

Transformation horizontale de PSMs: de EJB à .NET

Jamal Abd-Ali, Karim El Guemhioui

Département d'informatique et d'ingénierie
Université du Québec en Outaouais (UQO)
C.P. 1250 succursale Hull, Gatineau (Québec), H3A 2N4 Canada
{abdj01,karim}@uqo.ca

Résumé. Depuis une dizaine d'années, la technologie des composants a connu une expansion rapide d'abord avec EJB (Enterprise JavaBeans) et plus récemment avec .NET. De nombreuses compagnies n'hésiteraient probablement pas à embrasser la nouvelle vague technologique des composants .NET si elles pouvaient récupérer (une partie de) leur investissement dans EJB. Comme il est plus que probable que les applications EJB existantes ne disposent pas de modèles indépendants de la plate-forme technologique (PIM), nous proposons un chemin de migration horizontale entre ces deux technologies par la définition d'une transformation qui permet de passer de tout modèle spécifique aux EJB (PSM) à un modèle spécifique aux composants .Net. Les métamodèles de ces deux technologies étant indispensables à la définition de notre transformation; nous utilisons un métamodèle de EJB adopté par l'OMG et nous proposons, pour les composants .Net, un métamodèle de notre cru. La transformation est écrite dans un langage bien défini dérivant d'une soumission de proposition en réponse au RFP lancé par l'OMG pour la normalisation du langage d'expression de transformation QVT.

Mots clés : transformation de modèle, métamodèle, IDM, MDA, transformation horizontale, EJB to.NET

1 Introduction

En plus de recourir à la technologie des composants, tels que les Enterprise JavaBeans (EJB) [14], le développement des applications modernes se caractérise par une activité de modélisation qui va au-delà de la simple présentation et documentation des systèmes, pour prendre sa place au cœur de la production. Ceci se concrétise par le biais de générateurs de code, voire par toute une architecture de développement comme le *Model Driven Architecture* (MDA) [8] défini par le consortium *Object Management Group* (OMG) [10]. MDA et, de façon plus générale, l'ingénierie dirigée par les modèles, préconise une conception neutre par rapport à la technologie du moment, à partir de laquelle sont dérivés des modèles spécifiques à une ou plusieurs technologies d'implantation et servant à la génération automatique du code approprié.

Par ailleurs, chaque fois qu'une nouvelle technologie émerge en offrant des solutions à des problèmes existants et un ensemble d'outils de support, on constate une migration vers la nouvelle technologie. La récente technologie des composants .Net ne semble pas échapper à la règle.

Alors que la technologie des EJB dispose d'un métamodèle reconnu [6], la technologie des composants .Net n'a pas, à notre connaissance, de métamodèle publié qui nous permettrait l'écriture de modèles spécifiques à cette plate-forme (PSM). L'existence d'un tel métamodèle permettrait en outre la définition de règles de transformation pour une conversion automatique entre PSMs, évitant une adaptation coûteuse d'applications existantes que l'on voudrait porter sur la nouvelle plate-forme.

Dans cet article, nous proposons donc un chemin de migration horizontale entre ces deux technologies par la définition d'une transformation qui permet de passer de tout modèle spécifique aux EJB (PSM) à un modèle spécifique aux composants .Net. À défaut de disposer d'un métamodèle pour les composants .NET, nous en proposons un, certainement perfectible et que nous sommes prêt à amender ou remplacer si une meilleure proposition devenait disponible.

2 Le métamodèle des EJB

Le travail de plusieurs compagnies pionnières dans l'orienté objet et le calcul distribué a donné naissance à un métamodèle de l'architecture des Enterprise JavaBeans [6], qui a d'ailleurs été standardisé par l'OMG. Le noyau de ce métamodèle est schématisé dans le diagramme de la figure 2. Tout composant EJB est dérivé de *EnterpriseBean* qui fait lui-même partie d'un élément *EJBjar*. Ce dernier est la racine d'un descripteur de déploiement qui contient, entre autres, les informations nécessaires au déploiement du composant EJB et à sa gestion. L'élément *Assembly* concrétise le descripteur de déploiement, en organisant les informations qu'il contient selon une structure bien définie dans la spécification de cette technologie.

