Correspondances et transformations actives dédiées aux IHM

Olivier Beaudoux, Arnaud Blouin, Slimane Hammoudi

Correspondances et transformations actives dédiées aux IHM

Olivier Beaudoux — Arnaud Blouin — Slimane Hammoudi

GRI
Groupe ESEO
Angers, France
{olivier.beaudoux, arnaud.blouin, slimane.hammoudi}@eseo.fr

RÉSUMÉ. *La récente évolution vers le Web 2.0 a mis en évidence l'importance de l'interaction entre les utilisateurs et les applications Web. Cependant, cette évolution n'est pas basée sur la transformation de données sources en vues cibles (graphiques), malgré l'intérêt déjà démontré d'utiliser les transformations dans un tel contexte. Nous proposons d'utiliser le concept de transformation active en tant qu'objet central des systèmes interactifs. Une transformation active transforme des données sources en une vue cible mise à jour dès lors que les données sources changent. Ce papier définit les principes fondamentaux des transformations actives dans une approche IDM. Il explique comment notre environnement AcT permet la génération automatique de transformations actives à partir de correspondances entre méta-modèles source et cible, et comment ces transformations peuvent être exécutées sur une plate-forme.*

ABSTRACT. *The recent evolution to the Web 2.0 underlines that interaction between users and Web-pages has become a non-negligible aspect of the development of Web applications. However, this evolution is currently not based on transforming the source data into target views, despite the high interest and expressiveness of transformations. This paper proposes the use of active transformations as the first class object within interactive systems, in order to reap the benefits from the transformation concept. An active transformation transforms source data into a target view and updates the target view as soon as the source data changes. The paper gives the foundation of active transformations based on MDE (Model-Driven Engineering). It explains how the presented AcT framework allows the automatic generation of active transformations from a graphical mapping, that can be then executed on an implementation platform.*

MOTS-CLÉS : *Correspondance, Transformation active, Interfaces Utilisateur*

KEYWORDS: *Mapping, Active transformation, Graphical User Interface*

1. Introduction

Le Web a récemment évolué d'un usage centré sur la navigation à un usage autorisant l'édition de documents. Par exemple, la rédaction de courriels est possible avec GMail, tout comme l'édition collaborative de pages HTML peut être réalisée *via* des serveurs Wiki. Le Web 2.0 a introduit l'idée de « Rich Internet Application » (RIA) pour exprimer l'évolution des interfaces clientes du Web vers des interfaces aussi riches que celles des applications de bureau traditionnelles. Cette évolution a démarré par l'utilisation de langages standard du Web (XHTML, CSS et JavaScript) complétés par la technologie Ajax (Garrett, 2005). Cependant, ces technologies ont été introduites avant le Web 2.0 et, par conséquent, leur usage n'est pas correctement adapté à la construction de RIA. De nouveaux environnements RIA sont apparus, tels que Flex (Tapper *et al.*, 2006) et WPF (Sells *et al.*, 2005), afin de permettre le développement unifié d'applications RIA pour le Web et d'applications de bureau traditionnelles.

Cependant, programmer des Interfaces Homme-Machine (IHM) reste indéniablement une tâche lourde et consommatrice de temps. Les travaux récents autour de l'IDM montrent que la génération automatique d'IHM à partir d'un modèle suffisamment explicite peut être utilisée. GMF est un exemple illustrant les bénéfices de la fusion d'outils IDM (EMF) et d'outils IHM (GEF) (Moore *et al.*, 2004; Gronback, 2009). GMF permet de générer des éditeurs de diagrammes complets à partir de seulement deux méta-modèles et d'une relation de correspondance entre ces méta-modèles, réduisant ainsi drastiquement l'effort de programmation.

Le but principal de nos travaux est d'appliquer un tel principe d'usine logicielle au contexte particulier des IHM. La première étape, présentée dans cet article, est résumée par la figure 1 : elle consiste à maintenir un lien durable entre un modèle de données sources en un modèle de vues cibles grâce à une transformation active. Un tel lien permet la mise à jour incrémentale des vues cibles dès lors que les données sources sont modifiées. Nous introduisons le terme de transformation active de manière à refléter l'idée des transformations incrémentales (Villard *et al.*, 2002), et de l'implémentation active de correspondances (Akehurst, 2000) (voir section 2). Parce qu'il est généralement assez lourd de programmer des transformations actives, nous proposons l'utilisation d'un *langage de correspondance*, à la fois graphique et textuel ; les transformations actives sont alors générées automatiquement à partir de leurs correspondances. D'autres travaux de recherche ont été réalisés autour de l'idée de transformation active, mais n'ont pas encore été appliqués au contexte des IHM (voir la section suivante) ; nos travaux visent à combler ce manque. La seconde étape de nos travaux, non présentée dans cet article, concerne le modèle d'interaction qu'il est nécessaire de considérer dès lors que les IHM sont spécifiées, de manière indépendante de la plate-forme, par des correspondances.

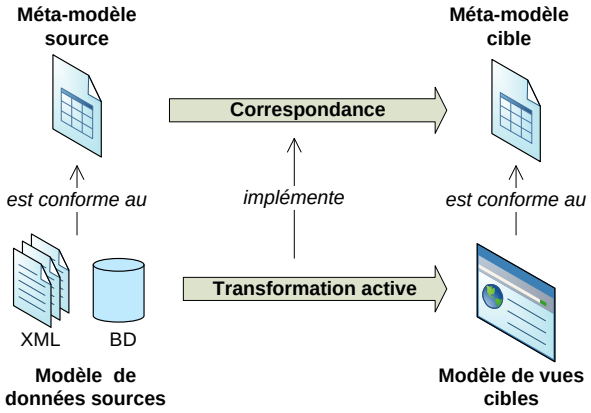

Figure 1 – Correspondance et transformation active

Cet article présente l'environnement *AcT* qui se compose du *langage AcT*, de l'éditeur graphique de correspondances *AcTeditor*, ainsi que d'une implémentation *AcT.NET* permettant d'exécuter les transformations actives sur la plate-forme .NET. Cet environnement est disponible sous licence LGPL à l'adresse *http://gri.eseo.fr/software/act*.

