Adaptation dynamique d’assemblages de dispositifs dirigée par des modèles

Daniel Cheung-Foo-Wo[*,] — Mireille Blay-Fornarino[*]**
Jean-Yves Tigli[*] — Stéphane Lavirotte[*,*] — Michel Riveill[*]**

[*] *Laboratoire I3S (CNRS - UNSA), 930 Route des Colles, 06 903 Sophia Antipolis*
{cheung,blay,tigli,lavirott,riveill}@polytech.unice.fr

[**] *CSTB, 290 Route des Lucioles, BP 209, 06 904 Sophia Antipolis*

[***] *IUFM, C. Freinet - Académie de Nice, 89 Av. George V, 06 046 Nice Cedex*

ABSTRACT. *Adaptable systems have been required since the emergence of pervasive computing. Adaptations are seen as direct consequences of the dynamic variations of devices surrounding the system. They consist in an integration cycle that is threefold : the discovery of new devices, the selection and the validation of adaptations. MDI allows to model the management of the variations of the devices availability. We use different representations of the same system at runtime according to different points of view. Transformations are applied at the metamodel level and maintain the coherence of the system. The metamodel and its tools are called “designer”. We detail two designers and discuss the accuracy of our approach.*

RÉSUMÉ. *L’informatique ambiante augmente la demande en systèmes adaptables à des situations non prévues à l’avance. L’adaptation est la conséquence de l’apparition et la disparition dynamique de périphériques environnants, notés “dispositifs” dans l’article. Elle constitue un cycle qu’on nomme “cycle d’intégration” dont les phases sont : la découverte de ces périphériques, le choix puis la validation des adaptations à effectuer. L’IDM offre des moyens rigoureux pour exprimer cette découverte dynamique. En effet, nous proposons de représenter suivant différents modèles un même système pendant son l’exécution. Chaque représentation est définie par un métamodèle au niveau duquel sont développés des outils de manipulation. Les transformations entre les métamodèles conservent la cohérence entre les modèles. Le but de cet article est finalement de montrer comment exprimer les cycles d’intégration en utilisant les modèles. Nous détaillons deux modèles ISL et ADL (Wcomp) et les transformations associées.*

KEYWORDS: *Dynamic Adaptation, Model, Component.*

MOTS-CLÉS : *Adaptation Dynamique, Modèle, Composant.*

1. Introduction

Nous nous intéressons à l'adaptation de systèmes informatiques à des utilisations non prévues à l'avance par programmation. On définit le fonctionnement d'un système comme étant la combinaison de plusieurs assemblages de composants qui définissent chacun un aspect précis d'une application. Il existe plusieurs approches pour adapter un logiciel en combinant des assemblages. Ces approches ont été répertoriées sommairement dans (McKinley *et al.,* 2004a) et puis exhaustivement dans (McKinley *et al.,* 2004b).

Adaptation par composition

Une des méthodologies d'adaptation consiste à adapter en ré-assemblant des composants logiciels. C'est ce que McKinley appelle la *composition* ou dans notre cas, re-*composition*. Mais la définition de *composition* a beaucoup dérivé depuis plusieurs années. *Composer* pour McKinley consiste à regrouper des fonctionnalités décrites en s'appuyant sur un vocabulaire propre à un ou plusieurs domaines techniques. Et c'est sur cette définition théorique que nous nous appuyons pour réaliser nos adaptations. Nous posons le vocabulaire propre à notre application en choisissant de construire un modèle pour chaque domaine technique et chaque vue de l'application. Nous décrivons un métamodèle en UML pour chaque domaine. Finalement, nous utilisons un formalisme logique pour décrire les transformations entre les modèles.

Adaptation en fonction du contexte opérationnel

L'ensemble des dispositifs (périphériques informatiques) à un instant donné constitue le *contexte opérationnel* de l'application. C'est le contexte dans lequel le système interagit avec son environnement. L'intégration dynamique de dispositifs consiste dans un premier temps à détecter leur apparition et leur disparition. Puis, il s'agit de sélectionner les adaptations à effectuer selon le contexte opérationnel courant pour finalement mettre en oeuvre ces adaptations.

Selon ces différentes étapes, certains formalismes tels que les ADLs (Architecture Description Languages) ou les *règles de réécriture* se révèlent plus adaptés. Les ADLs décrivent les architectures logicielles. Les règles de réécriture capturent et regroupent les adaptations sous forme d'ensembles de règles (schémas) et facilitent leur sélection.

L'adaptation d'assemblages de composants exige dans notre cas l'utilisation de plusieurs métamodèles correspondant chacun à un domaine technique. Nous manipulons les modèles en utilisant soit des représentations graphiques de leurs éléments de base (c'est-à-dire leur vocabulaire), soit leur en associant une syntaxe concrète à travers un langage de description. Lorsque nous développons à partir de ces représentations graphiques ou lorsque nous décrivons des fonctionnalités par l'intermédiaire

du langage dédié, nous utilisons des éditeurs que nous appelons "designers" (en français, "concepteurs") adaptés pour chaque domaine.

Cet article montre d'abord comment exprimer l'intégration d'un nouveau *dispositif* en utilisant les *modèles*, puis comment intégrer les transformations de modèles pour garder l'application cohérente lors de son exécution. Nous décrivons deux exemples de designers qui sont ISL4Wcomp et Wcomp.

Wcomp, le designer ADL, permet l'instanciation, la destruction de composants logiciels et la gestion des liaisons entre composants. Ce designer est plus proche de l'assemblage de composants et facilite la compréhension et la modification de ces assemblages en termes d'ajout et de retrait de composants et de liaisons. On y définit les notions d'instanciation (add_{ADL}), de destruction ($remove_{ADL}$) de composants, d'ajout (bind) et de retrait (unbind) de liaisons.

ISL4Wcomp, le designer ISL, permet d'éditer des schémas décrivant des *interactions* (Blay-Fornarino *et al.,* 2004) entre des composants logiciels. Ce designer permet une description déclarative et facilite par conséquent l'expression des adaptations. Il permet également leur validation a priori. Y sont définies les notions d'ajout (add_{ISL}) et de retrait ($remove_{ISL}$) de schémas.

