Vérification a priori de modèle de composition logicielle

Une approche basée sur la relation Tout-Partie

Nicolas Belloir, Fabien Romeo

Université de Pau et des Pays de l'Adour
LIUPPA, B.P. 1155
64013 Pau CEDEX, France
{nicolas.belloir, fabien.romeo}@univ-pau.fr

RÉSUMÉ. *La complexité toujours croissante des systèmes d'informations, ainsi que les contraintes de flexibilité et d'adaptabilité, plaident pour un développement de ces systèmes à base de composants logiciels. Dans ce cadre, les techniques de modélisation de la composition logicielle, et de vérification du respect de ces propriétés par le système développé, sont des besoins clairement identifiés. Nous présentons dans cet article une approche de modélisation de la composition logicielle basée sur la relation Tout-Partie. Nous montrons comment, grâce à un environnement de composition que nous avons développé, nous pouvons effectuer une vérification a priori du respect des propriétés de composition. Nous l'illustrons par une étude de cas.*

ABSTRACT. *Information systems are more and more complex. They must become more flexible and more adaptable. Component Based Software Engineering is a good response to these needs. In this context, design techniques, specially based on software composition and verification techniques, proving that the design properties are well implemented, are current active problematics. In this paper, we present a composition design technique based on the Whole-Part Relationship. We show with an in-house verification framework how to verify the well implementation of the composition properties. We illustrate our approach through a case study*

MOTS-CLÉS : *composant, composition, vérification, COTS*

KEYWORDS: *component, composition, verification, COTS*

1. Introduction

Les systèmes d'informations subissent la pression des nouvelles technologies de l'information. En plus d'être sûrs, fiables et performants, ils doivent aussi être rapidement évolutifs afin par exemple de permettre aux entreprises de suivre l'évolution du marché et des nouveaux supports de communication et/ou de vente. Dans ce cadre, l'ingénierie logicielle basée composant (CBSE) (Szyperski, 2002) offre une flexibilité intéressante puisqu'elle permet la réutilisation rapide de briques logicielles déjà développées. On appelle ces briques des « composants ». L'objectif est de construire facilement et rapidement des systèmes par assemblage de composants, mais également de remplacer aisément un composant par un autre au cours du cycle de vie de l'application.

Actuellement, une des limitations de la CBSE vient du fait que la composition logicielle est principalement traitée lors des phases d'assemblage de composant ce qui entraîne notamment des dépendances non spécifiées entre composants ou encore des « adaptations » des composants. Ces dépendances ou adaptations sont nécessaires pour intégrer les composants dans le système car ils ne sont pas toujours conçus pour prendre en compte leur composabilité et leur réutilisabilité. Cela va à l'encontre de la flexibilité attendue. Des problèmes se posent alors non pas sur les composants, mais sur leur composition : la méconnaissance du comportement futur des compositions par exemple.

La prise en compte de la composition, non plus seulement au niveau assemblage de composant, mais au niveau modèle, est une des recommandations des experts du domaine (High Confidence Software and Systems Coordinating Group, 2001) et est préconisée par la communauté, notamment par l'OMG au travers de son initiative MDA (OMG, 2001 b). Cette approche est d'ailleurs prise en compte dans UML 2.0., la nouvelle version de UML (OMG, 2003). Malheureusement, ces dernières avancées sont encore insuffisantes (Bruel, 2003a), car, même si la composition est prise en compte, elle ne l'est que par des schémas sans sémantique formelle associée.

Nous avons proposé dans (Belloir et al., 2003a) et dans (Belloir et al., 2003b) une approche basée sur la relation Tout-Partie pour spécifier la composition logicielle dès la phase de conception. Elle s'appuie notamment sur la spécification de propriétés sémantiques au niveau de la relation de composition. Dans cet article, nous allons montrer plus en détail comment mettre en œuvre cette spécification et l'illustrer à l'aide d'une étude de cas. Nous allons présenter également l'environnement de vérification que nous avons développé et qui supporte notre approche, permettant de vérifier les propriétés d'assemblage des composants. Pour ce faire, nous allons dans un premier temps faire un bref rappel de notre proposition. Ensuite, nous allons illustrer notre approche à l'aide d'une étude de cas. Enfin, et avant de conclure, nous présenterons l'avancée de nos travaux portant sur ce thème.

2. La relation Tout-Partie comme support à la composition logicielle

Dans cette section, nous rappelons les grandes lignes de notre approche. Nous illustrons l'intérêt de cette approche par des exemples concrets. Mais avant cela, nous positionnons notre approche vis-à-vis des travaux visant à améliorer la conception des assemblages de composants.