L'article est organisé comme suit. La section 2 présente les travaux de recherche connexes aux processus de transformation et de correspondance entre méta-modèles. La section 3 fournit les principes clés des transformations actives pour le contexte spécifique des IHM. La section 4 explique comment une transformation active est spécifiée par une correspondance *via* l'éditeur graphique *AcTeditor*. La section 5 donne le principe de la génération des transformations actives à partir des correspondances, ainsi qu'à leur exécution sur la plate-forme .NET. La section 6 conclut en synthétisant notre contribution et en donnant les perspectives de l'environnement AcT.

2. Travaux connexes

La séparation des données du domaine de leurs représentations est utilisée depuis longtemps en informatique. Dans le contexte des systèmes interactifs, le motif de conception Modèle-Vue-Contrôleur (MVC) précise explicitement une telle séparation : le modèle définit les données du domaine, la vue et ses contrôleurs présentent le modèle aux utilisateurs (Krasner *et al.*, 1988). MVC permet la synchronisation entre les modèles et leurs vues : dès lors que le modèle change, les vues associées rafraîchissent leur état. Cette synchronisation est implémentée par une liaison *vue-vers-modèle* complétée par un mécanisme de notification de changement d'état *modèle-vers-vues*. La notion de « binding », permettant d'établir le lien des vues vers les données abstraites, est présente dans les plate-formes RIA actuelles (Tapper *et*

al., 2006; Weaver, 2007; Sells *et al.*, 2005), ainsi que dans JFace[1]. Toute implémentation de MVC, basée sur le concept de « binding » ou non, implique que les vues restent dépendantes de leur modèle respectif : les liens sont définis dans les vues. De plus, ces liens dépendent de la plate-forme, contrairement aux correspondances.

L'externalisation du lien entre modèles et vues dans une transformation supprime cette dépendance. Pour cela, des langages de transformation sont utilisés, dont le plus connu est probablement XSLT (Fitzgerald, 2003). Il est largement utilisé pour transformer des documents XML en documents visualisables, tels que des documents (X)HTML ou SVG. Parce qu'XSLT est un processus « batch », il ne maintient pas le lien de synchronisation entre documents sources et documents cibles, contrairement à MVC. Cependant, des versions incrémentales de processus de transformations existent (Beaudoux, 2005b; Villard *et al.*, 2002), mais le codage de transformations incrémentales demeure plus complexe que la création de transformations « batch ».

L'approche IDM peut permettre de s'affranchir de la construction des transformations actives en se focalisant sur le niveau *méta-modèle*. De nombreux langages de transformation de modèles ont été créés (Czarnecki *et al.*, 2006; Blouin *et al.*, 2008b) ; cependant, seulement deux d'entre-eux permettent la spécification de transformations actives. (Akehurst, 2000) propose une *implémentation active* des correspondances entre deux diagrammes de classes. Le système VIATRA permet la construction de transformations dynamiques (« live transformations ») entre des modèles représentés par des graphes (Varró *et al.*, 2002). Ces deux approches illustrent le fait qu'une correspondance devrait être considérée comme la spécification d'une transformation active, dynamique ou incrémentale.

A notre connaissance, les langages de correspondance et leurs implémentations actives n'ont pas encore été spécifiquement appliqués dans le contexte des systèmes interactifs. La seule exception est GMF, mais elle reste spécialisée dans la génération d'éditeurs de diagrammes.

3. Principes fondamentaux

3.1. *Pourquoi les transformations doivent être actives ?*

Parce qu'une correspondance permet d'établir un lien durable entre la source et la cible, son implémentation doit maintenir ce lien à tout moment (Akehurst, 2000; OMG, 2005). Malgré cela, la plupart des langages de transformation ne maintiennent pas un tel lien. Cela résulte probablement de la complexité de l'implémentation des transformations actives et du côté facultatif d'une implémentation active dans le contexte des systèmes d'information. Cependant, Cela s'avère faux dès lors que la cible est une vue (graphique) de la source : dans ce cas, les données sources et les vues cibles doivent rester synchronisées au sens défini par MVC.

---

1. *http ://wiki.eclipse.org/index.php/JFace_Data_Binding*

Principe 1 – Les transformations pour les systèmes interactifs *doivent* synchroniser les données sources et leurs vues cibles, donc être *actives*.

3.2. *Méta-modèle des données sources*

Les données sources peuvent être enregistrées par différentes *plate-formes de données* : documents texte, documents XML ou bases de données. Puisqu'une correspondance doit rester indépendante de la plate-forme, les données sources doivent être spécifiées par un *méta-modèle*, exprimé par un diagramme de type diagramme de classes. Ce méta-modèle doit pouvoir opérer sur différentes plates-formes de données, d'une manière largement abordée dans la littérature (Jouault *et al.*, 2006; Muller *et al.*, 2006; Domínguez *et al.*, 2007).

Principe 2 – Les méta-modèles des données sources *doivent* être spécifiés par des diagrammes de classes simples. Ils *doivent* être implémentés en se basant sur des passages vers des bases de données, des documents XML ou des documents textes.

3.3. *Méta-modèle des vues cibles*

Les vues cibles présentent les données sources aux utilisateurs. Tout comme les données sources, les vues cibles doivent être spécifiées par un méta-modèle de manière à être indépendantes de la *plate-forme graphique*.