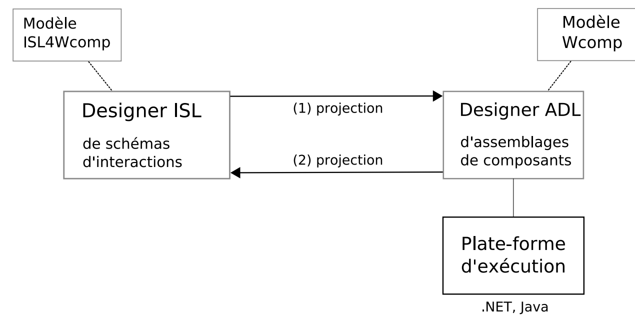


Figure 1. *Problématique provenant de nos applications*

Le but de ce travail est de gérer la cohérence entre différentes représentations d'un même système (cf. figure 1).

2. Vues différentes d'une application

Nous voyons dans cette section deux vues possibles d'une même application. Dans le domaine des systèmes multi-dispositifs, la vue la plus adaptée est celle décrite en ADL. D'une part, les systèmes *multi-dispositifs* sont des systèmes informatiques qui sont connectés à un très grand nombre de dispositifs. Ces systèmes n'interagissent avec leur environnement qu'à travers ces derniers. Notre expérience montre que le développement d'applications pour les systèmes multi-dispositifs se conçoit aisément en

s'appuyant sur les concepts des ADLs. D'autre part, le fait de représenter un logiciel comme un ensemble de *boîtes noires* (voir (Broy, 1996)) connaît une correspondance immédiate avec la structure des langages vers lesquels on projette tels que l'Assembleur, le C, le C++, le Java, le C# ou l'Objective-C. Dans ces langages, on peut faire correspondre la notion de boîte noire – ayant des entrées et des sorties – à une fonction (ou une procédure ou une méthode). La projection d'une application conçue à partir d'une description en ADL sur diverses plates-formes est par conséquent aisée.

2.1. *Wcomp*

Le premier designer que nous proposons s'appuie donc sur les ADLs. On l'appelle Wcomp. Ce designer permet de manipuler les éléments de première classe (principalement, les composants et les connecteurs) du paradigme des composants logiciels dans lequel une application multi-dispositifs est écrite.

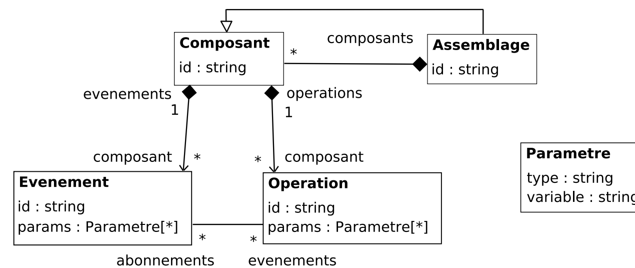


Figure 2. *Métamodèle Wcomp en UML (ADL)*

Dans la figure 2, nous avons établi le métamodèle de Wcomp. Une application est par définition un assemblage de composants. Cet assemblage peut être vu comme étant un nouveau composant. A l'image d'un composant matériel (par exemple, les circuits intégrés (McIlroy, 1968)), un composant logiciel est doté de ports d'entrée et de sortie qui sont respectivement implémentés par des évènements et des opérations. Les ports de sortie (les évènements) peuvent éventuellement être associés à certains ports d'entrée (les opérations).

2.2. *ISL4Wcomp*

Le designer ISL (ISL4Wcomp) s'appuie sur le langage de spécification d'interactions nommé ISL pour *Interaction Specification Language* (Berger, 2001, Blay-Fornarino *et al.,* 2004). ISL est un langage de description de schémas d'interactions entre des objets. Ce langage a la propriété d'être "composable" automatiquement. Cela signifie que l'on peut écrire plusieurs schémas indépendamment en ISL. Puis une fois ajoutés, ces schémas se composent pour ne donner plus qu'un seul schéma rassemblant *correctement* les fonctionnalités décrites séparément.

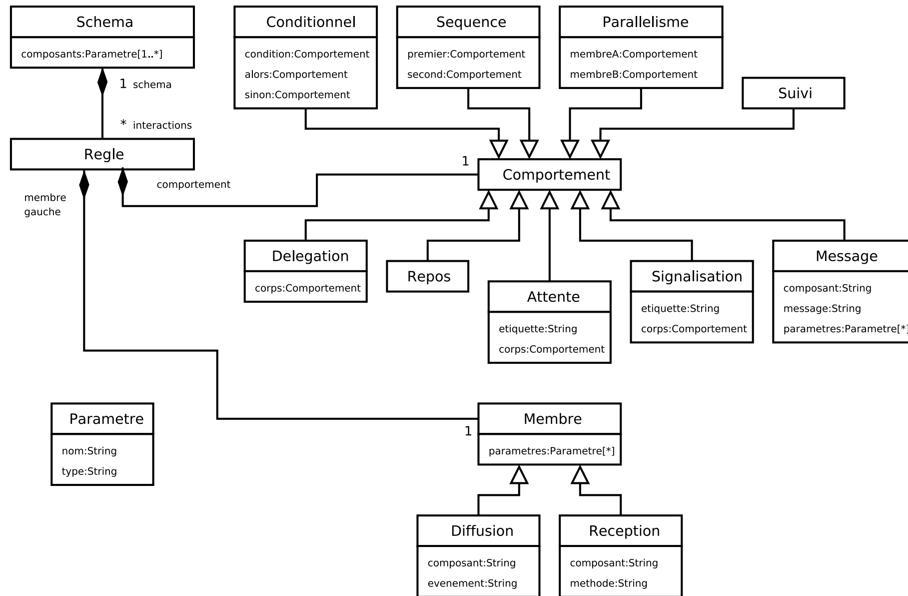


Figure 3. *Métamodèle ISL4Wcomp en UML (ISL)*

Afin d'utiliser les propriétés de ce langage dans nos expérimentations et dans notre formalisme, nous avons adapté ses opérateurs pour qu'ils prennent en charge la notion d'événement (Cheung-Foo-Wo *et al.,* 2005). Un schéma écrit en ISL4Wcomp est constitué d'un ensemble de règles de réécriture agissant sur des appels d'opérations ou de déclenchement d'événements. La grammaire du langage ISL pour le logiciel ISL4Wcomp et l'implémentation de la fonction de composition de schémas s'appuient sur les concepts de SRP (Système de Résolution de Problèmes).