Notre approche de la composition de composants est basée sur une hiérarchie compositionnelle réflexive de type Tout-Partie (*WPR* pour Whole-Part Relationship), où toute composition est vue sous la forme d'un composant de haut niveau (*Tout*) composé de sous-composants (*Parties*). Ce composant Tout s'appuie sur les services du composant Partie pour fournir lui-même d'autres services. Cette vision est également celle adoptée par le récent workshop WCOP 2003 (Bosh *et al.,* 2003) dans lequel il est dit : « *In fact, one can think of a component-based system as a triangle. At the top node a component-based system consists only of one component that actually represents the whole architecture of the corresponding application. A zoom operation can then be used to decompose the application* ». Cette approche présente des similitudes avec le patron de conception *Composite* défini dans (Gamma *et al.*, 1995). Elle s'en différencie cependant dans le sens où nous nous intéressons plus fortement à la formalisation des propriétés de la relation de composition. Or l'approche par patron de conception est un peu limitative en ce sens puisqu'elle généralise un concept et que, par conséquent, elle ne formalise pas toutes les combinaisons de propriétés possibles. Une approche formelle pour la composition de composants, comme par exemple l'utilisation du langage *B* (Petit *et al.*, 2003), ne nous semble pas non plus une solution complètement satisfaisante, notamment à cause du fait de sa difficile mise en œuvre par un non spécialiste, le concepteur d'application basée composant par exemple. Nous pensons que l'aspect formel de la composition doit être sous-jacent à la technique utilisée et si possible transparent pour l'utilisateur.

2.1. Vue d'ensemble de l'approche de modélisation

Dans cette approche, nous avons proposé d'adapter les propriétés formelles identifiées de la relation Tout-Partie (Barbier *et al.*, 2003) afin d'en tirer des propriétés de composition permettant de spécifier la composition entre les composants *Tout* (encore appelés *composés*) et les sous-composants *Partie* (encore appelés *composants*). Ces propriétés, dans le contexte des composants, ont été identifiées et présentées brièvement dans (Belloir *et al*., 2003a) et dans (Belloir *et al*., 2003b).

Les études citées précédemment ont montré qu'une relation Tout-Partie est caractérisée par un ensemble de propriétés classées en propriétés *primaires* et *secondaires* : les propriétés primaires sont les propriétés que doit vérifier obligatoirement une relation pour être qualifiée de Tout-Partie : nature binaire de la

relation, asymétrie au niveau instance, anti-symétrie au niveau type et existence d'au moins une propriété émergente et une propriété résultante. Les propriétés secondaires sont les propriétés qui caractérisent le type de relation Tout-Partie. Il s'agit de l'encapsulation, des dépendances des cycles de vie (9 cas), de la transitivité, de la partageabilité, de la séparabilité, de la mutabilité et de la dépendance existentielle.

Nous pensons que le respect strict des propriétés primaires pour qu'une relation soit qualifiée de Tout-Partie est trop contraignant. En effet, certaines de ces propriétés peuvent être considérées comme des heuristiques de modélisations (c'est le cas des propriétés résultantes et émergentes). En ce sens, une relation ne respectant pas strictement ces deux propriétés peut tout de même entrer dans le champ de notre approche. Nous préconisons d'étendre l'utilisation de cette méthode de composition à des composants ne respectant pas ces deux propriétés.

En fait, l'intérêt de cette approche consiste principalement en la possibilité pour le concepteur de spécifier des propriétés secondaires sur la relation de composition. En fonction des propriétés spécifiées entre deux composants, il est possible de prévoir le comportement de l'assemblage de ces composants, ce qui est une demande forte de la communauté (Bosch *et al.,* 2003).

2.2. Détail des propriétés secondaires de la relation Tout-Partie

Dans cette section nous discutons de l'impact de la spécification des propriétés secondaires sur une composition de composants. Tout d'abord, nous présentons les propriétés secondaires. Nous invitons le lecteur intéressé par le détail des propriétés primaires à consulter nos précédents articles.

2.2.1. Encapsulation

Dans une application, seul le composé dans lequel un composant est encapsulé peut accéder à ce dernier. Cela implique pour le composant de ne pas pouvoir être composant d'un composé différent via une autre relation et de ne pas entretenir de relation d'association (au sens large) avec un autre composant hors du composé. Notons tout de même que l'encapsulation pouvant être récursive, le composant peut avoir une relation de type Tout-Partie avec un autre composant pour lequel il joue lui-même le rôle de composé.

2.2.2. Partageabilité

La propriété de partage permet à un même composant de faire partie de plusieurs composés. Il peut être envisagé au *niveau local* ou au *niveau global*. Le partage au niveau local consiste à partager un composant entre plusieurs instances du même type de composé. Le partage au niveau global quand à lui consiste à partager un même composant entre plusieurs composés de différents types. Le contraire de la

partageabilité locale est l'*exclusion locale*. Elle signifie qu'une même instance d'un composant ne peut pas faire partie de plusieurs composés de même type. Le contraire de la partageabilité globale est l'*exclusion globale*. Elle signifie qu'une même instance d'un composant ne peut pas faire partie de plusieurs composés de différents types.

2.2.3. Séparabilité et mutabilité

La séparabilité est la propriété caractérisant le fait qu'un composant puisse être séparé de son composé. Malheureusement, cette propriété est source de confusion à cause de son manque de distinction avec, en particulier, la dépendance de vie. En effet, si on pause comme contrainte de dépendance de vie que le composant doit disparaître en même temps que (à la mort de) son composé, cela n'empêche pas la séparation. En cas de séparation préalable, cette règle n'est pas applicable, puisque la relation de composition n'existe plus. Pour préciser ce problème, il convient de s'intéresser à la mutabilité.

