Application de la théorie de la relation Tout-Partie à la composition de composants logiciels

Nicolas Belloir – Jean-Michel Bruel – Franck Barbier

Laboratoire d'Informatique de l'Université de Pau et des Pays de l'Adour,
BP 1155, 64013 Pau Cedex
{belloir, bruel, barbier}@univ-pau.fr

RÉSUMÉ. *La réutilisabilité est un facteur clé pour la réussite du développement d'applications à faible coût. L'ingénierie logiciel basée composant se propose de répondre à ce challenge en fournissant des composants logiciels modulables et aisément intégrables dans des applications. Dans ce cadre, la composabilité est un problème clé, notamment, à cause de la difficile collaboration des composants logiciels entre eux. Dans cet article, nous nous intéressons à une technique de composition à haut couplage basée sur une encapsulation des composants de faible granularité au sein de composants de granularité plus forte, ceci sur un même noeud de déploiement. Pour cela, nous proposons l'intégration des propriétés de la théorie de la relation Tout-Partie au sein d'un environnement de composition. Nous contraignons la modélisation de la composition de composants grâce à un ensemble de règles de compositions héritées de cette relation. Enfin, nous expliquons comment nous fournissons des fonctionnalités de test intégrées aux composants permettant la validation de leur composition.*

ABSTRACT. *Reutilisability is a key factor for the success of the development of low cost applications. Component-based software engineering (CBSE) proposes to answer this challenge by providing flexible and easily integrable software components in applications. Composability is a key question of CBSE, because software components collaborate with difficulty, in spite of their a priori capacity to do so. In this paper, we propose to formalize a special kind of composability based on a high coupling encapsulation of low granularity components within stronger granularity components, this on the same node of deployment. We study the integration of the properties of the whole-part relationship within a composition framework. We force the component composition design within the framework thanks to a set of composition rules inherited from the whole-part relationship. These constraints are generated as contracts at component level. Finally, we explain how we provide built-in test functionality into components allowing validation of their composition.*

MOTS-CLÉS : *Composabilité, Composant Logiciel, Relation Tout-Partie, Modélisation*

KEYWORDS: *Componability, Software Component, Whole-Part Relationship, Design*

1. Introduction

Avec l'avènement des nouvelles technologies, la société de l'information connaît une extension tout azimut tant en terme de supports (Internet ou les téléphones portables par exemple) que de services (e-Business ou e-Services par exemple). Cela a pour conséquence d'imposer aux applications de nouvelles obligations en terme de capacité de déploiement, d'adaptation ou encore de modularité, ce qui en accroît la complexité et donc le coût du développement. Une des approches classiques visant à réduire les coûts de développement consiste à réutiliser tout ou partie de code existant. La technologie objet a été une des réponses à ce besoin de réutilisation. Or force est de constater que, malgré ses apports considérables sur de nombreux aspects, l'objectif de réutilisabilité n'a pas été complètement atteint. Une des raisons est que la réutilisation est envisagée trop tard dans le cycle de vie du logiciel, i.e. au niveau de l'implémentation. En effet, même les approches liées à UML, qui permettent par exemple de prendre en compte au plus tôt les aspects de réutilisabilité, ne le font que de manière textuelle ou documentaire. Le concept de composant existe en effet dans la documentation UML 1.4, mais aucune sémantique ne lui est associé. Les explications sur ce concept font quelques pages, et aucune relation formelle entre l'utilisation de ces composants et les autres diagrammes du modèle n'est permise. En plus de cela, l'héritage des classes présente des défauts en ce qui concerne les objectifs de réutilisabilité, à cause, pour l'essentiel, de leur faible degré de granularité. Des structures plus conséquentes ont alors du être candidates à la réutilisation (comme par exemple, les patrons de conception de Gamma [GAM 95]).

L'ingénierie logicielle orientée composant (*Component Based Software Engineering*, CBSE) est née de ce constat d'échec relatif. A cela, plusieurs raisons : d'une part, la composition d'objets réutilisables va au–delà du simple mécanisme d'héritage. D'autre part, les composants sont des entités généralement de taille plus importante que les objets et sont donc envisagés plus tôt dans le cycle de vie. Enfin, les applications basées composants sont plus canoniques [JOH 97] que les applications objets (existence de modèles de composants et de composabilité).

Dans ce cadre la composabilité est un problème clé, notamment, parce que les composants logiciels collaborent difficilement, malgré leur capacité a priori à le faire. Cela se traduit par des dépendances entre les composants qui ne sont ni toujours visibles ni toujours spécifiées mais également par une méconnaissance du comportement des assemblages de composants. Dans ce cadre, la prédiction du comportement est un des challenges de la composabilité [CRN 02]. Nous envisageons cette dernière selon deux angles distincts : la *composabilité horizontale* correspond aux problèmes de liaisons et de coopérations entr composants au sein des systèmes distribués ; la *composabilité verticale* traite de *composants Tout* utilisant les services de sous-composants (ou *composants Partie*), sur un même nœud de déploiement. Ces parties, complètement encapsulées, ne sont par ailleurs pas des unités de déploiement mais agissent en tant qu'implémentation des services fournis par le composant Tout. A cause de cela, et du fait que ces composant Partie sont non partagés, nous pouvons les envisagés comme membres de l'interface requise du composant Tout. Cette approche offre singulière-

ment de la flexibilité, par l'idée d'interface de configuration [BOS 00], qui permet de paramètrer les composants statiquement ou dynamiquement afin de les mettre en accord avec le contexte d'exécution. Nous nous intéressons dans cet article à ce type de composition.