Prolog étant actuellement le langage le plus connu pour un SRP, c'est dans ce langage que nous avons implémenté les règles logiques de la composition. L'implémentation consiste en 25 règles de composition. Ceci nous a permis en outre de vérifier certaines propriétés essentielles comme la commutativité de la composition des schémas. Au niveau d'ISL4Wcomp, nous ne considérons que quelques opérateurs. Ces opérateurs regroupent les notions d'appel d'opérations, de déclenchement d'événements, de séquence entre deux opérations, de concurrence, d'attente, de notification (ou signalisation) et de condition. Donc, nous avons neuf opérateurs :

$\alpha;\beta$	la séquence, α s'exécute après β
$\alpha\|\beta$	la concurrence, α et β s'exécutent en parallèle
if$(\gamma)\{\alpha\}$else$\{\beta\}$	la condition, α s'exécute si γ est vraie, sinon β s'exécute
$c.m()$	l'envoi d'un message, appel de la méthode m sur le composant c
$call$	se réfère à l'opération ou l'événement du membre gauche.

nop	No OPeration, n'effectue rien.
delegate $\{\alpha\}$	la délégation ; l'appel de méthode ou l'envoi d'événement spécifié par le membre gauche de la règle est remplacé par α
wait(x, α)	l'attente ; l'exécution de α ne se fera qu'une fois que la variable x sera signalée (par l'opérateur de signalisation)
signal(x, α)	la signalisation ; signale la variable x lorsque α a achevé son exécution

On note f_n la fonction de composition. Notons $s_1, \ldots, s_n$ des schémas d'interactions, pour tout n, $f_n(s_1, \ldots, s_n)$ est invariante par permutation de ces n schémas. Cela signifie en pratique que l'état dans lequel se trouve le système à un instant donné est indépendant de l'ordre dans lequel nous avons ajouté les schémas d'interactions.

2.3. *Exemple d'utilisation*

Nous proposons l'exemple d'une application de commande à distance. Nous considérons sept composants logiciels qui sont regroupés dans la liste ci-dessous. Nous y trouvons le brochage du composant logiciel indiquant la signature de ses opérations (à gauche) et de ses événements (à droite) s'il en est doté, ainsi qu'une description intuitive de son comportement.

Voici un descriptif de la fonction de chaque composant :

BadgeRF

void click(string id)

Le composant *BadgeRF* déclenche un événement *click* avec un identifiant unique *id* lorsque l'utilisateur appuie sur le bouton. Un BadgeRF (dispositif) est constitué d'un bouton et d'un émetteur.

Bascule

void basculer(...) | void etat1(...) | void etat2(...)

Le composant *Bascule* déclenche alternativement les événements *etat1* et *etat2* après l'appel de l'opération *basculer*. Cet opération accepte tout type de paramètres qu'il retransmet aux événements en sortie.

PC

void login(string id)
void logout(string id)

Le composant *PC* permet de se connecter à un PC de bureau en ouvrant son compte personnel. L'identification est effectuée par l'identifiant *id*.

Antivol

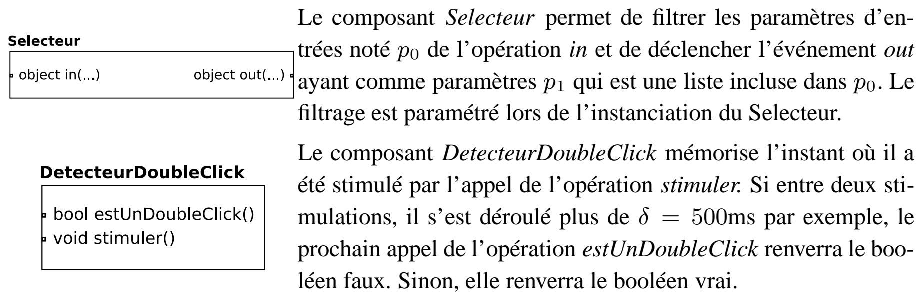
bool autorisé(string id)

Le composant *Antivol* (authentification de l'utilisateur) vérifie que l'identifiant unique *id* en paramètre de l'opération *autorisé* est valide. Si tel est le cas, alors l'opération renvoie le booléen vrai, sinon elle renvoie faux.

Television

void on()
void off()

Le composant *Television* permet de contrôler l'allumage et l'extinction d'une télé en appelant respectivement l'opération *on* ou *off*.



Le composant *Selecteur* permet de filtrer les paramètres d'entrées noté p_0 de l'opération *in* et de déclencher l'événement *out* ayant comme paramètres p_1 qui est une liste incluse dans p_0. Le filtrage est paramétré lors de l'instanciation du Selecteur.

Le composant *DetecteurDoubleClick* mémorise l'instant où il a été stimulé par l'appel de l'opération *stimuler.* Si entre deux stimulations, il s'est déroulé plus de $\delta = 500$ms par exemple, le prochain appel de l'opération *estUnDoubleClick* renverra le booléen faux. Sinon, elle renverra le booléen vrai.

L'application sur laquelle nous nous appuyons est schématisée dans la figure 4. Elle s'exécute sur un PC de bureau. Le composant BadgeRF est un capteur accessible via un réseau sans-fil et les autres composants sont liés localement au PC.

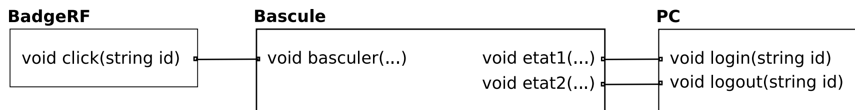


Figure 4. *Assemblage représentant l'application "télécommande".*

Scénario :

Cette application consiste à se connecter sur une machine de type PC à l'aide d'un badge RF (Radio Frequency). Le scénario est tel que lorsque l'on clique sur le bouton du badge RF, le composant BadgeRF déclenche l'événement *click* avec comme paramètre, l'identifiant de l'utilisateur. L'opération *basculer* est abonnée à ce dernier événement (voir la figure 4). Ainsi, nous avons : soit l'évènement *etat1* est déclenché, ce qui a pour effet d'appeler l'opération login sur le PC avec l'identifiant ; soit l'évènement *etat2* est déclenché, ce qui a pour effet d'appeler plutôt l'opération *logout* sur le PC. Dans le premier cas, la session appartenant à l'utilisateur est ouverte et dans le second, elle est refermée.

