction d'un concept spécifique spécifié au niveau modèle, au niveau de l'application finale. Certains concepts telles que les *classes* ou les *interfaces* ont une correspondance directe entre le modèle et le code. Pour d'autres, et particulièrement les nouveaux concepts introduits dans les modèles, on ne trouve pas de moyen direct pour les traduire au niveau du code et c'est alors souvent au développeur de choisir comment implémenter le concept. Notre préoccupation est alors que ce choix peut introduire une variation sémantique entre le concept du modèle et sa traduction au niveau code. Il serait préférable de fournir au développeur une traduction directe lors de l'implémentation de chaque concept énoncé au niveau du modèle. Dans l'idéal, l'application pourrait être entièrement générée à partir du modèle. Une des solutions préconisée actuellement par l'approche *Mda* [Ric 00] consiste à faire cela via des transformations successives de modèles. Cependant, même dans ce cas il faut bien disposer d'une traduction du-dit concept au niveau implémentation.

Dans ce contexte, nous présentons ici mise en œuvre de notre modèle au niveau du modèle de composant *Fractal* [BRU 04], et plus précisément dans son implémentation Java, *Julia*. Nous fournissons ainsi au développeur les moyens d'implémenter directe-

1. Nous utilisons par la suite l'acronyme anglais unanimement reconnu CBSE, pour *Component-Based Software Engineering*.

ment les propriétés de composition exprimées au niveau conceptuel. Notre extension de Julia assure que l'application, lors de son exécution, respecte les propriétés de composition spécifiées. Ce travail est présenté à la section 4. Le reste de ce papier consiste en une discussion sur la composition en UML en section 2, en une présentation brève de notre modèle de composition à la section 3, puis en une liste non exhaustive de travaux connexes à la section 5. Nous concluons et dressons des perspectives à la section 6.

2. Composition hiérarchique en UML

Le modèle de composition hiérarchique sur lequel nous nous appuyons est basé sur l'idée que la composition doit se focaliser sur les aspects sémantiques des assemblages [STA 02]. Une application construite par intégration de composants n'est pas forcément viable en exécution. Déterminer les combinaisons de composants valides implique d'envisager à la fois les dimensions architecturales et sémantiques lors de l'intégration des composants.

Au niveau conceptuel, le langage graphique UML est le plus utilisé pour modéliser les applications basées composant. Ce type de notation est souvent critiqué par la communauté scientifique à cause de son « manque » de sémantique. Cependant, ce langage offre un formalisme de lecture aisément compréhensible, bien connu des industriels et qui a largement prouvé son utilité en développement logiciel [HAR 04]. Ainsi, nous avons appliqué nos travaux sur l'amélioration de la sémantique de la relation de composition au niveau conceptuel à UML. Plus précisément, nous avons restreint notre étude à la composition hiérarchique. Ce type de composition est notamment présent avec des modèles de composants comme Koala [OMM 02] ou Fractal [BRU 04].

En UML 1.x, la composition de composant est exprimée en utilisant les relations de *composition* et d'*agrégation*. Malheureusement, ces deux relations sont mal définies et manquent de consistence [HEN 99]. De plus, les notions inhérentes à la CBSE y sont pauvres. La nouvelle version d'UML, la 2.0, tente de corriger cela en introduisant des concepts spécifiques autour des composants tels que les *structures composites*, les *ports* ou les mécanismes de *délégation*. Il est indéniable que du point de vue de la notation, UML 2 fournit une réponse aux attentes laissées par UML 1.x [BOC 04]. Cependant nous trouvons ces améliorations insuffisantes. L'approche que nous proposons a pu bénéficier des améliorations en termes de notation, mais pour ce qui est de la spécification des liens entre composite et partie, notre profil reste non seulement valide, mais toujours indispensable. Dans notre approche, la notion de Tout peut être rapprochée de celle de PackagingComponent et celle de partie de celle de BasicComponent.

