Définition d'un mécanisme de composition appliqué aux modèles métiers

Térence FERUT

Stage du Master PLMT, Laboratoire I3S, équipe OCL

Encadrement : Philippe Lahire et Pierre Crescenzo

12 juin 2006

Table des matières

Table des figures

1 Introduction

Les concepts mis en avant par la programmation par objets (encapsulation, polymorphisme, modularité...), ont représenté une avancée majeure pour la construction d'applications complexes. Ils favorisent la réutilisation et permettent ainsi une diminution importante des temps de développement et des délais de maintenance. Il est cependant avéré que les outils fournis par les approches à objets sont loin d'être suffisants pour répondre à tous les besoins en matière de réutilisation.

On peut facilement s'en convaincre pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de fonctionnalités données, mais qui nécessitent également la prise en compte de différentes propriétés non-fonctionnelles telles que la distribution, la journalisation, la sécurité ou la persistance. Dans de telles applications, l'implémentation des fonctionnalités et celle des différentes propriétés non-fonctionnelles se trouvent intimement mélées. *De facto*, la réutilisation se trouve compromise. Il n'est, en effet, pas toujours possible de réutiliser les différents aspects d'une application, indépendamment les uns des autres. Même si on ne considère que des propriétés fonctionnelles, dans des hiérarchies complexes, ces mêmes problèmes de réutilisabilité se posent.

Le besoin de combler les lacunes du paradigme des objets est à l'origine de nombreux travaux de recherche. Il est en particulier particulièrement important d'augmenter les possibilités d'adaptation du logiciel dans le but de le faire évoluer ou de le mettre au point. La réutilisation de classes, bibliothèques, modèles d'application ou encore code fonctionnel s'inscrit dans cette démarche d'évolution.

Dans cette optique, de nombreux paradigmes de programmation et de conception apparaissent comme des approches prometteuses pour combler certaines lacunes des approches par objets. Dans le cadre de notre stage de Master Recherche nous nous sommes intéressés en particulier à la *séparation des préoccupations* [16], et notamment aux paradigmes par aspects et par sujets, à la métamodélisation, et aux approches dirigées par les modèles. Dans l'équipe OCL deux travaux sont actuellement en cours : le premier d'un point de vue chronologique se situe au niveau applicatif et concerne l'adaptation du code des applications. Le second propose une possibilité de définir des modèles contenant des entités génériques. Notre travail est au carrefour de ces deux activités.

Plus précisément, l'approche développée par Laurent Quintian dans sa thèse [28] et améliorée dans [15] s'appuie sur les programmations par sujets et par aspects pour considérer une application comme un ensemble de préoccupations à composer. Un des aspects originaux de ce travail a été d'utiliser une approche dirigée par les modèles pour mettre en oeuvre la composition de ces préoccupations. Ainsi les auteurs ont proposé un métamodèle pour la description d'applications à objets ainsi qu'un concept d'*adaptateur*. Ce dernier sert de support *i)* pour définir un protocole de composition indépendant du contexte d'intégration qui accompagne toute préoccupation réutilisable et, *ii)* pour compléter ce protocole en l'adaptant aux conditions spécifiques de son intégration dans une application particulière. La description de ces adaptateurs repose sur un ensemble de types d'adaptation et elle est réalisée à travers l'utilisation d'un langage dédié (en anglais *Domain-Specific Language - DSL*). Cependant les adaptations proposées ne concernent que le code d'une application et non pas son modèle. Un premier prototype a été développé par Laurent Quintian, il s'agissait de *JAdapt*. Dans le cadre de mon projet de Master 1, l'an dernier, j'ai participé à la réalisation d'une seconde implémentation de ce prototype [5] qui s'appuie sur *EMF* [10] (pour *Eclipse Modelling Framework*), un plugin *Eclipse*[9] permettant de concevoir et manipuler des modèles respectant la norme *EMOF (Essential MOF*, sous-ensemble de la spécification de *MOF*)[23], pour définir le métamodèle de composition appelé *SmartAdapters*.

L'approche développée par Emanuel Tundrea dans le cadre de sa thèse concerne l'expressivité des modèles métiers et plus particulièrement celle des modèles qui décrivent des lignes de produits. Il propose en particulier de s'appuyer sur les travaux menés par Pierre Crescenzo dans sa thèse [7] pour prendre en compte les variations structurelles et comportementales qui peuvent exister entre les entités participant à la description d'un modèle [25]. Par exemple si on considère une

ligne de produits d'appareils photographiques, il faut pouvoir décliner les points communs et les différences des différentes sortes d'appareils. Cela revient à considérer un appareil comme une entité générique. Son degré de généricité est défini à un niveau méta à l'aide d'un ensemble d'informations spécifiques qui capturent la variation. Ces variations peuvent, si on reprend l'exemple de l'appareil photo, concerner le type de support (pellicule ou mémoire FLASH), le type d'appareil (numérique ou argentique) ou encore la documentation (en ligne, ou sur papier). Une première validation de ce travail consiste à proposer une extension (appelée *SmartModels*) du modèle *EMF* pour la définition d'entités génériques. Ce métamodèle enrichit *EMF* et l'éditeur produit à partir de ce dernier (qui sert pour l'instant de langage dédié) a été étendu pour prendre en compte la généricité.

L'objectif qui m'a été fixé est de transposer les idées relatives à la composition du code des préoccupations issues des travaux de Laurent Quintian à la composition de modèles décrits à l'aide *SmartModels*, en d'autres termes, il s'agit d'adapter le procédé d'adaptation des modèles analogue au procédé d'adaptation des applications.

2 Composition de modèles : présentation informelle

Afin de mieux appréhender ce qu'est la composition de modèles et ses applications, prenons un cas d'utilisation. Considérons une ligne de produits d'appareils photographiques et une modélisation objets de cette ligne de produits s'appuyant sur la *séparation des préoccupations*[16] et plus précisément sur le paradigme des *sujets*. Cette technique permet d'obtenir une modélisation décomposant cette ligne de produits en *sujets* (voir section **??** pour plus de détails). Un *sujet* est un sous-ensemble des entités d'un modèle ne contenant que les éléments définissant un thème particulier, et pouvant servir de « brique » dans l'élaboration d'autres sujets.

Dans notre exemple un des sujets pourrait modéliser le côté commercial, c'est-à-dire tout ce qui englobe les données relatives au commerce des appareils photos : la gamme de prix, les accessoires livrés avec, et les options.

Un autre sujet pourrait représenter le côté technique avec la spécification de chaque appareil, l'objectif, le type de support, la vitesse d'obturation, la puissance et le temps de rechargement du flash etc.

Un troisième sujet pourrait décrire la documentation, à la fois pour les techniciens et pour les utilisateurs.

Chacun de ces sujets est représenté par un modèle différent. Le site de vente de la marque, spécialisé dans la vente des appareils photographiques, a besoin à la fois du sujet commercial pour les prix et les offres de vente, et du sujet technique pour présenter aux visiteurs les spécificités de chaque appareil. Le modèle décrivant les besoins de ce site de vente, formant le sujet « vente en ligne », consiste en une composition des deux sujets sus-cités donnant naissance à un nouveau sujet.

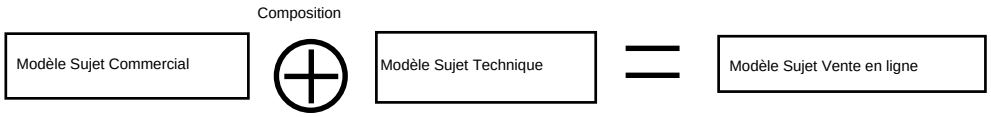

FIG. 1 – Composition de modèles sujets.

La figure 1 montre la composition entre les deux modèles. Plus précisément, il faut que le nouveau modèle « vente en ligne » contienne les éléments des constituants des deux modèles sources, commercial et technique. On appellera ce genre de composition une fusion de modèles et la figure 2 en montre un premier aspect.

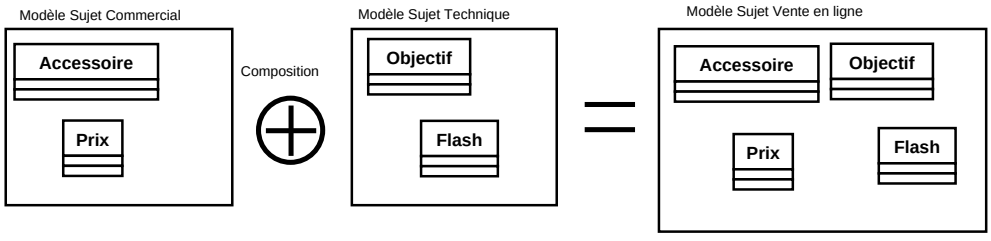

FIG. 2 – Exemple simple de fusion de modèles.

Les classes des modèles *Sujet Commercial* et *Sujet Technique* sont introduites dans un nouveau modèle *Sujet Vente en ligne*, contenant l'union des classes que contenaient les Sujets *Commercial* et *Technique* avant la composition. Ce cas, même s'il ne présente aucune difficulté, permet de donner un premier aperçu de la composition de modèles.

Mais que se passerait-il si ces deux modèles contenaient tous les deux une classe *AppareilPhoto* ? Ces classes pourraient contenir dans les deux sujets des informations différentes, dédiées à leurs

objectifs respectifs. Doit-on choisir une seule de ces deux classes au détriment des informations contenues dans l'autre ? et si oui laquelle ? Ou doit-on aussi essayer d'intégrer ces deux classes dans une nouvelle classe, en fusionnant leur contenu ?

