Actes des Journées du groupe Objets, Composants et Modèles (OCM'2005)

Ph. Collet, Philippe Lahire

LABORATOIRE

INFORMATIQUE, SIGNAUX ET SYSTÈMES
DE SOPHIA ANTIPOLIS
UMR 6070

OCM 2005

Actes des Journées du groupe Objets, Composants et Modèles

Berne, du 9 au 11 mars 2005

Philippe Collet & Philippe Lahire (eds.)

Rapport de recherche
ISRN I3S/RR–2005-07–FR

Février 2005

LABORATOIRE I3S: Les Algorithmes / Euclide B – 2000 route des Lucioles – B.P. 121 –
06903 Sophia-Antipolis Cedex, France – Tél. (33) 492 942 701 – Télécopie : (33) 492 942 898
http://www.i3s.unice.fr/I3S/FR/

Avant-propos

Après Vannes en 2003 et Lille en 2004, le groupe OCM du GDR ALP a tenu son troisième atelier de travail du 9 au 11 mars 2005 à Berne, en Suisse, en marge de la conférence LMO'2005. Ces sessions réparties sur trois jours ont permis d'améliorer encore les collaborations entre les équipes du groupe de travail. Les doctorants et les jeunes chercheurs étaient en particulier appelés à présenter les travaux qu'ils ont récemment entrepris de manière à les faire connaître à l'ensemble du groupe. Les soumissions venant de chercheurs et enseignants-chercheurs confirmés étaient aussi les bienvenues si elles se situaient dans le même esprit.

Cette année, le comité de sélection a reçu dix propositions, ce qui montre un intérêt croissant des membres du groupe pour cette manifestation. Après relectures, toutes ont été jugées acceptables pour être présentées lors des sessions. Les corrections demandées, plus ou moins importantes, ont été prises en compte par les auteurs et nous les remercions pour leurs efforts. Les thèmes abordés dans ces articles couvrent largement le spectre du groupe de travail OCM. On y retrouve d'ailleurs les préoccupations des trois domaines qui forment son sigle : Objets, Composants et Modèles. Le lecteur y trouvera ainsi des travaux proches de l'aboutissement, mais aussi des premières pistes de recherches et des démarches qui favorisent les interactions et les retours entre les membres du groupe.

Enfin, nous tenons à remercier les membres du comité de sélection, ainsi que les organisateurs de LMO 2005, dont le travail a rendu possible l'organisation de ces sessions.

Philippe Collet et Philippe Lahire
responsables du groupe OCM
Université de Nice - Sophia Antipolis, I3S CNRS UMR 6070

Comité de sélection

- Philippe Collet, Université de Nice - Sophia Antipolis, I3S
- Daniel Deveaux, Université de Bretagne Sud, VALORIA
- Christophe Dony, Université Montpellier-II, LIRMM
- Philippe Lahire, Université de Nice - Sophia Antipolis, I3S
- Roger Rousseau, Université de Nice - Sophia Antipolis, I3S
- Dalila Tamzalit, Université de Nantes, LINA

Table des matières

Un patron de génération de code pour le profil VUML

Xavier Crégut (1), Sophie Ebersold (2)
Mahmoud Nassar (2, 3), Bernard Coulette (2)

(1) Enseeiht – IRIT
2, rue Camichel 31071 Toulouse Cédex 7 – cregut@enseeiht.fr
(2) Equipe GRIMM-ISYCOM – Université de Toulouse le Mirail
5 allées A. Machado 31058 Toulouse cédex, France – {ebersold, coulette}@univ-tlse2.fr
(3) ENSAM – Université Moulay Ismaïl
B.P. 4024, Béni M'Hamed, Méknès, Maroc – nassar@univ-tlse2.fr

Mots-clefs – Keywords

Modélisation, profil UML, Vue, Point de vue, Classe multivues, Patron d'implémentation.

Modelling, UML profile, View, Viewpoint, Multiviews class, Implementation pattern.

Résumé – Abstract

VUML (View based Unified Modeling Language) est une méthode d'analyse/conception basée sur les vues. Elle offre un formalisme (profil UML) et une démarche pour modéliser un système logiciel par une approche combinant objets et vues. Le principal ajout à UML est la *classe multivues* qui permet de stocker et restituer l'information en fonction du point de vue de l'utilisateur, en offrant des possibilités de changement dynamique de points de vue tout en gérant la cohérence entre les vues dépendantes. Sur le plan sémantique, nous avons étendu le méta-modèle d'UML avec des règles formalisées en OCL. Cet article traite de la phase de génération de code en proposant un patron générique d'implémentation pour générer du code correspondant à un diagramme de classes VUML. Le patron proposé exploite la délégation, la technique de la poignée et le polymorphisme pour représenter les vues. Il a été testé avec le langage cible Java.

VUML (View based Unified Modelling Language) is a view-based analysis/design method. It offers a formalism (UML profile) and a process to model software systems through objects and views. The main extension to UML is the multiviews class whose goal is to store and deliver information according to user's viewpoint. VUML supports the dynamic change of viewpoints and offers mechanisms to describe views dependencies. On the semantics side, the VUML metamodel extends the UML one with OCL rules. This paper focuses on VUML code generation stage. We propose a generic implementation pattern to generate code corresponding to a VUML class diagram. The proposed pattern uses delegation, hook technique and polymorphism to represent views. It has been tested with Java.

1 Introduction

Les concepts de vue/point de vue ont été étudiés dans de nombreux domaines de l'informatique : bases de données, représentation des connaissances, analyse et conception, la ngages de programmation, outils de Génie logiciel, etc.[SS89][FKG90][AB91][CG91][C KM96][Deb98] [MDM00]. Ces concepts ont aussi été introduits sous d'autres termes tels que sujet [HO93], rôle [Kris95] ou programmation par aspects [KLMMV97], [Szy99]. Nos propres travaux menés sur ce thème ont donné lieu à la définition du langage VBOOL [MCK94], et à la méthode associée VBOOM [Krio95]. Prenant en compte d'une part le standard UML dans la modélisation des systèmes logiciels, d'autre part la difficulté à modéliser les vues avec de l'héritage multiple, nous avons décidé de reconsidérer l'approche développée dans VBOOM en privilégiant la réutilisation des standards de l'OMG (en particulier UML) et en ciblant les principaux langages à objet du marché. La notion de vue d'UML [OMG04] est un moyen offert au concepteur pour structurer une conception : cas d'utilisation, logique, composants, déploiement. Cependant, UML ne permet pas d'intégrer la notion de vue au cœur même de la modélisation, i.e. dans les diagrammes de classes. Aussi avons-nous décidé de développer une extension d'UML, appelée VUML (View based Unified Modeling Language) [Nas03],[NCCMK03] qui repose sur le concept de "Classe MultiVues" (C-MV). VUML a pour objectif d'offrir un formalisme et une démarche pour mener une modélisation par vues de l'analyse jusqu'à la génération de code. Par rapport aux propositions existantes et à l'approche adoptée dans VBOOM, la singularité de ce travail réside dans le fait d'associer de façon unique une vue à chaque type d'acteur du système. Cette décision simplifie tout d'abord la détermination des vues sur une entité donnée, et donc le fort indéterminisme qui caractérise cette détermination, et réduit aussi nettement les problèmes liés à la composition des vues inhérents aux propositions à base de vues multiples [Deb98][Krio95][MCK94][MDM00][MCCV03]. Dans cet article, nous focalisons notre attention sur la phase de génération de code dans VUML. Dans la section suivante, nous présentons brièvement le profil VUML. Le lecteur intéressé pourra en trouver une présentation détaillée ainsi que la démarche associée dans [Nas04]. La section 3 décrit la traduction d'une classe multivues en code objet multi-cibles à partir d'un patron générique d'implémentation. Dans la conclusion, nous faisons le point sur le travail effectué et présentons ses principales perspectives.

2 VUML

VUML est un profil UML fondé sur une approche centrée utilisateur, dont l'objectif est de représenter les besoins et les droits d'accès des acteurs du système (utilisateurs finaux ou non) de l'analyse jusqu'à l'implémentation. Pour atteindre cet objectif, VUML étend la notation UML en ajoutant le concept de classe multivues (C-MV). Le méta-modèle de VUML a ainsi été défini comme une extension de celui d'UML [Nas03][NCCMK03].

Classe et objet multivues

Une C-MV (ou classe multivues) est une unité d'abstraction et d'encapsulation composée d'un ensemble de vues appelées *view*, qui représentent les besoins et les droits spécifiques des acteurs et d'une vue par défaut appelée aussi *base* - partie de la classe accessible par tous les acteurs (au sens UML), donc commune à toutes les vues. Chaque vue correspond à un seul

acteur. Ces vues sont reliées à la base via une relation de dépendance stéréotypée par *viewExtension*. La figure 1 ci-dessous décrit le modèle VUML simplifié d'un concessionnaire de voitures. Dans ce diagramme de classes, il y a 3 classes multivues : Voiture, Constructeur, Accident. Pour chaque C-MV, nous avons mis en évidence les vues pertinentes. Ainsi, la classe Voiture est dotée de 3 vues correspondant aux points de vue du Maintenicien, du Commercial et du Client. Pour la classe Constructeur, seules les vues associées au Maintenicien et au Commercial sont pertinentes. VUML supporte le mécanisme d'héritage entre C-MV : une classe descendante d'une C-MV est également multivues, avec des possibilités d'ajout d'une nouvelle vue, de redéfinition d'une vue de la classe parente, etc. La description de ce mécanisme n'entre pas dans le cadre de cet article, mais peut être consultée dans [Nas04]. Un objet multivues est une instance d'une classe multivues. Comme les objets classiques, un objet multivues a un état et un comportement. Il est composé d'un *objet base* et d'un ensemble d'*objets vues* (dits tout simplement *vues* par la suite). La relation *viewExtension* n'est pas une relation d'héritage : les vues dépendent de la base au sens où les valeurs d'attributs et les méthodes de la base sont implicitement partagées par toutes les vues de la classe. Par exemple, les vues d'un même objet de type *Voiture* auront accès aux valeurs des attributs de la base : *ref*, *marque*, *couleur*, etc.

Gestion des vues

Au moment de l'exécution, une vue d'un objet multivues peut être activée. Elle est alors dite vue courante ou active et détermine le point de vue de l'utilisateur courant. Elle rend accessible les informations spécifiques à cette vue, sachant qu'il est possible de redéfinir des opérations de la base (mécanisme similaire à la redéfinition de l'héritage). L'activation d'une vue est faite durant l'exécution. La base est implicitement active si aucune vue n'est spécifiée. Plus généralement, il est possible d'activer une vue qui devient alors la vue courante (la vue précédente est automatiquement désactivée), de désactiver une vue, de bloquer une vue (empêcher les futures activations) ou de la débloquer. Les opérations bloquer/débloquer ne peuvent être réalisées que par l'administrateur du système qui est représenté par une vue implicite automatiquement ajoutée à toute classe multivues.

Propagation de la vue active

L'activation d'une vue sur un objet concerne tous les objets multivues ayant des relations avec cet objet. Aussi, cette activation doit être propagée, récursivement, sur tous les objets multivues qui lui sont liés. Sur le modèle VUML de la figure 1, l'activation de la vue *Commercial* sur une instance de la classe *Voiture* est propagée à l'objet *Constructeur* et à chacun des objets de la liste *accidents*. L'activation de la vue *Client* sur cette voiture provoque l'activation des vues par défaut de *Constructeur* et *Accident*, car elles n'ont pas de vue *client*.

Traitement des messages

L'appel d'une méthode existante sur un objet multivues se fait de la manière suivante : la méthode est cherchée tout d'abord dans la vue active ; si elle n'y est pas, elle est cherchée dans la base de l'objet multivues. Si cette méthode n'est pas trouvée, c'est une erreur. Si elle

appartient à une autre vue, c'est un accès illicite et une exception est levée (pour informer l'appelant que la méthode invoquée n'est pas accessible selon la vue courante).

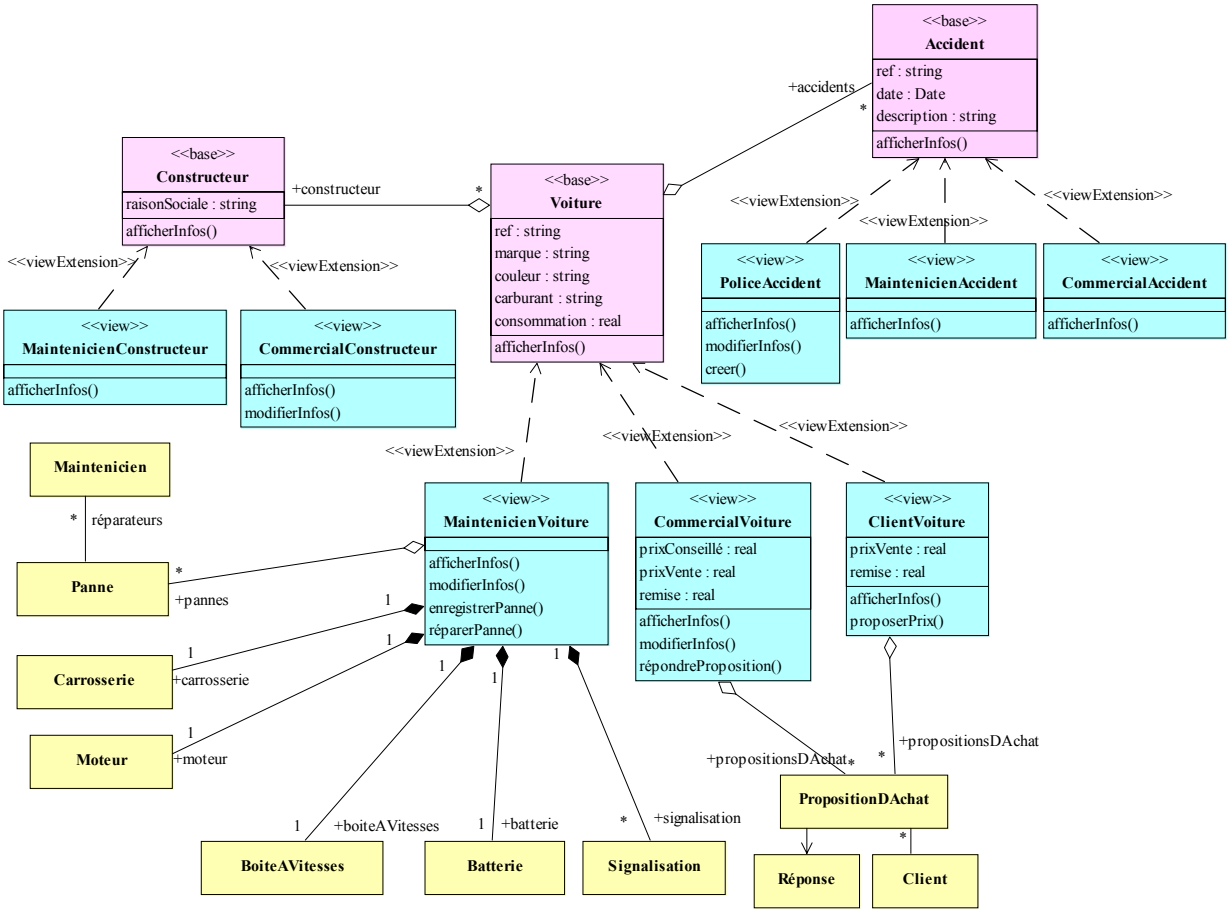

Figure 1 : Exemple de modèle VUML "Concessionnaire de voitures"

3 Le patron de génération de code

Pour illustrer les besoins de programmation par objets à l'aide de vues, nous donnons ci-dessous (figure 2) des exemples des différentes possibilités d'invocations de méthodes sur un objet multivues avec des changements dynamiques de vue.

Pour rendre notre approche facilement utilisable, nous avons décidé de cibler les principaux langages à objet du marché (Java, C++, Eiffel, …), et de proposer une traduction automatisée d'une **classe multivues** en un modèle d'implémentation multi-cibles. Dans la suite, les exemples sont donnés en Java mais la technique proposée est valable pour tous les langages objets supportant le polymorphisme.

Nous avons défini un patron générique d'implémentation. Il combine le patron Rôle pour implémenter la délégation et la technique de la poignée, et le patron Stratégie pour l'implémentation des vues avec la surcharge des méthodes [Gam95][Coa92]. Ce patron est utilisé dans le processus de génération de code relatif à une modélisation VUML (cf. figure 3). Nous ne donnons ci-dessous que les éléments clés permettant de comprendre ce patron, en

laissant de côté l'implémentation de certaines fonctionnalités : propagation des changements de vue, mise en cohérence des vues, traitement de l'héritage entre classes multivues.

```
Voiture v = new Voiture() ;            // Instanciation de Voiture
v.setView("ClientVoiture") ;           // activation de la vue ClientVoiture
v.afficherInfos() ;                    // appel de la méthode afficherInfos() de la vue ClientVoiture
v.modifierInfos() ;                    // lève une exception car la vue ClientVoiture ne donne pas accès à la méthode
                                       // modifierInfos()
…
v.setView("CommercialVoiture") ;       // activation de la vue CommercialVoiture
v.afficherInfos() ;                    // appel de la méthode afficherInfos() de la vue CommercialVoiture
v.modifierInfos() ;                    // appel de la méthode modifierInfos() de la vue CommercialVoiture
v.repondreProposition() ;              // appel de la méthode repondreProposition() de la vue CommercialVoiture
…
v.setView("");                         // activation de la base de l'objet v
v.afficherInfos() ;                    // appel de la méthode afficherInfos() de la base
v.reparerPanne() ;                     // lève une exception car la base ne donne pas accès à la méthode reparerPanne()
…
v.setView("MecanicienVoiture") ;       // activation de la vue MecanicienVoiture
v.reparerPanne() ;                     // appel de la méthode reparerPanne() de la vue MecanicienVoiture
…
```

Figure 2 : Exemple d'appels en Java sur un objet de type Voiture

Une classe multivues CMV est traduite par l'ensemble de classes suivant : une classe de même nom CMV reliée par composition à une classe Base_CMV (regroupant les attributs et les méthodes de la base), et à une classe View_CMV, et une liste de classes correspondant aux vues. Les vues sont décrites par des sous-classes de View_CMV. La classe CMV fournit la méthode *setView()* permettant d'activer/désactiver une vue et la méthode *getView()* qui retourne la vue active. La vue active est soit une instance de *View_CMV* (vue par défaut quand aucune vue n'est active), soit une instance de la classe décrivant une vue (*Vuek_CMV*). Les méthodes sont appliquées sur l'instance de la classe CMV qui contient toutes les méthodes définies dans la classe multivues. La sélection de la méthode à exécuter s'appuie sur le polymorphisme de *View_CMV*. Considérons une méthode m(args) ayant T (éventuellement Void) comme type de retour. La définition de la méthode m*()* engendrée dans la classe CMV se contente de rediriger l'appel vers la vue courante :

```
T m(args) {return getview().m(args);}
```

La définition de la méthode m() dans la classe View_CMV dépend du lieu de définition de m(). Si m() est uniquement dans la base, l'appel engendré passe par une indirection sur elle :

```
T m(args){getCMV().getBase().m(args) ;}
```

Si m() n'appartient pas à la base, l'exception *AccesInterditException* est levée :

```
T m(args){throw new AccesInterditException();}
```

La classe Vuek_CMV est obtenue en dupliquant la vue correspondante Vuek. Ainsi, si une méthode m() apparaît dans Vuek, elle est définie dans Vuek_CMV ce qui constitue une redéfinition de la méthode de View_CMV.

La figure 2 présentée plus haut permet de vérifier le mécanisme proposé ci-dessus sur l'application *Concessionnaire de voitures.*

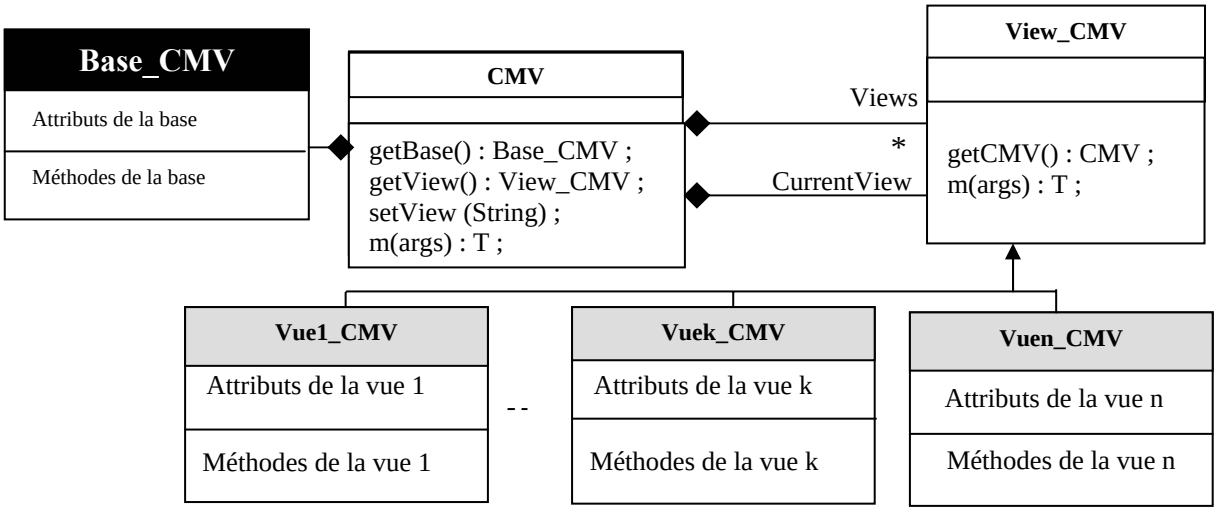

Figure 3 – Patron de génération de code associé à VUML

4 Conclusion et perspectives

Le travail présenté dans ce papier s'inscrit dans le cadre de la définition d'un profil VUML et d'une démarche centrée utilisateur pour modéliser un système à l'aide de classes multivues. Dans cet article, nous avons focalisé plus particulièrement notre attention sur un patron générique d'implémentation qui décrit la génération de code objet multicible relatif à une modélisation VUML (diagramme de classes). Ce patron permet notamment de générer le code correspondant à la structure d'une classe multivues, aux appels de méthodes, aux changements dynamiques de vue. Afin de gérer les droits d'accès aux services offerts par une C-MV, le patron proposé utilise le polymorphisme et la technique de la poignée, ce qui confère à un objet multivues un dynamisme permettant d'adapter son comportement à la vue active.

Actuellement, nous travaillons sur l'affinage du profil de génération de code associé à VUML. Un prototype opérationnel a été implanté sous Objecteering, à l'aide du langage J. Il doit être étendu afin de traiter certaines fonctionnalités telles que la mise en cohérence des vues dépendantes. Le patron a été testé avec un générateur de code Java, mais il sera validé avec d'autres langages cibles tels que le langage Eiffel.

Parmi les autres pistes de travail d'ores et déjà explorées dans le projet, nous pouvons signaler la finalisation de la démarche associée à une modélisation en VUML (en particulier la phase de fusion en VUML de modèles UML développés séparément), l'étude de l'impact de la modélisation par vues sur les diagrammes dynamiques d'UML (notamment les diagrammes d'activités et les diagrammes Etats-Transition), le développement de composants multivues réutilisables. A cet effet, nous avons étendu le méta-modèle d'UML pour intégrer dans VUML la notion de composants multivues correspondant à la notion de classes multivues, mais offrant des interfaces *vues* fournies ou requises [Nas04].

Ce travail s'inscrit dans le cadre des activités de recherche du réseau franco-marocain STIC en Génie Logiciel (http://www.grimm/isycom/reseauSTIC/reseauSTIC.html)

Références

[AB91] S. Abiteboul, A. Bonner. Objects and Views. In *ACM SIGMOD*, pages 238-247, 1991.

[CG91] B. Carré, J.M. Geib. The Point of View Notion for Multiple Inheritance. In *ECOOP/OOPSLA*. 1991.

[CKM96] B. Coulette, A. Kriouile, S. Marcaillou. L'approche par points de vue dans le développement orienté objet des systèmes complexes. Revue l'Objet, Vol. 2, No 4:13-20, 1996.

[Coa92] P. Coad. Object-oriented Patterns. Communications of the ACM, 1992.

[Deb98] L. Debrauwer. Des vues aux contextes pour la structuration fonctionnelle de bases de données à objets en CROME. Thèse de doctorat en Informatique. LIFL, France, 1998.

[FKG90] A. Finkelstein, J. Kramer, and M. Goedicke. Viewpoint Oriented Software Development. In *Eng. and Applications Conference*. Toulouse, France, pages 337-351, 1990.

[GAM95] B. Gamma, and al. Design patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

[HO93] W. Harrison, H. Ossher. Subject-oriented programming: a critique of pure objects. In *OOPSLA'93*, Washington D.C., pages 411-428, 1993.

[Krio95] A. Kriouile. VBOOM, une méthode orientée objet d'analyse et de conception par points de vue. Thèse d'état, Faculté des sciences de Rabat, Maroc, 1995.

[Kris95] B. B. Kristensen. Object Oriented Modeling with Roles. In *the 2nd International Conference on Object Oriented Information Systems (OOIS'95)*, Dublin, Irland, 1995.

[KLMMV97] G. Kiczales, J. Lampng, A. Mendhekar, C. Maeda, L.C. Videira. Aspect-Oriented Programming. In *ECOOP'97*. Springer-Verlag LNCS 1241. Finland, 1997.

[MCC03] A. Muller, O. Caron, B. Carré, and G. Vanwormhoudt. Réutilisation d'aspects fonctionnels : des vues aux composants. Revue RSTI - L'Objet, septembre: 241-255, 2003.

[MCK94] S. Marcaillou, B. Coulette, A. Kriouile. Visibility : A new relationship for complex system modelling. *TOOLS USA'94*, Santa Barbara, Prentice Hall, 1994.

[MDM00] H. Mili, J. Dargham, and A. Mili. Views: A Framework for Feature-Based Development and Distribution of OO Applications. In *Thirty-Third Hawaii International Conference on System Science*, Honolulu, 2000.