De nombreuse boîtes à outils IHM peuvent servir de *plate-forme graphique*. Cependant, la plupart d'entre-elles implémentent le motif de conception MVC d'une façon particulière en utilisant des stratégies différentes pour l'affichage graphique et la gestion des événements. Cette diversité accentue la difficulté du passage à la plate-forme graphique. Il est préférable de ne retenir que celles basées sur le concept de *graphe de scène* introduit par le domaine de la 3D dans le but de pouvoir spécifier des scènes 3D. Les boîtes à outils IHM modernes sont basées sur un tel concept de graphe de scène, telles que SVG, Draw2d (Lee, 2003) et WPF (Sells *et al.*, 2005). Puisque les graphes de scène décrivent les objets graphiques sous forme arborescente, le méta-modèle d'un graphe de scène peut s'exprimer aisément et de manière unifiée par un diagramme de classes.

Les transformations actives n'adressent pas le problème général du lien entre modèles sources et cibles, mais se focalisent plus spécifiquement sur le lien entre *données* sources et *vues* cibles. Dans le domaine de l'IHM, il est clairement admis que les utilisateurs doivent percevoir le modèle des données sources de manière à pouvoir construire leur propre modèle mental du modèle de l'application (Collins, 1995). L'application est alors considérée comme bien conçue si le modèle de l'application reste proche des modèles mentaux des utilisateurs. Ceci implique que le modèle des vues ne doit pas exagérément différer, en terme *structurelle*, du modèle des données sources, au risque de rendre ces vues peu ergonomiques. Par contre, cela ne signifie

pas que les correspondances soient nécessairement triviales : ce n'est en particulier pas le cas pour les IHM du domaine de la visualisation d'information.

Principe 3 – Les méta-modèles des vues cibles *doivent* être spécifiés par des diagrammes de classes simples, indépendamment de la plate-forme graphique, et permettant à l'utilisateur de percevoir le méta-modèle des données sources. Ils *devraient* être implémentés sur la base d'un modèle à graphes de scène.

3.4. *Correspondance et transformation active*

La figure 1 synthétise la différence essentielle entre correspondance et transformation. Une transformation opère directement au niveau modèle. Ceci offre l'avantage de la simplicité ; par exemple, construire un programme XSLT transformant un document XML en un document HTML reste une tâche simple si la grammaire du document XML n'est pas trop complexe. Cette approche a cependant deux inconvénients : premièrement, le programmeur doit lui-même s'assurer que la transformation fonctionne pour tous les documents XML sources conformes à la grammaire source, et qu'elle génère des documents HTML cibles conformes à la grammaire HTML ; deuxièmement, le programme XSLT reste dépendant de la plate-forme des données (XML) et de la plate-forme graphique (HTML), le rendant inutilisable dans d'autres contextes (*e.g.* avec une base de données relationnelle).

L'utilisation d'une correspondance plutôt qu'une transformation permet de travailler au niveau méta-modèle, supprimant ainsi les deux inconvénients précédents.

Puisqu'il est possible de représenter graphiquement les méta-modèles sources et cibles, les correspondances entre ces méta-modèles devraient également être représentées graphiquement. Les détails des correspondances devraient par contre être spécifiés textuellement.

Principe 4 – Une transformation active implémente une correspondance pour les plates-formes des données et graphique souhaitées. Elle *doit* être entièrement générée à partir de sa correspondance. Sa structure principale devrait être précisée graphiquement au dessus des diagrammes des méta-modèles sources et cibles.

4. Méta-modèle d'une transformation active

4.1. *Méta-modèle de la source et de la cible*

Le méta-modèle de données AcT possède certaines spécificités qui le rende bien adapté à notre problématique de correspondance (*e.g.* une classe *est un* ensemble d'instances). D'autres méta-modèles (*e.g.* Ecore (Steinberg *et al.*, 2008)) devraient par contre rester compatibles avec celui de AcT, voire être utilisés à la place de AcT.