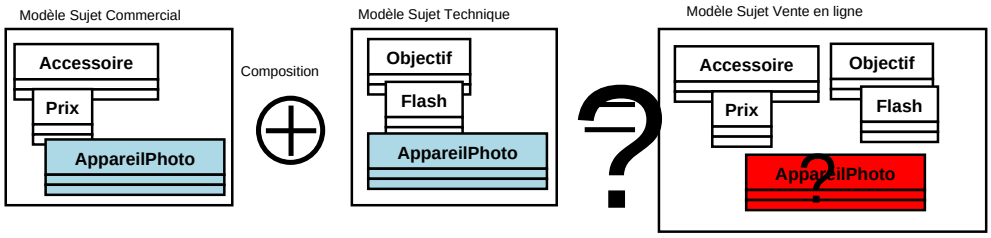

FIG. 3 – Exemple de composition ambigüe.

Cette dernière solution semble *a priori* intéressante mais que fait-on si les deux classes contiennent des méthodes dont la signature est identique ou des attributs de même nom (par exemple si les deux classes *AppareilPhoto* contiennent un attribut *reference*) ?

Inversement, supposons que les modèles qui correspondent aux sujets *Commercial* et *Technique* aient été conçus sans convention commune de nommage. On peut imaginer que les deux classes décrivant un appareil photographique dans ces deux modèles ne portent pas le même nom. Si on n'établit pas une correspondance entre ces deux classes au moment de la fusion, elles seraient toutes les deux introduites dans le nouveau modèle. Il est donc fort probable qu'un utilisateur souhaite pouvoir indiquer au moment de la composition que même si le nom des deux classes n'est pas identique, il s'agit bien de la même entité et donc que ces classes doivent être fusionnées. C'est ce cas qui est illustré dans la figure 4.

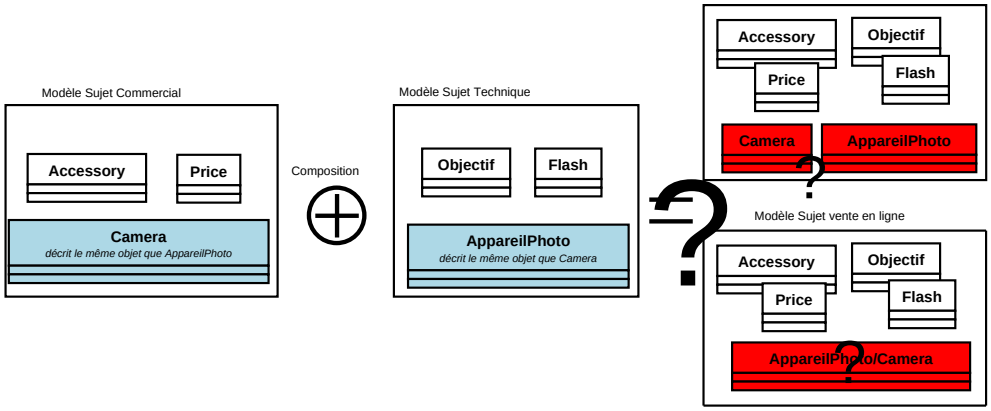

FIG. 4 – Classes correspondantes dans une composition.

Dans la suite nous nous appuierons sur cet exemple que nous continuerons à développer pour illustrer notre approche et ainsi de mettre en évidence les problèmes à résoudre et les solutions que nous offrons relativement à la composition de modèles.

3 La composition de modèles métiers : un état de l'art

Le succès rencontré par les approches dirigées par les modèles a favorisé l'émergence de nombreuses propositions et outils permettant de guider des compositions, ou transformations de modèles. Ce chapitre fait un état de l'art des différentes approches qui offrent un support à ces transformations ou compositions.

Ces approches se présentent sous la forme de métamodèles ou d'outils, et proposent dans certains cas une implémentation sous forme de *plugin* Eclipse[9]. Les adaptations de programme présentent souvent des similitudes avec des adaptations de modèles, nous avons donc souhaité inséré ici certains outils plutôt dédiés aux transformations de programmes, mais dont le principe nous semblait intéressant au niveau des modèles. Nous avons également inclus dans cet état de l'art des concepts qui, même s'ils ne sont pas directement liés à la composition de modèles, peuvent contribuer à l'enrichir.

La section 3.1 propose un panorama des différents métamodèles modélisant les préoccupations et les opérateurs de composition, dans les approches étudiées. La section 3.2 présente des modèles de composition. Enfin, la section 3.3 précise les langages utilisés par ces approches pour expliciter la composition.

3.1 Expressivité de la modélisation des préoccupations

Les différentes approches en matière de métamodélisation des préoccupations restent dans l'ensemble toujours assez proches les unes des autres.

Concernant la modélisation des éléments classiques d'un modèle, *EMF* occupe une place intéressante. Fourni sous forme d'un plugin pour la plateforme Eclipse, *EMF* met à disposition des développeurs tous les outils nécessaires pour la création de modèles ou la manipulation de ses éléments. *EMF* est comme on l'a dit précédemment, conforme à *EMOF* [23] et on y retrouve les principaux éléments nécessaires à la description des modèles (*paquetage*, *classifieur*, *opération*, *propriété*, *association*...). *EMF* ne prend pas en compte l'aspect comportemental, c'est à dire le contenu des opérations. D'autres approches proposent d'utiliser des diagrammes *UML*[20], permettant ainsi de décrire ce comportement et donc de pallier à cet inconvénient.

Dans [6], l'auteur propose une modélisation basée sur *UML* où la notion de paquetage est associé au stéréotype *Subject*. La différence entre un sujet et un paquetage se situe au niveau des éléments que ces deux entités peuvent regrouper. D'après la spécification *UML*, un paquetage peut contenir ou référencer d'autres paquetages (*Package*), des classifieurs (*Classifier*), des associations (*Association*), des généralisations (*Generalization*), des dépendances (*Dependancy*), des contraintes (*Constraint*), des collaborations (*Collaboration*), des états machines (*StateMachine*) et des stéréotypes (*Stereotype*). Dans cette approche un sujet ne contiendra (ou ne référencera) que d'autre sujets, classifieurs, associations, généralisations, dépendances, contraintes et collaborations. Un sujet est donc présenté comme un paquetage avec des restrictions. Cette aproche permet uniquement de manipuler des entités composables entre elles.

Dans [28] la modélisation d'un langage objet source est proposée. Elle réifie de manière très classique les constituants d'un langage à objets à savoir les classes, paquetages, attributs, signatures d'attribut, méthodes, signatures et corps de méthode, instructions et modificateurs de visibilité. Ce métamodèle est enrichi dans [15], puis implémenté dans *SmartAdapters* [5], où de nouvelles entités jusqu'alors absentes sont ajoutées comme par exemple les paramètres de méthode et le type d'un attribut ou de retour de méthode. Bien que limité à des langages, ce métamodèle pourrait être étendu à la description de modèles.

KerMeta, un langage de définition, de description et de manipulation de modèles, se base sur *EMF*, auquel il ajoute une couche supplémentaire permettant la description de l'aspect comportemental. Ces informations sont considérées par *EMF* comme de simples annotations, assurant ainsi une compatibilité bidirectionnelle entre *EMF* et *KerMeta*.

Dans [4], les auteurs proposent trois métamodèles sous forme d'une extension d'*UML*. Un premier métamodèle général, permet de modéliser des préoccupations. Un second métamodèle, tenant

compte des spécificités *d'AspectJ*[2], telles que les perfectionnements (*advices*) et les points de recouvrement (*pointcuts*) est aussi proposé. Ce métamodèle est accompagné d'un ensemble de règles de transformation permettant de transformer un modèle, instance du métamodèle général, en une instance du métamodèle spécifique à *AspectJ*. Le même travail est proposé pour *Hyper/J*[13], avec un métamodèle prenant en compte les spécificités d'*Hyper/J* telles que les *Hyperspace*, les *Hypermodules* et les *Dimensions*. De même que précédemment, un ensemble de régles permet de passer d'une instance du métamodèle général à une instance du métamodèle d'*Hyper/J*.

On peut aussi citer *SmartModels* [25], le métamodèle qui fait l'objet de la thèse d'Emanuel Tundrea. Il concerne l'expressivité des modèles métiers et en particulier des modèles décrivant les lignes de produits. En s'appuyant sur les travaux menés antérieurement par Pierre Crescenzo dans sa thèse[7], il propose la prise en compte de variations structurelles et comportementales entre entités participant à la description d'un modèle en introduisant la notion de paramètres métiers. Ces entités deviennent des entités génériques, modélisées sous forme « d'atomes », et les variations entre ces entités sont exprimées à un niveau méta, niveau modélisé sous le nom de « concept » dans le métamodèle.

3.2 Expressivité de la composition

Nous allons aborder dans les points suivants les principaux aspects concernant les modèles de composition existants. Dans la section 3.2.1, nous aborderons la composition structurelle et comportementale, c'est-à-dire la modélisation des opérateurs de composition portant aussi bien sur la structure des entités que sur leur comportement. Nous nous intéresserons en particulier aux problèmes que certains de ces opérateurs posent, notamment par rapport à la fusion d'assertions. Dans la section 3.2.2, nous étudierons l'apport des approches actuelles en terme de réutilisabilité de composition, c'est à dire leur degré d'abstraction par rapport au contexte d'intégration.