La mutabilité est la propriété qui permet de faire évoluer en nombre et/ou en identité l'ensemble des composants liés à un composé. Par opposition, l'immutabilité implique que l'ensemble des composants d'un composé soit le même tout au long du cycle de vie du composé, en nombre et en identité.

Il a été établi formellement que immutabilité => inséparabilité et que séparabilité => mutabilité (Barbier et *al.*, 2001).

2.2.4. Dépendances des cycles vie

Il existe neuf cas de dépendances de cycle de vie entre une instance de composant et une instance de composé. L'une d'entre elle est particulièrement remarquable : il s'agit de la *dépendance existentielle*, c'est-à-dire de la coïncidence de naissance et de mort des deux éléments de la relation. Elle est directement liée aux propriétés de mutabilité et de séparabilité. En effet, pour qu'il y ait dépendance existentielle, il faut que la propriété d'immutabilité soit vérifiée et donc celle d'inséparabilité.

2.2.5. Transitivité

Considérons un composant Tout A, composé d'un composant Partie B, lui-même composé d'un composant Partie C. La propriété de transitivité consiste à permettre à A d'utiliser directement un service de C sans que ce service soit fourni par l'interface de B. La non transitivité implique que A ne puisse utiliser ce service de C que s'il est fourni par l'interface de B.

2.3. Relation et combinaison des propriétés secondaires

Nous avons vu dans la présentation précédente que certaines propriétés ont des relations fortes entre elles. Par souci de place, nous ne décrivons pas ici l'ensemble de ces interactions (ou dépendances) mais nous allons en donner quelques exemples.

La propriété de dépendance existentielle, par exemple, est intimement liée avec les propriétés d'immutabilité et d'inséparabilité. La propriété d'encapsulation est également illustrative. Nous envisageons la composition logicielle comme *verticale*, c'est-à-dire que nous considérons toute composition sous la forme de relation Tout-Partie : un composant Tout est composé de composants Partie, ces derniers jouant le rôle de fournisseur de services pour ce composant Tout. En ce sens, lorsque la propriété d'encapsulation est spécifiée cela implique pour le composant d'être non partageable localement et globalement, puisqu'il ne peut fournir ses services qu'au composant Tout. De même, le fait que ce sous-composant ne soit pas partageable implique que, si le composant Tout est lui-même le composant Partie d'un troisième composant de plus haut niveau, ce dernier ne pourra pas utiliser directement les services du composant Partie de plus bas niveau. La propriété de non transitivité est alors spécifiée.

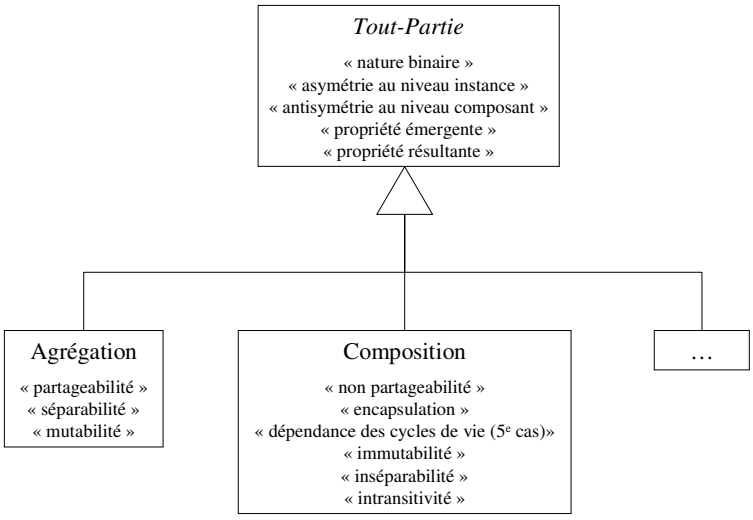

Figure 1 – Exemples de sous-types de la relation Tout-Partie

Ces exemples illustrent les dépendances pouvant exister entre les propriétés secondaires. Ces relations permettent de définir plusieurs sous-types de la relation Tout-Partie. Par exemple (*cf.* Figure 1), une relation vérifiant les propriétés de partageabilité, séparabilité et mutabilité caractérisera le sous-type appelé plus

communément *agrégation*. Une autre relation vérifiant les propriétés de non partageabilité, d'encapsulation, de dépendance existentielle, immutabilité, d'inséparabilité et de non transitivité caractérisera le sous-type appelé *composition*. Nous n'avons pour le moment étudié que ces deux sous-types puisqu'ils sont déjà traités dans UML. Cependant, ce n'est pas une réelle limitation de notre approche puisque le concepteur a toujours la possibilité de spécifier sa relation avec l'ensemble des propriétés secondaires. La définition de sous-types de relation Tout-Partie n'en est qu'une simplification. Cependant, nous travaillons dans ce sens à caractériser d'autres sous-types de relation Tout-Partie, et à définir toutes les interactions éventuelles entre propriétés pour (i) éviter les descripteurs contradictoires et (ii) définir les dépendances et implications entre ces propriétés.