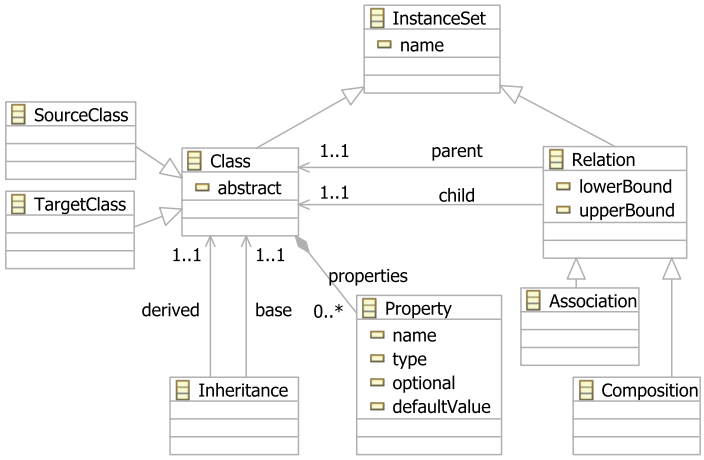

Figure 2 – Méta-modèle source et cible

La figure 2 donne le méta-modèle AcT pour les données sources et les vues cibles. Un méta-modèle source ou cible Σ est un ensemble de classes C, *i.e.* $\Sigma = \{C\}$. Une classe C (méta-classe *Class*) est considérée du point de vue ensembliste : c'est l'ensemble de toutes ses instances (méta-classe *InstanceSet*), noté $C = \{i\}$, dans lequel tout i partage la définition de *propriétés* et de *relations*. Une propriété est une donnée élémentaire représentée par un tuple (*nom, type, optionnel, valeurParDefaut*), où le type est l'un des types simples de *XML Schema* (van der Vlist, 2002). Les instances peuvent être en relation avec d'autres instances, au travers d'associations (méta-classe *Association*) ou de compositions (méta-classe *Composition*). Considérons la relation $r(C_P, C_F)$ entre une classe parente C_P et une classe fille C_F. Pour chaque $i \in C_P$, la relation r définit le sous-ensemble de C_F qui contient les instances en relation avec i, *i.e.* $r(i \in C_P, C_F) = \{j \,|\, j \subset C_F \wedge i \,\text{en relation avec}\, j\}$. Tout comme pour les classes, les *relations* sont des *ensembles d'instances*. Par conséquent, les correspondances entre classes ou entre relations sont traitées de manière unifiée : il s'agit de correspondances entre ensembles d'instances. Finalement, une classe dérivée C_D peut hériter d'une classe de base C_B (l'héritage multiple n'est pas autorisé). La classe dérivée est un sous-ensemble de la classe de base, *i.e.* $C_D \subset C_B$. Une classe de base peut être abstraite, ce qui signifie que $C_B = \bigcup_{k=1..n} C_{D,k}$ où $C_{D,k=1..n}$ sont les classes dérivées de C_B : chaque instance de C_B est nécessairement élément de l'une des classes dérivée $C_{D,k}$.

La distinction des classes sources (méta-classe *SourceClass*) et classes cibles (méta-classe *TargetClass*) est faite de manière à pouvoir définir différentes contraintes. Par exemple, une classe source ne peut pas hériter ni être en relation avec une classe

cible. De plus, la vue cible d'une correspondance est représentée par la classe cible racine qui représente la vue dans son ensemble ; cette classe racine est donc un singleton. Notons que les instances sources et cibles doivent avoir la capacité d'être enregistrées et chargées (dans et depuis un document XML par exemple), ce qui doit être géré au niveau de la plate-forme de données. Le fait que la cible puisse aussi être sauvegardée permet aux utilisateurs de modifier les vues cibles sans pour autant altérer les données sources (*e.g.* suppression d'un lien d'association dans la vue d'un diagramme UML, sans suppression dans le modèle source). Cette capacité implique que le chargement des données sources et des vues cibles soient suivies de leur mise en correspondance, ce qui n'est pas actuellement implémenté dans AcT (la vue est systématiquement construite à partir du modèle).

4.2. *Navigation*

La plate-forme AcT définit son propre langage de navigation permettant de naviguer dans les modèles sources et cibles. Ce langage utilise la notation pointée d'OCL (OMG, 2003), et est également inspiré des chemins et pas de localisation d'XPath (W3C, 2007). Une correspondance AcT utilise la navigation pour spécifier quelles instances et propriétés du modèle source sont en relation avec les instances et propriétés du modèle cible.

Un chemin C est la concaténation de $N \geq 1$ pas $P_{n=1..N}$. Chaque pas P_n s'applique à partir d'une instance donnée i, et désigne soit une propriété de i, soit une relation de i. Un pas respecte les règles de typage suivantes :

– une *valeur* : si $P_n = i.p$ où p est une propriété de i ;

– une *instance* : si $P_n = i.r$ où r est une relation avec une cardinalité égale à 1 ou $0..1$;

– un *ensemble d'instances* : si $P_n = i.r$ où r est une relation avec une cardinalité égale à $a..b$ avec $a \geq 0$ et $b > 1$.

La concaténation de deux pas P_n et P_m, notée $P_n.P_m$, applique le résultat de P_n à P_m en retournant différents types d'éléments comme suit :

– une *valeur* : si $P_n.P_m = i.r_n.p_m$ où r_n est une relation avec une cardinalité 1 ou $0..1$, et où p_m est une propriété de l'instance retournée par $P_n = i.r_n$;

– une *instance* : si $P_n.P_m = i.r_n.r_m$ où r_n et r_m sont deux relations ayant des cardinalités égales à 1 ou $0..1$;

– un *ensemble d'instances* : si $P_n.P_m = i.r_n.r_m$ où r_n et r_m sont deux relations ayant des cardinalités égales à $a..b$ avec $a \geq 0$ et $b > 1$. Si r_n et r_m vérifient $a > 1$, la concaténation $P_n.P_m$ retourne un ensemble d'ensembles d'instances qui est *aplati* pour devenir un ensemble d'instances.

Tous les autres cas de concaténation sont interdits. Une instance peut être sélectionnée dans un ensemble d'instances en utilisant l'accesseur [], ce qui est utile pour les rela-

tions ordonnées. Par exemple, $i.r[1]$ retourne la première instance en relation r avec i. Il est également possible de se déplacer d'un pas en arrière avec la fonction $Parent()$.

4.3. *Méta-modèle des correspondances*

Une correspondance établit une relation durable entre un méta-modèle source et un méta-modèle cible. Puisque sources et cibles sont composées d'ensembles d'instances (*i.e.* de classes et de relations) et de propriétés, une correspondance peut mettre en relation n'importe quel de ces éléments de modélisation.

Une correspondance entre un ensemble d'instances sources S et un ensemble d'instances cibles C définit, pour chaque instance source $s \in S$, l'instance cible associée $c \in C$ représentant la vue de s. Si S est ordonné, la correspondance précise la relation entre les positions de s dans S et les positions de c dans C. Si S n'est pas ordonné, la correspondance met simplement s et c en relation.

Toute correspondance entre les ensembles d'instances S et C est suivie d'une correspondance entre les instances $s \in S$ et $c \in C$. Cette dernière comprend la correspondance entre leurs propriétés ainsi que la correspondance entre leurs relations. La correspondance entre les propriétés de s et c est réalisée par des opérations mathématiques appliquées à des chemins sources et cibles. Par exemple, les propriétés $prenom$ et nom d'une instance source $s \in Personne$ peuvent être mises en relation avec la propriété $text$ d'une instance cible $c \in TextBox$ en utilisant l'opérateur de concaténation : $c.text = s.prenom +'' \ '' + s.nom$. La correspondance entre les relations de s et c est réalisée de manière identique à la correspondance entre les ensembles d'instances S et C, puisque les relations sont des ensembles d'instances.