3.2.1 Composition structurelle et comportementale

Dans le domaine de la composition, plusieurs approches existent ; elles se différencient notamment au niveau du type des opérateurs de composition proposés. Les adaptations sur lesquelles s'appuit la composition portent sur les constituants des modèles tels que les classes, les méthodes, les attributs ou les associations. Certaines adaptations sont comparables au niveau des modèles et au niveau applicatif : il s'agit d'adaptations concernant les éléments communs entre ces deux niveaux, principalement les classes, attributs, méthodes ou paquetages. Les adaptations portant sur les autres éléments spécifiques aux modèles n'ont évidemment aucun équivalent au niveau applicatif. Les deux principales approches existantes sont issues de la *séparation des préoccupations*[16] et concernent les paradigmes des *aspects* et des *sujets*.

Le paradigme des *aspects* est un modèle de séparation des préoccupations basé sur le paradigme objet. Il résulte de l'absence de séparation des préoccupations d'une application. Ce modèle formalise notamment le tissage des préoccupations plutôt mal pris en compte par le paradigme objet. Il permet d'implémenter des préoccupations qui s'entrelacent avec le reste du système d'une manière modulaire. Ce paradigme permet à la fois de bénéficier des avantages de la modularité (code plus simple et plus facile à mettre au point et faire évoluer). Son apport en termes d'opérateurs au niveau de la composition de modèles est très intéressant puisqu'il permet de décrire l'interception d'appels de méthodes (et de levées d'exception) et l'introduction d'éléments au sein d'élements-conteneurs. Une interception capture un appel de méthode et ajoute un traitement spécifique *avant*, *après*, ou *autour* de son exécution. Une interception permet aussi de réaliser un traitement spécifique si une exception est déclenchée lors de l'exécution d'une méthode. Par rapport aux opérateurs d'introduction, il est possible d'ajouter par exemple une classe dans un paquetage et une méthode ou un attribut dans une classe. Parmi les langages de programmation par aspects on peut notamment citer *AspectJ*[2] ou *JAC* [1].

Le paradigme des *sujets* est lui aussi une extension du paradigme objet. Un sujet est un fragment de programme (découpage vertical) qui peut être exprimé en utilisant le paradigme objet. La

composition d'un ensemble de sujets permet de produire l'application finale ; la composition porte ici le nom d'intégration. Les sujets peuvent être comparés à des composants et représenter soit des composants primitifs, soit des composites. Le procédé de composition est tout à fait comparable à celui des aspects, et on y retrouve ses principaux avantages mentionnés ci-dessus. Il s'articule essentiellement autour de deux familles d'opérateurs d'intégration : les fusions d'éléments et les redéfinitions[1]. Les fusions enrichissent la cible de l'adaptation. Lors d'une fusion de classes, par exemple, la classe cible se verra enrichie des constituants de la classe source tandis qu'une fusion de méthodes correspondra à une succession (dans un ordre spécifique) de l'exécution des corps des deux méthodes ou le plus souvent à un entrelacement des instructions qui les composent.

Dans [28], l'auteur propose un ensemble d'adaptations valides au niveau applicatif, regroupant les adaptations issues des paradigmes par aspects et par sujets. Ces adaptations peuvent être classées en trois grandes familles : les interceptions, les introductions (et redéfinitions) et les fusions. Cet ensemble d'adaptations peut être *invasif* (ou encore *in-situ*, la composition modifie l'entité cible de l'adaptation) ou non (*ex-situ*). Dans ce dernier cas, une nouvelle entité est créée, sans effet de bord sur les cibles. Par la suite, cet ensemble d'adaptations a été enrichi dans [15], puis implémenté dans *SmartAdapters* [5].

L'ensemble des opérateurs de composition proposé varie sensiblement d'une approche à une autre en fonction des besoins. Par exemple, dans [8] nous retrouvons les opérateurs d'introduction et de fusion similaires à ceux du métamodèle proposé dans [15] mais adaptés aux modèles, alors que la famille des interceptions disparaît. En revanche, une nouvelle famille est ajoutée, celle des opérateurs de modification (modifications de types, de noms, de visibilité).

Dans [6], l'auteur propose une approche dérivant directement du paradigme par *sujets*. Ses compositions structurelles correspondent donc aux *intégrations* des *sujets* et dans le métamodèle proposé, les deux seules familles d'intégration sont les redéfinitions et les fusions. Le métamodèle a été revu de sorte que toute entité contenue dans un *Subject* soit à la fois rédéfinissable et fusionnable avec d'autres entités du même type. Le métamodèle est donc très homogène par rapport à la composition.

Une approche de composition structurelle dédiée à des architectures de composants est proposée dans[3]. Ce canevas de conception est construit comme un système de tissages d'*aspects* au niveau d'une description d'architecture logicielle. Les *aspects* sont appelés *plans*, et prennent en charge une préoccupation. L'architecture logicielle impactée par les modifications décrites dans ces *plans* est appelée *plan de base*. Définir une composition revient à préciser les *points de jonction* ou la préoccupation modifie le *plan de base* pour pouvoir s'intégrer.

Dans [29], une approche très différente est proposée. A partir du code source d'un programme écrit en Java, un arbre de syntaxe abstraite (*AST*) est généré, servant à la construction d'une *base de faits* Prolog qui correspond aux entités du programme. A partir d'un ensemble de règles d'inférence, l'utilisateur peut rapidement exprimer des requêtes et des transformations sur cette base de faits. Les transformations sont dites conditionnelles car leur réalisation dépend d'une précondition. Ces préconditions sont exprimées à l'aide d'opérateurs logiques simples (*and*, *or* et *not*). Lorsqu'une transformation est réalisée, elle met automatiquement à jour la base de faits et le programme peut être resynchronisé pour correspondre à nouveau à la base de faits.

Au niveau du comportement des modèles, principalement deux types d'approches existent : l'approche déclarative, qui consiste à contraindre certains éléments, et l'approche procédurale, qui décrit le comportement à l'aide d'un pseudo-langage.

OCL[21], pour *Object Constraint Language*, est un langage de description de contraintes pour *UML*[20]. Il s'agit d'une approche déclarative permettant d'exprimer une partie du comportement, par exemple au niveau des assertions et des invariants de classes. La composition de contraintes *OCL* constitue quant à elle un problème ayant fait l'objet de peu de travaux encore actuellement. Cette composition est assez délicate car son résultat dépend fortement de la sémantique de ces contraintes.

[1] Les fusions et les redéfinitions sont les adaptations de base dans des langages de programmation par *sujets* tels que *Hyper/J*[13].

Dans [31], l'auteur propose un métamodèle basé sur *UML 2.0* permettant de générer des machines à états à partir de diagramme de séquences. *UML 2.0* propose plusieurs opérateurs d'intéraction permettant de composer des instances de diagrammes de séquences. Ces opérateurs de composition sont la *séquence*, la *séquence forte*, l'*alternative*, l'*itération*, la *composition parallèle* et l'*option*. Dans cette approche, un métamodèle permettant le support de la variabilité dans les lignes de produits en UML est également proposé, avec modélisation de contraintes OCL sur ces lignes de produits.

KerMeta[27] est probablement le meilleur exemple d'un langage de type impératif procédural permettant la description comportementale. Comme déjà indiqué précédemment, ce dernier se base sur la métamodélisation d'*EMF* pour l'enrichir avec des annotations d'*EMF* décrivant le comportement. Le métamodèle de *KerMeta* a été conçu pour permettre donc, avec une certaine souplesse, d'effectuer de la composition comportementale, en modélisant les principaux composants comportementaux tels que les structures de controle (itérations, blocs, ou expressions), les variables, les appels, les affectations, ou les lambda-expressions.

3.2.2 Capacité pour la réutilisation

La réutilisation est un point important dans la plupart des domaines, y compris dans la composition de modèles.

La première façon d'améliorer la réutilisation se situe au niveau de l'héritage. L'entité modélisant la composition peut hériter d'une autre, en la spécialisant d'avantage ou simplement en la concrétisant. C'est ce que propose *JAdapt* [28] et *AspectJ* [2]. D'autre part, il est important de pouvoir, en faisant abstraction de certains éléments, concevoir des compositions qui seront concrétisées *a posteriori*. Cela permet d'obtenir des composition génériques et indépendantes de leur contexte d'intégration, et par conséquent plus facilement réutilisables. Cette généricité est d'autant plus importante que les cibles effectives d'une composition ne sont souvent pas connues lors de la définition de la composition.

Dans *AspectJ* [2], il est possible de créer des modules abstraits représentant des préoccupations. Les informations rattachant la préoccupation à son contexte ne sont spécifiées que lors d'une concrétisation de ces modules. Les aspects abstraits sont ainsi rendus plus génériques. Malheureusement, concrétiser ces aspects n'est pas toujours une opération simple, et aucun protocole de composition n'est proposé pour guider cette concrétisation, à l'inverse de [28] par exemple.

Dans [18], les auteurs proposent un schéma de mise en oeuvre d'aspects fonctionels, ou vues, réutilisables par adaptation. Ce schéma s'appuie sur un métamodèle de composants-vues, défini lors de travaux antérieurs[17, 19], qui permet de décrire une structuration complexe d'application mettant en oeuvre de nombreux aspects, fonctionnels ou non. Ce schéma s'appuie sur différents patrons de conception, permettant de concevoir des vues sous la forme de représentations éclatées, et rendant compte de la structuration en aspects. Un patron adaptateur (qui peut être abstrait), est également introduit permettant de découpler les fragments de base et les fragments de vues. C'est cet adaptateur qui assure la réutilisation des aspects.