[Nas03] M. Nassar. VUML : a Viewpoint oriented UML Extension. In the *18th IEEE International Conference on Automated Software Engineering (ASE'2003)*. Montreal, Canada, 2003.

[NCCMK03] M. Nassar, B. Coulette, X. Crégut, S. Marcaillou, and A. Kriouile. Towards a View based Unified Modeling Language. In *ICEIS'03*, Angers, France, 2003.

[Nas04] M. Nassar. VUML : une extension UML orientée point de vue. Thèse de l'université Mohammed V, Rabat, 13 décembre 2004.

[OMG04] OMG, 2004. Unified Modeling Language, version 2.0, http://www.omg.org

[SP89] J. Shilling, P. Sweeny. Three Steps to Views. In *OOPSLA'89*. New Orleans, LA, pages 353-361, 1989.

[Szy99] C. Szyperski. Component Software, Addison-Wesley, 1999.

Modélisation distribuée par métiers des systèmes embarqués

Wolfgang Theurer , François Mekerke, Joël Champeau
ENSIETA - Laboratoire DTN
2 rue François Verny 29806 Brest Cedex 9
{mekerkfr, theurewo, champejo}@ensieta.fr

Mots-clefs – Keywords

Pivot, facettes-métier, aspects, meta-modèles, méthodes formelles, cohérence
Pivot, craft-oriented facet, aspects, meta-models, formal methods, consistency

Résumé – Abstract

Lors des développements "système", le plus important est de pouvoir garantir la bonne intégration des éléments de solution. Certaines techniques, dont la conception par aspects, permettent de fusionner des modèles dérivant de préoccupations différentes, donc d'ajouter des propriétés dans un modèle, mais seulement localement dans le temps. Manque aussi à ces techniques la prise en compte du découpage par métier qui est la norme dans l'industrie.
Nous présentons ici une méthode de maintien en cohérence d'un système modélisé de manière distribuée, qui supporte l'adjonction d'aspects. Elle peut se résumer comme suit : des "facettes orientées métier" se coordonnent en échangeant des informations abstraites au travers d'un "pivot" qui vérifie au passage que diverses propriétés, issues des aspects ou des spécifications, sont bien vérifiées, au moyen de méthodes formelles notamment. Nous espérons ainsi réaliser un bon compromis entre souplesse et rigueur, qui permet un développement assez libre tout en validant les choix régulièrement.

When developing systems, integration is what matters most. Some techniques allow us to weave models into one another, thus adding properties to a model, but only locally in time. They also forget to aknowledge the craft-oriented partition that is the norm in industry.
We present here a method to validate the consistency of a distributed model, which also supports aspects. It can be summarized as follows : "craft-oriented facets" work together, exchanging abstract data through a "pivot", which checks a number of properties coming from aspects or specifications, thanks to formal methods for example. Doing so, we hope we have found a good compromise between flexibility and rigor, which allows for free development while validating choices all along.

1 Introduction

Les systèmes embarqués sont de plus en plus complexes tant par leur technicité que par le nombre de spécialistes différents impliqués dans leur réalisation. Dans ce contexte il devient impossible d'appréhender un tel système dans sa globalité par un modèle monolithique le représentant en entier. En effet, en plus d'être terriblement volumineux, un tel modèle poserait le problème de son interprétation par des spécialistes de différents métiers n'utilisant par les mêmes formalismes.

Les objectifs de nos travaux[1] sont, d'une part d'identifier les problèmes posés par la multiplication des intervenants dans la conception des systèmes embarqués puis de proposer un support méthodologique basé sur l'utilisation de modèles semi-formels pour répondre à ces différentes problématiques. Ces travaux se posent comme une étude en largeur ayant pour vocation de proposer un cadre sur lequel peuvent se greffer des techniques existantes ou à venir répondant à des problèmes précis survenant lors de la conception d'un système embarqué temps-réel.

Nous nous plaçons dans le cadre général de la modélisation "par parties", notamment dans le cadre MDA[2] de l'OMG[3], qui apportent des solutions pour la gestion de la *separation of concerns* (v. [HL95], [OT00]). La conception par aspects (v. [SY99], [MGFA02]), issue de la programmation par aspects (v. [KLM+97]) ou les *composition filters* (v. [AT98]) préconise la modélisation séparée, avec les fonctionnalités d'un côté, et les préoccupations non fonctionnelles de l'autre, suivie du *weaving* de ces dernières dans les premières. Nous souhaitons d'une part élargir le champs d'application de ces techniques de modélisation "logiciel" au système dans sa globalité et surtout conserver au niveau modèle un découpage par métier, plus proche de la répartition réelle des tâches sur un projet industriel.

Pour ce faire nous nous intéressons plus particulièrement au point de vue et aux besoins du Maître d'Œuvre (MOE) (ou du chef de projet) qui seul possède une vue transversale du système. Nous introduisons un modèle "pivot" du système qui lui est spécialement dédié ainsi que des modèles "d'interaction" en le MOE et les autres intervenants. Enfin nous apportons un ensemble de modèles "actifs" permettant de gérer l'intégration continue des sous-parties du système développées par les intervenants garantissant ainsi la cohérence du tout au niveau le plus abstrait possible.

La suite de cet article est organisée de la façon suivante ; dans une première partie nous fournissons une vision globale de nos travaux, puis au cours des trois parties suivantes nous definissons plus précisément les concepts majeurs que nous introduissons.

2 Présentation générale de notre approche

Notre objectif est de fournir un cadre méthodologique ainsi qu'un ensemble de techniques permettant d'aider la maîtrise d'œuvre de systèmes mettant en jeu de nombreux spécialistes métier différents, chacun réalisant sa partie du système. Pour ce faire, il est impératif que la méthodologie que nous proposons ait les propriétés suivantes :

Tout d'abord il est souhaitable que chaque intervenant puisse travailler sur ses propres modèles

[1] Thèses de doctorat de W. Theurer et F. Mekerke

[2] Model-Driven Architecture

[3] Object Management Group

et que sa connaissance des autres parties du système réalisées par les autres intervenants soit limitée aux strictes informations dont il a besoin pour réaliser sa partie. Il est aussi important de ne pas forcer les intervenants à opter pour un formalisme de modélisation commun : chaque domaine métier possède des outils de modélisation adaptés à ses problématiques (DSL[4]) et fournissant des fonctionnalités qui ne peuvent être incluses dans un langage de modélisation général.

Ensuite, le MOE doit disposer d'un modèle adapté à ses préoccupations. La vision qu'il porte sur le système à tout moment du cycle de développement lui est propre, elle doit être synthétique et refléter l'intégralité des exigences système et lui permettre de savoir quels sous-traitants sont impliqués dans la réalisation de chacune d'elles. N'étant pas spécialiste d'un domaine, le MOE n'a pas besoin de connaître les détails de l'implantation des différentes parties de son système. Notre approche devra lui cacher le raffinement des modèles des intervenants tout en lui fournissant les informations qui lui sont nécessaires. Elle doit enfin fournir le moyen de vérifier la cohérence des différentes parties du système et permettre de gérer les préoccupations transverses[5] en calculant des informations générales à partir de données locales fournies par les intervenants concernant leur partie.

Pour réaliser ces objectifs nous proposons une méthodologie reposant sur un ensemble de modèles organisés de la façon suivante :

Chaque intervenant, travaillant au sein de sa "facette" (liée à un domaine d'expertise), fournit un modèle abstrait de la partie du système qui lui incombe. L'ensemble de ces modèles forme la base de la vue système du maître d'œuvre ; ils s'expriment dans un même formalisme. Bien sûr, le développement des parties privées des facettes peut conduire à l'utilisation d'un formalisme différent, mais alors il doit possèder une relation de transformation vers la partie publique.

Le modèle dédié au maître d'œuvre, ou "pivot", est structuré de manière à permettre la validation de propriétés liées aux spécifications.

La liaison entre "facettes" et "pivot" se fait au travers d'un ensemble de modèles dynamiques, appelés "layers", qui extraient les informations pertinentes des "facettes" pour les remonter au niveau du "pivot". Enfin nous fournissons un processus de développement intégrant notre méthodologie. (v. [TMR+04])

3 Les facettes

Nous considèrons ici que le système est distribué aux différents intervenants en fonctions de leurs domaines d'excellence, donc du "métier" qu'ils exercent. Nous dénommerons par le terme générique de "facette" à la fois les exigences liées à un métier, l'équipe physique spécialisée dans ce métier et l'ensemble de modèles du sous-système qu'elle conçoit. Par exemple, dans le cas du développement d'un avion on peut citer les facettes électronique, logiciel embarqué, hydraulique, automatique, aérodynamique, etc.

Une facette est constituée d'une partie privé et d'une partie publique. La partie privée, connue de la facette seule, est composée de l'ensemble des modèles nécessaires à la réalisation de sa partie du système. Le choix des formalismes employés dans cette partie est laissé à l'entière

[4] Domain Specific Languages

[5] cross-cutting concerns

discrétion de la facette. La partie publique, ou Public Model (PM), a plusieurs fonctions : elle sert de contrat entre l'intervenant et le MOE en spécifiant le modèle solution abstrait regroupant à la fois les informations architecturales et "comportementales" sur lesquelles portent les exigences de la facette. Il est en outre suffisamment précis pour permettre l'extraction d'informations transverses nécessaires au MOE pour vérifier les exigences système, et ainsi permettre la vérification a priori la bonne intégration finale des sous-systèmes.

Ce PM est constitué de deux types d'objets :
- les objets *offerts* sont des objets développés par la facette. Celle-ci en connaît les détails d'implantation. C'est elle qui doit spécifier les attributs de ces objets nécessaires au reste de parties prenantes. Par exemple la facette "logiciel" offre des processus et des données, la facette materiel offre des liens de communication, des ressources de calcul et des capteurs/actionneurs (à un niveau d'abstraction le plus élevé possible) ;
- les objets *requis* sont des abstractions d'objets d'autres facettes nécessaire pour le développement de la facette considérée. Par exemple la facette logiciel intègre les capteurs actionneurs en tant que processus requis ce qui lui permet de manipuler un modèle clos et ainsi des spécifier tous les échanges de données.

4 Le pivot

Étant données la taille et la complexité des grands systèmes actuels (avioniques, navals ...), le MOE ne peut ni ne doit connaître en détails l'intégralité des modèles du système dont il a la charge. De même, les équipes métier impliquées dans le développement (ou le maintien en condition opérationnelle), si elles doivent maîtriser à 100% leur domaine, n'ont besoin que d'une quantité réduite d'informations abstraites sur le reste du système.

Pour répondre à cette double problématique, nous définissons un modèle abstrait appelé pivot qui regroupe des informations de haut niveau permettant d'une part au MOE d'avoir une vision synthétique du système et d'autre part de servir d'interface entre les Public Models des différentes facettes. Le pivot est un modèle abstrait du système orienté exigences fonctionnelles : Les entités du pivot sont conçues comme des blocs logiques reflétant une fonctionnalité du système sur lequel porte des exigences globales (transverses).

Exemple 1 *Considérons le cas d'un distributeur automatique de billets. Les spécifications de celui-ci seront, entre autres, de fournir une interface conviviale, de valider l'identité du client par les informations de sa carte et de son code secret et de contacter l'agence bancaire afin de mettre à jour le solde.*

Même si la solution choisie pour implanter ce DAB est constituée d'un calculateur doté d'un OS multi-tâches et de plusieurs processus logiciels le pivot comportera un bloc "IHM", un bloc "authentification", et un bloc "communication" qui représenteront chacun un processus logiciel et la part de calculateur qu'il utilise. Ceci permet d'obtenir une vision fonctionnelle du système et de spécifier des exigences transverses telle que le temps de réponse sur un bloc logique.

5 Layers

Un des problèmes majeurs induits par l'utilisation de multiples modèles pour représenter un même système est le maintien en cohérence de l'ensemble. Dans le contexte du découpage des modèles par métier, la relative orthogonalité des préoccupations des divers intervenants permet de réduire ces problèmes d'assemblage et de cohérence à la gestion des exigences transverses. En choisissant de considèrer le raffinement des modèles comme une information privée, cette gestion se trouve limitée à un unique niveau d'abstraction. La structure de layers que nous présentons dans ce chapitre s'appuie sur ces propriétés de notre découpage en modèles métier pour répondre à la problématique de la cohérence globale.

Le pivot contient des informations transverses reconstituées à partir de données extraites des objets offerts (donc informatifs) des Public Models des facettes. Pour effectuer ces tâches d'extraction et de calcul d'informations, nous introduisons le concept de "layer"[6]. Ces layers ont pour but le calcul d'informations transverses, mais aussi la vérification d'invariants/contraintes provenant des exigences et stockés dans le pivot. La figure 1 montre le méta-modèle des layers. Ce méta-modèle indique qu'un layer applique des traitements sur des objets offerts provenant

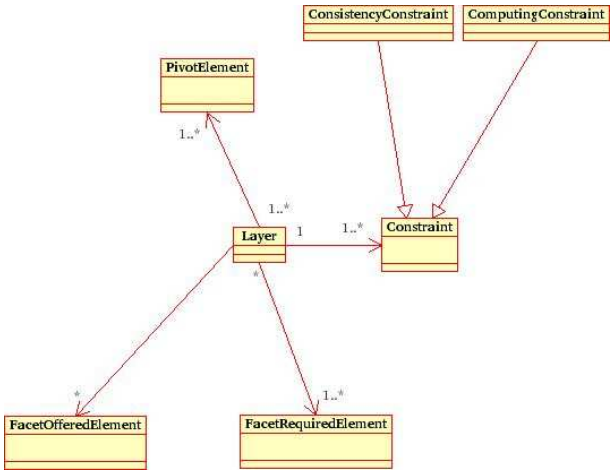

FIG. 1 – Méta-modèle des layer

des facette, de sorte à évaluer des contraintes validant un élément de pivot et/ou des éléments requis d'une facette. En effet un layer s'appuie obligatoirement sur les éléments internes des facettes car eux seuls possèdent de l'information. Il s'applique aussi sur un élément de pivot dans la mesure où, soit il vérifie une exigence système et donc se réfère à un élément du pivot, soit il calcule une information et doit la stocker dans un élément de pivot, ainsi qu'en informer les éléments demandeurs d'autres facettes concernées par les dites informations.

6 Conclusion, perspectives

Nous avons présenté une approche de modélisation distribuée des systèmes embarqués. Au delà des inévitables détails techniques, nous avons tenté de mettre en lumière les points importants inhérents à ce type de modélisation distribuée. Tout d'abord il apparaît que le choix du découpage du système en modèles restreints doit limiter au maximum la redondance d'informations

[6] calque,couche

tout en laissant aux intervenants la liberté d'utiliser les outils de leur choix pour réaliser leurs tâches. L'exploitation du découpage naturel par facettes métier nous semble être un excellent candidat répondant à ces contraintes. Ensuite, quelque soit le découpage choisi, il faudra s'assurer que les modèles soient cohérents entre eux et au regard des spécifications générales (dont une partie sera nécessairement transversale). Pour ce faire nous avons introduit, d'une part un modèle pivot fournissant une vision globale abstraite support des cross-cutting concerns, et d'autre part un ensemble de modèle "actifs" chargé de calculer des données transverses et de vérifier les exigences globales du système.

Le lecteur intéressé pourra trouver une description plus détaillée de nos travaux dans [MTC04] ainsi que dans [TMR+04] qui replace nos travaux dans une dynamique de processus de développement et d'ingéniérie système.

Références

[AT98] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, 1998.

[HL95] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical Report NU-CCS-95-03, College of Computer Science, Northeastern University, Boston, MA, February 1995.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[MGFA02] François Mekerke, Geri Georg, Robert France, and Roger Alexander. Tool support for aspect-oriented design. In J.-M. Bruel and Z. Bellahséne, editors, *Advances in Object-Oriented Information Systems OOIS 2002 Workshops, LNCS 2426*, pages 280–289, September 2002.

[MTC04] François Mekerke, Wolfgang Theurer, and Joël Champeau. Non-functional aspects management for craft-oriented design. In *UML 2004, WS : Models for Non-functional Aspects of Component-Base Software*, 2004.

[OT00] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology : The State of the Art in Software Development*. Kluwer, 2000.

[SY99] Junichi Suzuki and Yoshikazu Yamamoto. Extending uml with aspects : Aspect support in the design phase. In *Proceedings of the third ECOOP Aspect-Oriented Programming Workshop*, 1999.

[TMR+04] Wolfgang Theurer, François Mekerke, Emmanuel Rochefort, Joël Champeau, and Philippe Dhaussy. Vers la gestion de la cohérence dans les processus multi-modèles "métier". In *Congrès AFITEP 2004 : Entreprise, Projet, Intégration*, 2004.

Précision et Validation de Métamodèles avec EMF et OCL

Gilles Vanwormhoudt
ENIC Telecom Lille 1
Université de Lille I - Cité Scientifique
2 rue Marconi 59655 Villeneuve d'Ascq
vanwormhoudt@enic.fr

Mots-clefs – Keywords

métamodélisation, validation de modèles, OCL, EMF
metamodeling, models validation, OCL, EMF

Résumé – Abstract

L'ingénierie du logiciel est entrain d'évoluer vers un nouveau paradigme où les modèles sont utilisés comme élément de première classe. Dans le cadre de cette évolution, il est important que les modèles soient définis de façon la plus précise et la plus cohérente possible. L'emploi de techniques de métamodélisation comme le MOF et de langages formels comme OCL apportent des réponses pour préciser les modèles. Toutefois, il existe très peu d'outils qui combinent ces deux techniques pour offrir de la métamodélisation avec OCL. Dans cet article, nous proposons d'étendre EMF (*Eclipse Modeling Framework*) avec un support OCL. EMF est un ensemble d'outils offrant des capacités de métamodélisation proches de celles du MOF au sein de l'environnement Eclipse. Notre extension permet d'exprimer des contraintes OCL au sein de métamodèles EMF, de vérifier leur cohérence puis de les traduire en code Java pour les évaluer sur des modèles instances. Cette extension devrait être utile aux développeurs de plugins Eclipse ayant recours à EMF pour produire leurs métadonnées et à tous ceux qui veulent spécifier des métamodèles avec OCL et les valider.

Software engineering is evolving toward a new paradigm where models are first-class elements. In the context of this evolution, it is important that models are defined in a precise and coherent way. Metamodeling techniques like MOF and formal languages like OCL are solutions for precise modeling. However, there is currently very few tools that combines these two techniques to support precise metamodeling with OCL. In this paper, we propose to extend the EMF (Eclipse Modeling Framework) approach with OCL capabilities. EMF is a set of development tools that provides a metamodeling approach for the Eclipse environment. Our extension gives the abilities to attach OCL contraints to any EMF metamodel, to verify their coherence and to translate the contraints into Java code for evaluating them on instance models. This work should be useful for developers of Eclipse plugins that exploit EMF to produce their metadatas and for everyone that need to specify metamodels with OCL and validate them.

1 Introduction

L'ingénierie du logiciel est entrain d'évoluer vers un nouveau paradigme où les modèles sont utilisés comme des éléments de première classe. L'approche MDA de l'OMG participe à cette évolution en plaçant les modèles et leur transformation au coeur de la conception et de la production de logiciels. Dans le cadre de cette évolution, il est important que les modèles soient définis de façon la plus précise et la plus cohérente possible. L'emploi de techniques de métamodélisation comme le standard MOF [Gro01] (*Meta Object Facility*) et de langages formels comme OCL (*Object Constraint Langage*) [AJ03] apportent des solutions pour mieux préciser les modèles.

Il existe aujourd'hui des outils basés sur ces deux techniques. Nous trouvons d'une part des outils tels que dMOF [DST00] ou ModFact [Mod] qui permettent de spécifier des métamodèles en conformité avec le MOF et de générer une API d'implantation. D'autre part, nous avons des outils de modélisation comme ArgoUML, OCLE, Use [RG02] qui fournissent un support OCL pour le métamodèle d'UML. Par contre, il y a actuellement très peu d'outils qui combinent ces deux techniques pour offrir une approche de métamodélisation avec OCL. Pourtant, le fait de pouvoir exprimer des contraintes OCL sur un métamodèle correspond à un besoin important comme l'atteste les spécifications de métamodèles standards produites par l'OMG et les nombreux travaux de recherche utilisant OCL pour préciser des métamodèles. Sur l'utilisation d'OCL pour la métamodélisation, les travaux les plus significatifs sont [AO03] et [LO03]. Ces travaux ont montré que le langage OCL n'est pas limité au métamodèle d'UML et qu'il peut être généralisé à d'autres métamodèles comme le MOF. Ils ont donné lieu à des implantations d'OCL qui peuvent être adaptées pour différents métamodèles.

Le travail présenté dans cette article vise à combler le manque d'outils pour métamodéliser avec OCL. Il propose une extension de l'approche EMF (*Eclipse Modeling Framework*) [FT03] avec un support OCL. EMF est un ensemble de plugins qui introduit dans l'environnement Eclipse une approche de développement dirigée par les modèles basée sur une simplification du MOF. EMF permet de définir des métamodèles puis d'en dériver une implantation en Java pour construire des modèles instances.

Notre extension permet d'exprimer des contraintes OCL au sein de métamodèles EMF, de vérifier la cohérence de ces contraintes puis de les traduire en code Java pour les évaluer sur des modèles instances. Cette extension peut intéresser une quantité importante de développeurs Eclipse exploitant EMF pour produire les métadonnées de leurs plugins. Pour ces développeurs, l'extension apporte la possibilité d'ajouter des contraintes à leur métamodèle pour produire automatiquement le code de vérification de ces métadonnées. Cette extension peut également être utile à ceux qui font de la métamodélisation et qui veulent un outil pour définir leur métamodèle avec des contraintes OCL et le valider.

2 Aperçu de l'*Eclipse Modeling Framework*

EMF (Eclipse Modeling Framework) [1] est un ensemble d'outils de développement intégré à l'environnement Eclipse sous forme de plugins. EMF a été conçu afin d'ouvrir Eclipse

[1] http ://www.eclipse.org/emf/

au développement dirigé par les modèles [2]. L'objectif est de pouvoir appliquer cette approche aussi bien pour le développement d'applications métiers que pour la construction et l'intégration de nouveaux plugins.

EMF apporte deux fonctionnalités principales. La première est la définition de modèles à objet constitués de packages, de classes et de liens entre ces classes. Un modèle EMF s'apparente au diagramme de classes d'UML en plus simple. Il peut être élaboré à partir de code Java annoté, de documents XMI issus d'autres modeleurs, d'un éditeur arborescent intégré à EMF ou encore par programmation. La deuxième fonctionnalité fournie par EMF est la transformation d'un modèle EMF en code Java et la génération d'un éditeur arborescent spécifique pour construire des entités conformes à ce modèle.

La structure des modèles EMF est définie par le métamodèle Ecore. Ce métamodèle est conceptuellement proche de celui du MOF [3]. La figure 1 montre la hiérarchie des méta-concepts présents dans le métamodèle Ecore. On y trouve les notions de package (`EPackage`), de classe (`EClass`), de type de données (`EDataType`), d'attribut (`EAttribute`), d'opération (`EOperation`) et de lien de référence (`EReference`). EMF intègre une implantation Java du métamodèle Ecore qui forme le *framework* de modélisation. Ce *framework* sert à représenter les modèles EMF sous forme d'objets ce qui permet leur manipulation par des programmes Java.

EMF offre un support pour faire de la métamodélisation. Il est possible en effet de spécifier un métamodèle sous forme de modèle EMF et d'en produire une implantation Java pour représenter des modèles instances de ce métamodèle et les manipuler. [4] La figure 1 présente également un exemple de métamodèle que nous avons spécifié à l'aide des concepts d'Ecore (indiqués par les ellipses) et qui servira dans le reste de l'article. Le but de ce métamodèle est de modéliser des assemblages de composants. Il introduit les classes `Component`, `Port`, `Required` et `Provided` pour modéliser des composants avec des ports requis et fournis, la classe `Connection` pour relier un port requis à un port fourni et le classe `Assembly` pour décrire l'assemblage dans son ensemble.

La génération proposée par EMF comprend deux parties : la génération du code Java d'implantation d'un modèle EMF et la génération du code Java pour l'éditeur associé. Pour un métamodèle spécifié avec EMF, la génération du code d'implantation est similaire à la génération standard JMI [Pro02] à partir d'un métamodèle MOF. Cette génération produit des interfaces et des classes Java pour représenter des modèles instances sous forme d'objets et les manipuler conformément au métamodèle. Pour chaque package EMF, la génération produit un package d'interfaces et un package de classes implantant ces interfaces. Une interface et une classe d'implantation sont générées pour chaque classe EMF, pour le package EMF lui-même et pour une fabrique.

La génération liée à un modèle est paramétrée à l'aide d'un modèle EMF de génération dont la structure est définie par un métamodèle appelé GenModel. Le modèle de géné-

[2] Il s'agit d'une tendance actuelle des environnements de développent comme l'attestent l'intégration du module MDR (*Metadata Repositorty*) à Netbeans ou encore la dernière version de Visual Studio.

[3] Mais il contient également des différences au niveau de l'organisation des packages et de la modélisation des associations qui sont liées à son orientation vers la production de code plutôt que vers la production de référentiel.