2.4. Avantage de cette approche

L'intérêt de cette approche est de permettre l'ajout de propriétés sémantiques spécifiquement sur la relation de composition. C'est un enrichissement important vis-à-vis des langages tels qu'UML qui ne traitent la composition que, au mieux, syntaxiquement. Il est possible alors de spécifier des contraintes portant sur la

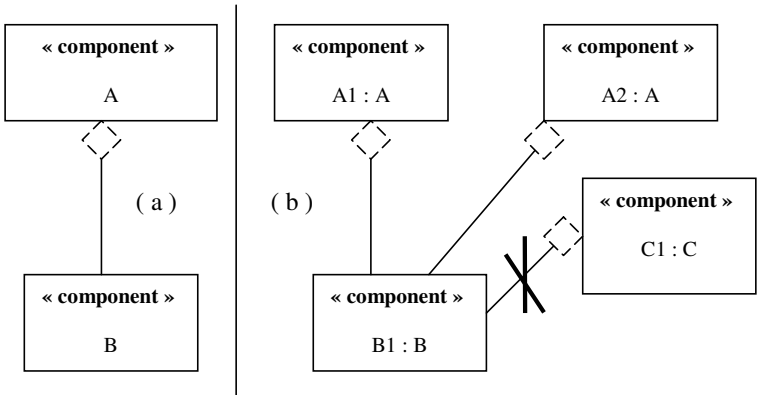

Figure 2 – Exemple de spécification

composition elle-même. Cela permet notamment d'envisager les interactions entre les composants dès la phase de modélisation et de fixer ainsi le comportement de ces interactions.

Nous illustrons ici un exemple de l'impact que peut avoir la spécification d'une de ces propriétés sur le système. Dans la Figure 2 (a), le concepteur a spécifié la partageabilité locale et la non partageabilité globale du composant *B* (utilisant une notation introduite dans (Barbier *et al.*, 2003) et où le losange pointillé représente une relation de type Tout-Partie), dans la relation Tout-Partie *A* – *B*. L'impact de ces

propriétés est illustré en partie (b) de la figure. Le composant peut-être partagé par plusieurs instances de type *A*, mais ne peut être mis en relation avec un composant d'un autre type.

2.5. Applicabilité de cette approche vis-à-vis du développement

L'intérêt d'une modélisation dans laquelle la spécification de propriétés sémantiques a une place accrue ne se justifie que si ces propriétés sémantiques se retrouvent implémentées et vérifiées dans l'application développée. En effet, un modèle, même validé, ne garanti en rien que l'application développée l'ait été en respectant les propriétés définies dans le modèle. De plus, un modèle valide peut cacher un certain nombre de problèmes, notamment des propriétés non implémentables techniquement. Partant de ce constat, nous proposons deux approches de vérification de nos modèles : une approche *a priori* qui permet de vérifier la cohérence de notre modèle avant intégration finale et une approche *a posteriori* qui permet une vérification finale de l'application (Belloir *et al*, 2003a), cette dernière, basée sur la technologie BIT, a été définie dans le cadre du projet européen Component+[1]. L'intérêt de la vérification *a priori* est évidemment de déterminer le plus tôt possible la cohérence du modèle et la faisabilité de son implémentation. Notre approche, présentée dans la section suivante, est basée sur une architecture JMX (Java Management eXtensions) (Sun, 2002). Elle est donc limitative au monde Java. Elle n'est donc pas une réponse totale à notre problématique mais elle est illustrative de la faisabilité de notre approche.

3. Environnement de vérification

Nous présentons maintenant un environnement de vérification basé sur le concept de relation Tout-Partie. Cet environnement contraint l'assemblage des composants logiciels à l'exécution, selon les propriétés intrinsèques de la relation que nous avons définies précédemment. Nous présentons tout d'abord la technologie que nous avons utilisée et détaillons ensuite l'implémentation des relations Tout-Partie que nous avons réalisée.

3.1. Présentation de JMX

Notre environnement est basé sur la technologie JMX. Cette librairie permet la gestion de composants Java, appelés MBeans (Managed Beans). Chaque MBean est référencé dans un serveur de MBeans par un nom unique, son « object name ». La communication inter-MBean s'effectue via le serveur de MBeans grâce aux « object

[1] Projet européen IST- 1999-20162 voir : http://www.component-plus.org

names ». Chaque MBean dispose également d'une interface de gestion dans laquelle il expose les attributs et les méthodes qui seront accessibles aux autres MBeans. L'architecture de JMX permet en outre de connecter un navigateur Web au serveur de MBeans : le serveur émet des pages HMTL permettant d'administrer les MBeans, en accédant notamment à leurs interfaces de gestion. La Figure 3 illustre cette architecture.