Dans *JAdapt* [28], un protocole de composition, permet de guider la composition et améliore *de facto* la réutilisabilité des compositions. Un protocole de composition est une abstraction de la composition d'une préoccupation. Les adaptations, formant la composition, sont encapsulées dans des entités appelées *adaptateur*. Pour pouvoir s'adapter aux différents contextes de réutilisation il est nécessaire de fournir une relation d'héritage entre adaptateurs afin de pouvoir raffiner progressivement la composition un peu comme dans *AspectJ* [12], ce qui permet de finir de concrétiser ces adaptateurs *a posteriori*. Un adaptateur abstrait peut donc être très générique puisqu'indépendant du contexte dans lequel on l'utilise. Un adaptateur, en plus des adaptations, contient aussi les variables permettant d'identifier les cibles des adaptations.

Un autre moyen d'accroître la réutilisabilité de la composition réside dans la possibilité de créer des correspondances entre les éléments de la source et de la cible de la composition. En effet, comme cela a déjà été évoqué dans la section 2, les modèles à composer peuvent avoir été conçus par

des personnes n'utilisant pas les mêmes conventions de nommage, ou plus généralement certaines compositions peuvent exiger la mise en correspondance d'éléments qu'*a priori* rien ne relie les uns aux autres. Plus il sera possible de retarder la concrétisation de cette correspondance, meilleure sera de fait la réutilisabilité. Ici encore, plusieurs mécanismes existent, tous relativement proches dans leur utilisation.

Dans [11], les auteurs proposent un moyen d'exprimer des *aspects* génériques, basés sur un mécanisme de méta-variables logiques. Une méta-variable est définie pour certains constituants d'un modèle et représente l'ensemble des instanciations valides possibles sur ce type. Ces méta-variables sont utilisées par des prédicats séquencés par des opérateurs logique, créant une correspondance entre les éléments pour chaque instanciation valide. Ces prédicats permettent d'exprimer des aspects très complexes. Une méta-variable logique est une méta-variable dont la valeur ne change pas à l'intérieur de sa portée. Dans beaucoup de langages à objets, on retrouve le concept de variable logique sous forme de *template* de classes. Les *aspects* n'utilisant que des méta-variables logiques sont de fait entièrement découplés de leur contexte et deviennent très génériques.

Par exemple, dans *KerMeta*[27] et *ATL*[26], deux langages permettant de décrire des transformations de modèles, ces mêmes transformations sont exprimées justement à l'aide de correspondances. Un modèle intermédiaire, conforme au *MOF*[23], décrit la transformation. Il établit une correspondance entre les éléments du modèle source et les éléments du modèle cible ; en d'autres termes il explique comment passer du modèle source au modèle cible. Ce modèle intermédiaire est décrit en utilisant le même le langage et peut donc être manipulé comme n'importe quel autre modèle, ce qui permettrait de décrire une transformation ayant pour cible une transformation de modèles[14].

Dans [6], le mécanisme de correspondance a une meilleure expressivité puisqu'il propose deux types de correspondances : les correspondances explicites et les correspondances implicites. Il est ainsi possible de préciser qu'un élément x correspond à un élément y, x et y devant être de même type. Par exemple pour indiquer qu'une classe A correpond avec une classe B on écrirait : « *class A corresponds with class B* ». Les correspondances implicites, quant à elles, reposent sur des schémas de correspondances prédéfinis. Un de ces schémas peut être l'utilisation d'une correspondance portant sur le type d'un attribut et son nom, ainsi tout couple d'attributs ayant les mêmes nom et type dans les modèles source et cible seront mis en correspondance. Parmi les autres schémas de correspondance prédéfinis, on peut noter des correspondances uniquement sur le nom ou sur le type de retour d'une méthode.

Dans SPOON[24], qui par rétroingénierie transforme des programmes *Java 1.5* spécialement annotés, aucune correspondance, ni protocole de composition ne sont au proposés. Le type de composition proposé est donc fortement lié aux programmes pour lesquels ces compositions sont écrites. Découpler les compositions de leur contexte pour améliorer leur réutilisabilité reste possible mais au prix d'un effort de structuration important.

3.3 Langage de composition

Le langage de composition est un élément essentiel du mécanisme de composition. C'est à travers lui qu'un utilisateur peut décrire une composition. Il doit donc être clair, simple, efficace et suffisamment expressif.

Ce langage peut être généraliste, comme *KerMeta*[27], un langage de description de modèles exécutables, qui n'est pas exlusivement dédié à la transformation de modèles. Il peut aussi être un langage évolué spécifique à la transformation ou la composition comme *ATL*[26], qui est un langage hybride, en partie déclaratif, et en partie impératif.

Lorsqu'il s'agit d'un langage exlusivement dédié à la composition de modèles métiers, on parle alors de *langage dédié* ou *DSL*. Ce langage se veut souvent extrèmement simple, réduit à la taille minimale, et adapté de par son expressivité à la description des compositions. C'est ce qui est proposé à la fois dans [28] et dans [8].

Dans [8], les auteurs proposent un langage dédié servant à illustrer les exemples de compositions présentées dans l'article. Il est dérivé de celui proposé dans [15] qui permet de décrire l'ensemble

des adaptations (voir section 3.2.1 pour plus de détails sur cet ensemble d'adaptations).

AspectJ[2] est une extension syntaxique pour Java pour la description d'*aspects* ; c'est le langage orienté aspects le plus connu. Il permet de décrire et de définir des *aspects* et des *points de jointures* reliant ces aspects au code fonctionnel de l'application. La façon dont les *aspects* seront tissés peut également être spécifié.

Dans [11], le moyen d'exprimer les aspects est proche de celui d'*AspectJ*. Les mêmes points de jointures et adaptations que celles proposées par *AspectJ* peuvent être décrits à l'aide d'une suite de prédicats contenant méta-variables ou méta-variables logiques, ce qui accroît leur expressivité.

Hyper/J[13] est aussi une extension de Java. Ce langage permet d'identifier des préoccupations, de spécifier des modules et synthétiser des systèmes ou des composants par intégration de ces modules. Les opérations ont lieu directement sur les classes compilées et non sur le code source. De nouvelles classes sont produites après l'intégration. Toutes les relations de composition sont décrites via un fichier de contrôle qui spécifie comment l'intégration doit avoir lieu.

Le langage de composition peut aussi s'intégrer au sein d'un langage qui n'est pas dédié à la composition, comme le fait *SPOON*[24], qui utilise le mécanisme des annotations de *Java 1.5*. Les transformations sont directement explicitées à l'aide de ces annotations qui font référence à des classes conçues par le développeur.

Dans [3], le langage de composition proposé est un langage simple, basé sur un ensemble minimal de primitives. Il permet de spécifier les modifications à apporter au *plan de base* au moment du tissage d'un nouveau *plan*.

On peut aussi mentionner *ModelScripting*[30] qui est lui aussi un langage très simple. Il représente une extension de *Javascript* dédiée à la manipulation des modèles *EMF*[10]. Il est aussi possible de décrire des transformations mais comme il n'est pas conçu pour cela, la difficulté d'expression de ces transformations évolue exponentiellement avec la complexité de la transformation.

Le langage de composition peut aussi être graphique, comme c'est le cas avec *UML*[22] *et* les approches basées sur *UML*[6]. *UML* permet à l'aide d'un ensemble de diagrammes de décrire un modèle et son comportement. Depuis *UML 2.0*, nous l'avons déjà précisé dans la section 3.2.1, les diagrammes de séquence peuvent être instanciés, et des opérateurs simples de composition sur ces diagrammes sont directement inclus.

4 Travail réalisé

Dans le but de pouvoir réaliser diverses compositions de modèles, le travail à effectuer s'articulait autour de différents axes. En tenant compte des travaux réalisés dans le domaine, nous avons défini un métamodèle pour la description des préoccupations, c'est-à-dire tout ce sur quoi portera les compositions. Ce métamodèle est présenté dans la section 4.1 tandis que le mécanisme de composition associé est décrit dans la section 4.2. La composition repose sur un ensemble d'opérateurs de composition, définissant l'ensemble des adaptations réalisables au niveau des préoccupations. Ces opérateurs sont présentés dans la section 4.3. La section 4.4 présente un aperçu du langage utilisé pour exprimer la composition. Enfin la section 4.5 abordera l'intégration de la généricité d'entités au sein d'un modèle, et son impact tant au niveau du métamodèle des préoccupations qu'au niveau des opérateurs de composition.

4.1 Le métamodèle des préoccupations

Dans [28], l'auteur proposait une métamodélisation d'un langage à objets servant de base à l'expression des préoccupations à un niveau applicatif. Cette métamodélisation, qui a été améliorée dans [5], ainsi que celle présentée dans [6], nous ont servi de base pour définir notre propre métamodèle, adapté aux modèles. Nous nous sommes aussi appuyés sur les spécifications d'*UML* et de *MOF*[23] pour définir notre propre métamodèle.