[4] Les concepteurs d'EMF ont mis à profit cette capacité pour autogénérer l'implantation du métamodèle Ecore servant à représenter les modèles EMF et pour produire également un plugin capable de spécifier des modèles conformes au métamodèle d'UML2 (http ://www.eclipse.org/uml2)

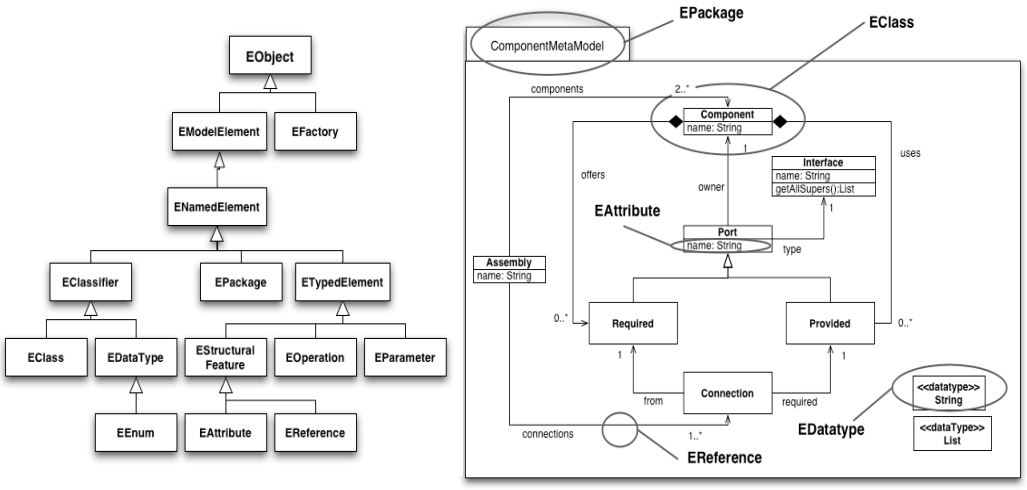

FIG. 1 – Un métamodèle pour des assemblages de composants

ration est construit automatiquement à partir du modèle EMF en associant un élément de paramétrage correspondant à chaque élement du modèle (un élément `GenPackage` est créée pour un élément `EPackage`, un élément `GenClass` pour un élément `EClass`, etc). Les informations détenues par ces éléments de paramétrage concernent différents aspects qui ont un impact sur la génération : règles de nommage, localisation des fichiers générés, mode de visualisation et d'édition, inclusion de fonctionnalités , etc. Cette approche de génération paramétrée par un modèle présente comme avantages de ne pas polluer les éléments de modélisation avec des ingrédients relatifs à la génération et de pouvoir appliquer plusieurs générations différentes à un même modèle.

3 Support OCL dans EMF

3.1 De l'intérêt de ce support

A l'aide des concepts présents dans Ecore, les seules contraintes qui peuvent être exprimées au niveau d'un métamodèle sont le typage des éléments, la composition d'éléments, la cardinalité et l'unicité des liens. En général, ceci n'est pas suffisant car d'autres types de contraintes doivent être pris en compte. Par exemple, pour notre métamodèle dédié à l'assemblage de composants, une contrainte importante au niveau du concept de connection est que le port requis et le port fourni mis en relation soient munis de types compatibles [5]. Si un développeur souhaite mettre en place ce type de contrainte et la vérifier lors de la construction de modèles d'assemblage de composants, la seule possibilité offerte avec la version de base d'EMF est de coder celle-ci de manière adhoc dans le code généré. Ce qui d'une part nécessite un effort de programmation important et d'autre part ne rend pas visible ce type de contrainte au niveau de la spécification.

[5] Cette contrainte peut être facilement exprimée en OCL de la façon suivante :
```
context Connection inv : self.from.type = self.to.type or self.to.type.getAllSupers()->
includes(self.from.type)
```

Ces constats et des besoins de précision de métamodèles dans le cadre de travaux en recherche nous ont conduit à étendre EMF avec un support OCL. Cette extension apporte de nouvelles fonctionnalités au niveau de la définition des modèles EMF mais aussi au niveau de leur transformation en code java. Au niveau définition, des contraintes OCL peuvent être associées à n'importe quel modèle EMF. Les types de contraintes OCL qui sont supportés actuellement sont celles de la spécification 1.3 [6]. Ainsi, une classe peut être enrichie avec des invariants, une opération pouvant l'être avec une précondition et une postcondition. L'association de contraintes OCL aux classes et aux opérations peut être réalisée soit par programmation, soit à l'aide de l'éditeur de modèles Ecore qui a donc été étendu. Au niveau de la génération Java, celle-ci a été étendue de façon à intégrer des comportements de vérification des contraintes sur les modèles instances. Une particularité de notre intégration est de pouvoir contrôler le mode d'évaluation de ces contraintes ainsi que le traitement de réaction à leur violation.

L'intégration du support OCL à EMF a nécessité l'enrichissement des métamodèles Ecore et GenModel et l'introduction de mécanismes pour vérifier et mettre en oeuvre les contraintes. C'est ce que nous allons expliquer dans les deux prochaines sous-sections.

3.2 Spécification et Vérification des Contraintes

Pour pouvoir spécifier des contraintes au niveau de n'importe quel modèle EMF, nous avons procédé à l'enrichissement du métamodèle Ecore. La solution adoptée diffère de celle qui est préconisée dans la spécification 1.4 du MOF pour spécifier des contraintes OCL au niveau des métamodèles. Nous n'avons pas retenu la solution du MOF car celle-ci se révèle trop général en ce qu'elle autorise l'attachement de contraintes sur tous les éléments du modèles, notamment les packages, les associations et même les contraintes. Notre solution ne présente pas cet inconvénient et définit précisemment les éléments qui peuvent recevoir une contrainte OCL tout en conservant des possibilités d'évolution vers OCL 2.0.

L'enrichissement du métamodèle Ecore est schématisé à la figure 2. Par rapport au métamodèle initial, nous avons ajouté deux nouvelles classes : la classe `EConstraint` pour modéliser les contraintes OCL et la classe `EConstrainedElement` qui capture les éléments de modélisation pouvant détenir des contraintes (via le lien `eConstraints`). Cette seconde classe est héritée par la classe `EClass` et la classe `EOperation` car leurs instances peuvent avoir des contraintes. Pour supporter la vérification, nous avons également enrichi deux classes existantes. Nous avons doté la classe EOperation d'un attribut `query` de type booléen afin d'indiquer les opérations qui ont le statut de requête. OCL étant un langage sans effet de bord, seules les opérations requête peuvent être employé dans une expression de contrainte. La class `EDataType` a été munie d'un attribut `oclDataType` afin de mettre en correspondance les types de données EMF spécifiques à un métamodèle et les types de données OCL. Enfin, nous avons introduit deux nouveaux types énumérés : le type `OclConstraintType` définissant les types de contraintes possibles (valeurs `inv, pre et post`) et le type `OclDataType` correspondant aux types prédéfinis d'OCL (valeurs `OclInteger, OclReal, OclAny, OclVoid, OclBag, OclSet, etc`). Tous ces enrichissements ont été incorporés à l'implantation du métamodèle Ecore en suivant les même principes que pour son autogénération. Nous avons enrichi le modèle EMF spéci-

[6] Le support d'OCL 2.0 est prévu dans une future version

fiant le métamodèle ECore puis nous avons regénéré le code Java correspondant à son implantation ainsi que l'éditeur de modèles associé.

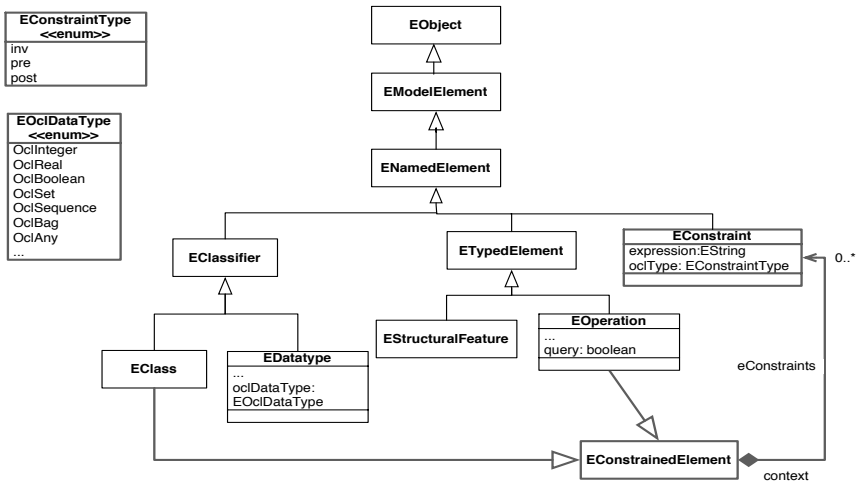

FIG. 2 – Le métamodèle Ecore étendu pour la spécification des contraintes

La cohérence des contraintes ajoutées à un modèle EMF est systématiquement vérifiée par notre extension. Deux vérifications sont appliquées : une vérification syntaxique et une vérification sémantique des expressions OCL. Cette seconde vérification assure que les éléments du modèle référencés dans l'expression d'un contrainte OCL existent et sont du bon type. Nous nous sommes appuyés sur la boite à outils OCL [HF00] développée à l'université de Dresden pour mettre en oeuvre ces vérifications. Cette boite à outils offre un ensemble de modules pour vérifier, normaliser des contraintes OCL mais aussi les mettre en oeuvre en Java. L'intérêt de cette boite à outils vient du fait qu'il est possible de faire fonctionner ses modules sur tout type de modèle implanté en Java, moyennant le respect d'un protocole donné. Dans notre cas, nous avons enrichi une partie des classes implantant le métamodèle Ecore (`EPackage, EClass, EStructuralFeature, EOperation, EReference, ...`) pour qu'elles respectent ce protocole et fournissent toutes les informations de typages nécessaires aux vérifications.

3.3 Mise en oeuvre et Activation des Contraintes

L'évaluation des contraintes requiert leur traduction en code Java exécutable et l'intégration de ce code dans celui implantant le métamodèle. Avant de préciser ces deux opérations, nous allons commencer par expliquer les deux moyens de contrôle que nous avons introduit pour offrir plus de flexibilité dans l'évaluation d'une contrainte. Il s'agit du mode d'évaluation et du mode de réaction à une violation.

Le mode d'évaluation est inspiré de la spécification du MOF [7]. Ce mode qui concerne principalement les invariants peut prendre deux valeurs : *immediate* ou *differed*. Le mode *immediate* signifie que l'invariant doit être vérifiée à chaque modification d'état de l'objet contraint et plus généralement du modèle. Le mode *differed* signifie que la vérification

[7]Dans cette spécification, la classe *Constraint* possède un attribut *evaluationPolicy* qui explicite le mode d'évaluation de la contrainte

de l'invariant pourra être déclenchée à n'importe quel moment, sur action de l'utilisateur par exemple. Pouvoir choisir entre ces deux modes nous parait important. En effet, pour certains invariants, le mode *differed* s'avère mieux adapté. Un exemple est l'invariant qui valide la connexion entre deux composants. Sa vérification ne doit être réalisée que si les deux ports sont reliés.

Le mode de réaction à une violation indique ce qu'il faut faire si une contrainte échoue. Trois modes sont prévus : *inform, action, restore*. Le mode *inform* qui est le plus simple notifie l'utilisateur de la violation. Le second mode *action* donne la possiblité de spécifier une opération qui sera invoquée pour remettre le modèle dans un état cohérent. Ce mode est le mieux adapté dans le cas où le modèle est manipulé par un outil plutôt qu'un utilisateur. Le dernier mode *restore* a pour but de ramener le modèle dans un état précédent la violation d'une contrainte. Pour l'instant, ce mode ne traite que des cas simples comme la modification d'une valeur d'attribut ou d'un lien ayant entraînée une violation d'invariant.

L'intégration de ces deux modes a été réalisée en enrichissant le métamodèle GenModel [8] . Le principale ajout à ce métamodèle est l'introduction d'une classe `GenConstraint` pour représenter les paramètres d'évaluation d'une contrainte : son mode d'évaluation, son mode de traitement des violations et le nom de l'opération à exécuter dans le mode "action". En plus de ces enrichissements, nous avons modifié le processus qui crée automatiquement le modèle de génération à partir d'un (méta)modèle muni de contraintes. Pour chaque contrainte, nous procédons à l'ajout d' un élément GenConstraint dans le modèle de génération résultat, autorisant ainsi le paramétrage de son évaluation.

A partir d'un modèle de génération étendu avec les paramètres d'évaluation, notre extension produit une implantation Java du métamodèle qui est complétée avec l'implantation des contraintes. Lors de la génération, chaque contrainte associée à une classe EMF ou à une opération EMF donne lieu à une méthode dans la classe Java correspondante. Le corps de ces méthodes est obtenu en faisant appel au module de génération de la boite à outils OCL. La partie gauche de la figure 3 donne en exemple le code de la méthode inv1 générée pour l'invariant considéré précédemment. Ce code est constitué d'appels à la librairie OCL inclut dans la boite à outils. Cette librairie fournit des classes implantant les types OCL prédéfinis et une classe `OclAnyImpl` que nous avons du spécialiser pour donner accès aux caractéristiques des objets représentant les éléments du modèle instance.

En plus des méthodes implantant les contraintes, des portions de code supplémentaires destinées à activer les contraintes sont également insérées au sein de la même classe. L'activation des invariants est séparée dans trois méthodes générées automatiquement : une méthode `immediateInvariantsCheck` pour activer les invariants immédiats , une méthode `differedInvariantsCheck` pour activer les invariants différés et une méthode `allInvariantsCheck` pour activer l'ensemble (cf la partie gauche de la fig 3). La méthode `immediateInvariantsCheck` est appelée dans chaque méthode générée pour la classe qui conduit à un changement d'état de l'objet. La méthode `allInvariantsCheck` est destinée à être invoquée lors de vérifications globale sur un modèle instance. Cette méthode est définie de façon à invoquer les méthodes `immediateInvariantsCheck` et `differedInvariantsCheck` puis à propager récursivement la vérification des invariants

[8] Une autre solution aurait été d'intégrer ces éléments au niveau du métamodèle Ecore comme le préconise la spécification du MOF. Nous avons plutôt privilégié une approche dans laquelle la description des métamodèles reste indépendante d'une exploitation particulière (transformation, génération, ...).

FIG. 3 – Activation des contraintes

aux sous-éléments.

L'activation des préconditions et des postconditions est accomplie dans le squelette des méthodes générées pour les opérations EMF. Un tel squelette est généré pour réaliser plusieurs appels de méthodes qui sont dans l'ordre : un appel à la méthode implantant la précondition si elle existe, un appel à la méthode implantant le corps de l'opération, un appel à la postcondition si elle existe et enfin un appel à la méthode `immediateInvariantsCheck` si l'opération n'est pas une requête. Ce schéma de génération qui est montrée dans la partie droite de la figure 3 garantit une vérification systématique des contraintes avant et après traitement. Pour terminer, il convient de préciser que toutes les activations des contraintes sont enrobées avec du code dédié au traitement de leur violation.

L'implantation du métamodèle qui est finalement obtenu permet de construire et de manipuler des modèles instances qui sont vérifiables en activant les méthodes décrites précédemment. Ces activations peuvent se faire par programme [9] mais nous avons également étendu la génération d'éditeur de modèles pour que ces possibilités d'activations soient offertes à l'utilisateur.

4 Conclusion

Dans cette article, nous avons présenté une extension de l'approche EMF avec un support OCL. L'outil générique qui en résulte vient compléter la panoplie des outils existants autour d'OCL et du MOF [RG02]. A notre connaissance, cet outil est l'un des seuls permettant de spécifier des contraintes OCL sur des métamodèles et a en proposer une évaluation sur les modèles instances.

Une expérimentation que nous avons pu réalisée grâce à l'extension est la vérification de contraintes OCL sur le métamodèle UML2 ayant servi à générer le plugin correspondant.

[9] et peuvent donc être exploité au sein d'outils pour valider les modèles manipulés.

Dans le cadre de travaux de recherche, nous avons proposé une formulation OCL qui précise la sémantique de liaison de template UML2 [OG04]. Cette formulation qui comporte des contraintes relativement complexes a pu être vérifiée grâce à l'extension. Nous avons également pu regenérer une nouvelle version du plugin UML2 qui incorpore une fonctionnalité de vérification des liaisons des templates.

Dans le futur, nous comptons faire évoluer cette extension vers la spécificiation 2.0 d'OCL. Notre intention est d'y intégrer les nouveaux types de contraintes apportés par cette version (`init, def, body`) afin de les utiliser dans la spécification des modèles et dans la génération de code.

Enfin, une autre amélioration envisagée pour cette extension se situe au niveau de l'évaluation des invariants dont les principes sont similaires aux approches d'assertions exécutables en Java comme JML [CL02] ou jContractor [KHB99]. Similairement à ces approches, notre évaluation des invariants est pour l'instant limitée à l'objet courant. Or un changement d'état d'un objet peut tout à fait invalider un invariant relatif à un autre objet. Actuellement, le seul moyen que nous proposons pour détecter ces violations d'invariant consiste à appliquer manuellement une vérification globale sur le modèle. Afin d'apporter une solution à ce problème, nous allons examiner des techniques d'évaluation plus sophistiquées prenant en compte les assertions de façon globale. Ce problème rejoint celui de l'utilisation des assertions pour gérer des contraintes d'intégrité [Col01]. Les techniques étudiées dans ce cadre nous semble une piste intéressante à explorer.

Références

[AJ03] A.Kleppe and J.Warmer. *The Object Constraint Language, Getting your Model Ready for MDA*. Addison-Wesley, 2003.

[AO03] D. Akehurst and O.Patrascoiu. OCL 2.0 - Implementing the Standard for Multiple Metamodels. Proc. of OCL Workshop, 6th Int. Conference on the Unified Modelling Language (UML'2003), 2003.

[CL02] Y. Cheon and G. T. Leavens. A Runtime Assertion Checker for the Java Modeling Language (JML). Int. Conference on Software Engineering Research and Practice (SERP '02), 2002.

[Col01] P. Collet. Fiabilité des systèmes à objets persistants. Langage et Modèles à Objets (LMO'01), 2001.

[DST00] DSTC. dmof users guide. DSTC report, http ://www.dstc.edu.au, 2000.

[FT03] E. Merks R.Ellersick F.Budinsky, D. Steinberg and T.Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[Gro01] Object Management Group. Meta Object Facility version 1.3, standard. 2001.

[HF00] B.Demuth H.Hussmann and F. Finger. Modular architecture for a toolset supporting OCL. Proc. 3rd Int. Conference Unified Modeling Language, LNCS 1939, Springer, 2000.

[KHB99] M. Karaorman, U. Hölzle, and J. Bruno. jContractor : A reflective Java library to support design by contract. Proc. of Meta-Level Architectures and Reflection, LNCS 1616, 1999.

[LO03] Sten Loecher and Stefan Ocke. A Metamodel-Based OCL-Compiler for UML and MOF. OCL Workshop Proceedings, 6th Int. Conference Unified Modelling Language (UML'2003), 2003.

[Mod] ModFact. http ://modfact.lip6.fr.

[OG04] A.Muller O.Caron, B.Carré and G.Vanwormhoudt. An OCL Formulation of UML2 Template Binding. Proc. 7th Int. Conference Unified Modeling Language, LNCS 3273, Springer, 2004.

[Pro02] Java Community Process. Java Metadata Inferface (JMI) Specification. 2002.

[RG02] Mark Richters and Martin Gogolla. Ocl : Syntax, semantics, and tools. Object Modeling with the Object Constraint Language, LNCS 2263, Springer, 2002.

Position de l'héritage multiple dans l'IDM - Transformation de hiérarchies à héritage multiple en hiérarchies à héritage simple

Aurélien Moreau
France Télécom R&D MAPS/AMS
38-40 rue du général Leclerc, 92794 Issy les Moulineaux Cedex 9, France
a.moreau@rd.francetelecom.com

Mots-clefs – Keywords

MDA, IDM, héritage multiple, héritage, UML, transformation de modèle
MDA, MDE, multiple inheritance, inheritance, UML, model transformation

Résumé – Abstract

L'utilisation de l'héritage, multiple ou simple, est toujours l'objet de discussion dans la communauté de l'objet. Nous présentons ici les objectifs et les pistes de cette thèse qui se propose d'étudier l'utilisation des hiérarchies de classes utilisant l'héritage multiple dans une démarche conforme à l'Ingénierie Dirigée par les Modèles (IDM, ou MDE en anglais ou encore MDA) dans l'objectif de générer du code ne supportant que l'héritage simple (par exemple pour Java). Nous tenterons de clarifier pour ceci les concepts manipulés et les différents choix stratégiques qui s'offrent à nous.
The use of multiple or single inheritance is a widely discussed issue among the object community. We introduce here some objectives and tracks of this thesis which focuses on the use of multiple inheritance in a Model Driver Engineering (MDE) framework allowing single inheritance code generation (as in Java). We will try to clarify concepts and present various strategies related to this concern.

1 Introduction

Les processus de développement de logiciels évoluent vers une plus grande abstraction et une réduction de la quantité d'informations à fournir pour obtenir un programme fonctionnel. Après l'utilisation de langages de haut niveau, une nouvelle évolution s'effectue, celle-ci tend à automatiser la généreration des applications à partir de modèles (vues structurelles et comportementales du système décrit).

Récemment l'OMG a proposé une base de travail dénommé Model Driven Architecture (MDA [MM03]), bien que le nom Model Driven Engineering ou « Ingénierie Dirigé par les Modèles » (IDM) serait plus adapté. Celle-ci tente de définir un cadre à la génération de code à partir de modèles. Ces abstractions sont décrites en UML et sont spécialisées de façon incrémentale par transformation pour tendre vers un système opérationnel.

Ces transformations successives soulèvent le problème de la prise en charge par l'IDM de hiérarchies de classes utilisant de l'héritage multiple. Il s'agit de pouvoir utiliser son pouvoir expressif dans la phase d'analyse/conception tout en employant dans la phase d'implémentation des langages de programmation ne supportant pas forcément cette capacité.

Nous préciserons dans un premier temps la notion d'héritage dans un modèle objet pour nous intéresser ensuite à sa signification dans un modèle UML et dans les langages à objets. Puis nous comparerons l'héritage simple à l'héritage multiple. Ensuite, nous discuterons des différentes positions et stratégies qu'il est possible d'adopter face à ce problème. Finalement nous conclurons sur les perspectives et les travaux qui doivent être effectués.

2 L'héritage

Le terme d'« héritage » peut être utilisé de différentes manières, nous allons poser quelques définitions informelles. Tout d'abord, une classe n'est pas forcément un type [AC96], les deux notions peuvent être séparées selon le modèle objet adopté. À partir de ce constat nous pouvons différencier deux relations possibles entre classes : une relation de spécialisation (sous-classe), et une relation de sous-typage.

Une sous-classe est une classe construite incrémentalement par l'ajout de méthodes et d'attributs à une ou plusieurs classes mères. Les attributs sont implicitement répliqués, ainsi que le code des méthodes si elles ne sont pas réécrites (surchargées) c'est-à-dire remplacées par une méthode de même signature. L'héritage est le mécanisme qui permet la réutilisation des attributs et du code de la super-classe.

Une relation de sous-typage permet de créer un treillis de types rendant possible l'utilisation d'un type par un autre dans un langage fortement typé , cette propriété se nomme la substituabilité [Duc02]. Sur cette base conceptuelle, il peut exister un lien plus ou moins fort entre ces deux notions.

L'héritage multiple est la possibilité d'hériter des attributs et méthodes de plusieurs super-classes en même temps. On dit qu'il y a conflit lorsqu'on hérite de deux attributs ou valeurs provenant de deux super-classes différentes et portant le même nom mais n'ayant pas la même définition. On peut différencier deux types de conflits [DHH+95] :

conflits de noms : Lorsque les deux super-classes n'ont pas d'ancêtre commun, *a priori* les propriétés entrant en conflit sont différentes (sémantiques différentes) malgré leur nom similaire.

conflits de valeurs : Lorsque les deux super-classes ont un ancêtre commun, *a priori* la propriété est unique (sémantiques identiques) mais présente plusieurs valeurs possibles.

2.1 Dans UML

Nous nous servons du métamodèle UML pour décrire nos modèles, celui-ci est semi-formel : une partie est définie par un langage bien formé et une autre dans un langage naturel. Nous proposons d'étudier brièvement la signification de l'héritage dans ce formalisme.

La norme UML2 [OMG03] définit la relation de généralisation comme *une relation de taxinomie (classification) entre un élément général et un plus spécifique. L'élément plus spécifique est entièrement consistant avec sa généralisation et peut contenir d'avantage d'informations. De plus une instance de la classe plus spécifique peut être utilisée à chaque emplacement faisant référence à son généralisé.* La première partie de la définition fait référence à une relation de spécialisation.

La seconde partie fait référence à une relation de sous-typage, nous pouvons de plus noter qu'une classe est un type dans le métamodèle UML. Le métamodèle implémente donc une relation « la spécialisation est du sous-typage » (lien fort entre les deux concepts).

La sémantique comportementale d'UML comporte différents points de variation [OMG03], ceux-ci sont des parties de la sémantique qui peuvent être changées sans répercution sur le modèle. L'un d'eux donne une résolution de l'héritage multiple. Celle-ci est assez simple, le modèle adopte la seule et unique définition accessible ou est dit mal formé en présence d'un conflit.

2.2 Dans les langages à objets

Dans les langages à objets, plusieurs approches de résolution de conflits d'héritage multiple sont envisageables. Par exemple C++ propose un mécanisme peu satisfaisant de résolution par opérateurs de portée et duplication des valeurs. Eiffel quant à lui demande clairement ce que le concepteur désire et offre ainsi une solution plus élégante. Toutefois ce langage est moins répandu [DEMN98]. Java [GJGB00], par souci de simplicité, ne propose pas d'héritage multiple mais préserve le sous-typage multiple par un système d'interfaces qui constitue une relation plus lâche dite « spécialisation implique sous-typage ». Il s'agit certainement d'un des langages à objets le plus utilisé aujourd'hui.

2.3 Pourquoi faire multiple quand on peut faire simple ?

Des études tendent à prouver que pour un même ensemble d'objets, une classification supportant la classification multiple est plus ordonnée (mesure d'entropie) qu'une classification ne supportant que l'héritage simple [OG97]. En représentation des connaissances, l'héritage multiple est aussi utilisé, entres autres, pour l'expression des points de vue [Dek94]. Ceci montre qu'il est assez « naturel » de définir une hiérarchie de classes de

cette manière, facilitant le travail du concepteur lors de la phase d'analyse et la relecture du modèle.

Nous pouvons donc trouver d'un côté des modèles à héritage multiple, et de l'autre des langages ne le supportant pas (ou demandant des précisions). L'IDM doit donc pouvoir apporter une ou plusieurs solutions à ces problèmes.