D'autre part, les approches compositionnelles s'appuient peu sur des théories scientifiques éprouvées dans d'autres domaines de l'ingénierie logicielle telle que la relation Tout-Partie. Cette dernière a notamment apportée des solutions probantes en modélisation objet [BAR 02b] et en particulier dans la résolution des incohérences qui existaient au niveau de la définition des notions d'agrégation et de composition dans UML 1.4. [OMG 01]. Nous proposons d'appliquer les propriétés de la relation Tout-Partie à la composition de composants à travers les concepts de *composant Tout* et de *composant Partie*. Une proposition dans ce sens a été amorcée dans [BAR 02a]. Cet article se propose d'étendre les propriétés sémantiques de la relation Tout-Partie en leur donnant une assise formelle dans le cadre de la composition logicielle. Notre objectif est de déterminer quelles propriétés s'appliquent à la composition de composants puis de définir formellement ces propriétés. Nous proposons ensuite de les intégrer ces propriétés niveau métamodèle afin de contraindre la spécification de compositions logicielles. A terme, nous fournirons un environnement de composition dans lequel, à l'aide d'un outil de modélisation basé sur ce métamodèle, la composition de composants sera contrainte dès la modélisation. Puis, ces contraintes se retrouveront générées sous forme de contrats au niveau des composants. Enfin, nous fournirons des fonctionnalités de test intégrées aux composants [BAR 03] permettant la validation de leur composition.

Après avoir discuté de la composabilité en relation avec la théorie Tout-Partie en section 2, nous introduisons notre proposition de modèle en section 3. Nous y discutons les conditions sous lesquelles un composant Tout peut être combiné avec des composants Partie. Enfin, et avant de conclure, nous montrons en section 4 comment nous proposons d'intégrer ces travaux à une librairie Java^{TM}, que nous avons développé, permettant d'intégrer aux composants des fonctionnalités de test. Ces fonctionnalités permettent de manipuler les composants afin de tester leur comportement dans un environnement de déploiement dans le but de les valider.

2. Composabilité et relation Tout-Partie

Dans cette section, nous présentons notre approche de composition basée sur la relation Tout-Partie. Pour cela, nous précisons ce que nous entendons par composabilité et les notions de composants Tout et de composant Partie. Ensuite, nous présentons la relation Tout-Partie ainsi que les propriétés pertinentes que nous avons identifiées. Puis nous discutons de l'adaptation de ces propriétés à la composition de composants. Enfin, nous précisons notre proposition d'application de la relation Tout-Partie à la modélisation du comportement des composants.

2.1. *Composabilité*

L'intégration de composant est le mécanisme permettant de relier des composants entre eux en mettant en relation leurs interfaces respectives. Cependant, contrairement au monde des composants électroniques, la simple intégration n'est pas suffisante pour assurer la qualité des interactions entre composants lors de leur déploiement. La composition doit fournir des moyens de détermination des propriétés des assemblages de manière à prévoir leur compatibilité lors du déploiement [STA 02].

La composabilité d'un composant logiciel est sa capacité à être systématiquement et facilement combiné avec d'autres composants en assurant un fonctionnement cohérent. Cette caractéristique est primordiale pour permettre l'utilisation et la réutilisation d'un composant. Cependant, bien que, comme le remarque Szyperski [SZY 97], "*les composants sont pour la composition*", l'expérience montre que les "parties de logiciel" ne sont pas toujours des composants car elles ne sont pas toujours intrinsèquement et directement incorporables dans les applications. Cette observation est confirmée dans [Hig 01] : "*However, this has led to design practices that assume components are "compositional" in all other behaviors : specially, that they can be integrated and still operate the same way in any environment. For many environments, this assumption does not hold*". La prise ne compte de la composabilité d'un composant lors de sa conception est donc fondamentale.

Cependant les problèmes de composabilité sont nombreux et complexes. Parmi ceux-ci, la taille croissante des applications rend nécessaire de décomplexifier les systèmes en envisageant la composition à divers niveaux de granularité [OMG 02] : "*Having a rigorous and consistent way to understand and deal with the hierarchy of parts and compositions, how they work and interact at each level and how one level relates to the next, is absolutely necessary for achieve the business goals of a flexible and scalable information systems*".

Dans ce cadre, nous proposons une technique d'assemblage de composant permettant de spécifier rigoureusement les connexions entre un *composant Tout* et ses *composants Partie*. Nous définissons un composant Partie comme un composant non partageable et complètement encapsulé par le composant Tout dont il fait partie. Nous détaillons ces notions dans la section suivante.

2.2. *Composants Tout et composant Partie*

La notion de composant Partie a été introduite dans [BAR 02a] sous le nom de "*sub-component*". Nous la rebaptisons ainsi afin de mieux correspondre à la terminologie Tout-Partie. Notons que [OMG 02] utilise les mots "*parts*" et "*compositions*" de manière similaires. Nous présentons dans la section suivante les propriétés pertinentes identifiées de cette relation. Parmi celles-ci, notons que le composant Partie est totalement encapsulé par le composant Tout et donc non partageable. Le composant Tout fournit un ensemble de services à travers son interface de service fourni (*pro-*

vided interface) à ses clients. Ces services sont, soit des services directement fournis par le composant Partie soit des services conjugant à la fois des services fournis par le composant Partie et par le composant Tout. Le composant Partie, du fait de l'encapsulation, ne peut les fournir directement aux clients du composant Tout. Ce service est proposé par le composant Tout à travers son interface de service fournis. Il le récupère au travers de son interface de services requis (*required interface*). Le composant Partie est en fait un élément vital pour les interfaces de service requis du composant Tout (voir Figure 1).