D'un point de vue purement structurel, les entités ajoutées au métamodèle de [5] sont les associations de classes (*Association*) et les terminaisons d'associations (*AssociationEnds*), c'est-à-dire les classes mises en relation par l'association, et leur multiplicité. La spécification de ces associations est conforme à celle d'*UML* 1.1[22], mais simplifiée au niveau des attributs de l'association (voir diagramme de classe en Annexe). D'après cette même spécification, l'unicité d'une association se répercute à tout l'espace de nommage d'un modèle. Nous avons donc choisi que toutes les associations appartiennent au paquetage racine du modèle pour lequel elles sont définies.

Au niveau des modèles nous perdons de l'information comportementale en comparaison avec le niveau applicatif. Au niveau des applications, le comportement est naturellement décrit à l'aide d'instructions spécifiques au langage. Concernant la description du comportement au niveau des modèles, celui-ci est décrit à l'aide de diagrammes et d'OCL [21] dans *UML*. Dans *KerMeta* le comportement est décrit à l'aide d'un langage proche de Java. Pour la description du comportement notre métamodèle contient des entités représentant les préconditions (*Precondition*) et les postconditions (*Postcondition*) de méthodes dans la classe *Method*, ainsi que des entités représentant les invariants de classe (*Invariant*) dans la classe *Classifier*. Ces assertions sont exprimées avec *OCL*. Une autre information comportementale est introduite par l'entité réifiant un appel de méthode (*MethodCall*) car pour les besoins de certains opérateurs de composition nous avions besoin de pouvoir manipuler les appels de méthodes. Au niveau de la classe *Method*, trois listes de *MethodCall* ont également été introduites (voir section 4.3) afin de pouvoir décrire un comportement réalisé *avant* ou *après* l'exécution du corps de la méthode, ou réalisé en cas de levée d'exception par la méthode. Pour le reste, il s'agit d'une modélisation très classique qui est présenté en annexe.

4.2 Le métamodèle de composition

Le métamodèle de composition que nous proposons est lui aussi une amélioration de [28] et de [8]. C'est dans [28] que la notion d'adaptateur (*Adapter*), clé de voûte du mécanisme de composition, et d'adaptation (*Adaptation*) a été introduite pour la première fois. Une composition est réalisée en éxécutant séquentiellement toutes les adaptations définies dans l'adaptateur, dans l'ordre de leur déclaration. Dans les sections suivantes nous ne détaillerons que les entités les plus importantes de ce métamodèle ; une description plus détaillée est disponible en annexe.

4.2.1 L'adaptateur

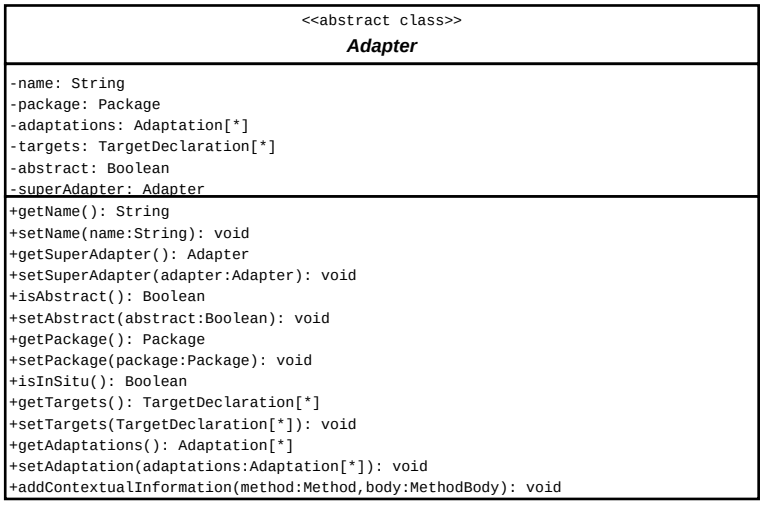

FIG. 5 – Diagramme de la classe Adapter

Un adaptateur peut être vu à la fois comme un conteneur et comme un *protocole de composition*. Il est donc possible d'expliciter une composition à l'aide d'un adapteur et de ses constituants.

Un adaptateur contient des adaptations (*Adaptation*), des déclarations de cibles d'adaptation (*TargetDeclaration*) et des définitions de schémas de correspondances (*Binding*). Il est identifié par un nom, unique dans le paquetage (*Package*) qui le contient.

Dans le but d'accroître la réulisabilité de nos compositions nous proposons deux types d'adaptateurs : les adaptateurs concrets (*ConcreteAdapter*) et les adaptateurs abstraits (*AbstractAdapter*). Toute la partie de la composition qui est indépendante du contexte peut être définie dans un adaptateur abstrait, et le contexte ne sera spécifié que lors de la concrétisation de cet adaptateur. La définition d'adaptateurs abstraits permet ainsi de guider la composition, en définissant les adaptations incomplètes (donc abstraites aussi) ainsi que des cibles ou des correspondances à compléter en fonction du contexte dans lequel on utilise la composition.

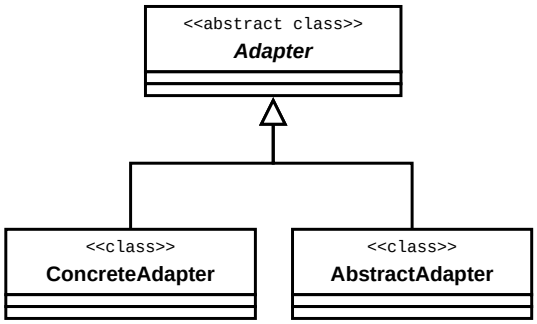

FIG. 6 – Adaptateurs abstraits et concrets

Un adaptateur abstrait peut contenir des adaptations concrètes ou abstraites, elles aussi. Si elles sont déclarées abstraites, elles devront être concrétisées dans un sous-adaptateur qui raffine l'adaptateur abstrait contenant l'adaptation abstraite.

Un adaptateur abstrait peut aussi contenir la déclaration de cibles d'adaptations concrètes (*ConcreteTarget*) ou abstraites (*AbstractTarget*), qui devront être elles aussi concrétisées lors de la concrétisation de l'adaptateur. Les cibles abstraites peuvent être contraintes par des déclarations

de type *TargetConstraint*. Ces contraintes permettent de guider la composition et de faciliter sa réutilisation.

Enfin un adaptateur peut aussi contenir des schémas de correspondances (*Binding*). Ces schémas permettent de faire correspondre des éléments de modèle source avec des éléments du même type dans le modèle cible. Le mécanisme de correspondance est donné en section 4.4. Une correspondance dépend du modèle source et du modèle cible ; en d'autre termes elle est fortement dépendante du contexte. Nous avons décidé de permettre à ces correspondances d'être abstraites et ainsi d'autoriser celui qui décrit le protocole de composition de ne spécifier qu'une partie de la correspondance. Cette partie peut, comme pour les déclarations de cibles d'adaptation, être contrainte (*BindingConstraint*). Une concrétisation de la correspondance sera alors nécessaire pour la compléter en fonction du contexte.

Un adaptateur peut être paramétré de façon à exécuter ses adaptations de façon intrusive *(in-situ)* ou non *(ex-situ)*. S'il est intrusif, ses adaptations modifieront leurs cibles, s'il est *ex-situ*, toutes ses adaptations seront réalisées dans un nouveau modèle, dont le nom sera spécifié.

Nous avons également accès aux informations contextuelles concernant les appels de méthodes. Ces informations sont nécessaires dans l'optique d'une évolution vers un modèle exécutable, afin de conserver les informations sur le contexte.

Héritage L'héritage entre adaptateurs est une spécialisation. Il est donc possible de créer une hiérarchie entre les adaptateurs permettant de spécialiser progressivement les compositions. Cet héritage est simple et un adaptateur peut donc avoir un seul adaptateur parent. Un adaptateur concret peut bien entendu hériter d'un adaptateur abstrait, adaptateur dont il devra concrétiser les cibles, correspondances et adaptations abstraites. Seuls les adaptateurs concrets peuvent être exécutables.

Redéfinition et concrétisation Les cibles ou les adaptations peuvent être redéfinies dans les adaptateurs d'une même hiérarchie. Les adaptations, correspondances et cibles concrètes peuvent aussi être redéfinies, dans ce cas il est important de noter que lors de l'exécution d'un adaptateur (*ie* réalisation de ses adaptations) si une entité a été redéfini on retiendra toujours la redéfinition se trouvant la plus basse dans la hiérarchie pour l'exécution (les définitions antérieures seront ignorées).

4.2.2 Les adaptations

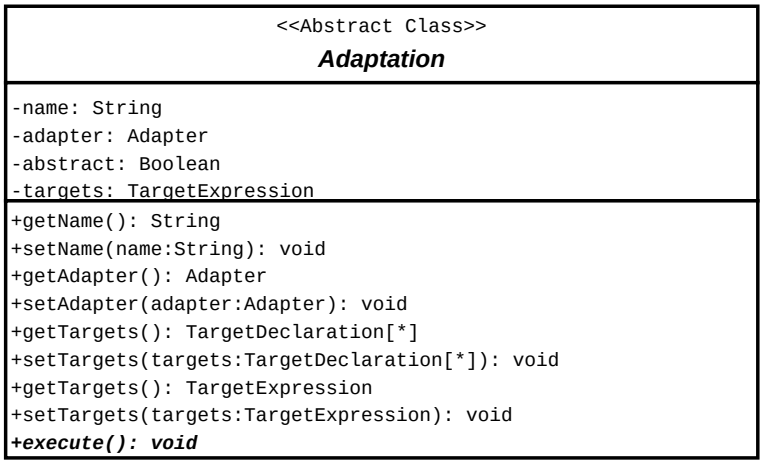

FIG. 7 – Diagramme de la classe Adaptation

Une adaptation est contenue dans un unique adaptateur, auquel elle a accès. Elle possède un nom et peut être déclarée abstraite dans un adaptateur abstrait. Une adaptation peut aussi être déclarée facultative. Une adaptation abstraite facultative n'aura pas besoin d'être obligatoirement concrétisée dans l'adaptateur concret alors que toute adaptation non-facultative devra être concrétisée. Une adaptation est associée à l'expression de cibles d'adaptations (*TargetExpression*) qui correspondent aux cibles sur lesquelles portera l'adaptation.