Les manques d'UML 1.x étaient pourtant explicites dans les documents qui sont à l'origine d'UML 2 (appelés RFP – *Request For Proposal* dans le jargon de l'OMG) [WEI 99] : support à l'assemblage de composants, concept de *plug-substitutability*, support pour la spécification de patrons d'interaction entre composants, support pour

la modélisation des contextes d'exécution des composants (comme pour les conteneurs EJB ou CCM), support pour la définition de profils spécifiques aux modèles architecturaux classiques. On peut dire que le nouveau standard répond effectivement en partie à ces exigences. Mais là où nous trouvons qu'il a échoué, c'est sur le premier point des attentes qui étaient liées à la composition [WEI 99] : (i) amélioration de la sémantique de la notation (notamment des associations) pour supporter le développement basé composant, (ii) meilleur support pour les interfaces et (iii) ajout des protocoles pour la formalisation des interfaces. En l'absence de retours encore probants, nous avons développé nos propres heuristiques sur l'utilisation de ces nouveaux concepts (voir aussi [RED 03]).

3. Extension du modèle de composition hiérarchique en UML

Notre modèle de composition hiérarchique [BEL 05] propose de définir une nouvelle relation de composition spécifique, que nous proposons d'ajouter au métamodèle UML, comme illustré par la figure 1. Nous avons pour l'heure défini un profil UML pour l'implémenter. Cette relation est une spécialisation d'une nouvelle relation de composition abstraite (*Component Relationship*) qui se spécialise en relation horizontale (*Horizontal Composition Relationship*) et en relation hiérarchique (*Vertical Composition Relationship*). La relation de composition hiérarchique est basée sur la sémantique de la relation Tout-Partie, utilisée notamment en orienté-objet pour caractériser l'agrégation et la composition lors de la conception [BAR 03]. Le principe consiste à spécialiser la relation de composition à partir de propriétés de composition bien identifiées.

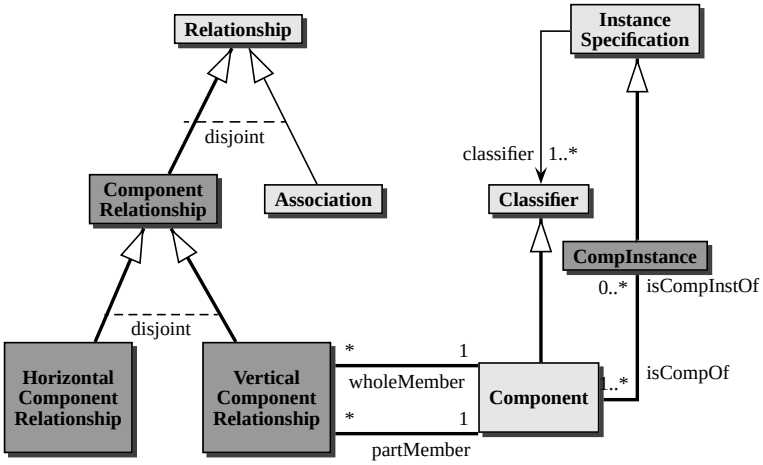

Figure 1. *Intégration d'une relation de composition dans le métamodèle* UML

Deux de ces propriétés sont toujours vérifiées. L'*asymétrie au niveau des instances* (signifie qu'une instance de composé ne peut pas être composant d'elle même et qu'une instance de composant ne peut pas être composé du composant qui la compose) et de l'*antisymétrie au niveau des types* (spécifie qu'un type de composé, `cA`, composé du type de composant `cB`, ne peut pas être lui-même composant de `cB`. Autrement dit, il ne peut pas y avoir de cycle entre composants et composés).