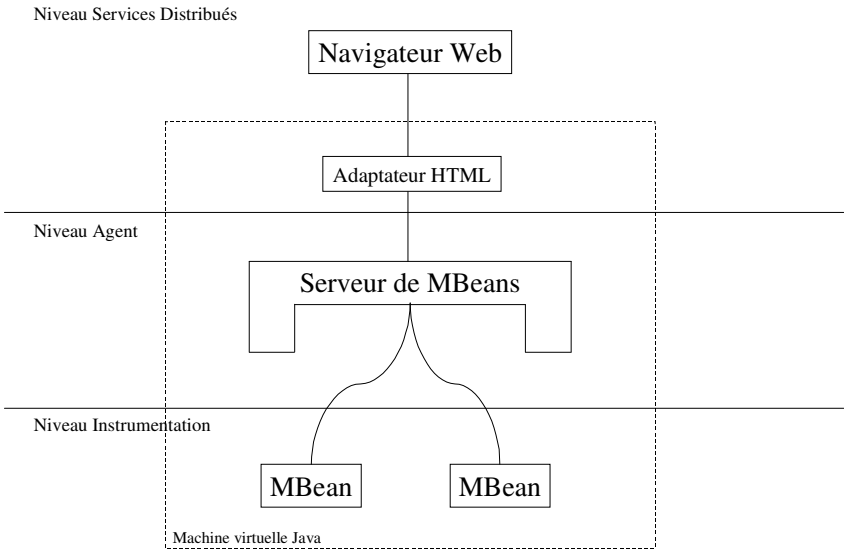

Figure 3 – Architecture simplifiée de JMX

JMX propose un service optionnel de relation, à partir duquel nous avons construit notre environnement. Ce service permet de créer et de gérer des relations entre MBeans et maintient également leur cohérence (e.g., respect des cardinalités). Ces relations sont de simples associations n-aires avec rôles nommés.

3.2. Implémentation de relations Tout-Partie

L'implémentation de WPR que nous avons réalisée a été initialement entreprise dans (Romeo, 2003). Elle est conçue sur la base d'une relation JMX. Les propriétés de la relation Tout-Partie sont satisfaites en utilisant conjointement les possibilités offertes par le serveur JMX et son service de relation. Par exemple, la nature binaire de la relation entre le composant Tout et son composant Partie est figée par un type

de relation JMX comprenant un rôle « Tout » et un rôle « Partie », et l'antisymétrie au niveau des composants de la relation est vérifiée grâce à un algorithme parcourant les types de relation déjà enregistrés dans le service de relation.

L'architecture que nous avons conçue permet d'implémenter simplement des sous-types de la relation Tout-Partie. Nous décrivons ce processus pour les deux sous-types définis dans UML (OMG, 2001a) : l'agrégation et la composition (voir Figure 1). La relation Tout-Partie est dotée par essence des propriétés primaires, tandis que les propriétés secondaires sont assignées à l'agrégation et la composition.

Dans notre environnement, nous avons défini une classe abstraite « WholePart » qui implémente toutes les propriétés, primaires et secondaires, de Tout-Partie et des interfaces de marquage pour chaque propriété secondaire. Ainsi, pour créer un sous-type de Tout-Partie, il faut étendre la classe « WholePart » afin d'hériter directement des propriétés primaires et utiliser seulement les interfaces de marquage correspondant aux propriétés secondaires que l'on veut activer. Pour l'agrégation et la composition cela se traduit par le code suivant :

```
public interface CompositionMBean
         extends Encapsulation, LifetimeDependency {
}

public class Composition extends WholePart implements
CompositionMBean {
      ...;
}

public interface AggregationMBean extends Shareability,
                                          Separability,
                                          Mutability {
}

public class Aggregation extends WholePart implements
AggregationMBean {
      ...
}
```

Les implémentations de ces deux relations sont extrêmement simples. Seuls les paramètres des constructeurs, relatifs à JMX, ont été omis. En plus de fournir un cadre pour la vérification *a priori* des relations entre composants, la flexibilité de cette approche facilite l'expérimentation de nouvelles relations en modulant leurs propriétés. Cela nous permet de déterminer pratiquement les propriétés adéquates à la composition logicielle et les sous-types de WPR correspondants au niveau modélisation.

4. Application à une étude de cas

Dans cette section nous présentons notre étude de cas, puis nous montrons l'application de notre méthode à cette étude de cas, en traitant d'abord la modélisation puis la vérification *a priori* de cette modélisation.

4.1. Présentation de l'étude de cas

Notre étude de cas consiste en un système de gestion de distribution de boisson décomposé en trois composants complexes i.e., non primaires (ou encore *composés*). Nous présentons celui gérant une machine a café. Le composant *CoffeeMachine* gère les services de saisie de monnaie ainsi que ceux de préparation et de livraison des boissons. Dans notre approche, il s'agit d'un composant *Tout*. Nous le représentons, en utilisant le formalisme UML 2.0. (OMG, 2003), sous la forme d'un *PackagingComponent*. Il contient deux sous-composants (ou composants *Partie*) représentés sous la forme de *BasicComponent* : le composant gérant la saisie et le remboursement le la monnaie, nommé *Coiner* et le composant gérant la préparation et la distribution des boissons, nommé *DrinkMaker* (voir la Figure 4).