Adaptation :

Concernant l'adaptation de cette application, nous avons deux possibilités : soit nous travaillons dans le modèle Wcomp et instancions des composants ou introduisons des liaisons, soit nous basculons vers le modèle ISL pour décrire et ajouter des schémas d'interactions.

Nous proposons l'écriture de deux exemples de schémas d'interactions. Dans un premier schéma, nous voulons intégrer un composant supplémentaire dans l'application qui est "l'*antivol*" (ou l'authentification de l'utilisateur). Dans un second schéma, nous ajoutons un dispositif supplémentaire qui est une télévision accompagnée d'un composant logiciel sélecteur de paramètres et d'un autre qui détecte les doubles clicks d'un bouton.

Le premier schéma d'interactions est décrit dans le tableau 1.

n°	Nom du schéma	Description
1	schéma_antivol	**Au lieu de** se connecter au PC, **si** l'antivol l'autorise, **alors** "**faire ce qui est prévu**" (c'est-à-dire, se connecter au PC), **sinon** "**ne rien faire**".

Table 1. *Schéma de l'antivol*

Puisque certains composants logiciels sont associés à des dispositifs, nous pouvons utiliser ISL4Wcomp pour les intégrer simplement en ajoutant des schémas d'interactions. Le second schéma que nous mettons en oeuvre exprime le comportement décrit dans le tableau 2.

n°	Nom du schéma	Description
2	schéma_tv	**Au lieu de** se connecter au PC, **si** le détecteur de double clicks en a détecté un, **alors** "**faire ce qui est prévu**" (c'est-à-dire se connecter au PC), **sinon** allumer la télévision.

Table 2. *Schéma d'intégration de la télévision*

Exemple de schémas

Nous reprenons la première description intuitive du schéma de "l'antivol de session" (voir tableau 1). Sa traduction en ISL, une fois appliquée aux composants *pc* et *antivol,* est la suivante :

```
schema schéma_antivol (pc,antivol) {
  pc.login(int id) {
    if(antivol.autorise(int id)) {call} else {delegate{nop}}
  }
}
```

Figure 5. *Schéma n°1 : l'antivol (ou authentification de l'utilisateur).*

Nous reprenons également la seconde description du schéma "d'intégration de la télévision" (voir tableau 2). Sa traduction en ISL4Wcomp une fois appliquée aux composants *pc*, *télévision* et *détecteur* est résumée par la figure 6.

Transformation T_{ISL} : composition de schémas

Le composition de schémas d'interactions se fait en plusieurs phases. On part de l'hypothèse que l'on compose des schémas quelconques.

```
schema schéma_tv (pc,television,detecteur) {
  pc.login(int id) {
    if(detecteur.unSeulClick()) {call} else {delegate{television.on()}}
  }
}
```

Figure 6. *Schéma n°2 : l'intégration de la télévision.*

1) Nous devons unifier les paramètres. Soient $p_1, \ldots, p_n$ les paramètres de ces n schémas. Pour créer un unique schéma ayant des paramètres p, on unifie tous les paramètres c'est-à-dire qu'on crée une liste de paramètres qui est l'union de tous les p_i.

2) Puis, on sépare les règles des schémas afin d'avoir les règles comme des entités de première classe et ne travailler que sur celles-ci. On se retrouve donc avec un ensemble de règles et une liste de paramètres unifiée p.

3) On groupe les règles ayant le même membre gauche (qu'il soit une réception ou une diffusion). Lorsqu'un groupe est constitué d'une seule règle, alors cette règle figure sans modification dans le schéma résultant. Lorsqu'un groupe est constitué de plusieurs règles, alors les corps de ces règles doivent fusionner. Le résultat de cette fusion est un nouveau comportement. L'ensemble est constitué du membre gauche et de ce nouveau comportement figure dans le schéma résultant.

4) La fusion est une fonction n-aire qui prend n comportements comme paramètres d'entrée. Elle est définie à partir d'une fonction de fusion binaire commutative. Par extension, nous pourrions considérer que la fonction de fusion est une transformation du même modèle. Toutefois, nous avons pour cet article développé les transformations entre les modèles Wcomp et ISL4Wcomp.

```
schema schéma_courant (pc,television,detecteur,antivol) {
 pc.login(int id) {
  if(antivol.autorise(id)) {
   if(compteur.unSeulClick()) { call }
   else { delegate { television.on() } }
  } else {
   if(compteur.unSeulClick()) {delegate {nop}}
   else {delegate {television.on()}
  }
 }
}
```

Figure 7. *Schéma résultant de la composition des schémas n°1 et n°2.*

Le résultat de la composition des schémas d'interactions a la signification suivante : lorsque l'opération *pc.login* est appelée, au lieu d'appeler directement l'opération sur le composant pc, on vérifie conditions suivantes. Si l'antivol nous l'autorise et qu'on a effectué qu'un click, alors on appelle effectivement *pc.login.* Si en revanche,

on a effectué deux clicks, alors on allume la télévision. Si l'antivol ne donne pas son approbation et qu'on a effectué qu'un click, alors rien ne se passe. Si en revanche, on a effectué deux clicks, alors on allume la télévision. Quelque soit la réponse de l'antivol, l'allumage de la télévision reste indépendant.

3. Cycle d'intégration d'un nouveau dispositif

Dans le paradigme orienté-composant, un composant est doté de ports. Une *liaison* représente l'abonnement d'un évènement qu'un composant peut diffuser (port de sortie) à un appel d'une *opération* (port d'entrée). Un évènement s'accompagne souvent de données sous forme de paramètres. Un *schéma d'interactions* ajoute des *composants de contrôle* entre les ports d'entrée et de sortie. Ces composants de contrôle se connectent soit avant la *réception* de certains messages, soit après la *diffusion* d'évènements. Ces schémas sont exprimés sous forme de règles de réécriture en ISL.