On peut se demander à présent s'il est pertinent, du point de vue ergonomie des IHM, de définir une correspondance entre un ensemble d'instances et une propriété. Par exemple, si l'on souhaite afficher le nombre d'instances d'une relation, ce nombre doit être affiché par un widget représentant, par exemple, la liste des instances de cette relation ; ainsi, l'utilisateur a un moyen de percevoir à quoi se rattache cette propriété : la mise en correspondance est donc assurée par le widget lui-même. De même, une correspondance entre une propriété et un ensemble d'instances n'a guère de sens dans notre contexte des IHM.

Enfin, plusieurs ensembles d'instances sources (*i.e.* de classes ou/et de relations) peuvent être considérées pour une correspondance : ceci est possible en considérant l'ensemble résultant de l'union de ces ensembles d'instances en tant qu'ensemble source d'une correspondance. Cette opération d'union n'est pour l'instant pas implémentée dans AcT (donc non incluse dans le méta-modèle des données). En ce qui concerne la capacité pour les correspondances à adresser plusieurs ensembles d'instances cibles, ceci est possible en utilisant soit plusieurs définitions de correspondance (solution triviale adaptée à l'idée de vues multiples), soit en autorisant cet adressage multiple pour une même correspondance en précisant la relation d'ordre (*i.e.* la relation entre les positions des instances des ensembles sources et celles des instances des

ensembles cibles), comme nous l'avons proposé dans nos récents travaux sur Malan (Blouin *et al.*, 2008a). Cette relation d'ordre est actuellement implémentée dans AcT, mais pour un unique ensemble cible (voir l'exemple donné à la section suivante).

La figure 3 donne le méta-modèle des correspondances AcT. Une correspondance est un ensemble d'opérateurs (méta-classe *Operator*). Un opérateur met en correspondance un ensemble d'instances sources S (association *Operator.source*) et un ensemble d'instances cibles C (association *Operator.target*). Deux catégories d'opérateurs sont définies : les boucles *ForEach*/*ForA* et l'opérateur *Map*. L'opérateur *ForEach* définit, pour chaque instance source $s \in S$, quelle instance cible $c \in C$ est en correspondance avec s. L'opérateur *ForA* définit quelle instance source s sélectionnée dans S correspond à une instance cible c, qui représente l'instance racine de la vue cible (voir la section 4.1). Une fois qu'un opérateur *ForA* ou *ForEach* a mis en correspondance s avec c, l'opérateur *Map* décrit comment les propriétés et les relations de s et c sont en correspondance.

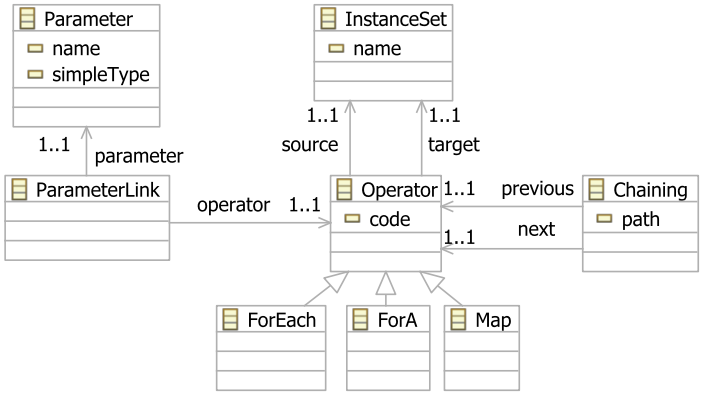

Figure 3 – Méta-modèle de correspondance

Les opérateurs sont chaînés (méta-classe *Chaining*) de la manière suivante. Une correspondance débute par au moins un opérateur *For*, suivi par un opérateur *Map*. A son tour, si l'opérateur *Map* établit une correspondance entre une relation source et une relation cible, il est chaîné à d'autres opérateurs *For* qui mettent en correspondance ces relations.

Une correspondance peut avoir plusieurs paramètres (méta-classe *Parameter*) liés aux opérateurs. Cela permet à l'utilisateur de paramétrer dynamiquement la vue cible, comme dans l'exemple du choix du critère de tri dans un widget de type liste.

4.4. *Exemple*

Cette section détaille un exemple assez complet réalisant la correspondance entre un planning scolaire et sa représentation graphique dans un agenda. La figure 4 est une capture d'écran d'une telle représentation graphique pour un planning enregistré dans un document XML.

La figure 5 (voir page suivante) présente la correspondance réalisée avec l'application *AcTeditor*. La partie gauche définit le méta-modèle source du planning. Une *Semaine* d'enseignement est décomposée en 5 *Jour*s d'enseignement, lesquels accueillent différents *Enseignement*s. Chaque enseignement concerne une *Matiere*, est assuré par un *Enseignant*, a lieu dans une *Salle*, et se déroule sur une ou deux *PlageHoraire*s consécutives. Si aucun enseignant n'est précisé pour un enseignement, le premier enseignant de la matière de l'enseignement (association *Matiere.enseignants*) est considéré. La partie droite de la figure représente le méta-modèle cible basé sur trois composants graphiques : le *CanvasAgenda* qui comprend des *LigneHeure*s ainsi que des *VignetteEvenement*s.