3 Les positions faces aux problèmes

3.1 A quel moment appliquer la transformation ?

L'IDM génère du code à partir de modèles en lui appliquant une série de transformations. Une RFP nommée RFP MOF2.0 Query/Views/Transformations a pour but de définir un langage de transformation de modèle à modèle, TRL [Sri03] est l'une des réponses. À chaque étape un modèle indépendant de la plateforme (PIM) est fusionné avec des informations extérieures pour donner un modèle dépendant (PSM) [MM03].

La place de la transformation liée à l'héritage au sein du processus de développement peut avoir une grande importance. Il nous faudra déterminer où situer notre intervention dans la chaîne classique : Descriptions des exigences, Analyse, Conception, Implantation, Tests. Les possibilités sont multiples : analyse → analyse, analyse → conception, conception → conception, conception → code ; l'analyse et la conception pouvant avoir chacun son modèle.

Cette place dans la chaîne sous-entend une quantité plus ou moins importante d'informations disponibles, comme des renseignements sur le comportement à l'exécution présents dans un modèle de conception mais pas d'analyse. Il nous faudra tirer les avantages et inconvénients de chacune de ces approches.

3.2 Quelle transformation appliquer ?

Définissons tout d'abord nos objectifs (en partie repris de [CMR02]) :
- Maintenir le plus possible la hiérarchie d'origine, permettant au concepteur de comprendre le résultat produit.
- Effectuer le maximum de modifications « locales », c'est-à-dire ayant le moins de répercutions possibles sur le code éventuellement déjà présent et sur les relations des classes non concernées.
- Réduire le plus possible les duplications et répétitions afin d'obtenir en fin de processus un code le plus lisible possible et le plus facile à maintenir.
- Permettre une remontée aisée des modifications apportées au code généré, et éviter les problèmes de cohérence.

Deux approches ont pour l'instant été envisagées. La première dite « combinatoire » [Rou02] coupe des liens d'héritages en tentant d'optimiser une métrique définit sur les hiérarchies (favorisant leur factorisation). Ne s'intéressant qu'à la structure, elle peut s'appliquer sans information supplémentaire du concepteur. Cette opération s'inscrit bien dans une vision « conception → conception » ou « analyse → conception ».

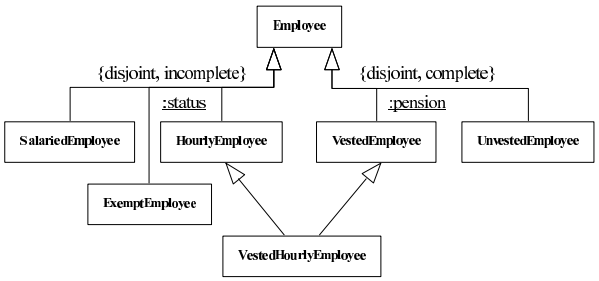

FIG. 1 – Un diagramme annoté

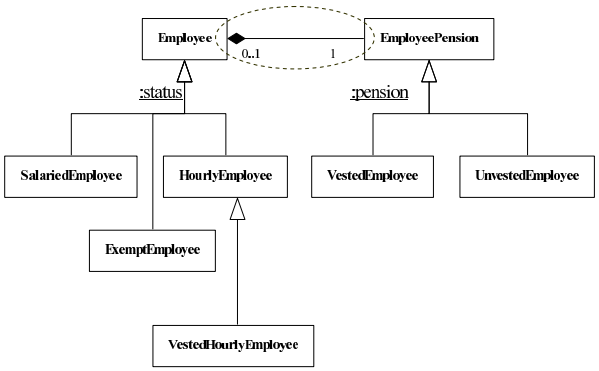

FIG. 2 – Une transformation possible

La seconde, vers laquelle nous nous dirigeons est dite « sémantique » [DHL+04]. Elle tente d'enrichir le métamodèle UML par des annotations porteuses de sens. UML2 propose déjà quatre marqueurs [OMG03] :*overlapping, disjoint, complete, incomplete* et la possibilité de regrouper différents liens de spécialisation sous un discriminant (« discriminator »). Il est de plus possible d'étiqueter un ensemble de liens grâce à un discriminant (« discriminator »). En ajoutant des marqueurs [DHL+04] nous espérons pouvoir repérer dans les hiérarchies de classes des situations typiques permettant d'associer à chaque sous-structure un traitement spécialisé. Pour cela, plusieurs techniques de substitution ont été répertoriées [TKH99, CMR02, DHL+04, VTB98].

La figure 1 présente un diagramme de classes annoté avec les marqueurs UML disponibles.
- `{disjoint, incomplete}` : une sous-classe d'`Employee` est sous-classe d'au plus une classe de l'ensemble portant le discriminant `:status`.
- `{disjoint, complete}` : une sous-classe d'`Employee` est sous-classe d'une et une seule classe de l'ensemble portant le discriminant `:pension`.

La figure 2 présente une transformation possible (appelée « Role aggregation » dans [DHL+04]) préférable lorsqu'on considère le discriminant `:status` comme plus important.

Notre but est d'étendre ou de modifier les marqueurs déjà proposés [DHL+04], en fonction des situations typiques que nous pourrons relever dans un corpus de modèles que nous constituons. Nous devons aussi trouver de nouvelles transformations adaptées. Une vision hybride combinant l'approche combinatoire et sémantique pourra sûrement être envisagée par la suite.

4 Conclusion

On peut noter d'autres pistes comme l'utilisation des modèles dynamiques d'UML pour en tirer de l'information supplémentaire. Si nous nous orientons sur la génération de code, les différentes techniques devront être explorées (insertion de code dans les modèles ou génération de patrons).

Nous nous sommes efforcés dans cet article de poser les bases de vocabulaire dont nous avons besoin pour continuer notre réflexion et pour comprendre les entités manipulées. De plus nous avons exposé les questions auxquelles nous devrons répondre afin de nous positionner correctement par la suite.

Références

[AC96] M. Abadi and L. Cardelli. *A theory of objects (Monographs in Computer Science)*. Springer-Verlag, hardcover edition, 1996.

[CMR02] Y. Crespo, J.-M. Marquès, and JJ Rodriguez. On the Translation of Multiple Inheritance Hierarchies into Single Inheritance Hierarchies. In Black, Ernst, Grogono, and Sakkinen, editors, *Proceedings of the Inheritance Workshop at ECOOP 2002*, pages 30–37, 2002.

[Dek94] L. Dekker. *FROME : Représentation multiple et classification d'objets avec points de vue*. PhD thesis, Université des Sciences et Technologies de Lille, Laboratoire d'Informatique Fondamentale de Lille, Juin 1994.

[DEMN98] R. Ducournau, J. Euzenat, G. Masini, and A. Napilo, editors. *Langages et modèles à objets : état et perspectives de la recherche*. Number 19 in Didactique. INRIA, Rocquencourt (FR), 1998.

[DHH+95] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, and A. Napoli. Le point sur l'héritage multiple. *Technique et Science Informatiques*, 14(3) :309–345, 1995.

[DHL+04] M. Dao, M. Huchard, T. Libourel, A. Pons, and J. Villerd. Proposals for Multiple to Single Inheritance Transformation. In *MASPEGHI'04 Workshop on Managing SPEcialization/Generalization Hierarchies*, pages 21–26, 2004.

[Duc02] R. Ducournau. Spécialisation et sous-typage : thème et variations. In *TSI : Technique et Science Informatique*, volume 21, pages 1305–1342, 2002.

[GJGB00] J. Gosling, B. Joy, Steele G., and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2 edition, 2000.

[MM03] J. Miller and J. Mukerji. *MDA Guide*. Object Management Group, Inc., june 2003. Version 1.0.1.

[OG97] H. Ouaggag and R. Godin. étude empirique de l'influence de l'héritage multiple sur l'entropie conceptuelle : comparaison avec l'héritage simple. In *Actes de Langages et Modèles à Objets (LMO'97)*, pages 161–174, 1997.

[OMG03] OMG. *UML 2.0 Infrastructure Specification*, 2003.

[Rou02] C. Roume. Going from Multiple to Single Inheritance with Metrics. In *Workshop (in ECOOP 2002) on Quantitative Approaches in Object-Oriented Software Engineering*, pages 126–136, june 2002.

[Sri03] P. Sriplakich. Techniques des transformations de modèles basées sur la métamodélisation. Master's thesis, Université Pierre et Marie Curie, 2003.

[TKH99] K. Thirunarayan, G. Kniesel, and H. Hampapuram. Simulating Multiple Inheritance and Generics in Java. *Comput. Lang.*, 25(4) :189–210, 1999.

[VTB98] J. Viega, B. Tutt, and R. Behrends. Automated Delegation is a Viable Alternative to Multiple Inheritance in Class Based Languages. Technical Report CS-98-03, 2, 1998.

Approche semi-formelle pour l'adaptation dynamique de composants

Audrey Occello (1), Anne-Marie Dery-Pinna (1)
(1) Laboratoire I3S - Université de Nice-Sophia Antipolis
930, Route des Colles, B.P. 145,
06903 Sophia Antipolis cedex, France
{occello, pinna}@essi.fr

Mots-clefs – Keywords

UML, OCL, simulation, sûreté de fonctionnement, adaptation de composants
UML, OCL, simulation, safety, component adaptation

Résumé – Abstract

Les modèles à composants prennent de plus en plus souvent en charge les adaptations dynamiques. Néanmoins, peu de plates-formes à composants fournissent les outils permettant de garantir la sûreté des adaptations qu'ils prennent en charge. Notre proposition pour rendre le processus d'adaptation plus sûr consiste à définir des propriétés de sûreté que les composants et les adaptations doivent respecter. Pour valider cette approche, il nous faut prouver formellement que ces propriétés de sûreté sont correctes. Cet article présente une première tentive pragmatique pour modéliser, formaliser et valider la sûreté des adaptations dynamiques dans les modèles à composants.

Component-based programming enables ever more runtime adaptations of applications. However, few component models provide tools to guarantee the safety of the adaptations they support. Our proposition to making the adaptation process safer is to define safety properties that components and adaptations must fulfill. To validate this approach, we need to formalize and validate these properties. This paper present a first pragmatic attempt to model, formalize and validate the safety of adaptations in components models.

1 Introduction

Les modèles à composants [OMGa, PSDF01, BCS02, BFCE+04, AP03] prennent de plus en plus souvent en charge les adaptations dynamiques permettant ainsi une évolution de plus en plus aisée des applications. Or, les modifications liées aux adaptations affaiblissent fréquemment la sûreté des applications. Pour vérifier la sûreté des adaptations, les modèles à composants existants fournissent essentiellement des solutions ad hoc dont la mise en œuvre est souvent informelle ou à la charge du programmeur.

Pouvoir aborder la problématique de la sûreté des adaptations d'applications à base de composants de façon générale en exhibant les propriétés de sûreté que doivent satisfaire les adaptations, nous a paru être un axe de recherche important. Cependant toute proposition doit s'appuyer sur des bases formelles solides afin de ne pas engendrer de problèmes de fiabilité supplémentaires.

Cet article décrit une première tentive pragmatique pour modéliser, formaliser et valider la sûreté des adaptations dynamiques de composants. La section 2 présente notre modèle UML de sûreté des adaptations et décrit la formalisation de propriétés de sûreté en OCL. La section 3 illustre notre méthodologie sur un exemple d'application adaptable dynamiquement. La section 4 montre comment améliorer le degré de confiance dans la spécification UML/OCL obtenue par simulation. Enfin, la section 5 conclut sur les limites et perspectives de ce travail.

2 Modèle de Sûreté des Adaptations avec UML et OCL

Le modèle Satin [ODP04] repose sur une description explicite des adaptations de composants et sur un ensemble de propriétés de sûreté permettant de déterminer à l'avance si une adaptation est sûre ou non. L'idée clé de l'approche est de proposer une solution indépendante des plates-formes technologiques afin de valider en une fois la complétude des contraintes et leur adéquation pour ces propriétés puis de faire profiter les différents modèles à composants ne vérifiant pas actuellement les adaptations qu'ils prennent en charge, par projection, de la sûreté vérifiée par le modèle Satin.

Un schéma d'adaptation est un ensemble de règles (les modifications à effectuer) définies sur un ensemble de rôles (les paramètres du schéma). Ainsi, les schémas d'adaptation décrivent les effets des adaptations sur la structure et le comportement des composants (on parle ici d'instances, d'unités d'exécution). Schématiquement, un rôle décrit ce que fournit et ce que requiert un composant ainsi que les adaptations appliquées au composant.

Adapter un ensemble de composants consiste à appliquer les règles d'un schéma d'adaptation à ces composants et a pour effet de modifier les composants et leurs rôles (modification comportementale, nouvelles fonctionnalités, création/modification d'assemblages de composants). De par l'évolution des rôles, nous devons assurer que les fonctionnalités initiales d'un composant ne sont pas supprimées (propriété de sûreté P_1) et que le comportement attendu des composants est conservé (propriété P_3). D'autre part, adapter un composant répétitivement peut introduire des conflits. C'est pourquoi nous devons assurer que de multiples adaptations d'un composant sont composées de manière cohérente[1] (propriété

[1]Par exemple, deux adaptations consistant à lever des exceptions différentes ne peuvent pas être

P_4). Enfin, les assemblages de composants étant modifiés par adaptation, nous devons garantir que la consistance des assemblages est préservée (propriété P_2) et être prévenus lorsque des cycles (boucles dans le flow d'appel de méthodes) apparaissent (propriété P_5).

L'approche adoptée pour formaliser ces propriétés de sûreté (P_1 à P_5) consiste à décrire des contraintes en OCL [WK99]. Ces contraintes correspondent à un ensemble de pré-conditions sur les méthodes et les classes du modèle Satin. Le tableau ci-dessous résume comment chaque propriété est assurée et sur quels éléments du modèle.

Propriété	**Contraintes**	**Eléments et opérations du modèle concernés**
P_1	C_1, C_5, C_9	Template.create, AdaptationPattern.instantiate
P_2	C_6, C_{10}, C_{11}	Component.replace ,AdaptationPattern.instantiate, AdaptationInstance.remove
P_3	C_2	Template.create
P_4	C_3, C_7	AdaptationPattern.create, AdaptationPattern.instantiate
P_5	C_4, C_8	AdaptationPattern.create, AdaptationPattern.instantiate

A titre d'exemple, nous détaillons dans ce papier les contraintes associées à la propriété P_2 en langage naturel puis en OCL.

- C_6 : Pour pouvoir appliquer un schéma d'adaptation à un ensemble de composants, il faut que chaque composant impliqué présente un rôle qui soit conforme au rôle du paramètre du schéma auquel il est associé.
- C_{10} : Une instance d'adaptation créée à partir d'un schéma d'adaptation comportant une règle d'ajout (de rôle ou de port) ne peut être détruite tant qu'il existe d'autres instances d'adaptation créées à partir de schémas comportant une règle d'adaptation impliquant le même composant et son port ajouté en question.
- C_{11} : Pour pouvoir remplacer un composant c par un composant c' dans un assemblage, le rôle de c' doit être sous-rôle du rôle de c.

Dans la contrainte C_6, l'opération `canPlayRole` se réfère à une relation de filtrage basée sur la correspondance de nom de méthode (`getX` correspond à `get*`) et sur un sous-typage covariant [Duc02].

```
context AdaptationPattern::instantiate(components : Sequence(Component)) :
AdaptationInstance
    pre C6 : Sequence{1..components->size}->forAll(index : Integer |
             components->at(index).canPlayRole(self.parameters->at(index)))
```

Dans la contrainte C_{10}, l'opération `containsDependancies` vérifie si un port ajouté d'une instance d'adaptation est utilisé ou adapté dans une autre instance d'adaptation. Il est impossible de supprimer l'instance d'adaptation tant qu'il y a des dépendances.

```
context AdaptationInstance::remove()
    pre C10: self.pattern.rules->forAll(ru | not self.containsDependancies(ru))
```

Dans la contrainte C_{11}, l'opération `isSubRoleOf`) implémente une relation de sous-typage, par défaut contravariant, mais d'autres formes de sous-typage peuvent être utilisées selon les besoins.

```
context Component::replace(cNew : Component)
    pre C11: self.roles->forAll(r1 | cNew.roles->exists(r2 | r1.isSubRoleOf(r2)))
```

composées car les deux exceptions ne pourraient être lancées simultanément.

3 Exemple d'utilisation du modèle de sûreté : des agendas collaboratifs

Considérons deux composants représentant des agendas (Figure 1) créés à partir de la même fabrique, `lizDiary` et `johnDiary`, et leur rôle associé, `BasicDiary`. Les ports fournis par `BasicDiary` sont : `addRdv, removeRdv, getRdv, isFree` permettant respectivement d'ajouter, de supprimer et de consulter des rendez-vous.

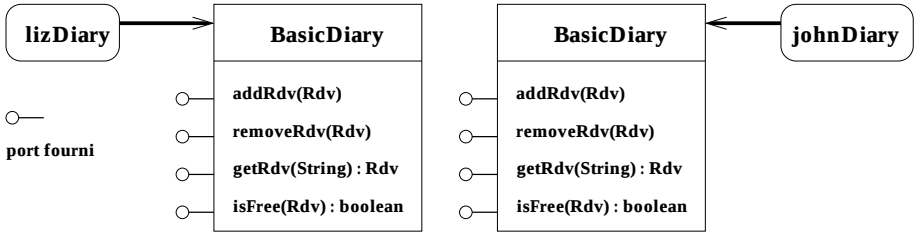

FIG. 1 – `lizDiary` et `johnDiary`

Supposons que l'on veuille ajouter une nouvelle fonctionnalité pour débuguer l'application d'agendas. Pour cela, nous utilisons le schéma d'adaptation `errorManagement` dont la règle spécifie comment ajouter un port `printError` à n'importe quel type de composants. En appliquant ce schéma d'adaptation à `lizDiary`, `printError` est ajouté au rôle de `lizDiary` en tant que port fourni[2] (Figure 2).

```
AdaptationPattern errorManagement(Any aComponent) {
   newPort aComponent.printError(Str s) -> throw exception(s) }
```

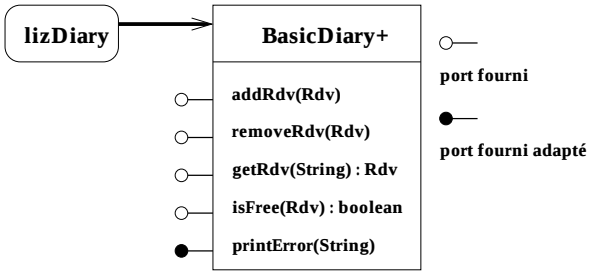

FIG. 2 – Résultat de l'application du schéma `errorManagement` à `lizDiary`

A présent, supposons que John veuille collaborer avec Liz. Il voudrait notifier ses nouveaux rendez-vous à Liz. Pour modifier le comportement de l'agenda, nous utilisons le schéma d'adaptation notification. `NotifyingDiary` (resp. `NotifiedDiary`) est un rôle déduit de la règle d'adaptation et fournit le port `addRdv` (resp. `addRdv`, `isFree` et `printError`). Ainsi, tout composant comportant au moins un port conforme à `addRdv` (resp. `addRdv`, `isFree` et `printError`) peut jouer le rôle `NotifyingDiary` (resp. `NotifiedDiary`).

```
AdaptationPattern notification(NotifyingDiary a1, NotifiedDiary a2) {
   replace a1.addRdv(Rdv r) ->
         if (a2.isFree(r)) then a1.addRdv(r) // a2.addRdv(r)
         else a1.addRdv(r); a2.printError("Diary not synchronized!")
         endif }
```

[2] A présent que `lizDiary` est adapté, les rôles de `lizDiary` et de `johnDiary` ne sont plus les mêmes.

Dans notre cas, nous voulons que `johnDiary` joue le rôle de l'agenda notifiant et `lizDiary` le rôle de l'agenda notifié. Comme toutes les propriétés (en particulier P_2) sont assurées, le schéma notification peut être appliqué. Un assemblage est créé (`johnDiary` et `lizDiary` sont connectés) et le rôle de `johnDiary` est modifié comme sur la figure 3. Notons que si nous avions permuté les rôles des composants, l'adaptation n'aurait pas été autorisée : la propriété P_2 (contrainte C_6) n'est pas assurée par cette adaptation car `johnDiary` ne peut pas jouer le rôle `NotifiedDiary` (son rôle ne fournit pas le port `printError`).

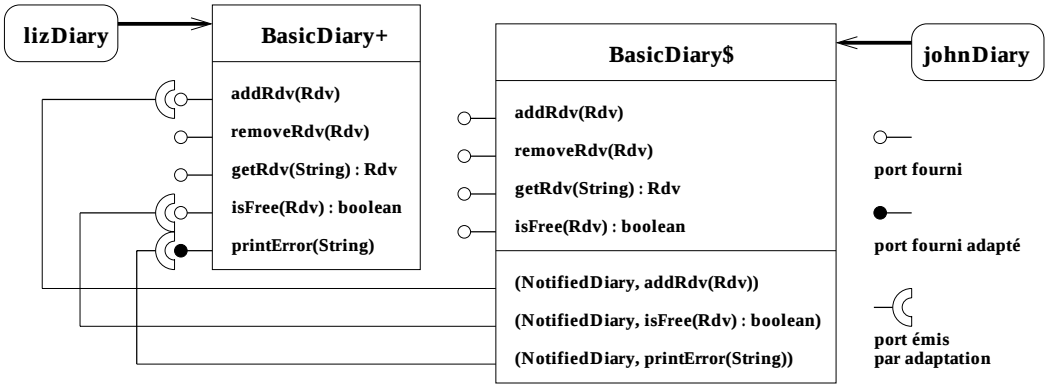

FIG. 3 – Résultat de l'application du schéma `notification` à `lizDiary` et à `johnDiary`

Après avoir débugué l'application, nous voulons désappliquer[3] le schéma d'adaptation `errorManagement` à `lizDiary` pour retirer le port `printError` du rôle de `lizDiary`. Cette adaptation n'est pas autorisée car P_2 (contrainte C_{10}) n'est pas assurée : `printError` est requis par `johnDiary` (via le schéma `notification`).

4 Validation par Simulation : Avantages et Limites de USE

Valider le modèle UML et les contraintes OCL consiste à prouver que les contraintes sont consistantes vis à vis de la propriété de sûreté à laquelle elles sont associées. La technique de validation par simulation consiste à construire explicitement des "snapshots" représentant des états d'un système. Un snapshot est un ensemble d'objets avec des valeurs d'attributs déterminés et des associations entre objets. Par exemple, les snapshots que nous avons testés par rapport à la contrainte C_1 correspondent à un système de 64 états. Pour un snapshot donné, si au moins une des contraintes testées est évaluée à "faux" ou a un résultat indéfini, l'état correspondant est considéré illégal et le snapshot est rejeté. Des outils "open-source" pour la validation de modèles UML par simulation sont proposés dans [Sof02, GBR05]. Objecteering [Sof02] fournit un framework pour tester dynamiquement l'intégrité des contraintes mais cet outil impose de définir les contraintes dans le langage propriétaire spécifique J. Par contre, Gogolla et al [GBR05] ont développé un outil, USE (UML-based Specification Environment), qui permet de valider des contraintes OCL à partir de définitions standard.

[3]Le processus d'adaptation doit être réversible : nous devons être capable de défaire les modifications exercées sur les composants et leurs rôles (suppression de fonctionnalités, destruction de certaines assemblages de composants, ...).

Deux étapes sont nécessaires pour prouver qu'un ensemble de contraintes n'est pas consistant par rapport à une propriété donnée. Premièrement, il faut vérifier que les contraintes ne soient pas trop faibles[4]. Pour cela, nous recherchons "à la main" des contre-exemples, c'est à dire, des snapshots qui seraient rejetés par une propriété P mais qui sont acceptés par les contraintes associées à P. Par exemple, nous avons testé si une fabrique pouvait être créé à partir d'une implantation et d'un ensemble de rôles dont certains ports ne correspondaient à aucun port de l'implantation. Nous avons évalué la contrainte C_1 avec plusieurs valeurs de ports et aucun des snapshots ainsi créés n'a été accepté par la contrainte comme nous l'attendions. Deuxièmement, il faut vérifier que les contraintes ne soient pas trop fortes[5]. Cette fois encore, nous recherchons "à la main" des contre-exemples, c'est à dire des snapshots qui seraient acceptés par une propriété P mais qui sont rejetés par les contraintes associées à P.

Pour valider rapidement un grand nombre de snapshots, nous utilisons le langage déclaratif ASSL (A Snapshot Sequence Language) [GBR05] au dessus de USE pour construire les snapshots de manière déclarative et ainsi explorer de manière systématique un sous-espace d'états. Les objets sont générés avec des valeurs d'attributs (choisies aléatoirement dans un ensemble de valeurs spécifié par l'utilisateur) et des associations entre ces objets (elles aussi choisies aléatoirement dans un ensemble d'associations spécifiées par l'utilisateur). Toutes les combinaisons sont testées jusqu'à ce qu'un état valide soit trouvé. Ainsi, les contre-exemples peuvent être trouvés plus rapidement. Néanmoins, comme ASSL interdit les effets de bord, seulement les invariants peuvent être testés. Nous avons donc du transformer les pré-conditions correspondant à nos contraintes en invariants. Par exemple, les deux pré-conditions correspondant aux contraintes C_3 et C_4 portent sur l'opération `create` de la classe `AdaptationPattern` donc sur les paramètres `roles` et `rules`. Il est très facile de transformer ces deux pré-conditions en invariants car les paramètres `roles` et `rules` appartiennent à l'état des objets de la classe `AdaptationPattern`. Il faut juste ajouter une condition supplémentaire pour indiquer que les associations de `AdaptationPattern` avec ses `roles` et ses `rules` doivent être effectives. Sinon, le premier snapshot construit (c'est toujours le même : celui où aucune association entre objet n'est effective) sera toujours choisi et l'exploration des espaces d'états se terminera.