Dans un assemblage, certains composants représentent des dispositifs tels que des capteurs et des actionneurs. L'adaptation d'un assemblage se fait de façon automatique. Elle s'appuie sur une boucle de découverte de dispositifs. La découverte d'un nouveau dispositif ou sa disparition (lors d'une panne, une déconnexion) implique la modification de l'application. Cette modification se passe pendant son exécution. Elle consiste en deux étapes :

1) en instanciant ou détruisant des composants logiciels

2) en sélectionnant et en ajoutant des schémas d'interactions

Scruter l'ensemble des dispositifs et réagir à des modifications éventuelles de cet ensemble est un mécanisme itératif qu'on appelle *cycle d'intégration d'un dispositif*. D'autres adaptations sont possibles en réaction à la découverte d'un dispositif. Citons par exemple les adaptations manuelles, la gestion de stratégies (Buisson *et al.,* 2005) et les adaptations par aspect (Barais *et al.,* 2006).

Designers

Au niveau d'ISL4Wcomp, l'ajout d'un schéma d'interactions a pour conséquence la construction d'un assemblage de composants. Comment se déroule cette construction ? Un algorithme permet de valider cet assemblage et de construire le graphe des interactions en conséquence. Si celui-ci est cohérent, il est projeté vers le Wcomp (étape 1, figure 1). Il a pour conséquence l'ajout de liaisons et la création de *composants de contrôle*. Un composant de contrôle est un composant qui a une sémantique spécifique et connu dans le graphe de composants. Si le graphe des interactions est incohérent, aucune projection n'est entreprise.

Au niveau du designer Wcomp, l'adaptation d'un assemblage de composants consiste à instancier ou détruire des composants logiciels afin d'intégrer des dispositifs. Elle consiste également à introduire ou rompre des liaisons entre des composants.

Cohérence

Ajout d'une liaison. Afin de maintenir la cohérence entre les deux designers, chaque introduction de liaison est transformée en un schéma d'interactions. Ce schéma est ensuite ajouté dans ISL4Wcomp (étape 2.1, figure 1).

Retrait d'une liaison. La rupture d'une liaison implique le retrait du schéma d'interactions correspondant dans le designer ISL4Wcomp.

Si la transformation due à l'introduction de liaisons génère une incohérence, alors les deux designers deviennent temporairement incohérents. Pour y remédier, l'intégralité de l'assemblage de composants dans Wcomp est d'abord transformée en un schéma unique dans ISL4Wcomp (étape 2.2, figure 1). Ce schéma sera le seul qui est présent à cet instant dans ISL4Wcomp. Cela signifie qu'on perd l'historique des schémas d'interactions précédemment ajoutés.

Nous avons décrit la manière dont ces deux designers collaborent selon deux points de vue d'une même application :

– les schémas et leur composition

– les composants et leurs liaisons

Nous voyons en détail dans la section suivante les deux transformations de modèles. Chacun a comme but de s'adresser à un domaine technique différent : la description d'architectures logicielles et la séparation de préoccupations.

4. Transformations de modèles

Nous analysons les modifications possibles dans chaque modèle. Nous allons voir deux transformations entre Wcomp et ISL4Wcomp. Nous distinguons les transformations totales d'un modèle à un autre et les transformations partielles.

Transformations totales. Ces transformations peuvent être des *transformations totales* d'un modèle à un autre. La totalité du modèle est transformée dans un autre modèle. Nous les appelons des *projections*. On projette un modèle sur un autre modèle.

Transformations partielles. Ces transformations peuvent être des *transformations partielles*, c'est-à-dire qu'elles ne concernent qu'un sous-ensemble des modifications. Nous les appelons des *perturbations*. Une partie du modèle est modifiée. Uniquement cette partie *perturbée* est transformée dans un nouveau modèle.

Nous proposons d'abord l'étude des modifications élémentaires dans le modèle Wcomp. Il s'agit de l'ajout, du retrait de liaisons. On identifie alors les opérations élémentaires suivantes : add_{ADL}(id, type), $\text{remove}_{\text{ADL}}$(id), bind(id, id_{source}, id de l'évènement,id_{cible}, id de l'opération), unbind(id). Dans ISL4Wcomp, les modifications élémentaires se résument aux actions suivantes : on peut poser ou retirer

un schéma d'interactions. On identifie alors les opérations élémentaires suivantes : add$_{\text{ISL}}$(id du schéma, schéma), remove$_{\text{ISL}}$(id du schéma).

4.1. *Transformation* $T_{ADL \to ISL}$

C'est l'étape 2 de la figure 1 de la page 77. On suppose que le modèle ISL se trouve dans un état noté E_0^{ISL}. Wcomp effectue une modification notée δ_w. Quelle est la modification à apporter au modèle ISL pour intégrer cette modification δ_w et se trouver dans un état noté E_1^{ISL} ?

$$E_0^{\text{ISL}} + T_{\text{ADL}\to\text{ISL}}(\delta_w) = E_1^{\text{ISL}}$$

La transformation que nous étudions se note $T_{\text{ADL}\to\text{ISL}}$. Cette transformation peut être séparée en deux sous-transformations, chacune travaillant sur un sous-ensemble du domaine des modifications de Wcomp. On les note :

$$T^l_{\text{ADL}\to\text{ISL}}(w) \text{ et } T^r_{\text{ADL}\to\text{ISL}}(w).$$

L'exposant l signifie que cette sous-transformation va prendre en compte uniquement les *liaisons* introduites entre deux ports. L'exposant r signifie que cette sous-transformation va prendre en compte les *ruptures* de liaisons du modèle.

Introduction d'une liaison : $T^l_{ADL \to ISL}(w)$

Le premier cas consiste à lier une opération o à un événement e. On note respectivement c_1 et c_2 les composants associées à ces deux pattes. On a δ_w= bind (id, c_1, e,c_2, o). La règle de transformation logique $T^l_{\text{WI}}(\delta_w)$ se note (similairement au formalisme Typol (Despeyroux, 1988)) :

$$\frac{}{\vdash bind(id, c_1, e, c_2, o) : add_{ISL}(id, s)}\,(T^l_{\text{WI}})$$ où s est le schéma suivant :

```
schema s (c1,c2) {
  c1.e(params) {
    call || c2.o(params)
  }
}
```

Rupture de liaison : $T^r_{ADL \to ISL}(w)$

Le second cas consiste à rompre la liaison entre une opération o à un événement e. On note respectivement c_1 et c_2 les composants associées à ces deux pattes. On a δ_w= unbind (id). La règle de transformation logique $T^r_{\text{WI}}(\delta_w)$ se note :

$$\frac{}{\vdash unbind(id) : remove_{ISL}(id)}\,(T^r_{\text{WI}})$$ où id est l'identifiant d'un schéma existant.