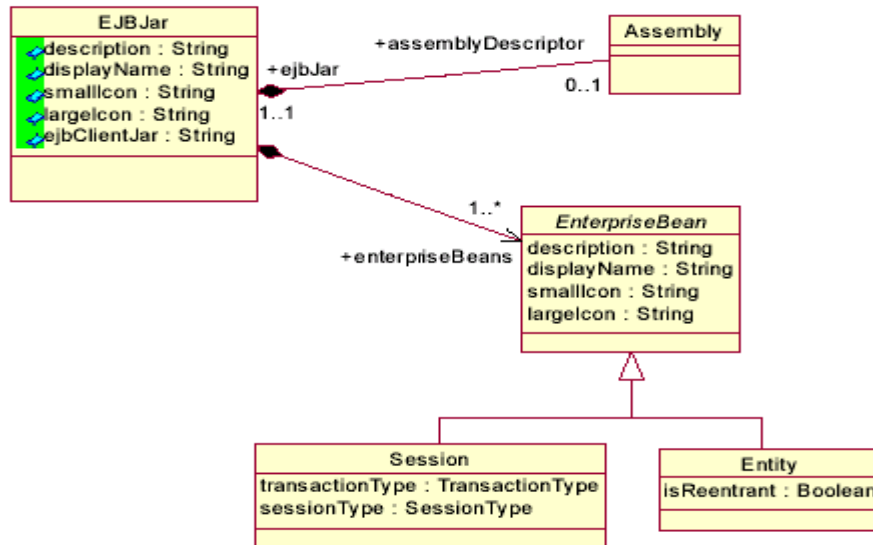


Fig. 1. Le diagramme principal du métamodèle des EJB

Un composant de type *Session* offre, via son interface, des services aux clients et définit des méthodes implémentant des fonctionnalités propres à un domaine métier. Un composant de type *Entity* modélise un concept métier et offre un service de gestion de persistance des états de ses instances.

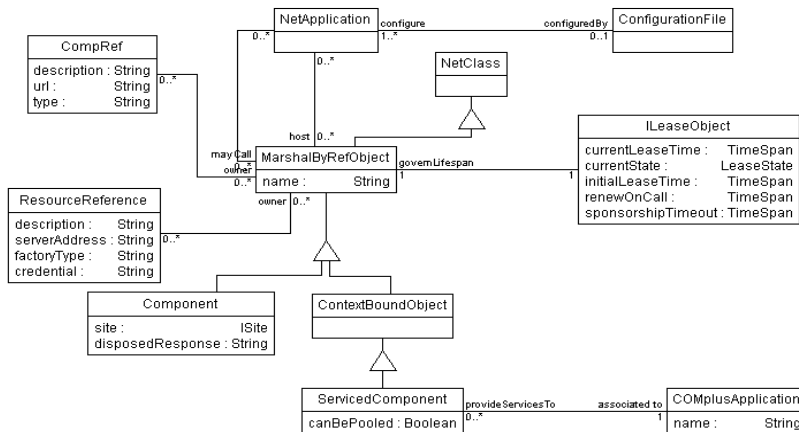
Le métamodèle des EJB comprend d'autres diagrammes et dépend du métamodèle de Java. Une consultation de la norme officielle [3] serait un bon préliminaire à la lecture des règles de transformation que nous présentons plus loin.