Les autres propriétés[2] de notre modèle sont l'*encapsulation* (qui traduit la contenance physique d'un composant à l'intérieur d'un composé), la *partageabilité* (qui spécifie si un composant ayant déjà une relation de composition avec un composé peut établir une autre relation de composition avec un autre composé), les *dépendances de cycles de vie* (qui indiquent les liens en terme de création/destruction entre le composant et le composé), la *séparabilité* (qui dit si une relation de composition peut être cassée) et la *mutabilité* (qui indique si une relation de composition peut varier en terme de type). Elles peuvent être vues comme des propriétés de caractérisation, permettant de définir des sous-types de relation de composition, et de ne plus se limiter aux seules agrégation/composition.

En effet, les relations d'agrégation/composition ne couvrent pas tous les points de variation sémantique que l'on peut exprimer sur une relation de composition au niveau de la description architecturale. Par exemple, si l'on considère les propriétés de *partage* et d'*encapsulation*, la composition peut être spécifiée comme vérifiant la non partageabilité et l'encapsulation, et l'agrégation la non encapsulation et la partageabilité. Or il se peut que l'on veuille spécifier une relation vérifiant par exemple la partageabilité et l'encapsulation et, dans ce cas, ni l'agrégation ni la composition ne sont recevables. Nous pouvons donc définir quatre relations de composition en combinant ces deux propriétés afin d'offrir plus de souplesse et de précision dans la phase de conception. Le tableau 1 montre les quatre relations de composition de notre modèle et leurs propriétés.

	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
dépendances de cycle de vie (cas n°) cf. [BAR 03]	1	4	1	4	9	9	9	9

Tableau 1. *Différents types de relation de composition*

2. Pour le détail de toutes les propriétés de composition, nous invitons le lecteur à consulter [BEL 05].

Nous avons intrumenté notre modèle de composition avec un profil UML nommé WPCP[3] [BEL 05]. Ce profil a été implémenté sous l'Atelier de génie logiciel Objecteering[4]. Notre choix s'est porté sur cet outil commercial car il est le seul à notre connaissance à offrir des fonctionnalités de création/utilisation de profil UML. Nous avons inclus dans notre profil des scripts de contrôles écrits en *langage J*, qui est le langage natif d'Objecteering et qui permet de réaliser toutes formes d'exploitation d'un modèle UML comme par exemple des requêtes, des règles de validation, des générations de code, ou des transformations de modèle [Sof 99]. Ils assurent que le concepteur ne peut pas construire des incohérences par rapport au modèle de composition dans ses diagrammes.

La figure 2 montre une capture d'écran d'un diagramme de type de composant créé à l'aide de WPCP. L'application décrite est une machine à café industrielle. Elle est construite à partir de cinq composants : *CoffeeMachine* est un composé construit à partir de deux sous-composants, *Coiner* qui gère la récupération de pièces de monnaie et *DrinkMaker* qui est un module de fabrication de boissons. *E-Money* est un composé assurant la gestion d'un porte-monnaie électronique, alimenté par le sous-composant *Coiner* et le sous-composant *aManager*. On a donc quatre relations de composition hiérarchique : deux sont des compositions fortes (SCOMP), une est une composition légère (LCOMP) et la dernière est une agrégation légère (LAGG). Les interfaces requises et fournies ne sont pas sur le diagramme par soucis de clarté.

4. Implémentation du modèle de composition dans le modèle de composant Fractal

Le modèle de composition présenté à la section précédente permet d'enrichir et d'améliorer le pouvoir d'expression d'UML. Cependant, si les modifications apportées au niveau des modèles ne peuvent pas être traduites aisément au niveau du code, ce sera au développeur de les traduire avec les moyens dont il dispose dans le langage/modèle à composant cible, c'est-à-dire qu'il aura un choix d'implémentation à faire. Cela peut être source d'erreur et peut induire une différence entre le modèle et l'application finale. Nous pensons donc qu'il est préférable de fournir les concepts du modèle de composition dans le modèle de composant choisi pour développer l'application. Il convient donc d'implémenter le modèle de composition dans le modèle de composant.