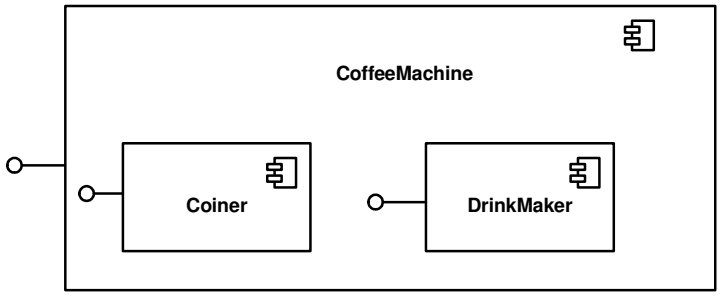

Figure 4 – Décomposition du composant CoffeeMachine

4.2. Utilisation des propriétés de la relation Tout-Partie pour la conception du système

Le composant *DrinkMaker* est une composante spécifique du composant *CoffeeMachine*. Il n'a pas d'autre utilité dans cette application. Nous avons donc spécifié que seul le composant *CoffeeMachine* pouvait y accéder. DrinkMaker est donc encapsulé dans *CoffeeMachine*. Pour assurer le fait que seul le Tout puisse y accéder, la propriété d'exclusion globale a été spécifiée. De même, comme ce

composant ne peut servir qu'à un seul composant *CoffeeMachine*, la propriété d'exclusion locale vient renforcer cette idée. Pour assurer la continuité du service, le composant doit être existant de manière permanente. Il ne doit donc pas être séparable du Tout. Cela implique son immutabilité et sa dépendance existentielle. Enfin, la non transitivité est assurée par l'encapsulation.

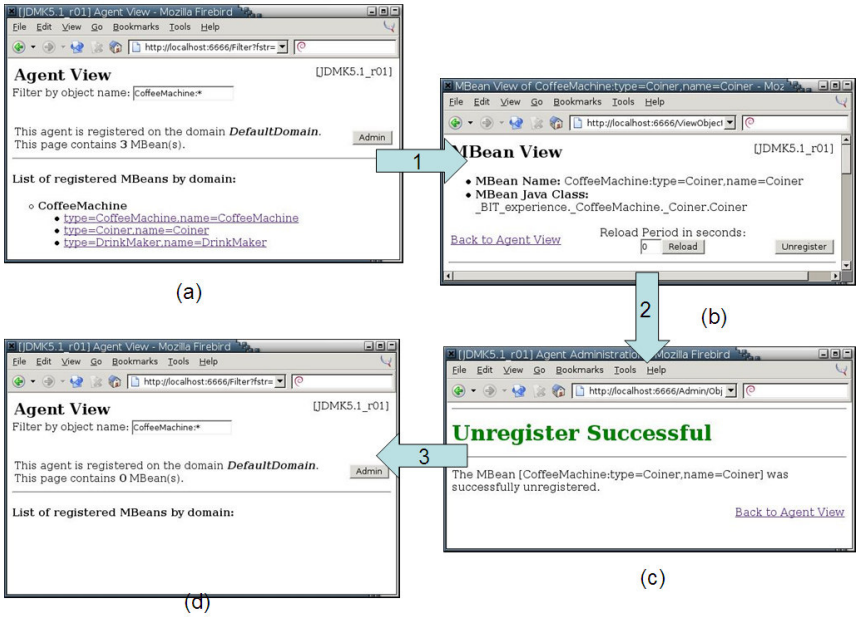

Figure 5 – Vérification de la dépendance existentielle

Le cas du composant *Coiner* est différent. En effet, dans le cadre de notre étude de cas, le composant *CoffeeMachine* est couplé avec une interface graphique et un mécanisme de porte-monnaie électronique. Les utilisateurs réguliers de la machine à café auront la possibilité de stocker de la monnaie virtuelle en utilisant le même monnayeur que celui permettant de rentrer de l'argent pour obtenir un café. Le porte-monnaie électronique n'est pas présenté ici mais il doit utiliser le même composant *Coiner* que le composant *CoffeeMachine*. Il faut donc partager ce composant. Il ne doit donc pas être encapsulé dans le composant *CoffeeMachine*. La propriété de partage globale est donc spécifiée. Par contre, le composant *CoffeeMachine* doit dépendre existentiellement du *Coiner* car on ne veut pas que la destruction du *Coiner* permette l'utilisation libre de *CoffeeMachine*.

4.3. Vérification

Nous avons mis en œuvre notre étude de cas dans une implémentation Java utilisant notre environnement. Cela nous permet de vérifier la cohérence des propriétés des relations que nous avons spécifiées dans la section précédente entre les composants du système. Selon la nature des propriétés, l'environnement effectue une vérification positive ou négative : soit l'environnement assure lui-même la propriété à la relation, 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 allons présenter un exemple de ces deux possibilités.