3. Un métamodèle des composants .Net

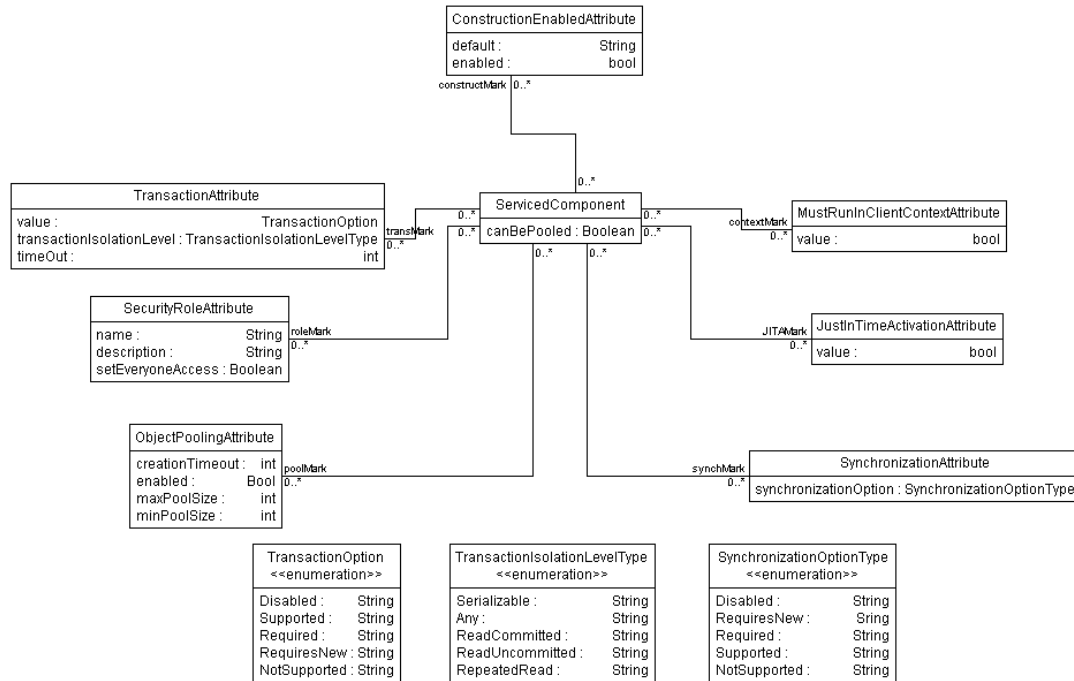
Le métamodèle proposé est conçu dans un souci d'inclure l'essentiel de la sémantique des éléments de la technologie des composants .Net. Vu la limitation d'espace à laquelle est assujéti cet article, on ne présentera que deux diagrammes de notre métamodèle, laissant le soin au lecteur intéressé de trouver de plus amples informations dans [1].

3.1 Diagramme principal et diagramme des services COM+

Le diagramme (a) de la figure 1 traduit le fait que la classe d'implémentation de tout composant .NET doit hériter de MarshalByRefObject (MBR) et qu'elle est généralement en relation avec une application hôte, avec une application cliente et éventuellement avec une application COM+ offrant un ensemble de services.



(a) Diagramme principal



(b) Diagramme des Services COM+

Fig. 1. Les deux diagrammes principaux du métamodèle des composants .Net

NetClass

Sémantique: cet élément représente une classe de la technologie .Net.

Éléments reliés: MarshalByRefObject (relation d'héritage). Cette relation indique qu'un composant est une classe de .NET qui jouit de certaines caractéristiques particulières.

MarshalByRefObject

Sémantique : La classe MarshalByRefObject représente un parent de toute classe permettant un accès à distance via une référence sur elle-même. Un objet appelant dispose alors d'un objet proxy qui achemine (*marshal*) les invocations de méthodes vers une instance de la classe du composant appelé. Dans la suite on appelle instance d'un composant tout objet instance de la classe héritant de MBR et implémentant ce composant. De plus, on ne fera pas de distinction entre un composant et son implémentation à moins qu'il y ait risque de confusion.

Éléments possédés

name.

Éléments reliés

ComPlusApplication, NetApplication et ILeaseObject.

ILeaseObject

Sémantique

Un objet ILeaseObject est associé à une instance d'un composant. Il encapsule un ensemble de valeurs de champs que le processus hébergeant utilise pour gérer la durée de vie de cette instance.