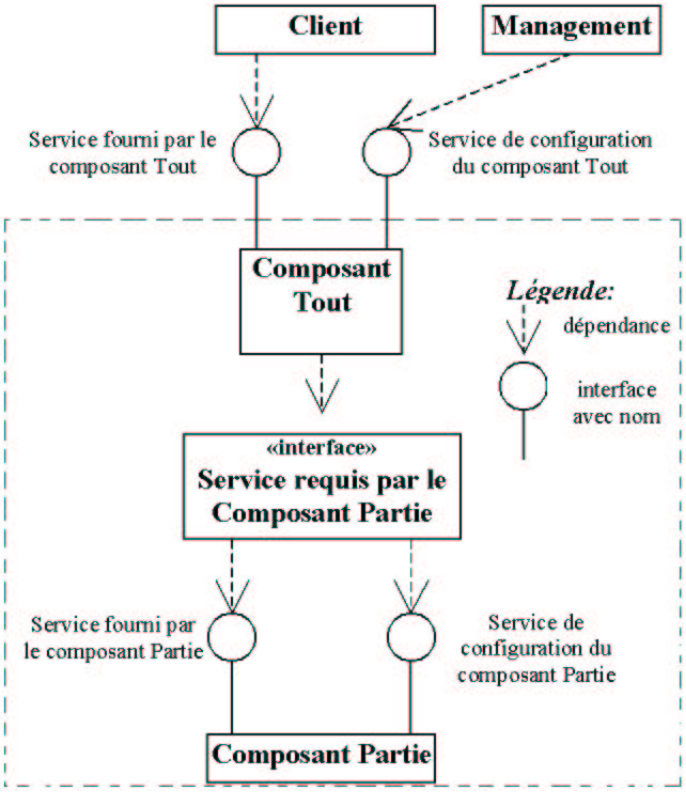

Figure 1. *Organisation canonique d'un composant Tout en regard d'un de ses composant Partie (formalisme UML)*

2.3. *La relation Tout-Partie : de l'approche orientée objet à l'approche composant*

De nombreuses disciplines se sont intéressées à la relation Tout-Partie. Nous avons mené une étude approfondie de ces différents travaux dans le but d'en extraire un ensemble de propriétés récurrentes et caractéristiques. À partir de ces propriétés, nous

Propriétés primaires	nature binaire, propriété émergente, propriété résultante, asymétrie au niveau des instances, antisymétrie au niveau des types
Propriétés secondaires	encapsulation, dépendance des cycles de vie (9 cas), transitivité, partageabilité, séparabilité, mutabilité

Tableau 1. *Propriétés primaires et secondaires de la relation Tout-Partie*

avons proposé dans [BAR 02b] une nouvelle implantation dans UML de la relation Tout-Partie afin de palier aux manques que présentent les relations d'agrégation et de composition. Ces propriétés sont classifiées en propriétés primaires (propriétés que doit respecter obligatoirement une relation pour être qualifiée de Tout-Partie) et en propriétés secondaires (propriétés qui spécialisent au besoin une relation Tout-Partie). Le tableau 1 récapitule cette liste.

Nous nous proposons de discuter de la pertinence de cette liste de propriétés dans la cadre de la composition de composant et au besoin de l'adapter.

2.3.1. *Propriétés primaires*

Les propriétés citées précédemment restent valables dans le cadre de la composition de composants. Cependant, dans le domaine des composants, on parle plutôt de composant et d'instance de composant que de type et d'instance. On parlera alors d'asymétrie au niveau des instances de composants et d'antisymétrie au niveau des composants. Pour plus de détails à propos de ces propriétés, nous invitons le lecteur à consulter [BAR 02b].

2.3.2. *Propriétés secondaires*

Les propriétés secondaires caractérisent le type de relation Tout-Partie. Nous allons spécifier le type de composition que nous traitons en fonction de ces propriétés. Un des intérêts de l'approche à haut couplage que nous proposons est que le composant Tout masque totalement à ses clients la manière dont il est lui-même composé. Les composants Partie qui le composent sont totalement encapsulés. Il s'agit d'une approche totalement boîte noire. Tout composant interagissant avec un composant Tout ignore la manière dont ce dernier est constitué. Il n'a qu'un interlocuteur qui lui propose des services. Cela facilite la composition et la maintenance. Cette approche est particulièrement utile dans le cas de composants sur étagère que nous appelons COTS (pour "*Commercial On The Shelf*") dans la suite de cet article.

Pour ce faire, la propriété d'encapsulation doit être vérifiée. Un composant Partie doit être totalement encapsulé par le composant Tout avec qui il est en relation. Cela signifie également qu'un composant Partie n'est pas déployable unitairement dans l'architecture logicielle. Seul le composant Tout est déployable. Notons également que le composant Partie peut être lui-même un composant Tout composé d'autres

composants Partie. Il y a donc récursivité. Cependant, dans ce cas, ce composant perd sa capacité de déploiement unitaire lorsqu'il devient lui-même un composant Partie.

La propriété de partageabilité est la capacité d'un composant à interagir avec plusieurs composants à un instant donné. Les composants Partie sont non partageable. Ils sont possédés par un et un seul composant Tout à un instant donné.