Un modèle de composant définit les éléments constituant ce modèle ainsi que la manière dont ces éléments peuvent être assemblés et peuvent interagir entre eux. Il existe de nombreux modèles de composant comme EJB [MIC] ou CCM [OMG 99]. Notre choix s'est porté sur Fractal pour plusieurs raisons. Tout d'abord, il fournit de manière native la construction d'applications hiérarchiques. Ensuite, Fractal est supporté par plusieurs implémentations telle que Julia, une plate-forme open-source

3. Whole-Part Composition Profile.
4. http ://www.objecteering.com/

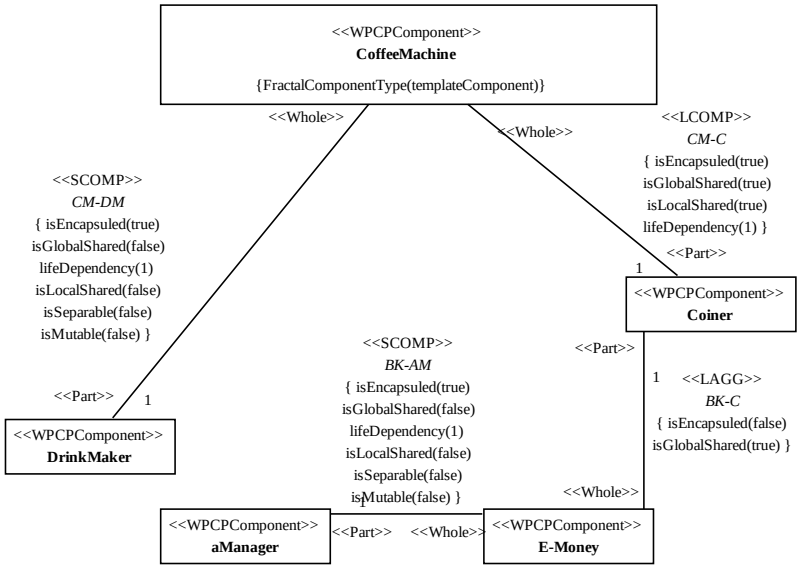

Figure 2. *Diagramme de type de composant généré avec le profil* WPCP

Java. Enfin il intègre implicitement plusieurs concepts proches de nos propriétés de composition telle que la partageabilité, ou le contrôle des cycles de vie présentées notamment dans [BRU 04]. Le tableau 2 résume la manière dont ces concepts sont traités dans Fractal.

Propriété	Asymétrie	Antisymetrie	Partageabilité		Séparabilité	Mutabilité	Dépendances de cycle de vie
			globale	locale			
Intégration dans Fratal	implémenté	implémenté	implémenté mais sans distinction entre les deux. Pas de verification de la non partageabilité		Sous-composant toujours séparable. Non séparabilité non implémentée	non implémenté	libre

Tableau 2. *Support des propriétés de composition par Fractal*

Nous avons rendu toutes les propriétés de composition, spécifiée au niveau UML, exprimable de manière explicite sous Fractal, et plus particulièrement sous son implémentation Java, Julia. Dans la suite de cette section, nous présentons brièvement Fractal, puis les modifications que nous avons apportées à Julia, et enfin un exemple d'utilisation.

4.1. *Le modèle de composant Fractal*

Fractal est un modèle de composant entièrement ouvert issu d'une collaboration entre l'INRIA et France Télécom R & D et est développé au sein du consortium ObjectWeb[5] depuis 2002. Il s'agit d'une norme définissant non seulement les divers éléments abstraits constituant le modèle mais aussi la manière dont ils peuvent être assemblés et peuvent interagir entre eux. Ce modèle est également fortement orienté vers la modularité et l'adaptabilité aux besoins spécifiques qui peuvent apparaître dans le développement d'applications orientées composant. En effet, il n'impose aucun langage de programmation particulier pour créer, déployer et maintenir des applications. De plus, ce modèle peut facilement être étendu et s'adapter pour intégrer d'autres modèles de composant tels que EJB ou CCM.