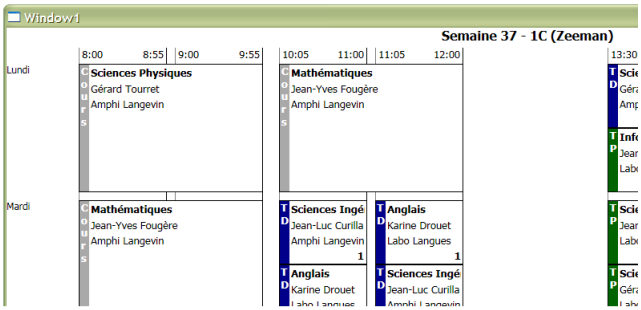

Figure 4 – Un exemple de vue cible

La structure générale de la correspondance est définie par les opérateurs *ForA*, *ForEach* et *Map*, représentés respectivement par les flèches «⟶, «⟷» et ↔. Une flèche → dénote une instance unique et une flèche ↠ dénote un ensemble d'instances. Les opérateurs sont chaînés entre-eux, chaînage représenté par la flèche ⇢. Le code associé aux opérateurs est donné dans des notes pour plus de lisibilité ; dans l'application *AcTeditor*, ce code est éditable dans une boîte de dialogue non représentée sur la figure.

La correspondance commence par l'opérateur *ForA* qui établit la correspondance entre la semaine sélectionnée (mot clé *select*) *via* le paramètre *indexSemaine* parmi les différentes semaines d'enseignement définies dans le document XML (classe source *Semaine*) et l'agenda (classe cible racine *CanvasAgenda*). L'opérateur est ensuite chaîné à l'opérateur *Map* qui établit la correspondance entre la semaine sélectionnée et

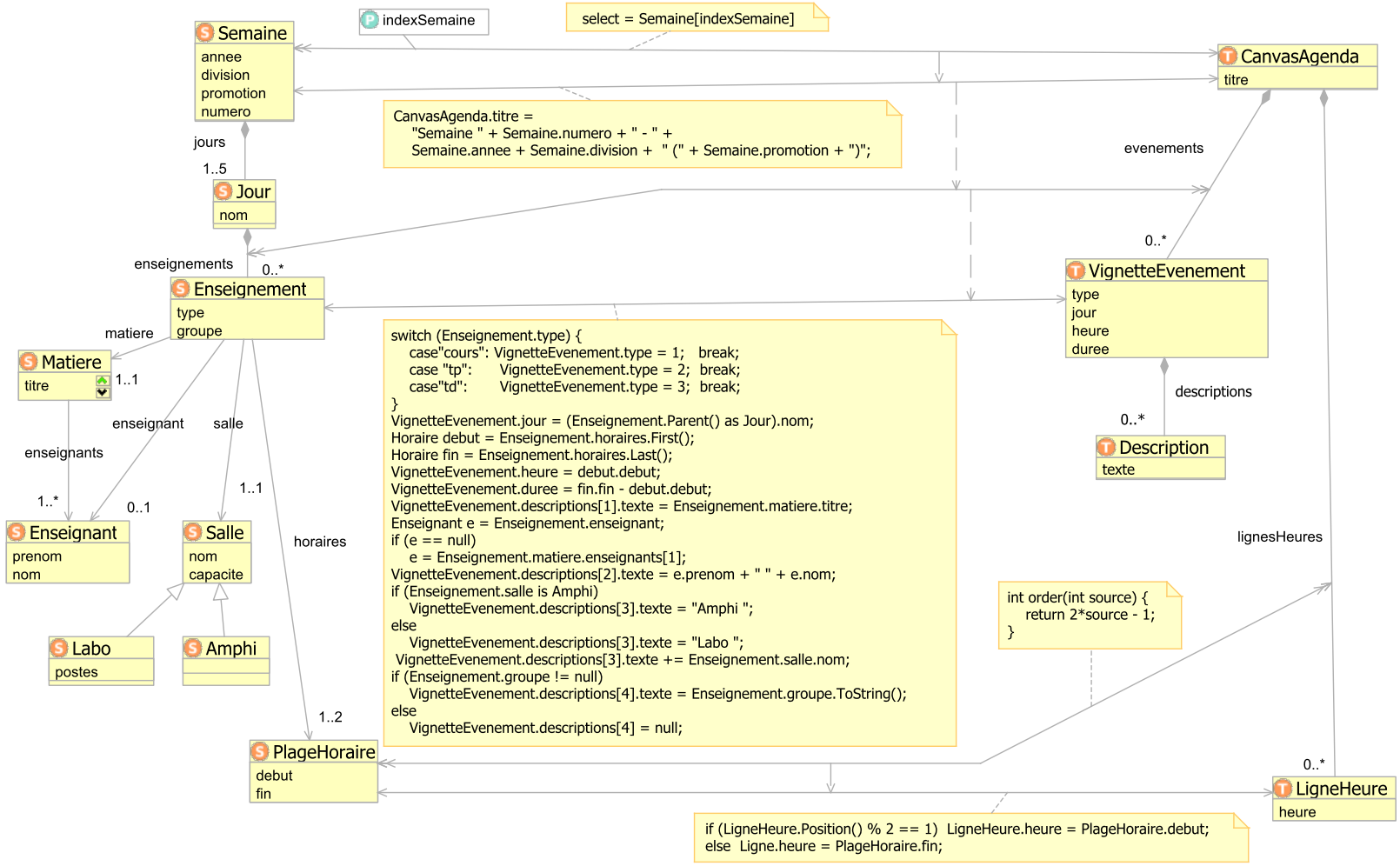

Figure 5 – Un exemple de correspondance

le canvas : le *titre* est formé à partir des différentes propriétés de la semaine. L'opérateur *Map* est chaîné à l'opérateur *ForEach* qui établit une correspondance de relations entre les enseignements de la semaine et les *evenements* du *CanvasAgenda* : il définit le fait que chaque enseignement de la semaine est associé à une vignette représentant un événement de l'agenda. Enfin, chaque enseignement (classe *Enseignement*) est mis en correspondance avec une vignette événement (classe *VignetteEvenement*) par l'opérateur *Map*. La spécification de l'enseignant, de la matière, de la salle ainsi que du groupe est effectuée en faisant correspondre les propriétés de l'enseignement à 4 *descriptions* de la vignette événement.