Les propriétés d'encapsulation et de non partageabilité ont un effet direct sur la propriété de transitivité de la relation Tout-Partie. Rappelons que la transitivité dans ce cadre est la propriété qui, pour un composant Tout A composé d'un composant Partie B, lui-même composé d'un composant Partie C, permet au composant A d'accéder directement au composant C. Si A nécessite un service de C, ce service sera proposé par le composant intermédiaire mais sans que le composant Tout le sache. Dans ce type de relation à fort couplage, la propriété de transitivité n'est donc pas vérifiée.

Nous pensons que le principe d'immutabilité doit être acquis pour le composant Partie. En effet, ce principe assure que les composants partis sont stabilisés en identité et en nombre. Cela garantit la qualité de service du composant Tout en évitant ainsi les risques de confusion interne.

Nous avons formellement démontré dans [BAR 01] que l'immutabilité implique l'inséparabilité. Donc le composant Partie ne peut être séparé du composant Tout avec lequel il est en relation. Cela a pour conséquence d'assurer la disponibilité du service fourni par le composant Partie.

La dernière propriété secondaire est la dépendance de cycle de vie. Il y a 9 cas possible selon l'ordre de création et de destruction du composant Partie vis-à-vis du composant Tout. La seule possibilité qui corresponde à notre modèle est celle de la simultanéité de la création et de la destruction du composant Tout vis-à-vis du composant Partie. En effet, le composant Partie doit être totalement encapsulé dans le composant Tout et ne peut être partagé. Cela signifie qu'il n'a de sens que si le composant Tout existe. De même, le composant Tout ne peut fournir ses services qu'à la condition que ses parties existes et soient synchronisées. Il va de soit que cette "existence simultanée" doit être considérée sur un plan transactionnel.

2.3.3. *Application au comportement des composants*

Nous travaillons actuellement sur un autre point de la composabilité, directement inspiré de nos travaux sur le métamodèle. Nous cherchons à étendre la notion de relation Tout-Partie au comportement des composants. La prise en compte lors de la composition de l'étude du comportement de la composition est d'ailleurs une préoccupation critique [STA 02].

Notre hypothèse est simple : l'état global d'un composant Tout est, récursivement, l'agrégation des états de tous ses composants Partie. La Figure 2 donne par exemple une représentation basée état d'un composant Tout ayant un composant Partie S0, lui-même composé des composants Partie S1, S2 et S3. Cette approche se justifie par le fait que les composants Partie sont des parties intégrantes et complètes du compo-

sant et leur comportement ne peut alors être disjoint du comportement du Tout. Nous comptons pour cela adapter à note problématique les travaux menés au sein de notre équipe sur le sous-typage comportemental [HAM 02].

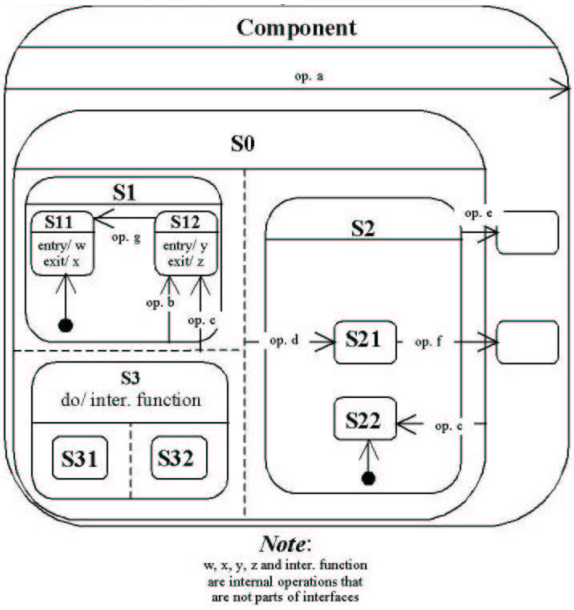

Figure 2. *Spécification de composant basée état en UML*

Notre approche est la suivante. Lorsque l'on compose des composants, à partir des diagrammes d'état de ces composants et des propriétés présentées dans les sections précédentes on peut dériver le diagramme d'état du composant Tout. Nous travaillons actuellement à définir une méthode permettant de systématiser cette approche.

3. Mise en œuvre concrète

Nous présentons dans cette section comment nous mettons en œuvre formellement le cadre théorique de la relation Tout-Partie au service de la problématique de la composition. Nous proposons un métamodèle modélisant les concepts précédemment discutés. L'idée est de rajouter des contraintes sémantiques fortes au métamodèle UML actuel afin de contraindre par la suite la composition de composants en utilisant notre approche. Les contraintes ainsi spécifiées au niveau du modèle se retrouverons lors du développement et lors du déploiement. Cela assure une composition plus sûre et permet de valider le comportement spécifié en vérifiant le respect de ces contraintes. Les contraintes n'apparaissant pas directement sur le schéma seront spécifiées à l'aide de règles formelles (cf. section 3.2). Cela permettra à terme de les inclure dans un outils

de modélisation. Nous présentons en premier lieu notre proposition de métamodèle puis un exemple en OCL de contrainte sémantique. Un exemple d'utilisation de cette approche sera détaillé en section 3.3.