Cas d'échec

Si le schéma précédent n'existe pas (c'est-à-dire, si $T^r_{\text{ADL}\rightarrow\text{ISL}}$ n'est pas valide), alors l'assemblage de composants dans Wcomp est transformé dans son intégralité en un schéma d'interactions. Nous expliquons l'algorithme de transformation (étape 2, figure 1, page 77) en se basant sur les métamodèles.

Algorithme de tranformation

On classe d'abord les composants Wcomp en deux catégories : ceux qui sont spécifiques à ISL4Wcomp (comme par exemple, le composant *Selecteur*) et ceux qui ne le sont pas comme les composants représentant les dispositifs et les autres composants logiciels. On note A l'ensemble des composants spécifiques. On distingue les liaisons dont les extrémités restent dans l'ensemble A des autres liaisons (voir figure 8). On les groupe dans un ensemble noté A_l. L'ensemble des composants de A liés par les liaisons de A_l constitue une forêt dont les noeuds sont les composants. Chaque arbre constitue une règle en ISL4Wcomp.

On considère deux cas : le cas où il existe une unique liaison entre un composant "autre" et un composant spécifique et le cas où il en existe plusieurs.

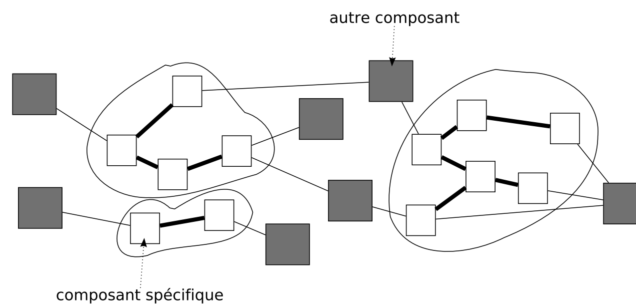


Figure 8. *Forêt de composants spécifiques.*

Dans le premier cas, on note l l'unique liaison entre un composant noté "*autre*" (c'est-à-dire non spécifique) et un composant spécifique. Le membre gauche de la règle est l'évènement associé à la liaison (voir un exemple de graphe de composants et de tri dans la figure 8). L'arbre de composants spécifiques est un "arbre ISL" selon (Berger, 2001). On dispose de toutes les informations nécessaires pour construire la règle.

Dans le second cas, on note $l_1, \ldots, l_n$ ces liaisons. On oriente le graphe dans le sens de l'évènement vers l'opération. Chaque arbre partant de l'événement associé à l_i représente alors le comportement d'une règle. Nous avons, dans ce cas, n règles. Toutes ces règles sont ensuite regroupées dans un unique schéma d'interactions.

L'apparition de cycles dans un arbre génère une erreur de transformation car cela n'a pas de sens en ISL4Wcomp.

4.2. *Transformation* $T_{ISL \rightarrow ADL}$

Dans le designer ISL4Wcomp, un schéma se compose de plusieurs règles. Chaque règle contient un unique corps qui explicite la réécriture du membre gauche. Ce corps est constitué d'un ensemble d'opérateurs imbriqués. Le corps d'une règle peut alors être représenté par un arbre dont les noeuds sont les opérateurs (voir l'exemple de la figure 9 et 10).

Réécriture d'un appel de méthode

Nous avons deux cas. Si le membre gauche (indiquant un composant c, une opération o) est une réception, alors le composant c se retrouve après le $call$ dans l'arbre ISL (voir figure 9). Ce sont en effet les interactions qui portent sur son opération o qui sont redéfinies. On note $M_g[c, m]$ le membre gauche d'une règle d'interactions. La variable c représente le composant sur lequel agit la réception (c'est-à-dire l'appel de méthode) ou la diffusion d'évènements ayant comme identifiant m. Le symbole $P[c, m]$ représente l'évènement du noeud (c'est-à-dire le composant) précédent. Soit la règle "call" :

$$\frac{\vdash M_g \text{ est une réception}}{M_g[c_0, m_0], P[c, m] \vdash call : bind(id, c, m, c_0, m_0)} \text{ (call) où } id \text{ est un identifiant unique.}$$

La règle précédente se lit de la manière suivante : si le membre gauche est une réception, alors on évalue $call$ et le résultat de cette évaluation consiste à effectuer la modification $bind(id, c, m, c_0, m_0)$ qui signifie que l'opération m_0 du composant c_0 pointés par le membre gauche est reliée au dernier évènement caractérisé par le composant c et la méthode m. L'identifiant id est généré à partir des quatre paramètres c, m, c_0 et m_0. Pour ne pas avoir de conflits, les liaisons ayant déjà cet identifiant sont renommées.

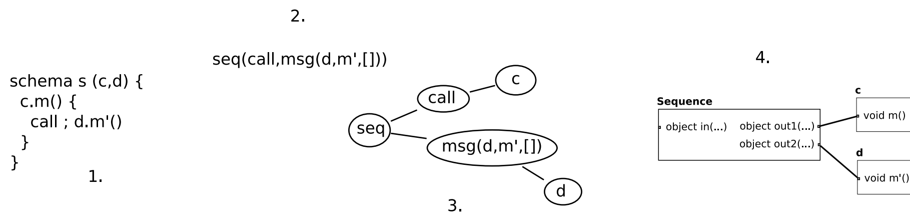


Figure 9. *Transformation d'une règle de réception.*

(La gestion des paramètres consiste en quelques règles logiques supplémentaires qui n'apparaîtrons pas dans cet article).

Pour ce qui est du problème du composant qui précède le premier noeud de l'arbre, sa valeur dépend du type du membre gauche. Si c'est une diffusion alors on établit une

liaison entre le composant indiqué par le membre gauche et la racine de l'interaction. Si c'est une réception, alors c'est une redirection qu'on réalise dans l'assemblage de composants. Il s'agit de propager la réécriture aux interactions des schémas résultants. Ainsi deux phases se révèlent alors nécessaires. D'abord, on doit travailler sur les diffusions et on crée un schéma partiel intermédiaire. Puis, on travaille sur les réceptions pour ajouter des composants de contrôle supplémentaires et compléter le schéma intermédiaire.