En plus des concepts inhérents au monde composant (*composants*, *interfaces*, *composites* ...) Fractal définit les *contrôleurs* qui permettent la gestion des composants et peuvent être étendus. Chaque contrôleur est spécialisé dans un domaine (démarrage, nommage, liaison ...). Plusieurs contrôleurs sont prédéfinis, comme le *binding-controller* qui assure la gestion d'une liaison (créer/supprimer), le *lifecycle-controller* qui permet de démarrer/arrêter un composant ou encore le *content-controller* qui gère le contenu d'un composite (ajouter/retirer des sous-composants). Enfin, Fractal permet l'introspection des composants, le partage entre composants, etc.

Julia est l'implémentation de référence de la spécification du modèle Fractal. Les différents éléments constituant le modèle concret ne sont pas directement implémentés dans l'API Fractal. Celle-ci n'est quasiment composée que d'interfaces, au sens Java du terme, dont les fonctions peuvent être redéfinies par l'utilisateur en fonction de ses besoins. Seules la gestion des exceptions et la classe Fractal, contenant la fonction de Bootstrap qui permet l'initialisation du modèle, sont implémentées par des classes Java. Julia permet d'obtenir une implémentation par défaut des différentes interfaces de contrôle, la gestion des interfaces clientes et serveurs restant à la charge du développeur. Julia permet également à l'utilisateur de créer ses propres contrôleurs.

4.2. *Modification de Julia*

Afin de spécifier et contraindre les propriétés de composition, nous avons créé un nouveau contrôleur. Nous décrivons ici la réalisation de ce contrôleur. Nous avons défini une interface *CompositionController* qui possède les services demandés par les autres composants. Celui-ci est modélisé à la figure 3. Le contrôleur est implémenté par le *BasicCompositionControllerMixin*. Celui-ci permet au concepteur de spécifier les contraintes des propriétés de composition. La classe *CompositionProperties* mémorise les propriétés de composition concernant un sous-composant. Notons que la partageabilité peut être envisagée entre composants de même type (partage local) et/ou entre composants de types différents (partage global).

---

5. http ://www.objectweb.org

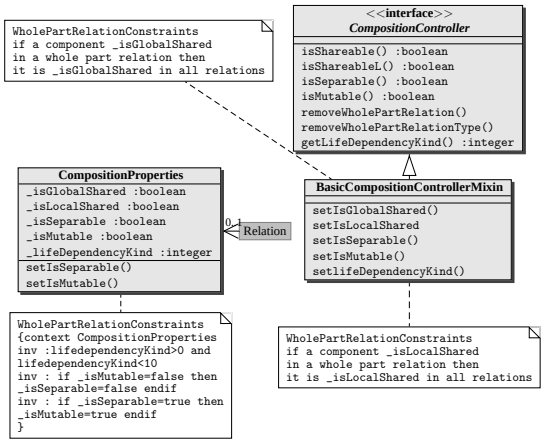

Figure 3. *Modèle* UML *du composition-controller*

Etant donné qu'un certain nombre de propriétés de composition sont déjà présentes dans Fractal, nous nous sommes appuyés sur ces mécanismes. Notre nouveau contrôleur a donc été confronté au problème de l'héritage multiple. Or Julia fournit un mécanisme permettant de contourner ce manque : les *mixins*, initialement utilisés dans Flavor [THO 83]. Ils permettent notamment la réutilisation du code de plusieurs classes. Soient trois classes *classA*, *classB* et *classC*. Si on veut que *classC* puisse réutiliser des méthodes de *classA* et de *classB*, la solution la plus simple serait de le faire hériter des deux classes. Or, en Java, il n'existe pas d'héritage multiple.

Une autre méthode serait d'avoir des classes combinant les méthodes des classes dont elles ont besoin. Cela entraînerait une multiplication du nombre de classes, des doublons de méthodes, etc. Les mixins offrent une solution à ce problème. Ils permettent donc de conserver la modularité et l'extensibilité de Fractal pour les contrôleurs. Afin de pouvoir implémenter l'intégration des propriétés de composition dans Fractal, nous avons donc dû créer un certain nombre de *mixins* qui utilisent tous le *CompostionController* que nous avons créé. Ils sont donnés par le tableau 3.