Une méthode *execute*, sans paramètre, sera toujours appelée pour executer l'adaptation, si cette dernière est concrète (une adaptation abstraite ne sera jamais exécutée). Cette méthode doit être définie dans chaque classe terminale de la hiérarchie des adaptations (voir 4.3 et l'ensemble des opérateurs définis).

Par l'intermédiaire de son adaptateur, l'adaptation accède à tous les schémas de correspondance définis au niveau de l'adaptateur mais les correspondances retenues pour l'adaptation sont associées à cette dernière.

4.2.3 L'expression des cibles

TargetExpression permet d'exprimer des cibles. Cette expression porte toujours sur un type unique de cibles (*TargetType*) et ces types sont *Method*, *Classifier*, *Package*, *Association* ou *Attribute*. Elle peut être explicitée de deux façons différentes : soit en spécifiant une ou des cibles connues, soit en décrivant une expression calculée (*ComplexExpression*).

Expression simple de cible Définir une cible revient à l'identifier de manière unique, c'est à dire en donnant son nom complet (nom et emplacement) et ses paramètres pour une méthode, nom complet pour un attribut, une classe ou un paquetage, ou simplement le nom pour une association. Se basant sur les conventions de nommage qui permettent d'identifier les éléments de manière unique, nous pouvons exprimer des cibles de manière claire, complète et non-ambigüe, à l'aide d'une simple chaîne de caractère. L'expression de ces cibles est détaillée en annexe.

Expression complexe de cibles Il est souvent utile d'indiquer en une fois un ensemble de cibles dont les noms ne sont pas forcément connus à l'avance. Un mécanisme d'expression proche de celui des expressions régulières, simplifié pour s'adapter aux besoins, est proposé dans ce but. L'entité *ComplexExpression* permet d'utiliser ce mécanisme, dont l'utilisation est détaillée en annexe. Une expression complexe est aussi capable de se référer aux cibles d'adaptations déclarées au niveau de l'adaptateur.

4.3 Les opérateurs de composition

Comme cela a été évoqué (section 4.3), les proposées par [16] ,[28],[8], [6] proposent quatre grandes familles d'opérateurs de composition : les introductions (et rédéfinitions), les interceptions, les fusions et les modifications.

Les opérateurs que nous proposons ici se trouvent au carrefour de ces approches. Les introductions d'éléments dans des éléments du modèle sont possibles, tout comme la fusion de certains éléments. Nous proposons aussi les mêmes interceptions que dans [28] et les modifications proposées dans [8], que nous avons enrichies. Ces opérateurs sont modélisés en tant qu'adaptations (voir 4.2) et leur hiérarchie, ainsi que leur spécification complète est donnée en annexe.

4.3.1 Les introductions

Les introductions permettent d'introduire des éléments dans des éléments conteneurs. Dans un paquetage, il est permis d'introduire une classe ou un autre paquetage. Dans une classe, on peut introduire des attributs, des méthodes ou des invariants. On peut aussi y introduire des super-classes ou des sous-classes. Dans une méthode, on peut introduire des préconditions, postconditions, et

dans sa signature, des paramètres. Une méthode abstraite peut-être concrétisée en recevant un corps de méthode.

Il n'est cependant pas possible d'introduire un élément qui existe déjà dans un conteneur. Cependant parfois ils serait opportun que certains éléments puissent correspondre même s'ils ne portent pas le même nom et donc qu'il ne soit pas possible de les introduire. Il faut donc pouvoir préciser ces éventuelles correspondances avant de réaliser les introductions.

4.3.2 Les interceptions

Les interceptions permettent d'indiquer un traitement spécifique en quatre endroits particuliers : *avant*, *après* l'exécution d'un corps de méthode, *à la place* de ce corps, ou encore lorsque la méthode lève une *exception*. Le comportement ajouté sera représenté sous forme d'un appel de méthode et c'est pour accepter ces opérateurs que nous avons introduit dans notre métamodèle une entité *MethodCall* (voir section 4.1 et en annexe), réifiant les appels de méthodes. A chaque fois qu'une méthode sera interceptée, un objet de type *MethodCall* sera ajouté à la liste de *MethodCall* que contient la méthode correspondante (voir en annexe pour plus de détails).

4.3.3 Les fusions

Nous proposons des fusions à propos de trois types d'éléments : fusions de classes, de méthodes et d'associations. Les fusions sont des opérations complexes car elle imposent de faire des choix quant à leur réalisation. Les algorithmes complets des fusions sont donnés en annexes, nous ne préciserons que les points délicats ici.

Lors de la fusion d'associations, il faut tenir compte du fait que des terminaisons peuvent porter sur les mêmes classes. Dans ce cas il faut prévoir de fusionner ces terminaisons, en adaptant la multiplicité en fonction des multiplicité des deux terminaisons.

Lors d'un fusion de classes, outre le fait d'ajouter les éléments des deux classes dans la cible, il faut prévoir une éventuelle correspondance entre des élements (méthodes ou attributs). Si une méthode est mise en correspondance avec une autre alors il faudra fusionner ces deux méthodes.

La fusion de méthodes est donc assez délicate. Il est en effet concevable de fusionner des méthodes ayant des signatures très différentes. La nouvelle méthode aura donc une nouvelle signature, et les corps des deux méthodes à fusionner seront exécutés séquentiellement, l'ordre étant paramétrable. Encore une fois il faut prévoir que deux éléments des deux méthodes à fusionner puissent correspondre, même s'ils ont un nom différent. Il faut donc que la fusion prenne en compte ces éventuelles correspondances entre paramètres.

Ces correspondances sont à utiliser avec précaution car, sémantiquement, dans la plupart des cas cela n'aura aucun sens de vouloir fusionner des méthodes ou des classes. L'utilisateur doit avoir conscience que fusionner des éléments n'ayant rien à voir dénature complètement les éléments modifiés.

Il pourrait être intéressant, dans le but d'améliorer encore notre approche, d'étudier plus en détails le mécanisme proposé dans [29] concernant les correspondances afin de voir si une approche similaire serait envisageable à notre niveau. Notre mécanisme des correspondances est détaillé à la section 4.4.

4.3.4 Les modifications

Les modifications portent sur la visibilité, le type et le nom des éléments. Il est possible de modifier le nom d'une classe, d'une association, d'un paramètre, d'un attribut, d'une méthode ou d'un paquetage.

Au niveau de la visibilité, il s'agit de la visibilité des classes, méthodes et attributs. Et pour les types, il s'agit du type d'un attribut ou du type de retour d'une méthode.

Ces adaptations ont été prévues dans le cadre exclusif d'une utilisation interne à d'autres adaptations, qui ont souvent besoin de modifier ponctuellement certains éléments. Il n'est donc

pas prévu de proposer leur description via le langage de composition (voir section 4.4) mis à la disposition de l'utilisateur.

4.4 Le langage de composition

4.4.1 Expressivité du langage

Le langage de composition est l'interface proposée à l'utilisateur lui permettant d'expliciter ses compositions. Nous voulions un langage à la fois simple d'utilisation, expressif, et lisible (aspect documentaire).

Contrairement à des langages comme *KerMeta* [27] et *ATL* [26], qui sont très riches et plutôt généralistes, notre choix s'est porté sur la proposition d'un langage dédié exclusivement à la composition, comme proposé dans [28] et [8]. Son expressivité doit nous permettre de décrire toutes les adaptations, d'une façon simple et claire.

Le langage proposé dans [8] pour décrire les exemples de composition présenté dans l'article a servi de base à notre travail. Ce langage accompagnait la métamodélisation du mécanisme de composition présenté dans ce même article. Notre métamodèle étant une extension de ce métamodèle, s'appuyer sur le langage qui lui était associé nous a semblé une bonne hypothèse de travail.

Nous l'avons donc enrichi de sorte qu'il puisse décrire toutes les adaptations mises à la disposition de l'utilisateur. La syntaxe de ce langage est fournie en annexe.

4.4.2 Les correspondances

Comme nous l'avons mentionné lors de la description de certains opérateurs de composition (voir 4.3), il est parfois souhaitable ou nécessaire de pouvoir préciser des correspondances entre des éléments de même type, appartenant aux deux modèles à composer. Nous pouvions comme dans [6], proposer un mécanisme implicite ou explicite, en proposant à l'utilisateur soit de spécifier au niveau du langage de composition, les schémas de correspondance entre éléments, soit d'utiliser des règles de correspondance prédéfinies.