Diffusion d'un évènement

$$\frac{\vdash M_g \text{ est une diffusion}}{M_g \vdash call : \varnothing} \text{ (call)}$$

Si le membre gauche (indiquant un composant c) est une diffusion, alors le composant désigné c se retrouve à la racine de l'arbre ISL (voir figure 10). C'est l'effet de la diffusion qui est alors redéfinie.

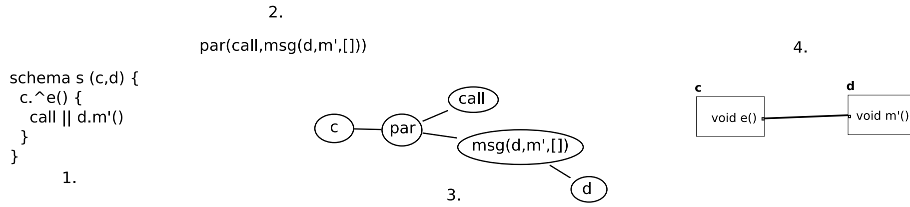


Figure 10. *Transformation d'une règle de diffusion.*

Projection dans Wcomp

La dernière phase consiste à faire correspondre l'arbre intermédiaire ISL à un assemblage de composants dans Wcomp.

Voici un exemple de règle de transformation concernant l'opérateur de concurrence. L'évaluation de l'opérateur de concurrence dans le contexte d'un membre gauche M_g et d'un parent P consiste à évaluer récursivement les deux branches α et β dans ce même contexte M_g,P.

$$\frac{\begin{array}{c} M_g, P \vdash \alpha : r_\alpha \\ M_g, P \vdash \beta : r_\beta \end{array}}{M_g, P \vdash par(\alpha, \beta) : \left\{ \begin{array}{l} r_\alpha \\ r_\beta \end{array} \right.} \text{ (concurrence)}$$

5. Discussion

Nous constatons que la multiplicité des métamodèles permet d'aborder l'interopérabilité entre plates-formes, mais suppose un enrichissement des plates-formes

au niveau "méta". En effet, la partie projection vers l'implémentation de Wcomp en .NET s'appuie actuellement sur une API de mise à jour des assemblages. De même, dans le cadre de notre application, nous avons besoin des remontées d'informations provenant des cibles (modification d'assemblage, ajout de schéma, découverte d'un nouveau dispositif) pour prendre en charge les modifications au niveau des modèles (Blay-Fornarino *et al.,* 2005). Ainsi l'interconnexion entre les modèles et leur mise en oeuvre est gérée par des mécanismes de réflexions sur les plates-formes cibles.

L'interopérabilité n'intervient pas de manière similaire en fonction des dépendances entre les modèles. En effet, la découverte de nouveaux dispositifs est basée sur un métamodèle non décrit dans cet article qui cible les mécanismes de découverte dynamique des infrastructures des réseaux et des middlewares sous-jacents. Celui-ci permet d'expliciter le procédé de mise à jour des adaptations indépendamment des plates-formes cibles. Néanmoins, la mise en oeuvre de Wcomp repose sur des *représentations* de ce métamodèle ; nous nous trouvons alors dans le cadre d'un empilement de modèles (Favre, 2004, Marvie *et al.,* 2006). En conséquence, celles-ci jouent le rôle de la plate-forme d'exécution pour les mises en oeuvre de Wcomp et les notifications de modifications qui sont aujourd'hui gérées directement entre ces implémentations sans passer par les modèles.

Dans une première approche, nous avons fait le choix de définir un métamodèle pivot à partir duquel serait gérée la cohérence de l'ensemble des designers via des transformations. Or, cette solution nous semble aujourd'hui peu efficace. En effet, nous avons vu qu'ISL4Wcomp gère la cohérence de certaines modifications ce qui implique qu'une modification qui est valide dans un métamodèle ne l'est plus dans un autre. Le modèle Wcomp est lui particulièrement bien adapté à définir des transformations efficaces vers des plates-formes d'implémentation. Par contre, l'analyse des graphes de propagation par transformation des graphes d'interactions est plus facile en partant du modèle ISL4Wcomp, tandis que des reconnaissances de *patterns* d'architectures tels que ceux présentés dans TranSat (Barais *et al.,* 2006) s'appliquera mieux à la représentation Wcomp. En conséquence dans notre cas d'étude, la composition des correspondances ou *mappings* (Pottinger *et al.,* 2003) pour simplifier la gestion de cohérence entre les différents designers nous semble mieux adaptée qu'une approche par métamodèle pivot. Une étude plus poussée devrait nous permettre de comparer ISL aux métamodèles définis dans le domaine des langages d'aspects (Ubayashi *et al.,* 2005, Han *et al.,* 2005).

6. Conclusion

L'ingénierie des modèles propose l'exploitation des modèles non seulement pour la production de code, mais également pour la génération de tests (Dubois *et al.,* 2005, Jézéquel *et al.,* 2006), la validation a priori des codes et des adaptations (Barais *et al.,* 2006) et la constitution de référentiels. Notre travail s'inscrit clairement dans cette démarche et aborde plus particulièrement la gestion de l'interopérabilité entre plates-formes via des métamodèles ainsi que des expérimentations de l'informatique

ambiante (Muñoz *et al.,* 2004). En l'occurrence, nos cibles sont aujourd'hui, pour la partie exécution des dispositifs, les middlewares usuels tels que .NET, Javaxxx. Pour la partie reconnaissance des adaptations à mettre en place, nous préférons actuellement une approche basée sur la logique des prédicats avec une mise en oeuvre en Prolog car les règles de projection s'y exprime de façon immédiate. Les transformations entre modèles sont définies au niveau des métamodèles et sont tout à fait indépendantes des cibles. Exprimées en Prolog, elles s'appuient sur des représentations exprimées sous la forme de faits : nous nous situons donc dans le cadre d'un même méta-métamodèle. D'autres solutions auraient probablement pu s'appliquer dont l'utilisation du MOF et les langages de transformation Kermeta (Fleurey *et al.,* 2006) ou ATL (Bézivin *et al.,* 2003).