3.1. *Le métamodèle*

Nous proposons de spécialiser le concept de composant UML en composant Tout et composant Partie (voir Figure 3). Cela nous permet d'ajouter explicitement le lien de composition entre ces deux spécialisations. L'intérêt de cette approche est que nous allons pouvoir ajouter une sémantique forte sur ce lien. La deuxième spécificité de notre métamodèle est la spécialisation de la notion d'interface en interface de service fourni (*provided interface*), en interface de service requis (*required interface*) et en interface de management (*management interface*). Cela permet de faire apparaître explicitement les notions acquises que sont les deux premières. La dernière représente un type d'interface spécifique permettant de configurer le composant, ce qui lui donne plus de flexibilité.

3.2. *Définitions de contraintes sur les propriétés de la relation Tout-Partie*

Toutes les propriétés qu'il est possible d'établir au niveau d'une relation Tout-Partie ne sont pas exprimées au niveau du métamodèle. A cela deux raisons : d'une part certaines sont optionnelles ; d'autre part, la notation graphique d'UML ne permet pas d'exprimer toutes les caractéristiques voulues. C'est d'ailleurs pour cela que, très rapidement, dans UML a été introduit un langage pour exprimer des contraintes sur des objets : OCL (*Object Constraint Language*) [OMG 01].

Nous avons formulé ces contraintes de manière formelle en OCL au niveau de notre proposition d'amélioration pour le métamodèle d'UML 2.0 [DST 02]. Nous sommes en train de valider et d'adapter à la composition de composants ces règles en les écrivant sous la forme de "règles de bonne construction" (*wellformedness rules*) dans un outil UML permettant leur description et leur utilisation.

A titre d'illustration d'OCL, considérons l'exemple de la propriété primaire d'asymétrie de la relation Tout-Partie. Asymétrique signifie irréflexive – f(x, x) doit être fausse – et antisymétrique – f(x, y) implique !f(y, x). En ce qui concerne l'irréflexivité, la relation Tout-Partie est irréflexive au niveau des instances, mais non des classes (cf. [BAR 02b] pour plus de détails). Ainsi la règle OCL concernant l'irréflexivité et complétant le nouveau métamodèle est la suivante [DST 02, p.10] :

```
context WPRelationship inv Irreflexivity:
theWhole.instance->forAll(w | w.oclIsKindOf(Part)
    implies not w.part->includes(w))
thePart.instance->forAll(p | p.oclIsKindOf(Whole)
    implies not p.whole->includes(p))
```

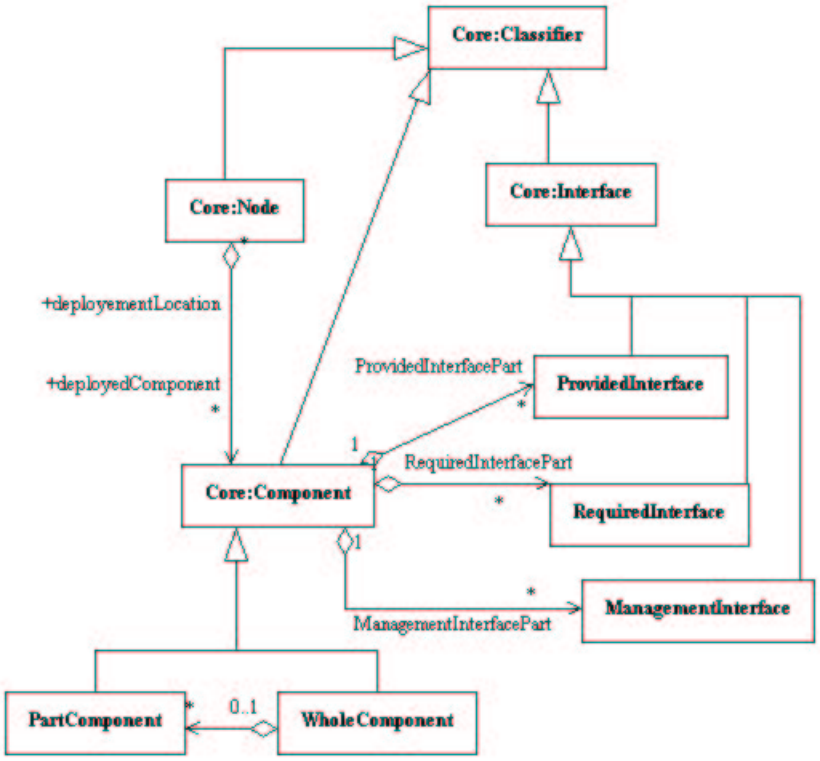

Figure 3. *Métamodèle intégrant la notion de composant Tout et de composant Partie*

3.3. *Bénéfices*

Les bénéfices de notre approche concernent la validation a priori qui peut être effectuée. Avant même le développement (ou l'achat de COTS, ou la réutilisation de composants existants) des éléments qui composent l'application, il est possible de valider un certain nombre d'éléments du développement, par exemple :

– Une violation des propriétés de la relation de composition peut être détectée automatiquement par l'outil au moment même de la conception du modèle. Le modèle en Figure 4 (b) ne peut être accepté par l'outil. En effet au moment où le concepteur va essayer de "tracer" un lien de composition entre un composant Partie (A) déjà membre d'un composant Tout (Tout 1), l'outil va détecter la violation de la propriété de non-partageabilité – exprimée au niveau du métamodèle par la cardinalité maximale 1, Figure 4 (a).

– Notre modèle de composition permet d'exprimer le comportement attendu du composant Tout à partir des comportements individuels (exprimés sous la forme de Statecharts) de ses composants Partie. On peut imaginer que ce comportement attendu est déjà exprimé (à partir des spécifications par exemple). Il devient donc alors possible de vérifier que les deux descriptions (l'exprimée, et la "calculée") correspondent.