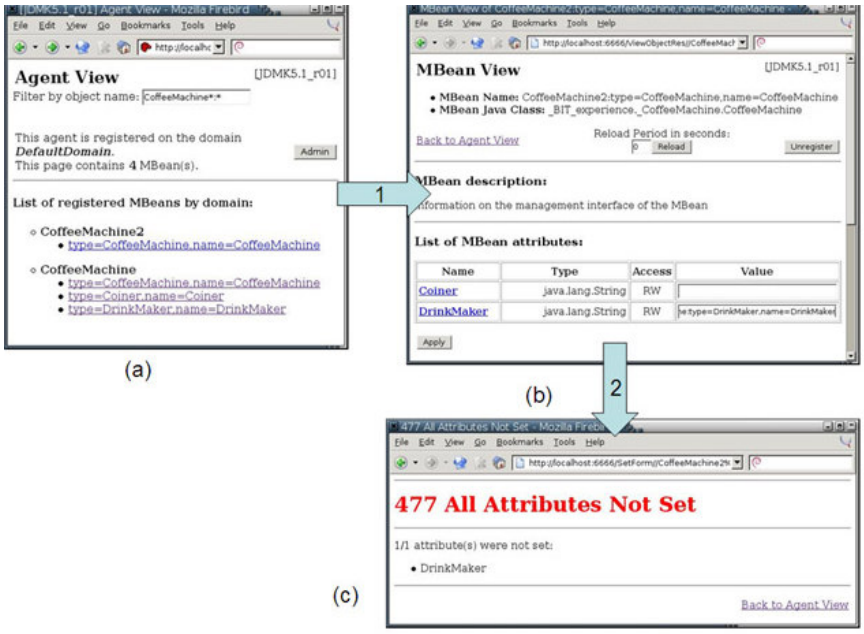

Figure 6 – Vérification de la partageabilité

Premièrement, dans notre étude de cas, le composant *CoffeeMachine* est en dépendance existentielle avec les composants *Coiner* et *DrinkMaker*. Ainsi la destruction du composant *Coiner*, par exemple, entraîne la destruction des deux autres composants par propagation de la propriété de dépendance existentielle. C'est un exemple de vérification positive. La Figure 5 illustre ce processus. En *(a)*, les trois composants sont manipulables par l'interface Web. Après avoir sélectionné le composant *Coiner*, ce composant est détruit en *(b)*, ce que valide JMX en *(c)*. Nous voyons bien de retour à l'interface Web en *(d)* qu'aucun des trois composants précédent n'est manipulable maintenant.

Deuxièmement, nous avons spécifié que le composant *DrinkMaker* est en relation exclusivement avec le composant *CoffeeMachine*. Cette relation est donc non partageable (i.e. exclusions globale et locale). Par exemple, un deuxième composant du même type que *CoffeeMachine* ne doit pas pouvoir être en relation avec le composant *DrinkMaker*. La Figure 6 illustre cette situation, où on voit que la violation de la propriété notifiée en *(c)* empêche l'attachement (en *(b)*) de *DrinkMaker* avec un deuxième composant *CoffeeMachine*.

Afin d'illustrer l'importance de la prise en compte de la composition au plus tôt, nous avons, à partir du modèle UML (*cf.*, Figure 4), développé les classes Java correspondantes sans donc tenir compte des propriétés de composition. Il est alors impossible de détecter une « mauvaise » implémentation du composant *DrinkMaker* (partageable par exemple). Alors que dans le cas où les propriétés ont été exprimées, celle-ci est automatiquement détectée.

5. Conclusion et perspectives

Pour répondre notamment aux besoins de flexibilité et d'évolution rapide des systèmes d'information, l'ingénierie logicielle basée composant se pose comme une solution particulièrement intéressante. Cependant, dans ce domaine, les besoins en termes de développement de qualité sont encore importants, notamment en ce qui concerne la modélisation et la vérification du respect des propriétés spécifiées.

Dans cet article, nous avons présenté une technique de modélisation pour les composants logiciels basée sur la relation Tout-Partie. L'idée est d'aborder la relation de composition comme une relation Tout-Partie. Nous avons montré, à l'aide des propriétés secondaires, notamment, comment nous pouvons spécifier la sémantique de la composition logicielle. Nous avons montré ensuite un outil permettant de vérifier *a priori* le respect, par l'assemblage de composants logiciels, des spécifications de la modélisation. Nous avons illustré notre approche à l'aide d'une étude de cas.

Le travail présenté dans cet article rentre dans le cadre d'un projet de plateforme de composition logicielle. Celle-ci doit permettre la modélisation, la mise en œuvre et la validation d'assemblages de composants. Nous avons présenté notre approche concernant la modélisation. Or, cette modélisation se fait actuellement « manuellement » et sans formalisme particulier. Nous travaillons, afin d'automatiser ce processus, sur la création d'un *profile* (OMG, 2001) UML dédié à la modélisation d'applications basées composants et s'appuyant sur notre approche. Un *profile* offre l'avantage de fournir un formalisme dédié à notre approche, mais aussi de permettre de coder les propriétés formellement à l'aide du langage OCL (OMG, 1997), mais de manière sous-jacente et transparente pour l'utilisateur. Il aidera donc le concepteur à mettre en œuvre cette méthode, et permettra notamment de vérifier la cohérence entre les propriétés de composition spécifiées, cela grâce à l'intégration des propriétés dans le *profile* sous forme de règles OCL. Ce *profile* sera intégré à un

environnement de modélisation UML. Nous explorons actuellement deux supports possibles, l'un du domaine public, SMW (Porres, 2003), et l'autre du domaine commercial, Objecteering (Softeam, 2003).