La correspondance comporte également l'opérateur *ForEach* entre les classes *PlageHoraire* et *LigneHoraire*. Il fait correspondre chaque instance de *PlageHoraire* avec 2 instances de *LignesHeure*, l'une correspondant au début de la plage horaire, l'autre à la fin. Cette mise en correspondance utilise la relation d'ordre entre deux ensembles d'instances *via* la fonction *int order(int s)* qui fournit, pour chaque position des instances de *PlageHoraire* la position des instances de *LigneHeure*. Au niveau de la transformation active qui implémente cette correspondance, cette relation d'ordre induit la création de 2 instances de *Ligne*Heure pour chaque nouvelle instance *PlageHoraire*, ainsi que leur insertion dans la relation *lignesHeures*. L'opérateur *Map* est finalement chaîné à ce dernier *ForEach* pour établir la correspondance entre les propriétés de *PlageHoraire* et de *LigneHeure*. Il utilise un test de parité de la position de *LigneHeure* (fonction *Position()*) de manière à faire correspondre à *LigneHeure* soit le début de la plage horaire, soit la fin.

5. Exécution d'une transformation active

La figure 6 résume le processus de génération des transformations actives pour la plate-forme d'exécution .NET. La plate-forme source est ADO.NET, ce qui autorise les données sources à être stockées dans des documents XML ou des bases de données relationnelles (Hamilton, 2008). La plate-forme graphique, basée sur le concept de graphe de scène et offrant une grand variété de widgets extensibles, est WPF (Sells *et al.*, 2005).

L'utilisateur crée en premier lieu la correspondance entre la *Source* et la *Cible* en utilisant le logiciel *AcTeditor*, comme indiqué dans la section précédente. Cette correspondance est sauvegardée dans le document XML *SourceVersCible.act*. Un menu contextuel de *AcTeditor* permet la génération de la transformation active, qui se compose de fichiers C# et *XML Schema*. Le méta-modèle source est spécifié par un schéma XML dans le fichier *Source.xsd*, lequel est utilisé par ADO.NET pour charger et sauvegarder les modèles sources dans des documents XML ou des bases de données. Le fichier C# *Source.cs* implémente toutes les classes du méta-modèle source, lesquelles permettent le chargement et la sauvegarde des instances du méta-modèle source, ainsi que la navigation à l'intérieur des ensembles d'instances sources. D'une manière similaire, le méta-modèle des vues cibles est spécifié dans le schéma XML *Cible.xsd* et implémenté dans le code C# *Cible.cs*. La transformation active, implémentée dans

le fichier *SourceVersCible.cs*, gère le lien entre la source et la cible en utilisant le motif de conception « observable - observateur » intégré à ADO.NET : la transformation est l'observateur des données sources observables. Le rendu graphique du modèle relationnel de la cible est ensuite réalisé par des vues WPF. Comme pour les transformations actives, les vues sont des observateurs des modèles cibles observables : elles observent le modèle cible de manière à synchroniser le rendu graphique. L'utilisateur doit créer, si nécessaire, ses propres contrôles *via* les fichiers *ControleUtilisateur.xaml* et *ControleUtilisateur.xaml.cs*. Ces contrôles correspondent soit à des formulaires utilisant d'autres contrôles, soit à de nouveaux contrôles spécifiques tels que ceux de notre exemple (*CanvasAgenda*, *LigneHeure* et *VignetteEvenement*). Les environnements *VisualStudio* ou *Expression Blend* peuvent être utilisés à cet effet.

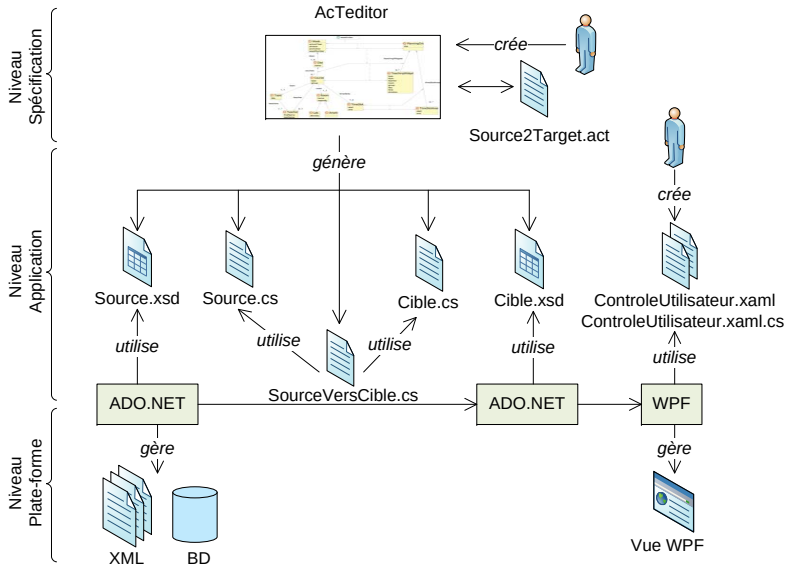

Figure 6 – Implémentation sur la plate-forme .NET

6. Conclusion

Cet article propose une utilisation, basée sur l'environnement AcT, du concept de correspondance adapté au contexte des systèmes interactifs. Une correspondance définit une relation entre un méta-modèle de données sources et un méta-modèle de vues cibles. Une correspondance est exprimée par le langage AcT : la structure générale de la correspondance est précisée graphiquement, tandis que ses détails sont spécifiés textuellement. Cette approche vise à être la plus simple possible et reste ainsi basée

sur un nombre réduit d'éléments de construction. Il ne s'agit pas d'un langage complet pour le contexte général de la correspondance, mais à pour but de répondre au contexte spécifique des IHM, où les vues cibles doivent présenter les données sources à l'utilisateur. L'environnement AcT inclut l'éditeur graphique *AcTeditor* dédié à la construction des méta-modèles sources et cibles, ainsi qu'à l'élaboration de la correspondance entre ces deux méta-modèles. Il permet la génération du code des transformations actives à partir d'une correspondance spécifiée avec le logiciel. Grâce à l'implémentation AcT.NET, ces transformations actives peuvent être exécutées sur le plate-forme .NET en tant qu'objets autonomes.