Nous avons dû modifier le code source de Julia afin que les initialisations de la relation de composition pour notre contrôleur soient prises en compte au plus tôt dans la création des composants c'est-à-dire dès l'utilisation des *templates*. Un template est une sorte de modèle auquel un composant doit se conformer et doit posséder les mêmes propriétés que ce template. Cette solution n'est pas idéale puisqu'ainsi notre contrôleur n'est pas un ajout isolé. Si on veut utiliser notre contrôleur, il est nécessaire d'utiliser également notre version de Julia.

Mixin	**rôle**
BasicCompositionControllerMixin	implementation du *CompositionController*
CompositionCompositeBindingMixin	gestion de la séparabilité à partir d'un composite
CompositionPrimitiveBindingMixin	gestion de la séparabilité à partir d'un primitif
ContentCompositionControllerMixin	gestion de la partageabilite et le retrait d'un sous-composant
BasicLifeCycleDependencyMixin	gestion des dépendances de cycles de vie

Tableau 3. *Liste des mixins développés*

4.3. *Utilisation*

Voici un exemple de code permettant de vérifier que le composant *component* est partageable. S'il est partageable, on crée une relation de composition entre le composant et le composite ayant appelé cette méthode du *Content-Controller.* Cette relation est conservée dans le *Composition-Controller* du composite. Puis on effectue les autres tests nécessaires avant un ajout via l'appel à *_super_addFcSubComponnent.* Si le composant n'est pas partageable, une exception est levée.

```
public abstract class ContentCompositionControllerMixin implements
ContentController{
    public CompositionController _this_weaveableCompC;
    public Component _this_weaveableC;
    ...
    public void addFcSubComponent(Component component){
    CompositionController compC;
    ...
    // if the component is shearable globally and locally
    if( compC.isShareable(component) && compC.isShareableL(component)){
        _this_weaveableCompC.setWholePartCompositionProperties(component, new
          Boolean(compC.isShareable(component)), new Boolean(compC.isShareableL(
          component)), new Boolean(compC.isSeparable(component)), null);
        _super_addFcSubComponent(component);
  }
    ...
}
```

Nous détaillons comment créer une relation de composition entre un composant *c* et un composite *cm.* On crée d'abord les deux composants, puis on récupère ensuite la référence au *composition-controller* du composite. On fixe les valeurs des propriétés de composition via la méthode *setWholePartCompositionProperties.* Le premier paramètre est le sous-composant, les suivants sont les valeurs des propriétés. Si une propriété n'a pas besoin d'être définie, ce paramètre peut être remplacé par *null.* C'est alors sa valeur par défaut qui est prise en compte.

```
// creation of two components
Component c = Coiner.createCoiner();
Component cm = CoffeeMachine.createCoffeeMachine();

// get the CompositionController by its interface name
CompositionController compC = (CompositionController) cm.getFcInterface("
 composition-controller");

// set the values of the composition properties
compC.setWholePartCompositionProperties(c,new Boolean(false), new
Boolean(true), new Boolean(true), null);

// signature de la méthode setWholePartCompositionProperties :
//public void setWholePartCompositionProperties(Component partComponent, Boolean
// isGlobalShared, Boolean isLocalShared, Boolean isSeparable, Boolean isMutable)
// throws CompositionException
```

Lorsque qu'une propriété de composition spécifiée est violée, le contrôleur lève une exception en empêche ainsi le non respect de la propriété.