– Un certain nombre de propriétés de la relation Tout-Partie concernent les dépendances de vie entre composant Tout et composants Partie. Il est donc possible de valider la cohérence des diagrammes de classe et des diagrammes d'états entre eux, en fonction de ces règles de dépendances.

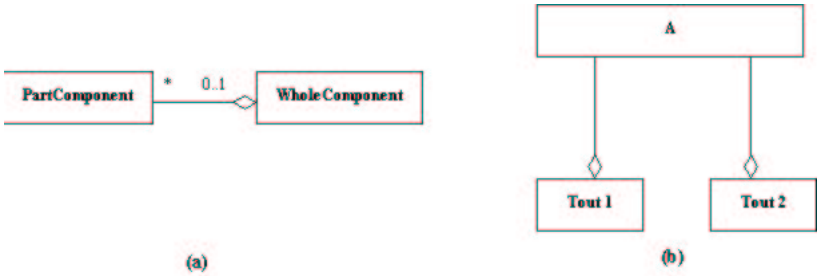

Figure 4. *Exemple de violation du métamodèle*

On voit donc que, grâce à la métamodélisation, il est possible de contraindre et donc de valider un certain nombre de caractéristiques, a priori, principalement structurelles, mais aussi comportementales.

4. De la validation aux tests

4.1. *Démarche*

Nous avons proposé un modèle de composition permettant de spécifier un certain nombre de contraintes sémantiques lors de la modélisation de compositions de composants logiciels. Ces contraintes permettent de prévoir le comportement de la future composition et ainsi d'effectuer certaines vérifications a priori, comme nous l'avons illustré dans la section précédente.

Cependant, afin d'accroître la confiance des utilisateurs de composants et notamment de composants COTS, il faut permettre à ceux-ci de vérifier que les propriétés spécifiées lors de la modélisation se retrouvent correctement implémentées, permettant ainsi une validation a posteriori. Pour cela, il est nécessaire de mettre à la disposition de ces derniers des outils leur permettant de vérifier le respect des contraintes spécifiées. Dans ce sens, nous avons proposé une technique[1], nommée Built-In Test

1. Développée dans le cadre du projet européen IST-1999-20162 COMPONENT+.

(BIT), permettant d'ajouter aux composants des fonctionnalités de test intégrées aux composants et nous avons développé une librairie Java^{TM} (nommée BIT/J) permettant de mettre en œuvre cette technologie, notamment avec des COTS [BAR 03]. Cette technologie permet à l'utilisateur d'un composant de le manipuler à l'aide d'une interface de test générique, et de le tester afin de vérifier son comportement d'une part, mais aussi la correcte implémentation des propriétés sémantiques spécifiées lors de la modélisation.

Cette technologie se base sur une extension du principe des contrats de Meyer [MEY 97] : chaque composant doit remplir un contrat, vis-à-vis des composants avec lesquels il est en relation. Ce contrat est spécifié lors de la modélisation en terme de pré- et de post-condition. Nous proposons d'inclure dans ces contrats les contraintes et propriétés liées à la composition spécifiée lors de la modélisation et de permettre ainsi leur vérification.

4.2. *La librairie BIT/J*

Nous sommes actuellement en phase de finalisation de la librairie BIT/J. Nous avons développé un outil de génération automatique qui, à partir d'un composant Java^{TM}, génère une partie importante du code des fonctionnalités de test ainsi qu'une interface de test permettant de manipuler le composant à distance à l'aide de la technologie JMX [SUN 02]. La Figure 5 représente l'interface du générateur. Pour cette exemple, nous avons choisi le composant *Stack* fourni par l'API Java^{TM}. Comme nous n'en avons pas le source, nous le considérons comme un composant COTS. Le générateur produit quatres fichiers : le composant BIT, le testeur et deux fichiers nécessaires à JMX pour le fonctionnement du test à distance, l'interface JMX et l'agent JMX. L'utilisateur doit ensuite finaliser le composant BIT.

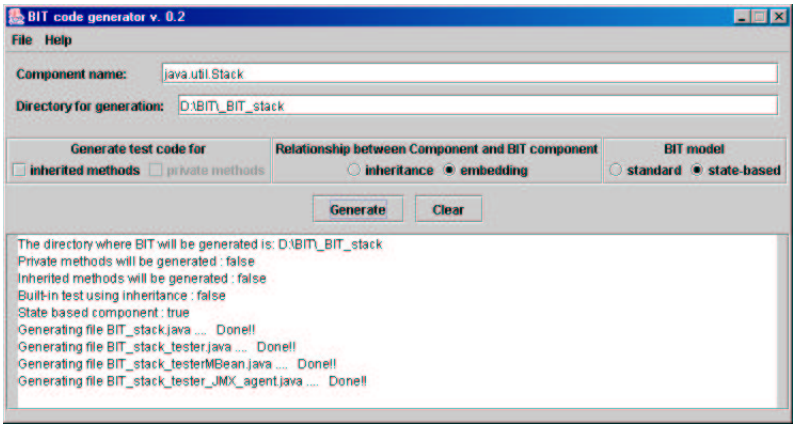

Figure 5. *Interface du générateur automatique*

Pour cet exemple, nous avons choisi de tester le composant selon le modèle BIT basé état. Pour cela, nous décrivons le comportement (voir Figure 6) du composant dans le composant BIT en se basant sur son diagramme d'état (le code non généré, c'est à dire entré par l'utilisateur, est en italique) :

```
protected void init_behavior() throws Statechart_exception
{
/* state definitions and formal relationships here */

_Empty = new BIT_state("Empty") ;
_Only_one =new BIT_state("Only one") ;
_More_than_one = new BIT_state("More than one") ;
_Not_empty = (BIT_state) (_Only_one.xor(_More_than_one))
.name("Not empty") ;
_BIT_stack = new BIT_state_monitor(_Empty
.xor(_Not_empty),"BIT stack") ;
_Empty.inputState() ;

}
```