Nous avons choisi de proposer uniquement une correspondance explicite : toutes les correspondances et schémas de correspondance devront être exprimés, au niveau de l'adaptateur. Cependant pour accroître la réutilisabilité du code de la composition, les correspondances définies dans un adaptateur peuvent être hérités dans un autre adaptateur. Ainsi, il est tout a fait envisageable de définir un ensemble de schémas de correspondances très générique inclus dans un adaptateur très abstrait hérité par défaut. L'utilisateur pourra ainsi choisir dans cet ensemble le schéma de correspondance qui lui convient et le compléter ou bien en proposer un nouveau.

Pour décrire une correspondance, nous définissons deux masques : l'un porte sur les éléments du modèle source et l'autre sur les éléments du modèle cible. Ces masques permettent de définir des sous-ensembles d'éléments en s'appuyant sur différents critères (nom, visibilité, type...). Nous proposons trois sortes de conditions de correspondance : *i)* fixer une valeur sur un des critères (visibilité « public », type « int »...), *ii)* exiger l'égalité entre les deux mêmes critères des deux masques, quelles que soient leur valeur (égalité sur le type, égalité sur le nom) ou, *iii)* ignorer le critère.

Par exemple, une correspondance générique entre toutes les méthodes publiques de même nom et ayant le même type de retour dans les deux modèles, indépendamment de la classe, du paquetage à laquelles elles appartiennent, ou des paramètres qui composent sa signature, s'écrirait :

```
Method binding ''method-binding based on names and type'' : generic-method-binding {

    source-model { // mask on the source model begins here
        visibility : « public » // fixed value
        class-element : « * » // other values don't matter
        type : « * »
        name : « * »
        class : « * »
        package : « * »
        parameters : « * »
    }
    target-model { // mask on the target model begins here
        visibility : « public » // fixed value
        class-element : « * » // this value doesn't matter
        type : « = » // equality is required on the type
        name : « = » // equality is required on the name
        class : « * » // other values don't matter
        package : « * »
        parameters : « * »
    }
}
```

Dans le cadre d'adapteurs réutilisables où le contexte d'intégration n'est pas connu à l'avance il faut pouvoir décrire des correspondances en partie seulement. Dans ce cas, la correspondance est considérée comme abstraite, et seuls les masques concernant le modèle source (*source-model*), c'est à dire précisant les critères de correspondance choisis, sont décrits. Au moment où l'utilisateur complètera la définition de l'adaptateur, il devra concrétiser la correspondance.

Comme pour les déclarations de cibles (voir section 4.2.3), des contraintes de correspondance peuvent aider l'utilisateur en le guidant lors de la concrétisation de ces correspondances. Ces contraintes portent sur deux points : la multiplicité de la correspondance et sa satisfaction. Les correspondances peuvent avoir trois types de multiplicité : *i) 1-à-1* si à un élément du modèle source ne correspond qu'un seul élément du modèle cible, *ii) 1-à-n* si un élément du modèle source peut correspondre à plusieurs éléments du modèle cible, ou *iii) n-à-1* si à un élément du modèle cible peuvent correspondre plusieurs éléments du modèle source. La mise en œuvre de certaines adaptations, telle que la *fusion*, peut n'être possible que si certaines règles de correspondances sont satisfaites. Pour d'autres adaptations, au contraire, c'est la non-statisfaction d'une règle de correspondance qui peut permettre leur exécution. C'est notamment le cas des adaptations de type *introduction* : une introduction n'est possible que s'il n'existe aucun élément qui corresponde à l'élement à introduire. Ces deux types de contraintes sont exprimables au niveau de l'adaptateur.

4.4.3 Exemple complet de composition

En reprenant l'exemple introduit dans la section 2, voyons comment nous pourrions décrire la composition.

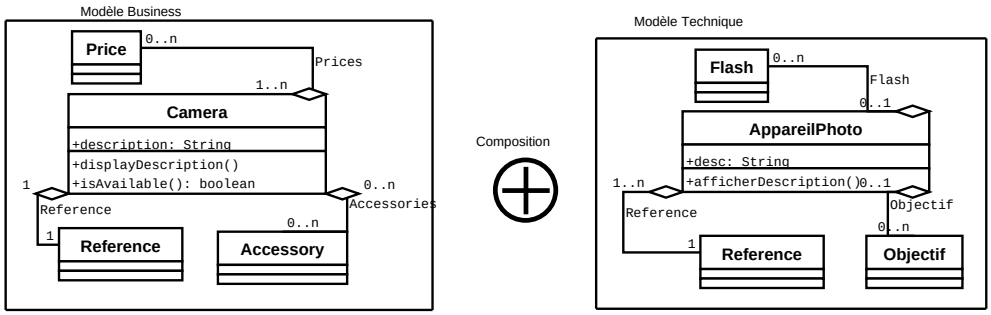

FIG. 8 – Modèles *Business* et *Technique* à composer.

Dans l'exemple de la figure 8 tout laisse croire que les classes *Camera* et *AppareilPhoto* correspondent, de même que les deux classes *Reference*. Les associations *Reference* correspondent aussi. Ces classes et associations devront donc être fusionnées.

Il en va de même des attributs *desc* de *AppareilPhoto* et *description* de *Camera* ou des méthodes *displayDescription* et *afficherDescription* contenues dans ces classes. Les autres classes et associations seront simplement introduites dans le nouveau modèle nommé *Business&Technique*. Le résultat attendu de la composition est décrit dans la figure 9.

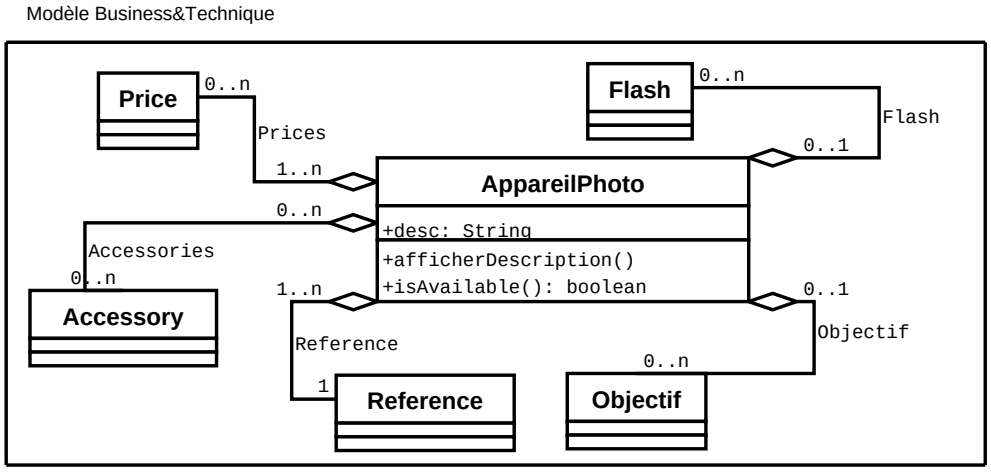

FIG. 9 – Résultat de la composition.

Le modèle *Business* est équipé d'un adaptateur réutilisable (et donc abstrait), décrivant le *protocole de composition*. Ce protocole ainsi que son adaptation au contexte d'intégration sont explicités dans notre langage de composition et présentés dans les codes sources numérotés 1 et 2.

Algorithm 1 Protocole de composition des modèles Business et Technique

```
01 concern example.business_subject
02 abstract adapter BusinessAdapter {
03
04 abstract Class target ”class(es) to introduce ” : toIntroduceClass
05 abstract Association target ”association(s) to introduce ” : toIntroduceAssociation
06 abstract Package target ” package where the classes will be introduced and merges ” : targetPackage
07
08 abstract Method binding ” method(s) matching in Camera Class with other ones in another Class ” :
09 cameraMethodBinding {
10      source-model {
11              visibility : *
12              class-element : *
13              type : *
14              name : "displayDescription"
15              class : "Camera"
16              package : "example.business_subject"
17              parameters : ""
18      }
19 }
20 require cameraMethodBinding 1to1
21 require cameraMethodBinding isVerified
22
23 abstract Attribute binding ” attribute(s) matching in Camera Class with other ones in another Class ” :
24 cameraAttributeBinding {
25      source-model {
26              visibility : *
27              class-element : *
28              type : *
29              name : "description"
30              class : "Camera"
31              package : "example.business_subject"
32              parameters : ""
33      }
34 }
35 require cameraAttributeBinding 1to1
36 require cameraAttributeBinding isVerified
37
38 adaptation introduceClass ”Introduce technical classes in Target package ” :
39 introduce toIntroduceClass in targetPackage
40
41 adaptation introduceAssociation ”Introduce technical associations in Target package ” :
42 introduce toIntroduceAssociation in targetPackage
43
44 optionnal abstract adaptation mergeCamera ”Merge Camera Class with another Class ” :
45 extends class example.business_subject.Camera
46 using binding cameraMethodBinding, cameraAttributeBinding
47
48 optionnal abstract adaptation mergeReference ”Merge Reference Association with another one ” :
49 merge association example.business_subject.Reference
50 }
```

Algorithm 2 Composition des modèles Business et Technique

```
01 concern example.business_and_technical_subject
02 compose example.business_subject with example.technical_subject
02 adapter BusinessAndTechnical extends TechnicalAdapter {
03
04 target toIntroduceClass = example.technical_subject.Price, example.technical_subject.Accessory
05 target toIntroduceAssociation = example.technical_subject.Accessories, example.technical_subject.Prices
06 target targetPackage = example.business_and_technical_subject
07
08 binding cameraMethodBinding {
09      target-model {
10              visibility : *
11              class-element : *
12              type : *
13              name : "afficherDescription"
14              class : "AppareilPhoto"
15              package : "example.technical_subject"
16              parameters : ""
17      }
18 }
19
20 binding cameraAttributeBinding {
21      target-model {
22              visibility : *
23              class-element : *
24              type : *
25              name : "desc"
26              class : "AppareilPhoto"
27              package : "example.technical_subject"
28              parameters : ""
29      }
30 }
31 adaptation mergeCamera :
32 extends class example.business_subject.Camera with example.business_subject.Camera
33 using binding cameraMethodBinding, cameraAttributeBinding
34
35 adaptation mergeReference :
36 merge association example.business_subject.Reference with example.business_subject.Reference
37
38 }
```

Le langage étant relativement simple et explicite nous n'entrerons pas d'avantage dans le détails. Pour plus de précisions quant à la syntaxe de ce langage, se référer aux annexes.