Nous avons présenté lors d'Inforsid 2003 nos travaux sur le test et la validation des composants (Belloir *et al.*, 2003a). Cette technique permet une validation *a posteriori* de la composition logicielle. Le second aspect sur lequel nous travaillons concerne l'intégration de ces différentes techniques dans un cadre de développement cohérent. Nous travaillons en particulier sur la génération de code de test et de validation depuis le modèle UML. Cela permettra a terme une aide accrue pour la mise en œuvre des validations *a priori* et *a posteriori*.

Enfin, nous cherchons à moyen terme à étendre notre travail en vue de proposer un véritable langage de modélisation de composition (Bruel, 2003b). Comme illustré dans cet article, le manque de support effectif et dédié au CBSE en terme de modélisation est un des principaux freins à une utilisation plus massive des composants logiciels, et notamment des composants « sur l'étagère » (Commercial Off-The-Shelf Components – COTS Components). Nous démarrons ainsi un projet de trois ans, dans le cadre des actions concertées incitatives du MERNT, visant à la définition d'un tel langage. A la frontière entre les notations de modélisation généraliste comme UML, et les techniques plus dédiées comme les ADLs (Architecture Description Language), ce projet nous permet de fédérer nos efforts en matière de composabilité (comme ceux décrits dans cet article) et ceux menés par ailleurs en matière de prédiction du comportement (et plus précisément des propriétés extra-fonctionnelles) des assemblages de composants (Constant *et al.*, 2003).

Remerciements

Les auteurs remercient Jean-Michel Bruel et les relecteurs qui par leurs commentaires et leurs remarques éclairés ont contribué à la qualité de cet article.

Bibliographie

Barbier F., Henderson-Sellers B., Le Parc-Lacayrelle A., Bruel J.M., « Formalization of the Whole-Part Relationship in the Unified Modelling Language », *IEEE Transactions on Software Engineering*, Vol 29, No 5, pp. 459-470, mai, 2003.

Barbier F., Henderson-Sellers B., Opdahl A.L., et Gogolla M., « The whole-part relationships in the Unified Modeling Language : a new approach », *journal Unified Modeling Language : System Analysis, Design and Developpement Issues* , Siau K and Halpin T, Idea Group Publishing, pp. 185-209, 2001.

Belloir N., Bruel J.M., Barbier F., « Application de la théorie de la relation Tout-Partie à la composition de composants logiciels », *Actes du XXIeme congrès Inforsid*, pp 35-50, Nancy, 3 - 6 juin , 2003.

Belloir N., Bruel J.M., Barbier F., « Whole-Part Relationships for Software Component Combination. », *Proceedings of the 29th Euromicro Conference on Component-Based Software Engineering*, IEEE Computer Society Press,1 - 6 septembre , pp 86-91, 2003.

Bosch J., Szyperski C., Weck W., « Ws5. The Eighth International Workshop on Component-Oriented Programming (WCOP 2003 », *report on the* Eighth International Workshop on Component-Oriented Programming, http://research.microsoft.com/~cszypers/ , 2003.

Bruel J.M. et Ober I., « The new UML 2.0 Component Model: Critical View.", dans Erwin Grosspietsch and Konrad Klöckner, editors, *Proceedings of the Work in Progress Session at the 29th Euromicro Conference*, September 2003.

Bruel J.M., « CML: Component Modeling Language », *Action Concertée Incitative Jeunes Chercheurs JC9067*, December, 2003.

Constant O., Monin W., Rougeot B., Barbier F., « Performance et composants logiciels : une démarche en conception », *Submitted at JC'2004*, December, 2003.

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

High Confidence Software and Systems Coordinating Group, « High Confidence Software and Systems Research Needs », *Interagency Working Group on Information Technology Research and Development*, http://www.ccic.gov/pubs/index.html, janvier 2001.

Object Management Group (OMG), « Object Constraint Language Specification V 1.1 », *http://www.omg.org/,* 1997.

Object Management Group (OMG), « Unified Modeling Language V 1.4 Specification», *http://www.omg.org/uml/,* 2001.

Object Management Group (OMG), « Model Driven Architecture », *http://www.omg.org/mda/*, 2001.

Object Management Group (OMG), « UML 2.0 Superstructure Specification », *http://www.omg.org/uml/*, document superstructure - 03-08-02.pdf, 2003.

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

Pores I., « SMW UML Tool », http://www.abo.fi/~iporres/html/smw.html, 2003.

Romeo F., « Composabilité des composants logiciels et test intégré », Mémoire de DEA Programmation et Système, Université Paul Sabatier, Toulouse III, 2003.

Softeam, « Objecteering UML Tool », disponible sur http://www.objecteering.com/, 2003.

Sun Microsystems, « Java™ Management Extensions Instrumentation and Agent Specification v1.2.», http://java.sun.com/products/JavaManagement/index.jsp, octobre 2002.

Szyperski C*., Component Software – Beyond Object-Oriented Programming – Second Edition*, Addison-Wesley and ACM Press, 2002, ISBN 0-201-74572-0.