En analysant les états acceptés et les états rejetés, nous avons pu améliorer notre spécification. Pour déterminer la non-consistance d'un ensemble de contraintes vis-à-vis d'une propriété, il suffit de trouver au moins un contre-exemple. Par contre, la consistance, elle, n'est garantie que vis-à-vis des états analysés. Par ailleurs, ASSL peut être amélioré de façon à ce que l'exploration des espaces d'états ne s'arrête pas au premier snapshot valide trouvé mais après avoir trouvé les n premiers snapshots valides ou même après avoir parcouru tout l'espace d'états. Pour conclure, la validation par simulation est une forme de prototypage permettant d'assurer rapidement un certain degré de confiance dans la spécification et ceci à moindre coût puisqu'il n'est pas nécessaire d'implémenter le modèle et les contraintes.

[4] Aucun état indésirable ne doit être accepté par les contraintes.

[5] Aucun état valide ne doit être rejeté par les contraintes.

5 Conclusion

Pour assurer la sûreté des adaptations, nous avons défini un modèle des adaptations et des propriétés de sûreté. Une première validation de cette modélisation a été décrite dans cet article. UML [OMGb] a permis d'établir rapidement et facilement une première modélisation des adaptations. Tous nos besoins en terme d'expression et des propriétés de sûreté ont pu être formalisés avec OCL [WK99]. Cette notation nous a aidé à clarifier nos besoins et à lever les ambiguïtés. Ce modèle est actuellement implémenté sous forme d'un service de sûreté accessible aux plates-formes à composants Java.

Néanmoins, les normes UML et OCL ne sont pas suffisantes pour avoir une formalisation complète du modèle. Aussi avons nous eu recours à l'outil de simulation USE pour établir la correction de notre spécification. Cette approche n'est qu'une première étape puisque l'exactitude de la spécification est garantie seulement vis-à-vis des états analysés. Aussi voulons nous compléter la validation de la modélisation en utilisant la technique de preuve par théorème avec B [Abr96] pour vérifier en particulier qu'il n'y a aucune contradiction entre les contraintes OCL décrites. Actuellement nous utilisons Atelier-B [Cle] dans cette optique.

Références

[Abr96] J.R. Abrial. *The B Book - Assigning Programs to Meanings.* Number ISBN 0-521-4961-5. Cambridge University Press, 1996.

[AP03] J. Adamek and F. Plasil. Behavior protocols capturing errors and updates. In *Proceedings of USE*, University of Warsaw, Poland, 2003.

[BCS02] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Proceedings of WCOP*, 2002.

[BFCE+04] M. Blay-Fornarino, A. Charfi, D. Emsellem, A-M. Pinna-Dery, and M. Riveill. Software interaction. *Journal of Object Technology*, 2004.

[Cle] Clearsy. Atelier B web site. http ://www.atelierb.societe.com/index_uk.html.

[Duc02] R. Ducournau. Spécialisation et sous-typage : thème et variations. *Technique et science informatiques*, 2002.

[GBR05] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Journal on Software and System Modeling*, 2005. To Appear.

[ODP04] A. Occello and A-M. Dery-Pinna. Safe runtime adaptations of components : a UML metamodel with OCL constraints. In *Proceedings of FUSE'04 (ETAPS Satellite Workshop)*, Barcelona, Spain, 2004.

[OMGa] OMG. CORBA 3.0 New Components Chapters. OMG Document ptc/2001-11-03.

[OMGb] OMG. Unified Modeling Language Specification. OMG Document formal/03-03-01.

[PSDF01] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC : A flexible and efficient solution for aspect-oriented programming in java. In A. Yonezawa and S. Matsuoka, editors, *Reflection*, volume 2192 of *LNCS*, pages 1–24. Springer-Verlag, 2001.

[Sof02] Softeam. The objecteering/UML tool suite. Internet : http ://www.objecteering.com/, 2002.

[WK99] J. Warmer and A. Kleppe. OCL : The constraint language of the UML. *Journal of Object-Oriented Programming*, 1999.

Vérification de conformité des interactions entre composants

P. André, G. Ardourel, C. Attiogbé, H. Habrias, C. Stoquer
LINA - FRE CNRS 2729
2, rue de la Houssinière, B.P.92208, F-44322 Nantes Cedex 3, France
Pascal.Andre@univ-nantes.fr

Mots-clefs – Keywords

Composants, Services, Spécification formelle, Conformité d'interaction
Components, Services, Formal Methods, Behavior consistency

Résumé – Abstract

Nous élaborons un cadre formel pour la définition, la composition et la vérification de propriétés des composants. Nous étudions les moyens d'effectuer rigoureusement des vérifications de conformité des interactions entre composants en nous basant sur les systèmes de transitions qui décrivent les services des composants.
We present a formal framework for the definition, the composition of components and the verification of the assemblies. First, we define a component model and a composition operator, then we describe the behavioral compatibility of components using transition systems to describe services.

1 Introduction

Le développement de logiciels à partir de composants [Szy97, MT00, BRS+00] suscite un certain intérêt mais soulève des questions non encore résolues. Plusieurs paramètres conditionnent la réussite du développement à grande échelle de logiciels à base de composants prédéfinis : la disponibilité de bibliothèque de composants fiables, la disponibilité d'outils de recherche de composants, la disponibilité de langages expressifs de composition des composants et surtout la disponibilité d'outils de vérification du bon usage de composants. Dans ce travail, nous étudions essentiellement le dernier paramètre.

Pour ce faire, nous nous appuyons sur un formalisme simple de modélisation et de composition de composants. L'utilisation d'un composant se fait à travers les services qui constituent son interface. Il est important de respecter les conditions d'utilisation d'un service pour bénéficier des fonctionnalités de celui-ci. Si le respect de la signature d'un service au moment de son appel apparaît comme un prérequis facilement vérifiable, le test de conformité a priori, de l'interaction entre un service et son appelant pendant le déroulement du service n'est pas une tâche facile. En effet l'interaction peut être très

simple et se résumer en un "appel-réponse" ou alors peut être plus complexe lorsque le déroulement du service nécessite la collaboration de l'appelant. En plus des signatures, la description des comportements des services est nécessaire aux appelants. Pour assurer un certain niveau de **correction des composants** et des **assemblages de composants**, l'analyse formelle des descriptions des services par rapport aux propriétés attendues du composant est nécessaire ; des tests de conformité de l'interaction entre les composants doivent aussi être effectués au moment de leur assemblage. On peut ainsi garantir la bonne marche d'un assemblage. L'utilisation d'un modèle formel permet de faire abstraction des détails d'implantation des composants afin d'avoir des techniques générales de raisonnement adaptables facilement à divers environnements spécifiques de mise en œuvre des composants.

Le papier est organisé comme suit. Dans la section 2 du document, nous donnons un aperçu du modèle formel utilisé dans ce travail. Nous présentons dans la section 3 les grandes lignes de l'analyse de conformité des interactions entre les services des composants. Une petite conclusion termine le document.

2 Un modèle formel pour les composants

L'idée de caractériser les composants par une *interface* constituée de *services offerts* et de *services requis* est aujourd'hui communément admise [AG97, MT00]. Un service offert réalise une fonctionnalité. Un service requis par un composant utilise un service offert d'un autre composant se trouvant dans son environnement. L'environnement du composant est défini lorsque celui-ci se trouve dans un *assemblage* (de composants). De plus, nous considérons que les composants peuvent avoir un espace d'états et des propriétés associées.

Un **assemblage** est un ensemble de composants qui coopèrent pour la construction d'une application logicielle. Les liaisons entre composants sont les liaisons entre les services requis par les uns et les services offerts par les autres. La figure 1 illustre une configuration pour un système client-serveur. Un service offert peut avoir lui-même besoin de services requis. Par exemple, pour réaliser le service de connexion (`cnx`), le composant `Serveur` requiert deux services (*demMdp* et *verifMdp*) pour demander un mot de passe et vérifier un mot de passe. Dans la définition du composant `Serveur` (Fig. 1), aucune hypothèse n'est faite sur les composants qui offriront les services requis, un composant qui offrirait les deux services pourrait convenir.

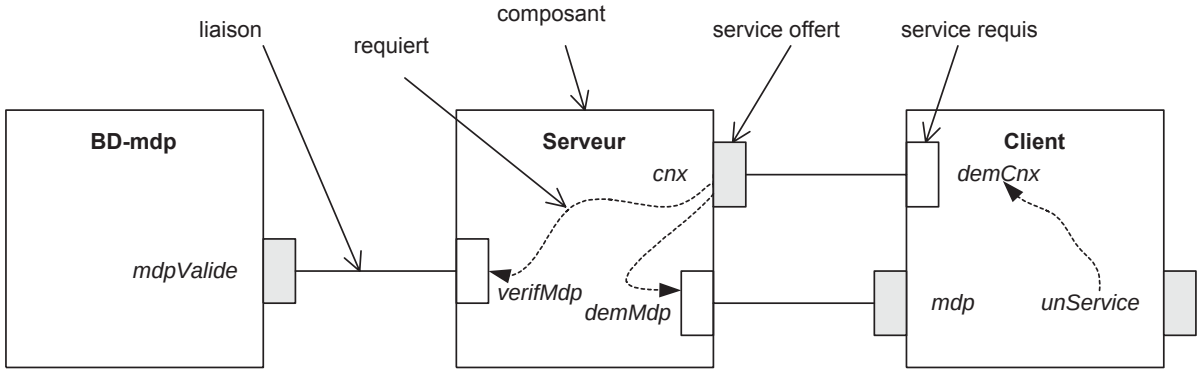

FIG. 1 – Assemblage de trois composants

Formellement, un **composant** est défini par
- T un ensemble de types, V un ensemble de variables, V_T un ensemble de variables ayant chacune un type $V_T \subseteq V \times T$ et I l'initialisation des variables de V_T ;
- A un ensemble fini d'actions élémentaires ;
- S un ensemble fini de noms de services, incluant S_{o_v} et S_{r_v} les ensembles des noms de services *visibles* offerts et requis.
- $\mathcal{D}_S$ ensemble de descriptions de services, partitionné en $\mathcal{D}_{S_o}$ (offerts) et $\mathcal{D}_{S_r}$ (requis).
- $\mathcal{C}_{S_o}$ les contraintes d'interaction entre les services du composant[1].

Un service est spécifié par une interface et éventuellement un comportement.

L'**interface** $I\langle\sigma,\ P,\ Q, N\rangle$ d'un service s comprend la signature σ, la précondition d'appel P, la postcondition Q du déroulement du service et un ensemble N de noms de services. Le déroulement d'un service peut être l'occasion d'un échange complexe d'informations entre l'appelant et l'appelé qui ne se résume pas au passage de paramètres et au retour de l'appel. Cet échange repose sur des services requis et offerts, donnés dans l'interface sous la forme de deux ensembles :

- N_{o_i} un ensemble fini de noms de services offerts non-visibles (avec $N_{o_i} \cap S_{o_v} = \emptyset$) ;
- N_r un ensemble fini de noms des services requis.

Le **comportement** d'un service offert est spécifié par $\mathcal{B}$, un système de transitions étiquetées par :

- Une action élémentaire, traitement qui ne nécessite pas d'autres composants.
- Une invocation de services (internes) du composant ou des services requis (externes) par le composant. Un service interne peut n'être utilisable que dans le cadre du service spécifié, on le qualifie alors de **sous-service**.
- Une communication, échange entre deux composants dans le cadre d'un service.

Par exemple, le service offert *cnx* du composant `Serveur` (Fig. 1) est spécifié comme suit :

```
cnx (nom : String) =
PRE -- précondition P
SPEC { init e0 :
       final e4 :
     e0 --demMdp()--> e1 ,
     e1 --demMdp?resMdp(m:String)--> e2 ,
     e2 --rep := verifMdp(nom, m)--> e3 ,
     e3 --!resCnx(rep) --> e4}
POST -- post condition Q
```

La syntaxe générale d'une communication est $serviceRequis(!\,|?)\ nomDeService(param*)$. Elle est inspirée du langage CSP de Hoare. Lorsque la variable *serviceRequis* dénote un composant qui offre ce service, par défaut c'est le composant qui a invoqué le service en cours d'exécution. Examinons maintenant les services (`unService` et `mdp`) du côté `client` (Fig. 1). On suppose que l'invocation de la demande de connexion apparaît dans le service `unService` du composant `Client`. Ensuite l'interaction est poursuivie avec l'appel de `mdp`.

```
unService () =
PRE true
SPEC { init e0 :
   e0 --cnx(monNom)--> e1 ,
```

[1] Cet aspect n'est pas développé dans ce papier.

```
   e1 <mdp>
   e1 --cnx?resCnx(b:Bool)--> e1}
POST true

mdp () =
PRE true
SPEC { init e0, e1 :
     final e1 :
     e0 --!resMdp(monMdp)--> e1}
POST true
```

A chaque état du système étiqueté, on peut associer un ensemble de services offerts, ou de sous-services invocables. La notation `e1 <mdp>` indique que le sous-service *mdp* est invocable dans l'état `e1` ; la suite du traitement est poursuivie dans le sous-service. Pratiquement, cela permet de simplifier la description du système de transitions (graphe orienté) d'un service, en étiquetant certains noeuds par des noms de sous-services x ; la description d'un sous-service est lui même un système de transitions. A partir d'un état `e` du système de transitions d'un service, on peut définir plusieurs sous-graphes que nous noterons $G(e, x)$ par la suite.

Le modèle formel présenté plus haut nous donne un cadre simple pour spécifier des composants. Pour **composer** ces derniers, nous introduisons un opérateur nommé compose, qui a pour paramètres un ensemble non vide de composants, la description explicite des *liens* entre les services (offerts et requis) des différents composants, une définition de l'interface du composant obtenu dans laquelle on peut mettre des services de ses composés, éventuellement redéfinis (par exemple, en changeant l'interface/le typage), et de nouveaux services éventuels. Cette composition, illustrée dans [AAA+04] est l'occasion d'une vérification de compatibilité comportementale décrite en section 3 .

3 Analyse des composants et assemblages

Les composants et leur assemblage peuvent être analysés sous diverses facettes. Ici nous nous préoccupons uniquement de l'analyse de la conformité des interactions entre services appelants et appelés de façon à détecter les incompatibilités. On a une *incompatibilité de comportements* lorsque l'appelant d'un service (supposé correctement décrit), n'est pas en mesure de satisfaire les exigences du service appelé au moment de son déroulement. La plupart des travaux sur les interactions utilisent des variantes du modèle des automates à états [dAH01, YS97, AL03]. Ils testent la compatibilité entre les automates représentant des services ; l'explosion combinatoire est la principale difficulté rencontrée. Dans notre proposition nous apportons une solution satisfaisante en considérant des environnements incomplets (par exemple des services requis non encore satisfaits). Contrairement à d'autres approches [PV02], les protocoles sont associés aux composants **et** aux services. Le service est une entité de première classe.

Nous proposons dans ce qui suit une première version de l'analyse de composants interagissants et l'analyse de compatibilité des comportements. Le lecteur trouvera des détails dans [AAA+04].

Nous traitons les services qui interagissent lorsque des composants sont assemblés ou com-

posés. Les *messages en entrée* d'un état s (notés $input_m(s)$) d'un comportement de service sont les messages venant de l'environnement et qui sont reçus dans cet état. Les *messages en entrée* du comportement $\mathcal{B}$ (notés $Input_m(\mathcal{B})$) d'un service sont définis par l'union des $input_m(s)$ des états de $\mathcal{B}$. Les *messages en sortie* d'un état s (notés $output_m(s)$) d'un comportement de service sont les messages émis de cet état à destination de l'environnement. Les *messages en sortie* du comportement $\mathcal{B}$ (notés $Output_m(\mathcal{B})$) d'un service sont définis par l'union des $output_m(s)$ des états de $\mathcal{B}$. Les *appels de service* d'un état s (notés $call_m(s)$) d'un comportement de service sont les appels de services d'autres composants ou des appels de sous-services du composant courant. Les *appels de service* du comportement $\mathcal{B}$ (notés $Call_m(\mathcal{B})$) sont en conséquence définis par l'union des messages $call_m(s)$ des états s de $\mathcal{B}$. De la même façon nous définissons les attentes d'appel $waitCall_m(s)$) et $WaitCall_m(\mathcal{B})$. En effet dans un état donné, l'appel d'un service ou d'un autre est attendu sinon il ne peut être traité.

Pour la correspondance entre messages en sortie d'un service et les messages en entrée du service impliqué dans une interaction, nous utilisons l'hypothèse de la communication synchrone. Les services utilisent dans ce cas un alphabet commun.

Définition 3.1 *Collaboration requise.*
Soit un service $se = \langle \sigma, I, \mathcal{B} \rangle$ *avec son comportement* $\mathcal{B} = \langle S, L, \delta \rangle$ *; on suppose que l'environnement avec lequel s'effectue l'interaction utilise un ensemble d'états* S' *et s'appuie aussi sur une modélisation à base de système de transitions.*

La collaboration requise pour interagir avec $\mathcal{B}$ *est tout comportement* $\mathcal{B}'$ *symétrique à* $\mathcal{B}$ *au sens où : i) pour chaque état s, depuis l'état initial de* $\mathcal{B}$*, les messages en entrée de s sont les messages en sortie de* s' *qui est l'état courant du comportement de l'environnement, ii) les messages de sortie de s sont les messages d'entrée de* s' *et iii) les messages d'appels correspondent aux attentes d'appel. De plus, les appels d'un service externe peuvent correspondre au déclenchement et au déroulement (avec interaction possible) d'un sous-système de transitions dans l'autre comportement (comme si on passait d'un état initial à un état final pour un sous-système de transitions ad hoc du côté du service appelé).*

$$\forall s \in S. \exists s' \in S'. \quad input_m(s) = output_m(s') \quad \wedge$$
$$\wedge \quad output_m(s) = input_m(s') \quad \wedge call_m(s) = waitCall_m(s')$$

Tout service ayant un comportement similaire à la *collaboration requise* peut interargir avec le service. Il a un comportement conforme (noté $\overline{\mathcal{B}}$) dans la suite.

Définition 3.2 *Comportement strictement conforme (Strict Matching Service).*
Un service se' *a un comportement strictement conforme pour un service* se *si le comportement* $\mathcal{B}'$ *de* se' *est équivalent (il s'agit ici de l'équivalence de systèmes de transitions) On note* $\mathcal{B}' \equiv \overline{\mathcal{B}}$

Définition 3.3 *Comportement non strictement conforme (Non-strict Matching Service).*
Un service se' *a un comportement* ($\mathcal{B}'$) *non strictement conforme à un service* se *lorsque la collaboration requise de* se *est un sous-ensemble du comportement proposé par* se'*. On note* $\mathcal{B}' \sim \overline{\mathcal{B}}$

$$\forall s \in S. \exists s' \in S'. \quad input_m(s) \subseteq output_m(s')$$
$$\wedge \quad output_m(s) \subseteq input_m(s') \quad \wedge \quad call_m(s) \subseteq waitCall_m(s')$$

Ces définitions constituent le fil rouge pour la mise au point des algorithmes de test

de conformité des interactions. Ces algorithmes sont cruciaux lors de l'assemblage ou la composition de composants.

4 Conclusion

Nous avons présenté un cadre pour le développement par composants élaboré avec un souci de généralité et de simplicité.

L'interaction entre composants est abordée ici au niveau des services, en prenant soin de limiter le problème de l'explosion combinatoire. Un de nos objectifs est l'expression du résultat d'une vérification comportementale de composition à l'aide d'un BIDL (Behavioral IDL) adapté qui fournira une abstraction du comportement de plus en plus haut niveau au fur et à mesure des compositions. Ce travail est en cours ; nous poursuivons la formalisation, la validation sur des études de cas et des expérimentations avec des outils de vérification.

Références

[AAA+04] P. André, G. Ardourel, C. Attiogbé, H. Habrias, and C. Stoquer. Vérification de conformité des interactions entre composants. Technical Report RR04.11, LINA - FRE CNRS 2729 - Nantes, December 2004.

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, July 1997.

[AL03] P. C. Attie and D. H. Lorenz. Establishing Behavioral Compatibility of Software Components without State Explosion. Technical Report NU-CCIS-03-02, College of Computer and Information Science, Northeastern University, 2003.

[BRS+00] K. Bergner, A. Rausch, M. Sihling, A. Vilbig, and M. Broy. A Formal Model for Componentware. In G. T. Leavens and M. Sitaraman, editors, *Foundations of Component-Based Systems*, pages 189–210. Cambridge University Press, New York, NY, 2000.

[dAH01] L. de Alfaro and T. A. Henzinger. Interface Automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE)*, pages 109–120. ACM Press, 2001.

[MT00] N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1) :70–93, january 2000.

[PV02] F. Plasil and S. Visnovsky. Behavior protocols for software components, 2002. IEEE Transactions on SW Engineering, 28 (9), 2002.

[Szy97] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming*. AddisonWesley Publishing Company, 1997.

[YS97] D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2) :292–333, 1997.

Un modèle de composant avec protocole symbolique

Sebastian Pavel, Jacques Noyé, Jean-Claude Royer
OBASCO École des Mines de Nantes - INRIA, LINA
4, rue Alfred Kastler, 44307 Nantes cedex 3, France
{Sebastian.Pavel, Jacques.Noye, Jean-Claude.Royer}@emn.fr

Mots-clefs – Keywords

CBSE, protocole explicite, système de transitions symboliques, contrôleur, canaux de communication
CBSE, Behavioural IDL, Explicit Protocols, Symbolic Transition Systems, Controllers, Channels

Résumé – Abstract

Les composants proposent une approche du développement logiciel reposant sur des assemblages d'entités réutilisables. Un aspect important est la détection et adaptation des incompatibilités lors de la composition. Notre travail suggère d'utiliser des descriptions des interactions entre composants basées sur une notion de systèmes de transitions symboliques (STS). Nous décrivons dans ce résumé les principes de l'implémentation d'un modèle de composants qui prend en compte les STS. Elle est basée sur une notion de contrôleur qui encapsule le STS et contrôle son comportement. Les communications entre les composants sont réalisées par l'intermédiaire de canaux de communications.
Component-Based Software Engineering (CBSE) has now emerged as a discipline for system development. Important issues in CBSE such as composition incompatibility detection and (dynamic) adaptation can only be addressed with the help of formal component models with Behavioural Interface Description Languages (BIDLs) and explicit protocols. The issue is then to fill the gap between such high-level models and implementation. This paper suggests to do so by relying on Symbolic Transition Systems (STSs). It describes a component model with explicit symbolic protocols based on STSs, and its implementation principles. This implementation is based on controllers that encapsulate protocols and channels devoted to (possibly remote) communications between components.

1 Introduction

Les composants proposent une approche du développement basée sur l'assemblage d'entités réutilisables. Il existe actuellement plusieurs travaux dans le domaine académique comme dans le monde industriel, on peut penser à des exemples comme Olan [BAKR95], Fractal [CBS02] ou encore aux EJB [DeM03]. Les langages d'architectures [MT00] proposent aussi des approches très voisines dans l'esprit mais souvent à un niveau plus spécification que programmation. Nous nous intéressons ici aux modèles ou langages de composants permettant la définition de composants hiérarchiques. Une difficulté importante est de pouvoir assurer certaines propriétés des assemblages. Il ne s'agit pas seulement de propriétés de typage mais par exemple d'assurer l'absence de blocage au sens dynamique du terme ou encore la vivacité de certaines actions. Par exemple dans le contexte du projet DISPO [DIS] nous voulons pouvoir assurer que les composants vérifient des propriétés de disponibilité. Pour aborder ce problème nous proposons un modèle de composants disposant, en plus de la notion habituelle d'interface, de protocoles. Il existe déjà quelques langages de spécifications formelles d'architectures proposant de telles constructions, par exemple WRIGHT [All97]. Par ailleurs, des langages de programmation comme SOFA [KT02] ont utilisé les expressions régulières dans ce but. Notre approche est plutôt d'utiliser des systèmes de transitions symboliques (STS [CMS02]), c'est-à-dire des machines à états finis avec des étiquettes autorisant des variables et des gardes. Ce formalisme a le principal avantage d'être lisible et d'éviter les problèmes d'explosion du nombre d'états et de transitions des machines à états et transitions étiquetées. Nous présentons dans ce document les principes d'implémentation de ce modèle en Java. Ceux-ci reposent sur l'utilisation de contrôleurs qui se chargent de gérer les interactions de l'implémentation du composant avec son environnement et cela conformément au protocole défini par le STS. Des canaux de communications permettent de représenter les communications entre composants et d'offrir des possibilités intéressantes de dynamicité.

2 Un modèle de composant basé sur les STS

Comme d'autres modèles ou langages de composants notre approche considère des composants munis d'un ensemble de services accessibles par des ports de communication. Ces services sont en fait regroupés dans une ou des *interfaces*. Les communications peuvent être en mode synchrone avec retour de valeur ou asynchrone. Nous proposons dans ce modèle une description de la dynamique des composants par l'utilisation de machines à états et transitions. Ces machines, appelé systèmes de transitions symboliques, décrivent la dynamique du composant. Elles autorisent l'utilisation de variables pour les communications mais aussi de gardes pour contrôler le déclenchement des transitions.

Nous allons décrire brièvement dans ce qui suit les éléments du modèle et les principes de son implémentation en Java. Notre cible est Java mais notre approche propose un modèle formel pour des besoins de vérifications qui peut être décliné vers d'autres langages concrets. Des détails complémentaires peuvent être trouvés dans [PPNR05] notamment une sémantique des STS et des éléments relatifs à la vérification.

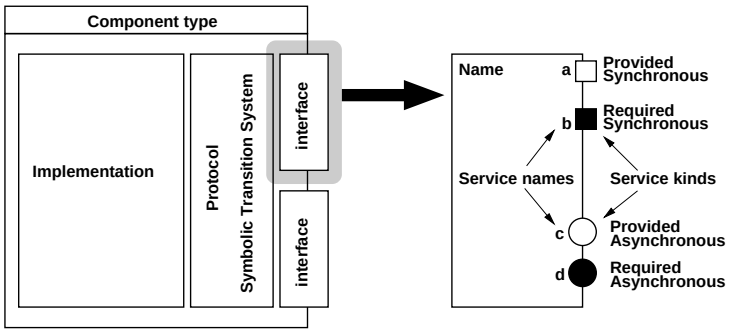

FIG. 1 – Représentation graphique des composants

2.1 Composant primitif

La partie visible ou publique d'un composant primitif est un nom, des interfaces nommées et un protocole. Ce composant primitif encapsule soit un type de données soit une autre application (éventuellement un autre composant) appelé son *implémentation*. Les services des interfaces correspondent à des fonctions ou des actions de la partie implémentation du composant. Les composants interagissent par l'intermédiaire d'interfaces nommées chacune correspondant à un *port* de communication. Une interface est un regroupement de services du composant. Un service est la donnée d'un nom unique dans l'interface et d'un type correspondant aux valeurs reçues ou émises par le service. Un service peut être *fourni* ou *requis* et fonctionner de manière *synchrone* ou *asynchrone*.