5. Travaux connexes sur la composition

De nombreux travaux portent sur la composition de composants. La communauté des méthodes formelles est très active sur le sujet et depuis longtemps. Par exemple Nierstrasz et Meijler [NIE 95] en 1995 définissent la composition logicielle comme un mécanisme, sans préciser lequel, permettant la construction d'une application à partir de composants. Se focaliser sur l'aspect composition est maintenant une chose acquise comme le montrent les travaux visant à proposer des langages de composition comme CL [IVE 02] ou PICCOLA [ACH 02]. Ceux-ci sont des langages opérant à un haut niveau d'abstraction. Ils représentent un amalgame entre le scripting, les langages de description d'architecture et les langages de coordination. Cependant, le coté formel de ces langages peut les rendre moins attrayants pour les industriels. C'est notamment pour cette raison que notre approche est semi-formelle et qu'elle s'appuie sur UML. Parmi les approches moins spécialisées sur la composition, on peut noter les approches utilisant les contrats [JéZ 01]. En effet, ceux-ci sont une solution élégante, bien que moins spécialisée, pour décrire la composition. Par ailleurs des travaux ont déjà porté ce mécanisme sur Fractal [COL 05]. D'autre part, on note maintenant une évolution des travaux portant sur la composition vers la composition dynamique, portés notamment par l'intérêt croissant pour l' « autonomic computing » [MCK 04].

6. Conclusion et perspectives

Améliorer les moyens de modélisation permet d'accroître la qualité des applications finales. Cependant, la traduction et l'application des modèles en code exécutable peut présenter des difficultés car, par exemple, un concept de modélisation peut ne pas trouver de traduction évidente dans un modèle de composant cible. Dans ce cadre, nous avons présenté dans ce papier une approche centrée sur la composition de composant. Les concepts spécifiques à l'expression de la composition sont définis à la fois

dans un modèle de composition basé UML, ainsi que dans la plate-forme à composant Julia, qui est une extension du modèle de composant Fractal. Le modèle de composition est basé sur l'expression de propriétés de composition définies en OCL. Certaines de ces propriétés étaient présentes implicitement dans Fractal. Au niveau de la plate-forme Julia, nous avons créé un nouveau contrôleur qui est chargé de surveiller le respect des propriétés lors de l'exécution de l'application.

La prochaine étape concernant ce travail sera de mener une étude visant à valider l'intérêt de cette approche. Nous comptons pour cela développer une application avec notre extension de Julia dans un premier temps, puis avec la version initiale de Julia et comparer les deux approches. D'autre part, Cette expérience laisse entrevoir plusieurs perspectives intéressantes. En effet, les applications tendant à devenir de plus en plus mobiles et étant amenées à fonctionner sur des supports de plus en plus légers en terme de capacités de traitement, nous évaluons actuellement les capacités de notre approche à fonctionner sur ce type d'environnement. La surcharge liée à l'ajout d'un contrôleur est-elle compatible avec la transformation du code Julia en J2ME ? D'autre part, il serait intéressant de s'intéresser à une autre implémentation de Fractal : *AOkell*[6]. Celle-ci permet de construire les contrôleurs et les membranes comme des assemblages de composants Fractal. Le contrôleur de composition pourrait être construit à l'aide d'un assemblage de sous-contrôleur et se passerait alors des mixins.

Enfin, nous travaillons à développer cette approche suivant un processus MDE. Nous avons le modèle indépendant de tout support technologique et le support technologique. Il est donc naturel de développer des moyens de génération automatique par transformation de modèles. Nous avons pour cela développé un modèle spécifique de composition pour Julia et nous l'avons implémenté sous la forme d'un profil UML. Nous avons défini des règles de transformation pour passer du modèle initial jusqu'à la génération de code via le modèle spécifique à Julia. Nous sommes en train de finaliser ce travail en intégrant des moyens de validation des différentes transformations.

7. Bibliographie

[ACH 02] ACHERMANN F., « Forms, Agents and Channels - Defining Composition Abstraction with Style », PhD thesis, University of Berne - Institute of Computer Science and Applied Mathematics, Berne, janvier 2002.

[BAR 03] 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, 2003, p. 459–470.

[BEL 05] BELLOIR N., BRUEL J.-M., « Développement basé composant : une approche centrée composition », *Méthodes Avancées de Développement des SI. Numéro spécial de la revue Ingénierie des Systèmes d'Information*, vol. 10, n° 6, 2005, p. 59–80, Hermes.