La plate-forme AcT en est aujourd'hui à sa première version et devra évoluer sous plusieurs aspects. Nous travaillons actuellement sur l'implémentation de notre modèle d'interaction adapté au concept de correspondance appliqué aux IHM. Nos précédents travaux sur eXAcT ont défini les concepts de bases à utiliser pour l'interaction, lesquels sont basés sur la notion d'interacteur de (Myers, 1989). L'interaction devra également inclure le partage des données sources ou/et des vues cibles, en temps réel ou en temps différé, sur la base du concept de point d'événement (Beaudoux, 2005a). D'autres plate-formes d'implémentation et d'exécution seront également à considérer (*e.g.* JavaFX et Flex), de manière à pouvoir générer une même application sur différentes plates-formes. Enfin, le code des correspondances, qui est actuellement un sous-ensemble du langage C#, devra évoluer vers un langage de correspondance de plus haut niveau et indépendant de la plate-forme d'exécution, sur la base de nos travaux sur Malan (Blouin *et al.*, 2008a). Il devra permettre la spécification de correspondances complexes, telles que l'on peut les rencontrer dans les applications de visualisation d'information (*e.g.* à base de zoom sémantique).

7. Bibliographie

Akehurst D. H., Model Translation : A UML-based specification technique and active implementation approach, PhD thesis, Computing Lab., University of Kent, 2000.

Beaudoux O., « Event Points : Annotating XML Documents for Remote Sharing », *Proc. of DocEng'05*, ACM Press, p. 159-161, 2005a.

Beaudoux O., « XML Active Transformation (eXAcT) : Transforming Documents within Interactive Systems », *Proc. of DocEng'05*, ACM Press, p. 146-148, 2005b.

Blouin A., Beaudoux O., Loiseau S., « Malan : A Mapping Language for the Data Manipulation », *Proc. of DocEng'08*, ACM Press, 2008a.

Blouin A., Beaudoux O., Loiseau S., « Un tour d'horizon des approches pour la manipulation des données du Web », *Document numérique - Les documents et le Web 2.0*, 2008b.

Collins D., *Designing Object-Oriented User Interfaces*, Benjamin/Cumming, 1995.

Czarnecki K., Helsen S., « Feature-based survey of model transformation approaches », *IBM Systems Journal*, vol. 45, n° 3, p. 621-645, 2006.

Domínguez E., Lloret J., Pérez B., Áurea Rodríguez, Rubio A. L., Zapata M. A., « A Survey of UML Models to XML Schemas Transformations », *proc. of WISE'07*, Springer, p. 184-195, 2007.

Fitzgerald M., *Learning XSLT*, O'Reilly, 2003.

Garrett J., Ajax : A New Approach to Web Applications, Technical report, Adaptative Path, 2005.

Gronback R. C., *Eclipse Modeling Project : A Domain-Specific Language Toolkit (to be published)*, Addison Wesley Professional, 2009.

Hamilton B., *ADO.NET 3.5 Cookbook*, O'Reilly, 2008.

Jouault F., Kurtev I., « TCS : a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering », *Proc. of GPCE'06*, ACM Press, p. 249-254, 2006.

Krasner G. E., Pope S. T., « A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80 », *Journal of OOP*p. 26-49, 1988.

Lee D., Display a UML Diagram using Draw2D, Technical report, IBM, 2003.

Moore B., Dean D., Gerber A., Wagenknecht G., Vanderheyden P., *Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework*, IBM, 2004.

Muller P.-A., Fleurey F., Fondement F., Hassenforder M., Schneckenburger R., Gérard S., Jézéquel J.-M., « Model-Driven Analysis and Synthesis of Concrete Syntax », *Proceedings of the MoDELS/UML 2006*, 2006.

Myers B. A., « Encapsulating interactive behaviors », *Proc. of CHI'89*, ACM Press, p. 319-324, 1989.

OMG, UML 2.0 OCL Specification, Technical report, OMG, 2003.

OMG, MOF QVT Specification, Technical report, OMG, 2005.

Sells C., Griffiths I., *Programming Windows Presentation Foundation*, O'Reilly, 2005.

Steinberg D., Budinsky F., Paternostro M., *EMF : Eclipse Modeling Framework*, Addison Wesley Professional, 2008.

Tapper J., Talbot J., Boles M., Elmore B., Labriola M., *Adobe Flex 2 : Training from the Source*, Adobe Press, 2006.

van der Vlist E., *XML Schema*, O'Reilly, 2002.

Varró D., Varró G., , Pataricza A., « Designing the Automatic Transformation of Visual Languages », *Science of Computer Programming*, vol. 44, n° 2, p. 205-227, 2002.

Villard L., Layaïda N., « An Incremental XSLT Transformation Processor for XML Document Manipulation », *Proc. of WWW'02*, ACM Press, p. 474-485, 2002.

W3C, XML Path Language (XPath) 2.0 Recommendation, Normative recommendation, W3C, 2007.

Weaver J. L., *JavaFX Script : Dynamic Java Scripting for Rich Internet/Client-side Applications*, APress, 2007.