4.5 La généricité

Dans *SmartModels*[25], les entités génériques sont modélisées à la fois par une réification standard, portant le nom d'*atome*, et contenant les éléments partagés par toutes les instances de ces entités génériques, et par un *concept*, une nouvelle entité encapsulant leur variabilité. Le mécanisme que nous proposons ici doit donc être adapté à ces modèles, et doit prendre en compte ces entités génériques. Le métamodèle de composition, ainsi que certains opérateurs, devront être modifiés pour que la composition de ces modèles soit réalisable.

C'est le cas notamment des *fusions*. Fusionner deux entités génériques supposera de fusionner les atomes, bien entendu, mais aussi leur concept.

Ces problèmes font l'objet de nos travaux de recherche actuellement et nous proposerons sous peu une nouvelle modélisation, prenant en considération les modifications à apporter à notre mécanisme.

5 Conclusion et perspectives

Nous avons la conviction que le mécanisme que nous proposons offre des solutions intéressantes pour la composition de modèles, en particulier pour le guidage et le contrôle de la composition (et donc pour sa réutilisation). Cependant cette proposition est encore perfectible et surtout certains points essentiels doivent être approfondis. Par exemple, la composition des assertions n'a pas été étudiée alors qu'il est primordial de pouvoir les composer en même temps que les autres éléments du modèle.

D'un point de vue comportemental, il serait aussi très intéressant de profiter des passerelles offertes par *KerMeta*[27]. A partir de simples modèles *EMF*, ce dernier permet de décrire le comportement. Puisque *EMF* se trouve aussi être le format de base de nos modèles, il est tout à fait envisageable de considérer *KerMeta* dans une évolution future.

Par ailleurs, le langage de composition n'est pas encore complet et il évolue en fonction des modifications que nous apportons au niveau des opérateurs et de l'expressivité que nous souhaitons lui donner.

La description des règles de correspondances n'était pas un problème sur lequel nous avions prévu de nous attarder. Mais en approfondissant notre approche nous nous sommes rendus compte que, même si c'est un point souvent peu développé, un mécanisme souple mais expressif était nécessaire pour composer efficacement des modèles. Le mécanisme de correspondance que nous avons développé, est certes intéressant, mais il est encore loin d'avoir atteint une maturité suffisante. Nous étudions aujourd'hui diverses possibilités pour affiner notre approche et ainsi améliorer son expressivité et son utilisation. Nous nous intéressons en particulier aux travaux réalisés par Gunther Kniesel [29] pour atteindre cet objectif.

Enfin, l'implémentation prévue sous forme de plugin pour Eclipse[9] n'a pas encore été réalisée, mais devrait débuter dans les semaines à venir. Le langage de composition y sera représenté graphiquement sous forme d'un éditeur, fourni par *EMF* et adapté à nos besoins.

Références

[1] Java aspect component. http ://jac.aopsys.com, 2003.

[2] AspectJ team. Aspect J. http ://www.aspectj.org, 2003.

[3] Olivier Barais. *Construire et Maîtriser l'évolution d'une architecture logicielle à base de composants.* Thèse de doctorat, Laboratoire d'Informatique Fondamentale de Lille, Universié de Lille, Lille, France, novembre 2005.

[4] Daniel Bardou and Ouafa Hachani. Modélisation par aspects et transformation vers AspectJ et Hyper/J. In *LMO'2005*, pages 127–142, 38402 Saint Martin d'Hères Cedex, France, juin 2005.

[5] David Bonfils, Térence Férut, Nicolas Julien, and Sébastien Leroy. Plugin eclipse pour la composition de préoccupations dans les langages à objets. Technical report, Université de Nice-Sophia Antipolis, 2005.

[6] Siobhan Clarke. *Composition of Object-Oriented Software Design Models.* Thèse de doctorat, Dublin City University, Dublin, Ireland, January 2001.

[7] Pierre Crescenzo. *OFL : un modèle pour paramétrer la sémantique opérationnelle des langages à objets - Application aux relations inter-classes.* Thèse de doctorat, Université de Nice-Sophia Antipolis, France, décembre 2001.

[8] Pierre Crescenzo and Philippe Lahire. Une approche pour améliorer la réutilisabilité des modèles métiers. In *Actes de la 2ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*, pages 51–73, Lille, septembre 2005.

[9] Eclipse foundation. Eclipse Environment. http ://www.eclipse.org, 2004.

[10] Eclipse foundation. EMF : Eclipse Modelling Framework. http ://www.eclipse.org/emf/, 2006.

[11] Tobias Rho Günter Kniesel. A definition, overview and taxonomy of generic aspect languages. *L'Objet*, 11(3), 2006. to appear.

[12] Stefan Hanenberg and Rainer Unland. Using and Reusing Aspects in AspectJ. In *Proceedings of the Workshop ASoC at OOSPLA'01*, 2001.

[13] Hyper/J team. Hyper/J. http ://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm, 2003.

[14] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference : MoDELS 2005 International Workshops OCLWS, MoDeVA, MARTES, AOM, MTiP, WiSME, MODAUI, NfC, MDD, WUsCAM, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers, LNCS 3844*, pages 128–138. Springer Berlin / Heidelberg, 2006.

[15] Philippe Lahire and Laurent Quintian. New perspective to improve reusability in object-oriented languages. *Journal Of Object Technology (JOT)*, 5(1) :117–138, 2006.

[16] Christina Videira Lopes. and Walter Hursch. Separation of Concerns. Technical Report TR NU-CCS-95-03, North-eastern University, Boston, February 1995.

[17] Alexis Muller, Olivier Caron, Bernard Carré, and Gilles Vanwormhoudt. Réutilisation d'aspects fonctionnels : des vues aux composants. In *Actes de la conférence Langages, Modèles et Objets, LMO 2003*, pages 241–255, Vannes, France, janvier 2003. Hermés Sciences.

[18] Alexis Muller Gilles Vanwormhoudt Olivier Caron, Bernard Carré. Mise en oeuvre dáspects fonctionnels réutilisables par adaptation. page 14, 2004.

[19] Alexis Muller Gilles Vanwormhoudt Olivier Caron, Bernard Carré. A framework for supporting views in component oriented information systems. *OOIS*, 2817 :164–178, Springer, septembre 2003. Lecture Notes in Computer Science.

[20] OMG. *Unified Modeling Language Specification (UML)*. Object Management Group, February 2001. Version 1.4.

[21] OMG. *Unified Modeling Language (UML) OCL - Final Adopted Specification*. Object Management Group, October 2003. Version 2.0 - undergoing finalization.

[22] OMG. *Unified Modelling Language Specification version 1.5*. Object Management Group, 1^{st} edition, March 2003. http ://www.omg.org.

[23] OMG : Object Management Group. MOF : MetaObject Facility. http ://www.omg.org/mof/, 2006.

[24] Renaud Pawlak. Spoon : Annotation-driven program transformation - the aop case. In *Proceedings of the 1st workshop on Aspect-Oriented Middleware Development (AOMD'05), ACM/IFIP/USENIX 6th International Middleware Conference*, volume 118 of *ACM International Conference Proceeding Series*, pages 1–6, Grenoble, France, November 2005. ACM Press.

[25] Philippe Lahire Pierre Crescenzo and Emanuel Tundrea. La généricité paramétrée au service des modèles métiers. In R. Rousseau and C. Urtado, editors, *Actes de LMO'2006, conférence nationale sur les Langages et Modèles à Objets.*, pages 151–166, Nîmes, France, mars 2006. Editions Hermès Lavoisier.

[26] Projet Atlas, INRIA. ATL : Atlas Transformation Language. http ://www.sciences.univ-nantes.fr/lina/atl/, 2005.

[27] Projet Triskell, IRISA. Projet Triskell : KerMeta. http ://www.irisa.fr/triskell, 2005.

[28] Laurent Quintian. *JAdaptor : Un modèle pour améliorer la réutilisation des préoccupations dans le paradigme objet.* Thèse de doctorat, Université de Nice-Sophia Antipolis, France, juillet 2004.

[29] Tobias Windeln (Rho). Logicaj - eine erweiterung von aspectj um logische meta-programmierung. Diploma thesis, CS Dept. III, University of Bonn, Germany, Aug 2003.

[30] Christophe Tombelle and Gilles Vanwormhoudt. Manipulation de modèles avec un langage de script orienté objet : une approche basée sur les mécanismes dynamiques des apis de manipulation. Technical report, LIFL, novembre 2005.

[31] Tewfik Ziadi. *Manipulation de lignes de produits en UML.* Thèse de doctorat, Université de Rennes I, Rennes, France, décembre 2004.

Annexes