7. References

Barais O., Lawall J., Meur A.-F. L., Duchien L., « Safe Integration of New Concerns in a Software Architecture », *13th Annual IEEE International Conference on Engineering of Computer Based Systems ECBS'06*, Potsdam, Germany, March, 2006.

Berger L., Mise en Oeuvre des Interactions en Environnements Distribués, Compilés et Fortement Typés : le Modèle MICADO, Thèse de doctorat, Université de Nice-Sophia Antipolis - Faculté des sciences et techniques, Ecole doctorale STIC - Informatique, Octobre, 2001.

Blay-Fornarino M., Charfi A., Emsellem D., Pinna-Dery A.-M., Riveill M., « Software interactions », *Journal Of Object Technology*, vol. 3, n° 10, p. 161-180, 2004.

Blay-Fornarino M., Franchi P., Nano O., « Vers une sémantique « plug-in » pour les modèles », 2005.

Broy M., « Towards a Mathematical Concept of a Component and Its Use », Components' Users Conference CUC'96, 1996.

Buisson J., André F., Pazat J.-L., « A framework for dynamic adaptation of parallel components », *ParCo 2005*, 2005. To appear.

Bézivin J., Dupé G., Jouault F., Pitette G., Rougui J., « First Experiments with the ATL Transformation Language : transforming XSLT into Xquery », *OOPSLA Workshop*, 2003.

Cheung-Foo-Wo D., Blay-Fornarino M., Tigli J.-Y., Dery A.-M., Emsellem D., Riveill M., « Langage d'aspect pour la composition dynamique de composants embarqués », *2ème Journée Francophone sur le Développement de Logiciels Par Aspects*, 2005.

Despeyroux T., TYPOL : a formalism to implement natural semantics, Technical report, INRIA - Sophia Antipolis, 1988.

Dubois H., Gérard S., Mraidha C., « Un langage d'action pour le développement UML de systèmes embarqués temps-réel », *IDM05, Actes des 1ères Journées sur l'Ingénierie Dirigée par les Modèles*, p. 175-192, 2005.

Favre J. M., « Towards a Basic Theory to Model Driven Engineering », *3rd Workshop in Software Model Engineering*, 2004.

Fleurey F., Drey Z., Didier V., *Kermeta language, Reference Manual*, IRISA. February, 2006.

Han Y., Kniesel G., Cremers A. B., « Towards Visual AspectJ by a Meta Model and Modeling Notation », *Proceedings of 6th International Workshop on Aspect-Oriented Modeling*, Held in conjunction with AOSD'05, Chicago, Illinois, USA, March, 2005.

Jézéquel J.-M., Gérard S., Mraidha C., Baudry B., *Le génie logiciel et l'IDM : une approche unificatrice par les modèles*, chapter 1, p. 1, 2006.

Marvie R., Duchien L., Blay-Fornarino M., *Les plates-formes d'exécution et l'IDM*, Edition Hermes, chapter 4, 2006.

McIlroy M. D., « Mass produced software components », *in* , B. R. E. P. Naur (ed.), *Software Engineering : Report on a Conference Sponsored by the NATO Science Committee*, 1968.

McKinley P. K., Sadjadi S. M., Kasten E. P., Cheng B. H. C., « Composing Adaptive Software », *IEEE Computer*, vol. 37, n° 7, p. 56-64, 2004a.

McKinley P. K., Sadjadi S. M., Kasten E. P., Cheng B. H. C., A Taxonomy of Compositional Adaptation, Technical Report n° MSU-CSE-04-17, Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, May, 2004b.

Muñoz J., Pelechano V., Fons J., « Model Driven Development of Pervasive Systems », *International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES)*, vol. 1, p. 3-14, june, 2004.

Pottinger R., Bernstein P. A., « Merging Models Based on Given Correspondences », *29th International Conference on the Very Large Data Bases*, 2003.

Ubayashi N., Moriyama G., Masuhara H., Tamai T., « A parameterized interpreter for modeling different AOP mechanisms », *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ACM Press, New York, NY, USA, p. 194-203, 2005.

ANNEXE POUR LE SERVICE FABRICATION
A FOURNIR PAR LES AUTEURS AVEC UN EXEMPLAIRE PAPIER
DE LEUR ARTICLE ET LE COPYRIGHT SIGNE PAR COURRIER
LE FICHIER PDF CORRESPONDANT SERA ENVOYE PAR E-MAIL

1. ARTICLE POUR LES ACTES :
   IDM'06, Lille – Secondes Journées sur l'ingénierie des modèles

2. AUTEURS :
   Daniel Cheung-Foo-Wo[*,**] — *Mireille Blay-Fornarino*[*]
   Jean-Yves Tigli[*] — *Stéphane Lavirotte*[*,***] — *Michel Riveill*[*]

3. TITRE DE L'ARTICLE :
   Adaptation dynamique d'assemblages de dispositifs dirigée par des modèles

4. TITRE ABRÉGÉ POUR LE HAUT DE PAGE MOINS DE 40 SIGNES :
   Adaptation dynamique d'assemblages

5. DATE DE CETTE VERSION :
   8 juin 2006

6. COORDONNÉES DES AUTEURS :

   – adresse postale :

   [*] Laboratoire I3S (CNRS - UNSA), 930 Route des Colles, 06 903 Sophia Antipolis

   {cheung,blay,tigli,lavirott,riveill}@polytech.unice.fr

   [**] CSTB, 290 Route des Lucioles, BP 209, 06 904 Sophia Antipolis

   [***] IUFM, C. Freinet - Académie de Nice, 89 Av. George V, 06 046 Nice Cedex

   – téléphone : 04 92 96 51 82

   – télécopie : 04 92 96 50 55

   – e-mail : cheung@polytech.unice.fr

7. LOGICIEL UTILISÉ POUR LA PRÉPARATION DE CET ARTICLE :
   LaTeX, avec le fichier de style `article-hermes.cls`,
   version 1.2 du 03/03/2005.

8. FORMULAIRE DE COPYRIGHT :
   Retourner le formulaire de copyright signé par les auteurs, téléchargé sur :
   `http://www.revuesonline.com`

SERVICE ÉDITORIAL – HERMES-LAVOISIER
14 rue de Provigny, F-94236 Cachan cedex
Tél : 01-47-40-67-67
E-mail : revues@lavoisier.fr
Serveur web : http://www.revuesonline.com