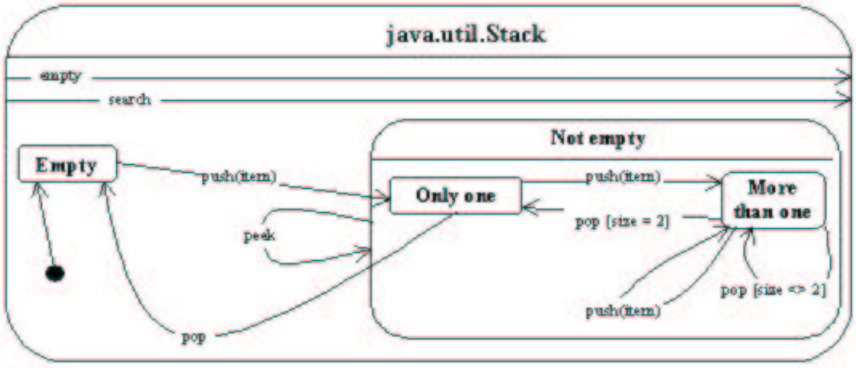

Figure 6. *Statecharts du composant* Stack

Ensuite, le code faisant le lien entre les transitions d'états et les opérations du composant est rajouté manuellement dans le composant BIT.

```
public java.lang.Object push(java.lang.Object o1) throws
Statechart_exception {

java.lang.Object result = _stack.push(o1);
/* state transitions here : _BIT_stack.fires(fromState,
toState); */

 _BIT_stack.fires(_Empty, _Only_one);
_BIT_stack.fires(_Only_one, _More_than_one);
_BIT_stack.fires(_More_than_one, _More_than_one);
_BIT_stack.used_up();

_BIT_stack.used_up();
return result;
}
```

Une fois les modifications terminées, l'utilisateur peut lancer l'exécution du composant BIT et le manipuler, en local ou à distance, à l'aide d'un navigateur Web comme illustré par la Figure 7. Un manuel utilisateur est disponible en ligne avec un example détaillé à l'URL : *http ://liuppa.univ-pau/themes/aoc/aoc/bitj.php*

5. Conclusion et Perspectives

Nous avons présenté dans cet article un modèle de composition de composant basé sur la théorie de la relation Tout-Partie. Il s'agit d'une approche de composition à fort couplage dont l'originalité porte sur la spécification de contraintes sémantiques au niveau de la modélisation, grâce aux notions de composant Tout et de composant Partie. Nous pensons que cette approche favorise la prédiction du comportement de l'assemblage de composants dès la modélisation de la composition en ce sens qu'elle contraint fortement et sémantiquement la relation de composition. Nous avons également étudié comment les propriétés de la relation Tout-Partie pouvaient être applicables à ce type de composition.

D'autre part, nous proposons aux utilisateurs de composants de valider cette composition à l'aide de l'ajout de fonctionnalités de test dans les composants eux-mêmes. La librairie BIT/J, que nous avons développé et qui permet de mettre en œuvre cette technologie, permet l'utilisation de composants simples ou de composants COTS Java^{TM}. Il est en effet indispensable de tenir compte du fait que, dans la composition, que tous les composants qui participent à l'assemblage n'ont pas été nécessairement développés par l'assembleur (la principale difficulté dans la validation de l'assemblage de COTS étant l'absence du code source du composant).

Nous travaillons actuellement à intégrer ces travaux au sein d'un environnement complet de composition utilisant pour la modélisation un outil UML supportant la définition de métamodèles et l'intégration de modules spécifiques. Nous proposons