Les STS [CMS02] ont été développés à l'origine comme une solution au problème d'explosion du nombre d'état et de transition dans les algèbres de processus avec passage de valeurs. Nos STS sont une généralisation de cette idée. Ils associent un état symbolique et un système de transition avec une implémentation [Roy03]. La description abstraite de la partie implémentation peut être donnée en utilisant des spécifications algébriques, des formalismes orientés modèle ou encore des classes Java. Les STS peuvent être comparés aux *Statecharts* mais leur sémantique formelle est plus simple et plus stricte. Le STS va donner les ordonnancements autorisés des services requis et fournis du composant. Il n'y a qu'un STS pour l'ensemble des interfaces ou services du composant.

La partie implémentation du composant est une application encapsulée. Une hypothèse est que cette partie (une application Java dans notre cas) ne fournit pas seulement des services mais aussi l'ensemble des gardes qui sont utilisées dans le STS.

Au niveau syntaxique, nous avons défini des notations graphiques pour l'interface statique (Fig. 1) et pour le STS associé (Fig. 2). La figure 2 décrit le protocole STS associé à un composant appelé *Company* et qui représente une compagnie proposant des vols d'avion. Les services requis sont préfixes par *!* et les services fournis par *?*. Le signe ^ dénote un message asynchrone. La syntaxe textuelle dépend de la cible du langage considéré, Java ici (voir [PPNR05] pour plus de détails).

Au niveau sémantique ,plusieurs propositions ont été étudiées, voir par exemple [CPR00, MPR04]. La sémantique d'un composant avec STS peut être exprimée comme un graphe de configurations (voir [MPR04, PPNR05]). Un tel graphe est un système de transitions étiquetées avec des valeurs associées aux états. Ainsi les notions habituelles de traces, d'arbre d'atteignabilité, etc. sont transposables dans ce contexte et ouvrent la voie vers

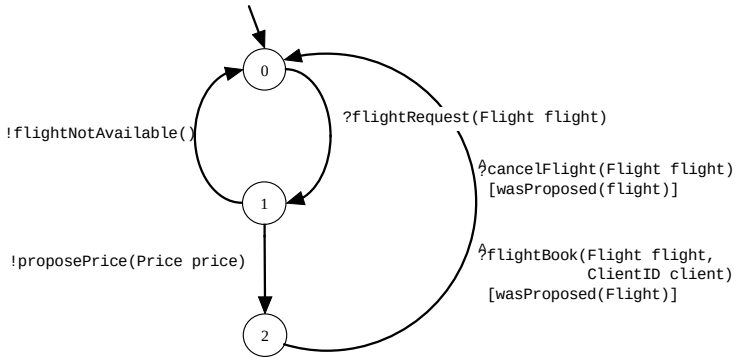

FIG. 2 – Représentation graphique d'un STS

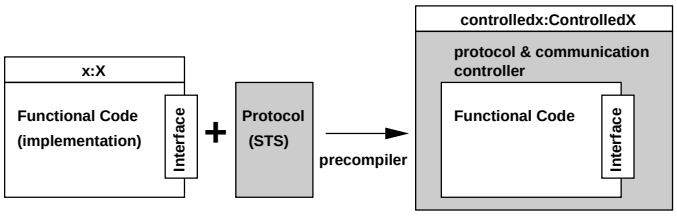

FIG. 3 – Principe de l'implémentation

des vérifications.

2.2 Composant composite

Un composant composite (ou hiérarchique) est un assemblage de composants (les sous-composants) et d'un ensemble de connexions entre les ports de ceux-ci. Les communications de base sont point à point et orientées mais des modes plus évolués (broadcast, ...) sont possibles. La correspondance entre deux interfaces liées est faite par un appariement des noms des services avec éventuellement renommage. Les services en correspondance doivent être compatibles au niveau du type et posséder le même mode de communication. Un composite est défini par une interface, les connexions entre ses sous-composants et les connexions entre ses sous-composants et son interface. Un composant composite peut être décrit sous une forme graphique ou sous une forme textuelle. La sémantique d'un composite est décrite par l'adaptation du produit synchronisé aux STS [CPR00, Roy03].

3 L'implémentation du modèle

L'idée générale de l'implémentation est donnée par la figure 3. Le principe est d'attacher un protocole à une application existante (l'implémentation). Après cette encapsulation le résultat est un composant primitif dont le STS contrôle la partie implémentation. Le composant fournit au plus les services définis par l'implémentation et inclut un mécanisme pour contrôler et imposer le protocole choisi. Le composant peut communiquer dans une architecture soit par une connexion locale ou par l'intermédiaire de RMI (si ses correspondants sont distants). L'implémentation des connexions d'une architecture est réalisée par l'utilisation de canaux de communication.

3.1 Implémentation du protocole

Pour adjoindre un protocole à une implémentation nous utilisons une entité active complémentaire (un *thread* Java) qui joue le rôle de contrôleur pour le composant. Le but de ce contrôleur est : (1) d'intercepter les messages reçus ou émis par l'implémentation et (2) de décider quand ces messages sont autorisés ou interdits. L'implémentation est réalisée en utilisant le principe de l'état logique (*logical state pattern* [GHJV95]). Ce patron est implémenté en deux étapes. Tout d'abord, les états possibles du protocole sont déclarés comme des variables privées du contrôleur. Ensuite, les actions à exécuter quand l'implémentation reçoit un message sont définies dans la méthode `run` du contrôleur. Les actions à réaliser quand l'implémentation doit envoyer un message sont définies dans les méthodes privées implémentant les opérations requises.

3.2 Canaux de communications

Pour la connexion des composants, nous réutilisons le principe des canaux de communications [ASdBB02]. Un canal représente une connexion anonyme entre deux composants. Il est orienté de la source vers le destinataire, ainsi il faut généralement deux canaux par connexion, un dans chaque sens possible de communication. Les canaux sont créés dans l'espace de visibilité du composite évitant ainsi des problèmes d'exportation hors de celui-ci. L'intérêt des canaux est qu'ils permettent facilement de prendre en compte le mode de communication synchrone ou asynchrone, la mobilité, des conditions, des priorités, etc. En plus de son rôle de coordination, un canal peut être utilisé pour faire des adaptations (d'interfaces et/ou protocoles), des connexions plus complexes que de simples connexions point à point (broadcast par exemple). Finalement ils uniformisent les communications (distantes ou locales) et autorisent la création et la connexion dynamique.

4 Conclusion

Dans ce résumé nous avons présenté l'introduction de protocoles explicites dans un modèle de composants hiérarchiques. Nous préconisons l'utilisation des systèmes de transitions symboliques pour des raisons de lisibilités et d'abstraction. Le modèle développé supporte des interfaces multiples hétérogènes et des communications synchrones et asynchrones. Une implémentation en Java de ce modèle a été prototypée. Elle repose sur la définition d'un mécanisme de contrôleur réalisant la partie protocole du composant. Ce contrôleur intercepte les messages reçus ou émis et déclenche le service correspondant uniquement si les conditions stipulées dans le STS sont réalisées. La communication entre les composants d'une architecture est réalisée par des canaux de communication. Ces canaux offrent beaucoup de souplesse et de puissance d'expression, ils autorisent de nombreux modes de communications et la mobilité.

Références

[All97] R. J. Allen. *A formal approach to software architecture*. PhD thesis, Carnegie Mellon University, 1997.

[ASdBB02] F. Arbab, J. V. Guillen Scholten, F.S. de Boer, and M. M. Bonsangue. A channel-based coordination model for components. Technical report, Centruum voor Wiskunde en Informatica, 2002.

[BAKR95] L. Bellissard, S. B. Atallah, A. Kerbrat, and M. Riveill. Component-based programming and Application Management with Olan. In J. Briot, J. Geib, and A. Yonezawa, editors, *Object-Based Parallel And Distributed Computation*, volume 1107 of *LNCS*, pages 290–309. Springer-Verlag, Berlin, 1995.

[CBS02] T. Coupaye, E. Bruneton, and J.-B. Stefani. The Fractal composition framework. Technical report, The ObjectWeb Group, 2002.

[CMS02] M. Calder, S. Maharaj, and C. Shankland. A Modal Logic for Full LOTOS Based on Symbolic Transition Systems. *The Computer Journal*, 45(1) :55–61, 2002.

[CPR00] C. Choppy, P. Poizat, and J.-C. Royer. A global semantics for views. In T. Rus, editor, *International Conference, AMAST'2000*, volume 1816 of *Lecture Notes in Computer Science*, pages 165–180. Springer-Verlag, 2000.

[DeM03] L.G. DeMichiel. *Enterprise JavaBeans^{TM} Specification*. SUN Microsystems, November 2003. Version 2.1, Final Release.

[DIS] DISPO home page. http ://www.irisa.fr/lande/jensen/dispo.html.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley Professional Computing Series. Addison Wesley, 1995.

[KT02] T. Kalibera and P. Tuma. Distributed Component System Based on Architecture Description : The SOFA Experience. In R. Meersman, Z. Tari, and al., editors, *CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 981–994. Springer-Verlag, 2002.

[MPR04] O. Maréchal, P. Poizat, and J.-C. Royer. Checking asynchronously communicating components using symbolic transition systems. In R. Meersman, Z. Tari, and al., editors, *CoopIS, DOA, and ODBASE*, volume 3291 of *Lecture Notes in Computer Science*, pages 1502–1519. Springer-Verlag, 2004.

[MT00] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE - Transactions on Software Engineering*, 26(1) :70–93, 2000.

[PPNR05] S. Pavel, P. Poizat, J. Noyé, and J.-C. Royer. A formal component model with explicit symbolic protocols and its Java implementation. Technical report, École des Mines de Nantes, January 2005.

[Roy03] J.-C. Royer. The GAT approach to specify mixed systems. *Informatica*, 27(1) :89–103, 2003.

COMMODE: Un Modèle d'Adaptation Structurelle pour les Composants Logiciels

G. BASTIDE (1,2), A. SERIAI (1), M. OUSSALAH (2)

Département GIP – Ecole de Mines de Douai
941 rue Charles Bourseul, 59558 Douai, Cedex, France
{seriai, bastide}@ensm-douai.fr
(2) LINA – CNRS FRE 2729
Université de Nantes, 2 rue de la Houssinière, 44322 Nantes Cedex
{bastide,oussalah}@lina.univ-nantes.fr

Mots-clefs

Composant logiciel, Adaptation, Adaptation structurelle, Re-factorisation (refactoring), Fractal.

Résumé

Nous proposons dans cet article un nouveau type d'adaptation appliquée aux composants logiciels : l'adaptation structurelle. Au contraire des approches d'adaptation existantes où l'objectif est de mettre à jour les services ou le comportement des composants, l'adaptation structurelle vise plutôt à mettre à jour la structure de ces composants. Les applications de ce type d'adaptation peuvent être, par exemple, l'adaptation du déploiement par rapport à la configuration de l'infrastructure sous-jacente ou l'adaptation de l'exécution par rapport aux ressources disponibles (CPU, mémoire). Nous avons développé un processus et un modèle de composants logiciels supportant l'adaptation structurelle statique et pouvant être étendus pour supporter l'adaptation dynamique. L'étude théorique a été réalisée en s'appuyant sur le modèle UML2. Cette approche a été transposée dans le modèle de composant Fractal puis implémentée en utilisant la plate-forme Julia.

1 Introduction

1.1 Contexte

Dans la lignée des approches orientées objets, celles à base de composants visent à accroître la réutilisation du code et à faciliter le développement d'applications logicielles [Mar02, Szy98]. Les développeurs de ces composants ont à faire face à une grande diversité de plates-formes d'exécution et d'environnements logiciels associés. Le nombre et la variété de ces plates-formes n'ont jamais été aussi grands, allant du simple téléphone portable doté de capacités minimales au cluster de plusieurs dizaines d'ordinateurs multiprocesseurs, en passant par l'ordinateur de bureau [Led01]. La solution réside évidemment dans le développement de composants logiciels capables de s'adapter à ces conditions d'exécutions nouvelles ou changeantes. Ainsi un composant logiciel est dit adaptable s'il est capable de fonctionner dans des conditions d'exécution non prévues au moment de sa conception.

Pour répondre à cet objectif concernant l'adaptation des applications à base de composants, différentes approches ont été proposées. Toutes se focalisent principalement sur l'adaptation des services des sujets à adapter. En effet, ces dernières proposent d'augmenter ou de diminuer les services proposés par le sujet d'adaptation en question ou à mettre à jour leurs implémentations (i.e. leurs algorithmes).

1.2 Problématique : Adaptation structurelle de composants logiciels

Dans cet article, nous proposons d'étudier un nouveau type d'adaptation pour les composants logiciels : l'adaptation structurelle. Nous définissons l'adaptation structurelle d'un composant logiciel comme la capacité de modifier la structure de ce dernier tout en préservant ses services et son comportement. La structure d'un composant est l'ensemble des éléments structurels définis dans l'espace de noms de ce composant et identifiables de manière unique. Ainsi, l'adaptation peut concerner la structure externe d'un composant (e.g. ses ports et ses interfaces, etc.) et / ou sa structure interne (e.g. ses sous-composants, ses classes d'implémentation, etc.). Par exemple, dans le cas d'un composant logiciel conforme au modèle UML2 [Eri03], l'adaptation structurelle peut consister en la réorganisation des services de ce composant dans des nouvelles interfaces ou en la décomposition de la structure interne monolithique d'un composant en un ensemble constitué d'autres (sous-) composants (voir figure 1). L'invariant pour le composant à adapter est que l'ensemble des services offerts ou requis par le composant doit rester le même avant et après l'adaptation. Les applications de ce type d'adaptation sont nombreuses. A titre d'exemple, le fractionnement d'un composant logiciel en plusieurs sous-composants permet un éventuel déploiement distribué de ce résultat de l'adaptation (voir section 2).

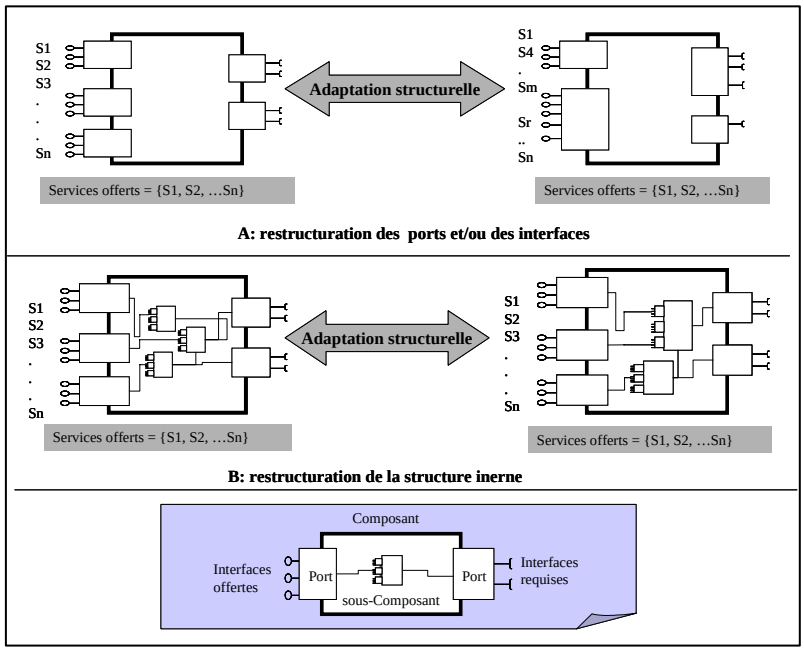

Figure 1. *L'adaptation structurelle*

Nous nous restreignons dans ce papier à l'étude de l'adaptation structurelle en considérant les choix suivants :

- L'adaptation d'un composant est réalisée par la mise à jour de sa structure interne.

- L'adaptation proposée est orientée par les interfaces. En effet, la spécification de la nouvelle structure interne d'un composant est orientée par la spécification des interfaces de chaque nouveau sous-composant. Ces interfaces sont considérées comme des éléments structurels indivisibles.

- L'adaptation est effectuée sur des composants non binaires. En effet, le processus d'adaptation manipule le code source du composant à adapter.

- L'adaptation s'opère sur des composants logiciels orientés objets. Notre approche d'adaptation structurelle est basée sur le modèle de composants logiciels proposé par UML2 [Eri03]. Ce modèle défini par le consortium OMG est basé sur une conceptualisation objet d'un modèle de composant.

Dans la suite de cet article, nous illustrerons notre approche comme suit : dans la section 2, nous présenterons quelques applications de l'adaptation structurelle. La section 3 est consacrée à la description du processus et du modèle de composant supportant l'adaptation structurelle. Une conclusion et quelques perspectives de ce travail seront présentées dans la section 4.

2 Motivations : les applications de l'adaptation structurelle

Les applications de l'adaptation structurelle sont multiples :

- L'adaptation du déploiement des composants logiciels

Le déploiement d'un composant est dépendant de la structure de celui-ci. En effet, il est impossible de déployer séparément une partie d'un composant, définissant un ensemble de services, si cette partie n'est pas structurellement identifiable et donc séparable. Ainsi, l'adaptation structurelle permet un déploiement flexible d'un composant logiciel. Ce déploiement pourra être adapté à la configuration de l'environnement qui peut être centralisé, distribué, sous forme de grappe (i.e. cluster), etc... Il est possible d'assurer cette propriété en permettant de redéfinir et de recréer, suivant les besoins du déploiement, les différentes entités structurelles à manipuler de manière séparée pour pouvoir leur appliquer des procédures de déploiement différentes.

Par exemple, si on reconsidère un composant qui offre les services d'un agenda partagé, l'administrateur de ce composant peut vouloir, pour des raisons de répartition de charges, déployer certaines parties du composant concernant, par exemple, les services liés à la gestion de réunions, la gestion des absences et la gestion des agendas personnels sur différents serveurs. Pour satisfaire cet objectif, il est nécessaire de fractionner ce composant en trois (sous-) composants offrant chacun un des ensembles de services spécifiés par l'administrateur, i.e. gestion des réunions, gestion des absences et gestion des agendas personnels.

- L'adaptation par rapport aux ressources disponibles

L'infrastructure d'exécution d'un composant peut, par exemple, permettre ou non des exécutions parallèles, disposer ou non des ressources nécessaires pour l'installation et l'exécution de l'ensemble des services de ce composant. Ainsi, dans le cas d'infrastructure permettant des exécutions parallèles (e.g. multiprocesseurs, cluster), il serait possible, grâce à l'adaptation structurelle, de partager l'ensemble des services du composant en différents sous-ensembles définis par différents sous-composants pouvant être exécutés comme des processus parallèles. Aussi, dans le cas de ressources non satisfaisantes ou limitées, il serait possible, dans une première étape, de définir certains services du composant (les moins critiques) dans une nouvelle entité (sous-composant) créée par adaptation structurelle et après de déconnecter (masquer) ce dernier du reste de l'application.

En reprenant l'exemple de l'agenda partagé, on peut imaginer que cette application soit installée sur un ensemble d'assistants personnels (PDA) attribués à chacune des personnes concernées par cette application. Suivant les ressources disponibles sur chaque assistant, il serait possible de charger uniquement une sous-partie des services offerts par cette application (ce composant). Ceci serait possible par le fractionnement du composant initial en deux sous-composants. Le premier offre uniquement les services qui vont être disponibles sur l'assistant. Le deuxième offre les services masqués dans l'assistant. Ce dernier composant sera déconnecté du premier. Par exemple, les services à charger sur un assistant pourraient être ceux liés à la gestion d'agendas personnels et à la gestion d'absences. Les services à masquer seront ainsi ceux liés à la gestion des réunions.

- L'adaptation pour l'interfaçage avec des composants structurellement hétérogènes

L'interfaçage d'un composant logiciel avec d'autres composants nécessite la vérification de l'adéquation des services fournis et requis mutuellement par ces composants. Cependant, dans certains cas, malgré que les services requis par l'un puissent être fournis par l'autre, l'assemblage de deux composants ne peut être réalisé. En fait, cette impossibilité d'assemblage peut être due au fait que les composants logiciels, structurant leurs services en interfaces, peuvent proposer d'organiser ces services de manières différentes (interfaces différentes), de sorte que les interfaces requises et fournies ne peuvent pas être mises en correspondance[1].

[1] La solution la plus préconisée dans la littérature, pour résoudre le problème de l'interopérabilité des interfaces, consiste à utiliser des composants adaptateurs ou les connecteurs adaptateurs quant il s'agit des architectures logicielles.

Dans ce cas, l'adaptation structurelle permet de restructurer les services des composants assemblés en redéfinissant les interfaces requises et fournies. Cette mise à jour de la structure des interfaces permet, ainsi, d'offrir une correspondance exacte entre les interfaces des composants. Cette adaptation réalisée sans changement de la sémantique des services des composants leur permet d'être correctement assemblés.

3 Processus et modèle d'adaptation structurelle

L'adaptation structurelle d'un composant logiciel est réalisée suivant un ensemble d'étapes qui constitue le processus d'adaptation. Ce dernier se compose de quatre grandes étapes qui sont la spécification de l'adaptation, la mise à jour de la structure, l'assemblage des entités créées et enfin l'intégration du résultat de l'adaptation.

3.1 Spécification de l'adaptation

Cette première étape du processus d'adaptation consiste à spécifier les résultats attendus de l'adaptation en précisant la liste des sous-composants constituant la nouvelle structure interne du composant à adapter. Elle consiste à spécifier au travers d'une directive[2] spéciale le composant sur lequel porte l'adaptation (i.e. CompAadapté) ainsi que l'ensemble des nouveaux composants résultats de l'adaptation. De cette manière, sont spécifiés, pour chaque nouveau composant, ses ports, et pour chaque port, les interfaces associées (i.e. CompDef {PortDef{[||] InterfaceDef}+}+}*).

```
StructuAdapt (CompAadapté, {CompDef = {PortDef = {[||] InterfaceDef}+}+}*)
```

Par exemple, l'objectif de l'adaptation structurelle du composant Agenda-partagé (cité précédemment) est de restructurer l'ensemble des services du composant en un ensemble de sous-parties où chaque ensemble sera défini et offert par un sous-composant. Ainsi, les sous-composants pourront être déployés séparément sur des sites différents qui peuvent être distribués ou non. Par exemple, les parties structurelles qui concernent respectivement les bases de données, la gestion de l'agenda et la gestion des réunions peuvent être spécifiées pour être installées sur différents sites (*Droit, MiseAjourDroit, Absence, Dates…* sont les interfaces du composant Agenda-Partagé à adapter ; *BaseDeDonnées, Agenda, GestionnaireDeRéunion* sont les nouveaux sous-composants à créer).

```
AgendaP.AdaptInterface. StructAdapt (
{ BaseDeDonnées          = {BDD = { Droit, MiseAjourDroit } },
  Agenda = {      P-Absence = { Absence, MiseAjourAbsence },
                  P-Date = { Dates, MiseAjourAgenda }      },
  GestionnaireDeRéunion = { P-Réunion= {Réunion, MiseAjourReunion}}     })
```

3.2 Mise à jour de la structure

La mise à jour de la structure interne d'un composant consiste à découper cette structure en plusieurs sous-parties, chacune représentée par un sous-composant. Pour garantir le bon fonctionnement des sous-composants obtenus par cette opération, nous avons identifié un ensemble de problèmes à résoudre et par conséquent nous avons proposé les solutions adéquates.

- **Garantir l'intégrité structurelle et fonctionnelle des composants issus de l'adaptation.** Il s'agit d'assurer que chaque entité structurelle liée de manière directe ou indirecte à toutes les interfaces du composant soit accessible dans l'espace de nom de ce composant. Dans ce cadre, une entité structurelle désigne tout élément identifiable de manière unique faisant partie de la structure du composant (i.e. faisant partie de son espace de noms). Les entités structurelles peuvent être composées d'autres entités structurelles (e.g. les ports sont composés d'interfaces et les interfaces sont composées de services). Elles peuvent être dépendantes entre elles à travers deux types de liens : les liens

[2] Cette directive est spécifiée en utilisant une notation proche du langage XML [XML05]. Les symboles « + », « * », « {} » et « || » désignent respectivement « un ou plus », « zéro ou plus », « un ensemble » et « le degré de priorité ». Le symbole « || » est utilisé quand une interface est définie par plusieurs sous-composants. Dans ce cas, le symbole « || » placé devant le nom d'une interface indique que, lors de l'étape d'intégration, c'est cette dernière qui sera utilisée.

structurels et les liens comportementaux. Dans ce sens, un port composé d'une interface est structurellement lié avec cette dernière. Cependant, si par exemple, une interface I1 composée d'une méthode M1 qui fait appel à une méthode M2 représentant un service offert par l'interface I2, alors les interfaces I1 et I2 sont liées par un lien de dépendance comportementale.

Pour assurer l'intégrité structurelle des composants issus de l'adaptation, il est nécessaire de déterminer le graphe des dépendances structurelles et comportementales d'un composant créé par adaptation (voir figure 2). Les sommets de ce graphe sont l'ensemble des ports et des interfaces spécifiés par la directive d'adaptation ainsi que de l'ensemble des entités avec lesquelles l'ensemble de départ est lié directement ou indirectement au travers de liens structurels ou comportementaux. Ce graphe de dépendance est construit grâce à l'analyse du code source du composant à adapter.