[BOC 04] BOCK C., « UML 2 Composition Model », *Journal of Object Technology*, vol. 3, n° 10, 2004.

6. http ://fractal.objectweb.org/tutorials/aokell

[BRU 03] BRUEL J.-M., OBER I., « The new UML 2.0 Component Model : Critical View », GROSSPIETSCH E., KLÖCKNER K., Eds., *Proceedings of the Work in Progress Session at the 29th Euromicro Conference*, septembre 2003.

[BRU 04] BRUNETON E., COUPAYE T., LECLERCQ M., QUEMA V., STEFANI J. B., « An Open component Model and Its Support in Java », *Seventh International Symposium on Component-Based Software Engineering (CBSE-7)*, LNCS 3054, 2004, p. 7–22.

[COL 05] COLLET P., ROUSSEAU R., COUPAYE T., RIVIERRE N., « A Contracting System for Hierarchical Components », *SIGSOFT Symposium on Component-Based Software Engineering (CBSE 05)*, LNCS, St-Louis (Missouri, USA), mai 2005, Springer-Verlag.

[GOG 80] GOGUEN J., BURSTALL R., « Cat, a system for the structured elaboration of correct programs from structured specifications. », technical report n° CSL-118, 1980, Computer Science Laboratory, SRI International.

[HAR 04] HAREL D., RUMPE B., « Meaningful Modeling : What's the Semantics of "Semantics" ? », *IEEE Computer*, vol. 37, n° 10, 2004, p. 64–72.

[HEN 99] HENDERSON-SELLERS B., BARBIER F., « Black and Whites Diamonds. », 1999, p. 550–565.

[IVE 02] IVERS J., SINHA N., WALLNAU K., « A Basis for Composition Language CL », Technical Report n° CMU/SEI-2002-TN-026, 2002, Carnegie Mellon, SEI.

[JéZ 01] JÉZÉQUEL J.-M., PLOUZEAU N., WEIS T., GEIHS K., « From Contracts to Aspects in UML Designs », 2001, QCCS project, available at http ://www.qccs.org/aodcontracts.pdf.

[MCK 04] 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, 2004, p. 56–64, IEEE Press.

[MIC] MICROSYSTEMS S., « Enterprise Java Beans Specifications, V1.1. ».

[NIE 95] NIERSTRASZ O., MEIJLER T. D., « Research Directions in Software Composition », *ACM Computing Surveys*, vol. 27, n° 2, 1995, p. 262–264.

[OMG 99] OMG, « CORBA Component Model », april 1999, http ://www.omg.org/.

[OMM 02] OMMERING R. V., « The Koala Component Model », CRNKOVIC I., LARSSON M., Eds., *Building reliable component-based software systems*, p. 223-236, Artech House Publishers, Boston, 2002.

[RED 03] REDDY P. V., « Toward Better Logical Models in UML », *Journal of Object Technology*, vol. 2, n° 5, 2003, p. 101–121.

[Ric 00] RICHARD SOLEY AND THE OMG STAFF STRATEGY GROUP, « Model Driven Architecture », White paper, novembre 2000, OMG.

[Sof 99] SOFTEAM, « UML Profiles and the J language : Totally control your application development using UML », 1999, White Paper.

[STA 02] STAFFORD J. A., WALLNAU K., « Component Composition and Integration », CRNKOVIC I., LARSSON M., Eds., *Building reliable component-based software systems*, p. 179–191, Artech House Publishers, Boston, 2002.

[THO 83] THOMPSON C. W., ROSS K. M., TENNANT H. R., SAENZ R. M., « Building Usable Menu-Based Natural Language Interfaces To Databases », *In Proceedings of the 9th international Conference on Very Large Data Bases*, 1983, p. 43-55.

[WEI 99] WEIGERT T., « UML 2.0 RFI Response Overview », Dec. 28 1999.