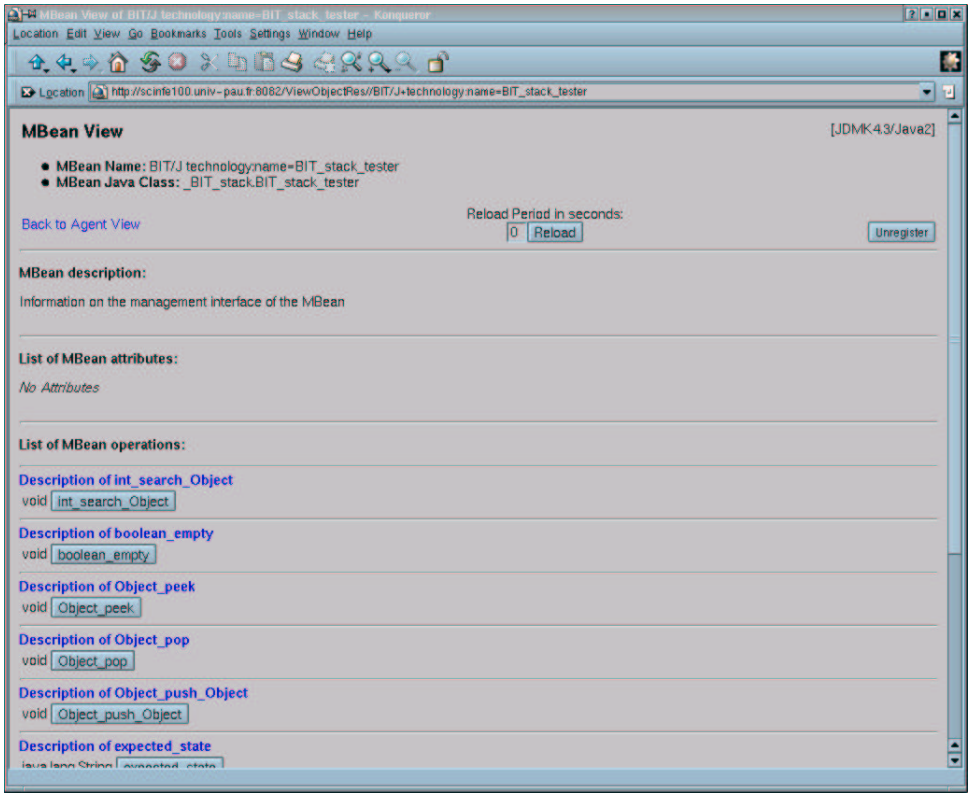

Figure 7. *Interface de test à distance*

d'ajouter un module permettant de modéliser l'assemblage de composants selon notre approche. L'intérêt est double. Premièrement, cela permettra de vérifier le modèle de composition au sein de l'outil. Deuxièmement, nous pensons ajouter un générateur de code implémentant automatiquement les propriétés spécifiées lors de la modélisation et notamment les propriétés intrinsèques à la relation Tout-Partie. Cela viendra en complément de notre librairie permettant de générer automatiquement les contraintes au niveau du développement mais également de générer automatiquement les fonctionnalités de test intégrés ainsi que les cas de test vérifiant ces contraintes. Enfin, nous envisageons d'étendre notre démarche à la composition de composants sur des nœuds de déploiements différents.

6. Bibliographie

[BAR 01] BARBIER F., HENDERSON-SELLERS B., « The Whole-Part Relationship in Object Modelling : A Definition in cOlOr », *Information and Sofware Technology*, vol. 43, n° 1,

2001, p. 19–39.

[BAR 02a] BARBIER F., « Composability for Software Components : an Appoach Based on the Whole-Part Theory », *Proceedings of the 8th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2002), Greenbelt, Maryland*, December 2002.

[BAR 02b] BARBIER F., LACAYRELLE A. L. P., BRUEL J.-M., « Agrégation et Composition dans UML - Révision basée sur la théorie Tout-Partie », *TSI (Techniques et sciences informatiques)*, vol. 21, n° 10, 2002, Hermès.

[BAR 03] BARBIER F., BELLOIR N., BRUEL J.-M., « Incorporation of test functionality into software components », *Proceedings of the 2nd International Conference on Commercial Off-The-Shelf (COTS)-Based Software Systems (ICCBSS'2003)*, Lecture Notes in Computer Science 2580, Ottawa, Canada, February 10-12, 2003, Springer.

[BOS 00] BOSCH J., *Design and Use of Software Architectures - Adopting and Evolving a Product-Line Approach*, Addison-Wesley, mai 2000.

[CRN 02] CRNKOVIC I., SCHMIDT H., STAFFORD J., WALLNAU K., « Anatomy of a Research Project in Predictable Assembly », *Fifth ICSE Workshop on Component-Based Software Engineering - White paper*, , 2002.

[DST 02] DSTC PTY LTD, « UML 2.0 Infrastructure – Revised Submission », , 2002, Document ad/2002-09-07, available at : http ://cgi.omg.org/cgi-bin/doc ?ad/02-09-07.

[GAM 95] GAMMA E., HELM R., JOHNSON R., VLISSIDES J., *Design Patterns : Element of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[HAM 02] HAMEURLAIN N., « Behavioural Subtyping and Property Preservation for Active Objects », *Proceedings of IFIP TC6/WG6.1 Fifth International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS 2002*, kluwer, 2002.

[Hig 01] HIGH CONFIDENCE SOFTWARE AND SYSTEMS COORDINATING GROUP, « High Confidence Software and Systems Research Needs », rapport, january 2001, Interagency Working Group on Information Technology Research and Development.

[JOH 97] JOHNSON R. E., « Frameworks = Components + Patterns », *Communications of the ACM*, vol. 40, 1997, p. 39–42.

[MEY 97] MEYER B., *Object-Oriented Software Construction*, Prentice Hall, Second Edition, 1997.

[OMG 01] OMG, « OMG Unified Modeling Language Specification, verion 1.4 », OMG document, 2001, Object Management Group.

[OMG 02] OMG, « UML Profile for Enterprise Distributed Object Computing Specification », may 2002, http ://www.omg.org/.

[STA 02] STAFFORD J. A., WALLNAU K., « *Building reliable component-based software systems (Ivica Crnkovic and Magnus Larsson editors)* », chapitre Component Composition and Integration, p. 179–191, Artech House Publishers, Boston, 2002.

[SUN 02] SUN, « JavaTM Management Extensions (JMX) 1.2 », 2002, http ://java.sun.com/products/JavaManagement/.

[SZY 97] SZYPERSKI C., *Component Software – Beyond Object-Oriented Programming*, Addison-Wesley, New York, NY, 1997.