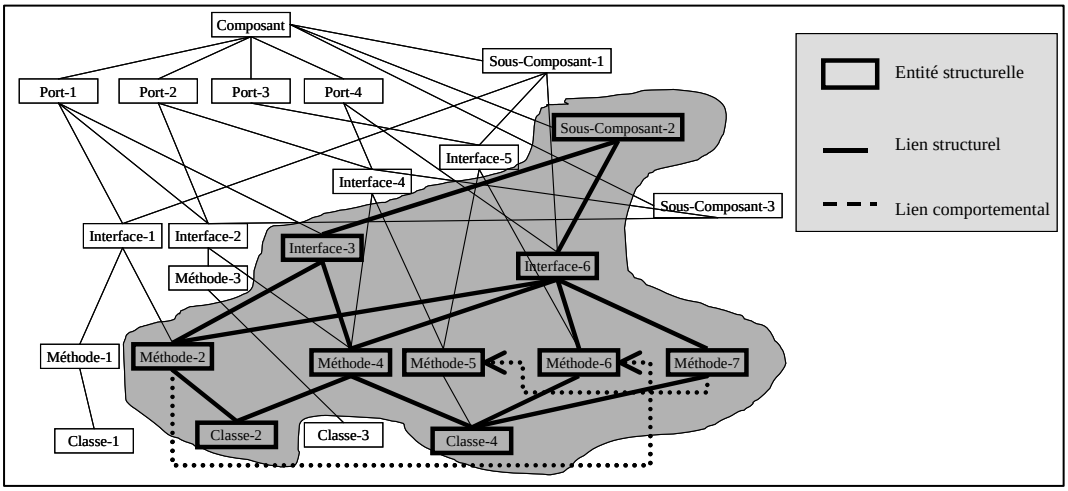

Figure 2. *Les dépendances structurelles et comportementales*

- **Gérer le partage des ressources.** Une ressource désigne n'importe quelle entité structurelle faisant partie de la définition du composant à adapter, qui est identifiable de manière unique, manipulable et possédant un état qui peut être mis à jour. Par exemple, pour des composants logiciels implémentés en utilisant l'approche objet et où chaque composant est un objet, un attribut de cet objet est une ressource. Aussi, un objet défini dans l'espace de noms d'un composant est une ressource. Ainsi, suite à l'étape de mise à jour de la structure et la création de nouveaux composants, une ressource peut se trouver nécessaire pour assurer l'intégrité de plusieurs composants issus de l'adaptation. De ce fait, le problème consiste à proposer une solution adéquate pour garantir ce partage. Nous avons opté pour une solution basée sur la duplication des ressources. Elle consiste à créer une copie locale de chaque ressource partagée pour chaque composant la partageant. Par conséquent, il est nécessaire de garantir la cohérence des états locaux d'une ressource partagée (un état dans chaque nouveau composant partageant la ressource). La gestion de la cohérence du partage de ressources est assurée par l'étape d'assemblage des composants (voir section 3.3).

3.3 Assemblage des nouvelles entités

La troisième étape du processus d'adaptation est l'assemblage des nouveaux composants générés par l'étape de mise à jour. Cette étape permet de produire par l'assemblage de l'ensemble des composants issus de l'adaptation les mêmes services (à travers les mêmes interfaces) proposés par le composant avant son adaptation. Ainsi, l'assemblage matérialise l'ensemble des liens de dépendances existants entre les différents composants. Comme, nous l'avons expliqué précédemment, ces liens consistent en le partage de ressources. En effet, la réalisation du partage des ressources par duplication nécessite la définition d'un mécanisme permettant d'assurer la cohérence des états des différentes copies locales des ressources partagées. Ce mécanisme de gestion de partage se charge, ainsi, de deux tâches :

La première tâche consiste en la communication entre les différents composants définissant la ressource partagée pour pouvoir préserver des états cohérents de cette ressource. Cette tâche est accomplie au travers de l'interface de cohérence (voir figure 3). Pour la satisfaction de cette condition, nous définissons, pour chaque composant utilisant la ressource, deux interfaces.

- La première interface est requise et synchrone. Elle permet au composant en question de notifier au reste des composants partageant avec lui une ressource, le changement d'état de cette dernière. Cette interface est synchrone parce que le composant ayant mis à jour l'état de la ressource ne peut continuer son exécution qu'après que les autres composants partageant cette ressource prennent effectivement ce changement d'état en compte. C'est la garantie que les composants se trouvent dans des états cohérents et par conséquent leurs services le sont aussi.

- La deuxième interface, définie comme fournie, permet à un composant de recevoir les notifications de mises à jour de ressources partagées.

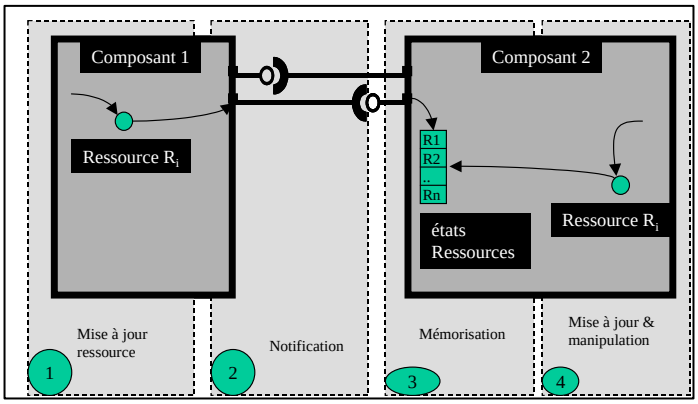

Figure 3. *Interfaces de communication*

La deuxième tâche concerne l'interdiction de l'accès multiple et simultané à une ressource. Elle est réalisée au travers de l'interface de synchronisation (voir figure 4). Toutes les opérations de manipulation des ressources doivent être mutuellement exclusives. Pour cela, nous définissons, pour chaque composant utilisant la ressource, deux interfaces.

- La première interface définie comme requise et synchrone lui permet d'acquérir le droit d'accès à la ressource. Cette interface garantit qu'à un instant donné, un et un seul accès à la ressource est possible.
- La deuxième interface définie comme fournie et synchrone permet à un composant de notifier aux composants demandeurs la disponibilité d'une ressource.

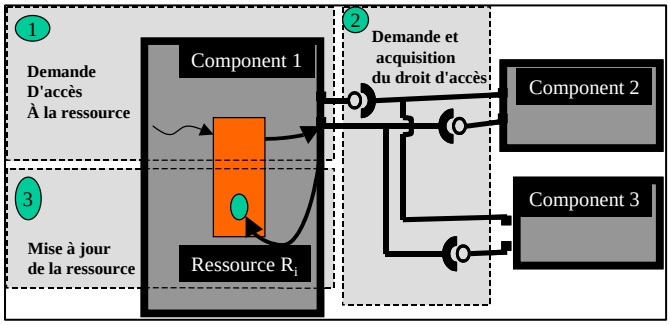

Figure 4. *Interfaces de synchronisation*

3.4 Intégration du résultat de l'adaptation structurelle

La troisième étape du processus d'adaptation est l'intégration, dans l'application sous-jacente, du résultat de l'adaptation structurelle produit par les deux premières étapes du processus. Il s'agit de réaliser l'assemblage des nouveaux composants, résultats de l'adaptation, avec le reste des composants de l'application. L'intégration doit permettre à l'application de continuer à s'exécuter sans aucun changement par rapport à sa configuration de départ (avec le composant avant son adaptation structurelle). Pour cela, il est nécessaire de satisfaire les propriétés suivantes :

- Les autres composants de l'application ne doivent pas pouvoir accéder, après l'adaptation, à d'autres services que ceux disponibles par le composant avant son adaptation. Les interfaces créées, pour la

réalisation du processus d'adaptation, doivent être des services internes aux nouveaux composants créés (i.e. interface de communication et synchronisation).

- Les nouveaux (sous-) composants doivent être manipulables, dès leur création, comme des entités indépendantes. Par exemple, il serait possible de déployer de manière indépendante ces composants sans que cela entraîne des modifications sur le reste de l'application.

Deux solutions sont possibles pour assurer l'intégration du résultat de l'adaptation :

La première consiste à considérer tous les composants issus de l'adaptation comme des entités autonomes et donc accessibles individuellement par les autres composants de l'application. Le problème est que les autres composants de l'application ne doivent pas accéder aux interfaces issues de l'adaptation. Il est donc indispensable de mettre en place un système d'authentification des composants.

La deuxième approche qui est celle que nous avons retenue consiste à encapsuler la nouvelle structure, qui est le résultat de l'adaptation, dans un nouveau composant que nous appelons composant virtuel. On définit ainsi un composant virtuel comme étant un composant composite qui offre les mêmes services que le composant adapté. Il a la même structure externe que ce dernier mais pas la même structure interne (résultat de l'adaptation du composant initial). Les interfaces d'intégrité, de cohérence et de synchronisation sont inaccessibles de l'extérieur du composant virtuel, ce qui rend sécurisé l'accès aux différentes ressources rendues disponibles à travers les interfaces des composants issus de l'adaptation.

4 Conclusion et perspectives

Nous avons présenté dans cet article une approche d'adaptation de composants logiciels. Cette dernière est basée sur la considération d'une nouvelle facette d'adaptation : la structure du composant. L'approche d'adaptation structurelle que nous proposons peut répondre à de multiples besoins : déploiement flexible, assemblage flexible, adéquation par rapport aux ressources disponibles, etc. Nous avons expérimenté notre approche par une implémentation réalisée en utilisant la plate-forme Julia [Bru03], qui est l'implémentation Java du modèle de composants Fractal [Bru04]. L'approche présentée dans ce papier traite principalement l'adaptation structurelle statique d'un composant logiciel. Dans ce cas, elle nécessite l'arrêt du composant à adapter, le lancement du processus d'adaptation dans une plate-forme d'adaptation et le remplacement du composant initial par le nouveau composant généré. Néanmoins, l'ensemble de l'approche présentée est assez générique pour être applicable à l'adaptation dynamique. La réalisation de ce processus de manière dynamique nécessite le développement des mécanismes de gestion de la connexion / déconnexion du composant, le chargement de dynamique de code source et la génération du code binaire. Cette gestion de la dynamicité constitue une perspective du travail présenté dans cet article. L'adaptation structurelle de composants logiciels binaires constitue une autre voie pour nos travaux futurs.

Références

[Bru00] E. Bruneton et M. Riveill, JavaPod : une plate-forme à composants adaptables et extensibles", INRIA, 2000.
[Bru03] E. Bruneton. Julia Tutorial. http://fractal.objectweb.org/tutorials/julia/
[Bru04] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani: An Open Component Model and Its Support in Java. CBSE 2004: 7-22
[Eri03] H-E Eriksson, UML 2 Toolkit, Wiley editon, ISBN: 0471463612, October 2003.
[Har02] E. R. Harold , W. Scott Means. XML in a Nutshell. O'Reilly, 2002.
[Ket02] A. Ketfi , Automatic Adaptation of Component-based Software: Issues and Experiences, PDPTA, 2002
[Led01] T. Ledoux. Etat de l'art sur l'adaptabilité. Projet RNTL ARCAD D1.1 12/12/2001.
[Mar02] R. Marvie et M.-C. Pellegrini. Modèles de composants, un état de l'art. RSTI- L'objet – 8/2002.
[Szy98] [Szy98]. Component Software. Beyond Object-Oriented Programming - Addison-Wesley / ACM Press, 1998.

Services de modélisation et Web Services Application sur le ModelBus[1]

Xavier Blanc, Marie-Pierre Gervais, Prawee Sriplakich

Laboratoire d'Informatique de Paris 6 (LIP6),University Paris VI
8, rue du Capitaine Scott, 75015 Paris
{Xavier.Blanc, Marie-Pierre.Gervais, Prawee.Sriplakich}@lip6.fr

Mots-clefs – Keywords

MDA, Modèle, Interopérabilité, Plate-forme, Outils
MDA, Model, Interoperability, Middleware, Tools

Résumé – Abstract

La publication par l'OMG (Object Management Group) de sa nouvelle ligne directrice Model Driven Architecture (MDA) a propulsé les modèles au coeur du processus de développement des applications réparties. Un objectif majeur du MDA est de rendre les modèles productifs via l'automatisation des étapes du cycle de développement, assurée par différents services de modélisation (ex. édition, transformation, vérification). Apparaît alors la nécessité de disposer d'environnement de développement intégré orienté modèle (iMDDE : integrated Model Driven Development Environment). Considérant la préexistence d'outils de modélisation qu'il conviendrait de réutiliser, nous préconisons une solution fondée sur un ensemble ouvert d'outils offrant chacun des services de modélisation. Se pose alors un problème d'interopérabilité entre les services de modélisation. Pour y répondre, nous avons proposé ModelBus, une approche générique pour la construction d'iMDDE fondée sur l'adaptation de plates-formes d'exécution existantes, pour assurer la prise en compte des modèles. Nous présentons dans ce papier les techniques d'adaptation de la plate-forme Web Services. Un prototype a été réalisé avec la plate-forme Axis d'Apache et Eclipse Modeling Framework (EMF).

OMG's Model Driven Architecture (MDA) emphasizes on using models in the software development life cycle. The main objective is the productivity via the automation of development life cycle using modeling services (e.g. model edition, transformation, verification…). This leads to the needs of an integrated Model Driven Environment (iMDDE) enabling various modeling services to interoperate. In order to build an iMDDE, our approach "ModelBus", is based on adapting available middleware to support modeling services. ModelBus is adopted by the IST European project "ModelWare". In ModelWare, one of the target middleware is the Web Services. However, the Web Services do not support the particular requirements of modeling services. Therefore, in this paper, we propose techniques for adapting Web Services to support such requirements and also implement a prototype using Axis application server and Eclipse Modeling Framework (EMF).

---

[1] Le travail présenté dans ce papier est supporté par le projet européen IST ModelWare, contrat 511731

1 Introduction

Depuis la publication du MDA (Model Driven Architecture) par l'OMG (Object Management Group), il est de plus en plus évident que les modèles sont placés au centre du cycle de développement des applications réparties. En effet, les modèles sont aussi bien utilisés pour l'expression des besoins ou pour l'analyse et la conception, que pour le déploiement d'applications ou même leur supervision.

Cependant, comme cela est clairement précisé dans [OMG03a], le MDA n'est pas uniquement dédié à l'élaboration d'un ensemble de modèles. C'est aussi et surtout la possibilité d'automatiser certaines étapes du cycle de développement en faisant en sorte que ces modèles soient productifs ! L'objectif étant bien évidemment de gagner en rapidité et en qualité pour le développement d'applications réparties.

La mise en œuvre du MDA nécessite donc la mise à disposition de plusieurs services de modélisation tels que le stockage [Blanc04a], l'édition [Ledeczi01], la transformation [Czarnecki03] [Bézivin03], la vérification [Gogolla00] et l'exécution [Jørgensen02] de modèles (ceci est une liste non exhaustive). Un service de modélisation est, pour nous, une opération d'automatisation qui prend des modèles comme paramètres d'entrée et de sortie. Ainsi, si nous prenons un exemple relativement classique de scénario MDA tel que : « un concepteur édite un modèle UML [OMG01], puis il vérifie des contraintes Object Constraint Language (OCL) [OMG01] sur celui-ci, et enfin le transmet à un autre concepteur qui va transformer le modèle UML en un modèle Enterprise JavaBeans (EJB) », alors nous pouvons dire que ce scénario nécessite la mise à disposition de trois services de modélisation : édition de modèles UML, vérification de contraintes OCL sur les modèles UML et transformation de modèles UML vers des modèles EJB.

Les environnements de développement permettant cette mise en œuvre du MDA (que nous appelons iMDDE : integrated Model Driven Development Environment[2]) doivent alors offrir un ensemble non négligeable de services de modélisation et assurer une intégration idéale de ceux-ci. L'idée est de pouvoir utiliser différents services de modélisation conjointement.

Etant donné qu'il existe déjà plusieurs outils fournissant certains services de modélisation, il nous semble optimal de réutiliser ces services. Pour ce faire, les iMDDEs doivent donc être conçus pour intégrer des outils existants. Etant donné que ces outils échangent entre eux des modèles et qu'ils partagent leurs services de modélisation, la construction de ces environnements doit faire face à une problématique d'interopérabilité au niveau des modèles.

Les plates-formes d'exécution existantes (telles que CORBA, EJB ou Web Services) permettent l'interopérabilité de services en général; cependant elles ne répondent pas aux besoins particuliers des services de modélisation (ex. la représentation et le transport des modèles). Elles sont donc insuffisantes pour résoudre notre problématique. Des spécifications comme Meta Data Facility (MOF) [OMG02], Java Metadata Interface (JMI) [JCP02], Eclipse Modeling Framework (EMF) [Eclipse04] et XML Metadata Interchange (XMI) [OMG03b] ont été créées pour résoudre ces problèmes de représentation des modèles dans les plates-

[2] MDD est ici utilisé pour généraliser l'approche MDA et ne pas la restreindre à la vision OMG.

formes d'exécution, mais elles n'offrent pas de la notion de services (telle que celle proposée par Web Services). Elles ne sont donc pas non plus suffisantes pour notre problématique.

Nous avons proposé dans [Blanc04b] une approche générique permettant d'adapter les plates-formes existantes afin qu'elles supportent la notion de service de modélisation. Cette approche, nommée ModelBus, permet ainsi d'intégrer différents services de modélisation quel que soient les outils fournisseurs des services. Par ailleurs, ModelBus fait partie du projet européen ModelWare visant à fournir un environnement de développement MDD (iMDDE).

Selon les attentes de ModelWare, la plate-forme cible à adapter est celle des Web Services. Cet article présente donc l'adaptation de cette plate-forme. L'article est organisé comme suit. La partie 2 explique le problème de la mise en œuvre de services de modélisation sur la plate-forme Web Services. La partie 3 présente notre solution pour l'adapter au ModelBus. La dernière partie conclut notre travail.

2 Web Services vs Services de Modélisation: Difficultés

La plate-forme Web Services propose une solution d'interopérabilité grâce à deux concepts. Le premier propose **la définition de services**. En effet, c'est un langage de description de services (WSDL) [W3C03b] qui permet d'identifier la *signature de services* (les inputs et outputs) et le *mécanisme d'invocation de services*. Notons que la version actuelle du WSDL propose un seul mécanisme qui est « SOAP HTTP Binding» [W3C03a]. L'autre concept propose **la mise en œuvre des services** décrits par WSDL. En effet, il est possible de réaliser les services avec plusieurs langages de programmation. C'est pourquoi il existe plusieurs *plates-formes concrètes* pour des Web Services telles que Axis [Apache04], J2EE de Sun, Microsoft .Net, Perl-PHP-Python Web Services [ASPN].

La plate-forme Web Services est conçue pour les services en général. Toutefois, les services de modélisation ont des besoins particuliers qui sont dédiés aux modèles. Ainsi, des difficultés apparaissent dans la réalisation de services de modélisation sur cette plate-forme car celle-ci ne prend pas en compte ces besoins particuliers. Nous justifions ci-dessous pourquoi les deux concepts de Web Services (la définition de Web Services, la mise en œuvre de Web Services) ne sont pas bien adaptés aux modèles.

2.1 Définition de Web Services vs. Models

Les signatures de services proposées par WSDL sont basées sur XML. C'est-à-dire, les inputs et outputs de services sont des données XML. Cependant, ceux de services de modélisation sont des modèles. Ainsi, le WSDL n'est pas bien adapté pour décrire les services de modélisation. En plus, **le mécanisme d'invocation** proposé par WSDL (SOAP http binding) est basé sur XML. Plus précisément, les paramètres d'invocation de services sont transportés dans un message SOAP via le protocole HTTP. Cependant, ce mécanisme ne spécifie pas comment transporter les paramètres de type modèle dans le message SOAP. Ainsi, il a besoin d'extension pour supporter l'invocation de services de modélisation.

2.2 Mise en œuvre de Web Services vs. Modèles

Afin de mettre en œuvre les Web Services, les plates-formes concrètes proposent la génération de « stubs » permettant traiter la réception et l'émission de messages SOAP. En effet, ces « stubs » permettent de traduire l'invocation à distance (via les messages SOAP) vers l'invocation locale (par exemple l'invocation d'une méthode Java). Le traitement de messages SOAP inclut en particulier **la sérialisation (marshalling) et la désérialisation (unmarshalling) de paramètres de services**. L'objectif est d'effectuer la traduction entre le format d'échange (données XML dans les message SOAP) et le format de traitement (ex. objets Java, C# etc).

Nous constatons que, parmi toutes les plates-formes actuelles, aucune ne propose la sérialisation et la désérialisation appropriée aux modèles. Par exemple, la plate-forme Axis propose un format de traitement qui est trop proche du format XML alors qu'il existe d'autres formats dédiés à la représentation de modèles (ex. Java Metadata Interface–JMI, Eclipse Modeling Framework–EMF). Pour la mise en oeuvre de services de modélisation, les plates-formes Web Services manquent ainsi un support pour la (dé) sérialisation de modèles.

En plus, les plates-formes Web Services permettent d'assurer la fiabilité des paramètres des services. Plus précisément, ils proposent **la validation de paramètres** par rapport à leurs types définis. En effet, en Web Services, les paramètres de services étant considérés comme des données XML, les Schémas XML sont donc utilisés pour la validation. Cependant, cette validation ne correspond pas aux besoins de services de modélisation car les paramètres de ces derniers sont des modèles et peuvent avoir différentes représentations XML. Pour la mise en oeuvre de services de modélisation, les plates-formes Web Services manquent ainsi un support pour la validation de modèles.

3 Adaptation de la plate-forme Web Services : Solution proposée

3.1 Concepts globaux

Rappelons que, conceptuellement notre approche « ModelBus » permet d'adapter n'importe quelles plates-formes aux services de modélisation. Ainsi, nous présentons d'abord les concepts d'adaptation de manière globale. ModelBus est composé de deux parties : Abstraite et Concrète. La partie **Abstraite** permet de décrire des services de modélisation grâce à un méta-modèle. Ce méta-modèle conceptualise la notion de services de modélisation (Figure 1). Il est indépendant des plates-formes d'implémentation de services. Sa valeur ajoutée est qu'il permet de définir les paramètres de type modèle (nous considérons qu'un type modèle est un ensemble de méta-classes). Ainsi, il permet de décrire des services de modélisation, tels que le vérificateur des contraintes OCL sur un modèle UML, le transformateur UML vers EJB, etc.

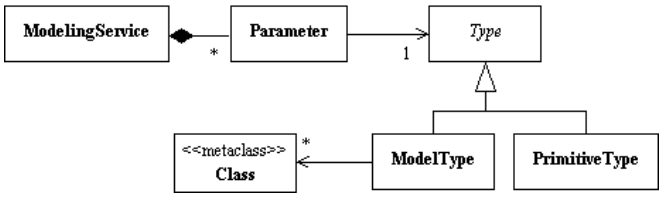

Figure 1 : méta-modèle Abstrait du ModelBus (version simplifiée)

La partie **Concrète** permet la mise en œuvre de services de modélisation sur des plates-formes d'exécution. Elle a deux concepts. Le premier est la génération automatique d'adaptateurs liant les outils fournisseurs de services et les plates-formes d'exécution. Le deuxième est l'extension de la plate-forme d'exécution afin d'ajouter les supports de modèles nécessaires (comme identifiés dans le paragraphe 2.2). Grâce à cette partie concrète, il est possible de réutiliser les plates-formes existant pour mettre en œuvre les services de modélisation.

La Figure 2 illustre l'utilisation du ModelBus. Un fournisseur de services qui souhaite brancher son outil doit, premièrement, décrire les services de modélisation offerts par l'outil (❶). Pour faire cela, il doit élaborer un modèle grâce au méta-modèle « Abstraite ». Deuxièmement il doit générer, à partir de la description des services de modélisation, un adaptateur pour une plate-forme d'exécution spécifique (❷). Une fois que ces deux étapes ont été réalisées, les services de modélisation sont accessibles pour les utilisateurs de la plate-forme (❸).

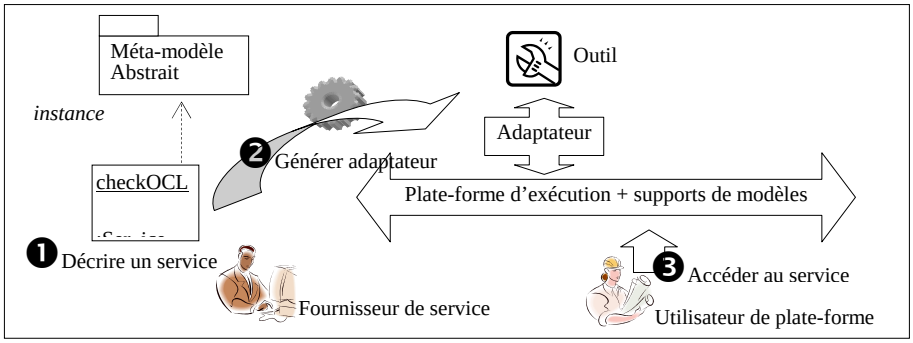

Figure 2. Utilisation du ModelBus pour brancher un outil

3.2 Concepts dédiés à la plate-forme Web Services

Comme présenté ci-dessus, seule la partie Concrète de ModelBus est spécifique aux plates-formes d'exécution. Pour réaliser la partie Concrète sur Web Services, nous proposons donc d'une part **la génération d'adaptateurs** pour la plate-forme Web Services et d'autre part **l'extension de la plate-forme** Web Services pour ajouter les supports de modèles.

Les adaptateurs correspondent à des interfaces WSDL et des « stubs » qui sont spécifiques à la *plate-forme concrète* utilisée (par ex. pour Axis, les « stubs » sont des classes Java). Afin de générer cet adaptateur, dans premier temps, nous traduisons donc la description de services de modélisation (modèle « Abstrait ») vers des interfaces WSDL. Afin de résoudre le problème de définition de paramètres (paramètres en XML dans WSDL vs paramètres modèle

dans ModelBus), nous utilisons le standard XMI, qui permet de représenter les modèles en XML.

Dans un deuxième temps, pour que les « stubs » prennent en compte les modèles, nous proposons l'extension de « stubs » dédiés aux modèles. Cette extension est illustrée dans la Figure 3. Le *stub de base* généré par la plate-forme concrète sert à gérer la réception et l'émission de messages SOAP (requête et réponse). Notre extension (*Stub4Model*) permet d'ajouter les supports de modèles (correspondant aux besoins de services de modélisation : la (dé) sérialisation, la validation de modèles). Grâce à notre extension, les services de modélisation peuvent être mis en œuvre sous forme Web Services.

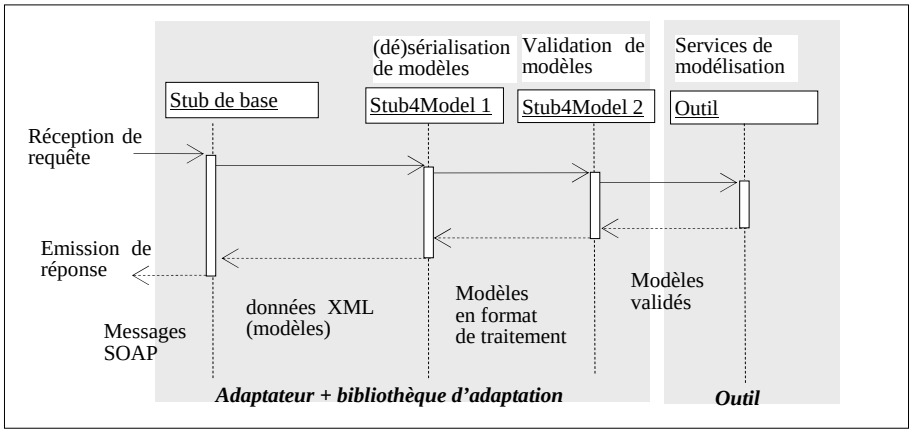

Figure 3. ***Extension de stubs pour services de modéliasion***

Nos concepts sont validés par le prototype de l'adaptation de la plate-forme concrète Axis. Axis a été choisie car elle est basée sur Java et ainsi cohérente avec des formats de traitement de modèles les plus utilisés par les outils actuels : MOF-IDL, EMF et JMI. Dans ce prototype, les supports de modèles sont implémentés à l'aide de l'implémentation d'EMF [Eclipse04] et de JMI [Blanc04a] qui permet de manipuler le contenu de modèles afin de le sérialiser et le valider.

4 Conclusion

Nous avons, dans cet article, présenté la nouvelle problématique de l'interopérabilité au niveau de modèles, qui est au cœur de la construction des environnements de développement support au MDA (iMDDE : integrated Model Driven Development Environment). Fondée sur l'existence des plates-formes d'exécution qui traite l'interopérabilité de services en général, notre solution, ModelBus, les adapte afin d'ajouter les supports dédiés aux modèles. Nous nous sommes focalisé sur un cas concret de l'adaptation de la plate-forme Web Services avec ModelBus. Grâce à cette adaptation, il est possible de mettre en oeuvre des services de modélisation sous forme de Web Services.

Pour les perspectives, nous envisageons de valider ModelBus dans le développement réel. Pour atteindre cet objectif, de nombreuses difficultés sont posées par l'hétérogénéité des modèles et celle des outils. Nous traitons ces problèmes dans le cadre du projet européen ModelWare qui vise à outiller intégralement le MDA en utilisant le ModelBus pour assurer l'interopérabilité des différents outils de modélisation réels.

Références

[ASPN] ActiveState Programmer Network (ASPN), http://aspn.activestate.com/ASPN/ WebServices.

[Apache04] Apache Web Services Project, Axis User's Guide, http://ws.apache.org/axis, 2004.

[Bézivin03] Bézivin J. *et al.*: « First experiments with the ATL model transformation language: Transforming XSLT into XQuery », *Second OOPSLA Workshop on Generative Techniques in the context of MDA*, 2003.

[Blanc04a] Blanc X., Bouzitouna S. et Gervais M-P., « A Critical Analysis of MDA Standards through an Implementation: the ModFact Tool », *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, 2004.

[Blanc04b] Blanc X., Gervais M-P. et Sriplakich P., «Model Bus: Towards the Interoperability of Modeling Tools », *Model Driven Architecture: Foundations and Applications (MDAFA 2004)*, 2004.

[Czarnecki03] Czarnecki K., Helsen S., « Classification of Model Transformation Approaches », *Second OOPSLA Workshop on Generative Techniques in the context of MDA*, 2003.

[Eclipse04] Eclipse Project, The Eclipse Modeling Framework (EMF) Overview, http://www.eclipse.org/emf/, 2004.

[Gogolla00] Gogolla M *et al.*, « Validating UML Models and OCL Constraints », *Third International Conference on the Unified Modeling Language*, 2000.

[JCP02] Java Community Process, Java Metadata Interface (JMI) Specification version 1.0, http://www.jcp.org, 2002.

[Jørgensen02] Jørgensen J., Christensen S., « Executable Design Models for a Pervasive Healthcare Middleware System », *Fifth International Conference on the Unified Modeling Language*, 2002.

[Ledeczi01] Ledeczi A. *et al.*: «The Generic Modeling Environment», *Workshop on Intelligent Signal Processing*, 2001.

[OMG01] OMG, Unified Modeling Language Specification version 1.4, document no: formal/01-09-67, 2001.

[OMG02] OMG, Meta Object Facility version 1.4, document no: formal/2002-04-03, 2002.

[OMG03a] OMG, MDA Guide Version 1.0.1, document no: omg/2003-06-01, 2003.

[OMG03b] OMG, XML Metadata Interchange (XMI) Specification version 2.0, document no: formal/03-05-02, 2003.

[W3C03a] W3C, SOAP Version 1.2, W3C Recommendation, 2003.

[W3C03b] W3C, Web Services Description Language (WSDL) Version 1.2, W3C Working Draft, 2003.

Vers un typage pour les composants logiciels

Bart George
Laboratoire Valoria Université de Bretagne Sud
Campus de Tohannic
F-56000 Vannes Cedex
Bart.George@univ-ubs.fr

Mots-clefs – Keywords

Typage, Composants logiciels, Propriétés non-fonctionnelles
Typing, Software Components, Non-Functional Properties

Résumé – Abstract

Les composants logiciels sont un concept encore récent et mal défini, manquant de base théorique universelle. Nous montrons que le typage pourrait apporter une réponse à ce problème et contribuer à établir une théorie viable pour les composants, notamment dans le domaine de la composabilité et de la substituabilité. Nous montrons également qu'une telle réponse ne peut pas se baser sur la seule approche "classique" du typage. Enfin, nous posons les bases de ce que doit être un typage réellement adapté aux composants

Software components are still a young and badly defined concept, with a lack of a universal theoretical basis. We show that typing could bring an answer to this problem, and help building a reliable theory for software components, especially for composability and substituability. We also show that such an answer cannot be based only on a "classical" approach of typing. Finally, we bring basic ideas on what a really component-oriented typing must be.

1 Introduction

L'architecture logicielle est actuellement une discipline à part entière, et le concept qui la sous-tend, celui de composant logiciel, a déjà donné lieu à de nombreux modèles. Cependant, malgré les efforts accomplis, il manque aux composants une théorie unifiée, qui serait capable d'apporter une réponse claire aux questions fondamentales "qu'est-ce qu'un composant ?" et "comment assemble-t-on correctement des composants ?" [SH04], surtout quand il s'agit de composants "sur étagère".

Nous pensons que le mécanisme de typage, qui a déjà servi de base à la vérification partielle de la correction des programmes et à la substitution d'éléments, pourrait apporter une réponse à ces différents problèmes. En effet, la notion de type abstrait a eu une influence majeure sur les concepts précurseurs de l'architecture logicielle, c'est-à-dire la programmation modulaire d'une part, et la programmation orientée objet d'autre part, et à ce titre les types abstraits sont considérés eux-mêmes comme un concept précurseur de l'architecture logicielle [Sha01]. De plus, bien que le typage ait fait l'objet de critiques, celles-ci visaient moins la notion en elle-même que l'usage qui en est fait, et leurs auteurs ont rappelé l'utilité de cette notion dans l'architecture logicielle. Ainsi, Dewayne Perry et Alexander Wolf [PW92] ont jugé le type (tel qu'on le connaissait) peu apte à décrire les propriétés architecturales, mais en même temps ils en ont recommandé l'usage au niveau de la conception des architectures.

Ce qui est critiqué n'est donc pas le concept de typage en lui-même, mais davantage la définition actuelle d'un type de composant (un ensemble d'interfaces fournies et requises, d'après la définition d'un composant donnée par Clemens Szyperski [Szy02]), qui n'est pas adaptée aux problèmes inhérents à ce domaine. En particulier, un véritable type de composant ne peut pas seulement décrire les services fournis et requis par ce composant (aspect fonctionnel), mais également la manière de fournir ou de requérir ces services (aspect non fonctionnel). De plus, les composants "sur étagère" étant créés par des tierces personnes, dans des environnements, des contextes et des langages différents, un véritable système de types de composants doit pouvoir être adapté à cette hétérogénéité. Nous pensons donc que le typage ne peut être utile aux composants qu'à condition qu'il soit défini *ad hoc* pour le paradigme composant.

Dans la deuxième section, nous présenterons notre vision de ce que doit être un type de composant. Nous expliciterons les relations entre les différentes formes de typage et de sous-typage dans la troisième section, avant de dresser dans la quatrième section un bilan comparatif des travaux qui ont été faits sur la question. Enfin, dans la dernière section, nous parlerons des perspectives possibles pour notre démarche.

2 Typage et sous-typage dans le monde composant

2.1 Définition d'un type de composant

Il convient tout d'abord de donner au type de composant une véritable définition qui mette en valeur son utilité : *un type de composant est l'ensemble des propriétés* ***structurées*** *et* ***comparables*** *de ce composant.* En effet, la spécification d'un type détermine quelles propriétés nous pouvons prouver [LW94], et pour substituer un composant par

un sous-type tout en conservant certaines propriétés, il faut savoir si les propriétés du sous-type sont les mêmes que celles du supertype, donc les comparer (voir plus bas pour les définitions de sous-type et de supertype). Cette définition s'accorde avec le **subtype requirement** défini par Barbara Liskov et Jeannette Wing [LW94] :

Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S were S is a subtype of T.

La comparaison des propriétés sous-entend leur structuration et leur hiérarchisation. En effet, l'utilité de comparer certaines propriétés peut changer d'un contexte à l'autre, et il est nécessaire de savoir quels éléments x et y doivent être comparés, et à quel moment.

2.2 Sous-typage et propriétés non-fonctionnelles

Un type `T` est un **sous-type** du type `U` (noté `T` $<:$ `U`) si une valeur du type `T` peut être utilisée à chaque fois qu'une valeur du type `U` est attendue. `U` est alors appelé **supertype** de `T`. Cette relation est concrétisée par la **règle de substitution**, qui stipule que si `S` $<:$ `T` et que `e` est de type `S`, alors `e` est aussi de type `T`. Une telle relation est réflexive et transitive. Selon ce qu'on entend par "utiliser une valeur d'un type quand on attend celle d'un autre type", il existe trois niveaux de sous-typage.

Le premier niveau est un **sous-type de signature** : le sous-type conserve des signatures compatibles (pour les services décrits dans les interfaces du composant) avec celles de son supertype. En particulier, dans les interfaces fournies des composants, les méthodes du composant sous-type `A->A'` sont sous-types des méthodes du composant supertype `B->B'` quand il y a contravariance sur les arguments (`B` est sous-type de `A`) et covariance sur les résultats (`A'` est sous-type de `B'`). A l'inverse, dans les interfaces requises des composants, il y a covariance des arguments et contravariance sur les résultats.

Le deuxième niveau est **sémantique** ou **comportemental** : en plus de la compatibilité de signature, le sous-type doit aussi conserver (pour les services décrits dans les interfaces des composants) les invariants de son supertype tout en proposant, d'une part des préconditions plus faibles et des postconditions plus fortes pour les interfaces fournies, et d'autre part des préconditions plus fortes et des postconditions plus faibles pour les interfaces requises. A ce niveau, le sous-typage est séparé de l'héritage, car il spécifie non plus la syntaxe, mais le comportement. Barbara Liskov et Jeannette Wing [LW94], à partir de leur "subtype requirement", ont ajouté deux règles supplémentaires (et orthogonales) au sous-typage comportemental : la première règle impose au sous-type des contraintes plus fortes que celles de son supertype, tandis que la seconde étend le sous-type en lui donnant plus de méthodes que son supertype.

Le dernier niveau de sous-typage est **non-fonctionnel** : non seulement le sous-typage doit satisfaire les exigences décrites aux niveaux précédents, mais il doit aussi garantir la préservation des **propriétés non-fonctionnelles**. Ce terme mérite d'être explicité, car il est au coeur de notre démarche : contrairement aux propriétés fonctionnelles, propres au code et au comportement, qui, dans le cadre des composants logiciels, correspondent aux services fournis et requis d'un composant, les propriétés non-fonctionnelles décrivent la manière dont le composant rend ces services ou les attend. L'aspect non fonctionnel ne peut être omis. En effet, toute application a des exigences relevant de cet aspect. Cela peut être des exigences en terme de performance (l'exécution du service ne doit pas excéder 0,2

secondes, ou prendre plus de 16MB de RAM...), de maintenabilité (on doit pouvoir intégrer facilement des protextions par mot de passe...) de portabilité (l'application doit pouvoir s'exécuter autant sous Windows que sous Linux...), etc. Il est donc impensable de choisir tel ou tel composant sur simple critère d'adéquation fonctionnel sans avoir une idée de leur contribution vis-à-vis de ces exigences. C'est également l'aspect le plus difficile à décrire formellement, car si des tentatives ont été faites pour standardiser les différents sous-aspects non fonctionnels, aussi appelés "attributs qualité" (voir par exemple le standard ISO 9126), il y a en réalité autant de descriptions possibles que de types d'attributs, certains semblant plus "descriptibles" que d'autres (par exemple, les propriétés relatives à la performance semblent plus formelles que celles reliées à la maintenabilité).

3 Une nouvelle hiérarchie de sous-typage

Idéalement, un composant doit être une "boîte noire", dont on ne doit connaître que le strict minimum nécessaire à son utilisation (en particulier, le code doit être inaccessible). Plus cette description est minimale, et plus les possibilités de faire évoluer le composant (soit en lui ajoutant des fonctionnalités, soit en le substituant) sont nombreuses. Dans le cadre de tels composants "sur étagère", créés par des tierces personnes, dans des environnements et des langages différents, des informations telles que le nom du composant, de ses interfaces ou de ses opérations, ne veulent plus rien dire. Et la question se pose de savoir comment substituer un composant par un autre d'origine quelconque, dont on ne connait que la description minimale, et qui, vraisemblablement, n'a pas été déclaré comme sous-type du composant à substituer.

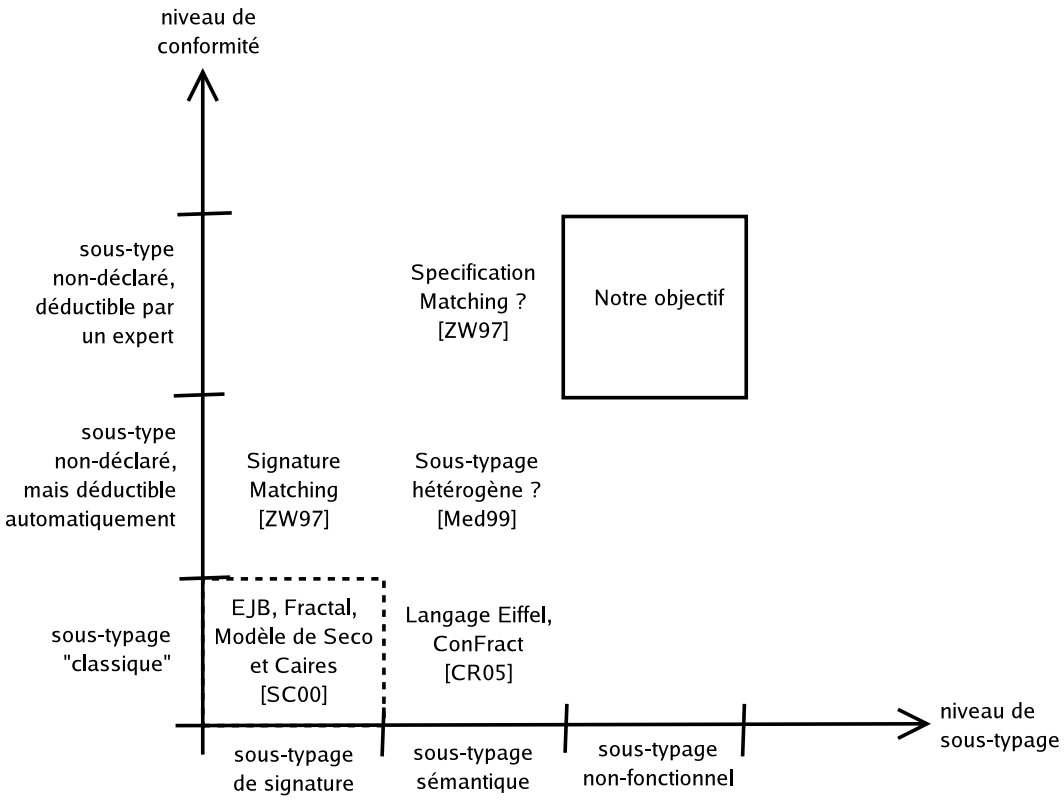

FIG. 1 – Hiérarchies de sous-typage

Face à une telle hétérogénéité, il est nécessaire d'introduire une autre hiérarchie de sous-typage, orthogonale à la hiérarchie "classique" (voir figure 1) qui prendrait en compte la difficulté de déterminer si un composant est sous-type d'un autre quel que soit leur environnement et leur origine :
- Au premier niveau se trouve le sous-typage "classique", où le sous-type est identifié par une simple déclaration ;
- Au deuxième niveau, le sous-type n'est pas déclaré comme tel, mais on peut le déduire automatiquement par la découverte de signatures compatibles ;
- Enfin, au troisième niveau, non seulement le sous-type est non-déclaré, mais il n'est pas déductible automatiquement. Une telle déduction exige la présence d'un expert (automate ou humain). Ce niveau contient les deux précédents.

Les deux axes de la figure 1 manifestent deux problématiques complémentaires :
- selon l'axe horizontal, aller vers une description de plus en plus fine et complète des propriétés d'un composant ;
- selon l'axe vertical, permettre une comparaison dans un contexte de plus en plus hétérogène entre deux composants.

Les propositions données par ConFract [CR05] portent le niveau de définition du type de composant au niveau sémantique grâce aux contrats. Ce travail peut être complété par les travaux sur les contraintes architecturales [FTS05] pour atteindre en partie le niveau non-fonctionnel. Pour répondre aux problèmes posés par l'axe vertical, l'introduction d'un intermédiaire a été le plus souvent utilisée. Dans le monde des composants, cet intermédiaire trouve sa place dans les connecteurs. Pour atteindre le troisième niveau de cet axe, nous pouvons bénéficier des résultats sur les abstractions de services [SKM01] qui proposent une définition des compatibilités de services allant au-delà de la simple signature.

4 État de l'art et comparaisons avec notre démarche

Des travaux ont naturellement été faits pour investiguer la notion de type de composant. Cependant, ils sont peu nombreux par rapport aux travaux sur d'autres domaines propres aux composants. Parmi eux, nous trouvons tout d'abord les modèles "basiques", qui se concentrent sur la définition formelle des services fournis et requis [SC00]. Plus intéressants sont les travaux qui cherchent à aller au-delà. Certains cherchent une méthode originale de structurer les types, par exemple en proposant plusieurs implémentations d'une même interface de collaboration [MO02] afin de créer des composants en combinant ces implémentations. D'autres utilisent une voie plus "naturelle", qui consiste simplement à étendre la notion de type. Dans le cas d'une adaptation dynamique des composants (c'est-à-dire à leur modification au cours de l'exécution d'un programme), le type pourrait être lui aussi adapté dynamiquement [ODP04]. Une autre méthode serait de contractualiser le type [Nie03] en rajoutant, en plus des services fournis et requis, un ensemble de contraintes qui conditionnerait la composition.

D'autres travaux, plutôt que d'investiguer le contenu d'un type de composant, préfèrent s'intéresser à "ce qu'il y a autour". Nous entendons par là deux sortes de choses. Nous avons d'une part les éléments d'un composant susceptibles d'être typés plutôt que le composant lui-même. On y trouve les connecteurs, qui peuvent contenir le comportement du

composant associé et la "glu" qui permettrait d'assembler les composants [Gar94]. On y trouve également les contrats, qui spécifient les interfaces du composant, et qui sont à plusieurs niveaux, investigués par Antoine Beugnard *et al.* [BJPW99] et quasi-similaires aux niveaux de sous-typage. Le niveau non-fonctionnel des contrats, en particulier, pourrait être réalisé en paramétrant les contrats [Reu01] en liant l'interface fournie d'un composant avec l'interface requise du même composant, et en la codant en fonction du contexte. Cela dit, il s'agit là de "négociation", et pas de substitution.

D'autre part, nous avons le typage à un autre niveau, non pas à l'intérieur d'un modèle, mais dans un univers constitué d'environnements, de contextes et de modèles différents. Les éléments architecturaux pourraient être vus comme des styles [GAO94] qui engloberaient les modèles de composants existants, chacun étant un style particulier, et qui seraient reliés entre eux via le sous-typage. Et en parlant de sous-typage, le monde composant pourrait être une opportunité pour supporter simultanément les différents niveaux de sous-typage dans un même univers hétérogène [Med99], et ainsi gérer l'assemblage et la substitution de composants d'environnements distincts. Enfin, on pourrait étendre le sous-typage vers une relation de "matching" [ZW97] afin de déterminer si deux composants sont substituables sans avoir été déclarés comme sous-types (on n'exigerait plus une compatibilité totale entre deux composants, mais juste la compatibilité d'un de leurs aspects, comme le comportement).

Que faut-il retenir de ces travaux par rapport à notre démarche ? Tous ont en commun la notion de services fournis et requis. Cependant, la plupart d'entre eux manquent de réflexion sur l'universalité de leurs modèles et la gestion des contraintes non-fonctionnelles. Les travaux qui nous semblent les plus intéressants par rapport à notre démarche sont ceux sur les types contractuels d'Oscar Nierstrasz [Nie03], ceux sur le sous-typage hétérogène de Nenad Medvidovic [Med99], et ceux sur le "matching" de spécification d'Amy Zaemski et Jeannette Wing [ZW97]. En effet, le premier papier est l'un des seuls à modéliser clairement et formellement cet "autre chose" contenu dans le type, le deuxième introduit un sous-typage à la fois spécifique aux composants et commun aux différents modèles existants, et enfin le troisième est l'un des seuls à essayer de substituer des composants dans le cas où le sous-typage n'est pas déclaré.

5 Conclusion et perspectives

A travers cet article, notre intention était de montrer l'utilité d'un typage dédié aux composants logiciels, en particulier pour répondre aux problèmes de la composabilité et de la substituabilité dans un environnement hétérogène. Pour cela, nous avons proposé d'adjoindre à la hiérarchie "classique" de sous-typage une hiérarchie "contextuelle", dont le but est d'identifier des composants substituables provenant d'environnements différents. Nous avons également mis l'accent sur l'importance des propriétés non-fonctionnelles, dont la description est un objectif majeur de notre démarche.

Pour le moment, nous avons posé les bases de ce que devait être un véritable typage pour les composants logiciels. Notre objectif immédiat est donc d'approfondir ces bases, d'une part en explorant un à un les différents aspects non-fonctionnels d'un type de composant, d'autre part en investiguant chaque point d'interaction entre les deux hiérarchies de sous-typage.

Références

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *IEEE Computer*, 32 (7), 1999.

[CR05] Philippe Collet and Roger Rousseau. Confract : un système pour contractualiser des composants logiciels hiérarchiques. In *LMO 2005*, March 2005.

[FTS05] Régis Fleurquin, Chouki Tibermacine, and Salah Sadou. Le contrat d'évolution d'architectures : un outil pour le maintien de propriétés non fonctionnelles. In *LMO 2005*, March 2005.

[GAO94] David Garlan, Robert Allen, and John Ockerbloom. Exploiting style in architectural design environnments. In *The second ACM SIGSOFT FSE*, pages 170–185. ACM Press, December 1994.

[Gar94] David Garlan. Formalizing architectural connection. In *Proceedings of the 16th ICSE*, May 1994.

[LW94] Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. In *ACM Transactions on Programming Languages and Systems 1994*, 1994.

[Med99] Nenad Medvidovic. *Architecture-Based Specification-Time Software Evolution.* PhD thesis, University of California, Irvine, 1999.

[MO02] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings OOSPLA*, 2002.

[Nie03] Oscar Nierstrasz. Contractual types. Technical report, University of Bern, Institute of Computer Science and Applied Mathematics, August 2003.

[ODP04] Audrey Occello and Anne-Marie Dery-Pinna. Safe runtime adaptation of components : a uml meramodel with ocl constraints. In *International Workshop on Foundations of Unanticipated Software Evolution*, 2004.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT SEN*, 17 (4) :40–52, October 1992.

[Reu01] Ralf Reussner. The use of parametrised contracts for architecting systems with software components. In Wolfgang Weck Jan Bosch, Clemens Szyperski, editor, *International Workshop on Component-Oriented Programming*, 2001.

[SC00] João Costa Seco and Luis Caires. A basic model for typed components. In *Proceedings of the 14th ECOOP*, 2000.

[SH04] Jean-Guy Schneider and Jun Han. Components - the past, the present, and the future. In Wolfgang Weck Clemens Szyperski and Jan Bosch, editors, *International Workshop on Component-Oriented Programming*, 2004.

[Sha01] Mary Shaw. The coming-of-age of software architecture research. In *Proceedings of the 23rd ICSE*, 2001.

[SKM01] S. Sadou, G. Koscielny, and H. Mili. Abstracting services in a heterogeneous environment. In *IFIP/ACM International Conference on Distributed Systems Platforms, Middleware 2001*, Heidelberg, Germany, November 2001.

[Szy02] Clemens Szyperski. *Component Software : Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, second edition, 2002.

[ZW97] Amy Moorman Zaremski and Jeannette Wing. Specification matching of software components. In *ACM TOSEM 1997*, 1997.