Éléments possédés

- ❑ CurrentLeaseTime: dont la valeur représente un laps de temps de type TimeSpan défini dans la technologie .Net. C'est le temps restant pour une instance.
- ❑ CurrentState de type LeaseState : c'est une énumération exprimant l'état du cycle de vie d'une instance et dont les membres sont : Active, Expired, Initial, Null et Renewing.
- ❑ InitialLeaseTime de type TimeSpan : donne la durée de vie dont une instance dispose au moment de sa création. Cette durée diminue avec le temps et dépend de plusieurs autres événements.
- ❑ RenewOnCall de type TimeSpan : c'est l'extension de la durée de vie restante d'une instance suite à une invocation d'une méthode sur cette instance.
- ❑ SponsorshipTimeout de type TimeSpan : Le client d'un MBR peut se mêler explicitement de ILeaseObject et créer un objet appelé sponsor pour cette tâche. Le processus hébergeant doit alors notifier ce sponsor et

attendre une réponse, durant un temps égal au `SponsorshipTimeout`, avant de finaliser une instance d'un composant.

Éléments reliés

`MarshalByRefObject` : `ILeaseObject` fournit les valeurs des paramètres qui conditionnent la gestion de la durée de vie d'une instance d'un composant héritant de la classe `MBR`.

NetApplication

Sémantique

Une application `.Net` peut héberger ou être cliente d'un (ou plusieurs) composant et cette relation doit être déclarée par configuration en utilisant soit un fichier de configuration soit la programmation.

Éléments reliés

`MarshalByRefObject` et `ConfigurationFile`.

ServicedComponent

Sémantique

La classe `ServicedComponent` (`SC`) est une classe qui hérite de la classe `MBR` de `.Net` et qui est caractérisée par son aptitude à profiter des `.Net Enterprise Services` (appelées aussi services `COM+` [4],[5]et[16]). Notons que `ServicedComponent` hérite de `ContextBoundObject` dont les instances gardent toujours un contexte constant de services. Donc un tel composant ne peut pas changer de contexte (sinon il sera dit contexte agile). Lorsqu'il n'y a pas de risque de confusion, on désignera par `SC` ou `ServicedComponent` aussi la classe que l'objet.

Éléments reliés

`ComPlusApplication`

ComPlusApplication

Sémantique

Une application `COM+` sert à configurer à la fois plusieurs composants instances de `ServicedComponent` (`SC`) et qui partagent des services communs. Généralement, ces composants font partie d'une seule application.

Éléments possédés

`name`: c'est une chaîne de caractères désignant le nom de l'application `COM+` que référencera un `SC` pour être associé à l'application.

Éléments reliés

`ServicedComponent`: l'association de `ComPlusApplication` à cet élément; traduit le fait qu'un type serveur héritant de `ServicedComponent` doit être enregistré auprès d'une application `COM+` pour profiter des services `COM+`.

Component

Sémantique

En plus d'hériter de `MarshalByRefObject`, une classe `Component` de `.Net` est caractérisée par son appartenance logique à un conteneur, et par ses méthodes virtuelles (à définir dans les classes dérivées) `Dispose()` et `Finalize()` pour relâcher les ressources qu'une instance utilise. De plus, elle offre une action de réponse à l'évènement `Disposed` déclenché pour permettre de détruire une instance.

Éléments reliés

`NetApplication` : représente un application `.Net` jouant le rôle d'hébergement ou de client.

CompRef

Sémantique

Chaque élément représente une référence à un composant `.Net` et contient les informations d'accès à ce dernier.

Éléments possédés

`description`, `url` et `type` sont les attributs de cet élément représentant respectivement une description, l'url d'accès à distance et un string donnant le nom de la classe d'implémentation et le nom de l'*assembly* du composant référencé.

Élément relié

`MarshalByRefObject`

ResourceReference

Sémantique

Un tel élément indique que le composant qui y est relié utilise une ressource; il fournit les informations d'accès à cette dernière.

Éléments possédés

description, serverAddress, factoryType et credential sont les attributs fournissant les informations sur la ressource référencée.

Élément relié

MarshalByRefObject

ConfigurationFile

Sémantique

Un fichier de configuration associé à une application .Net renferme les informations nécessaires à la configuration d'hébergement ou d'appel d'un composant MarshalByRefObject. Il spécifie, entre autres, le port d'écoute pour les appels et les modes d'activation des composants.

Éléments possédés

Tous les autres éléments du diagramme de configuration expliqué ci-après. Car ces éléments représentent les éléments XML du fichier de configuration.

Le diagramme (b) de la figure 1 indique les principaux services COM+ offerts à un composant de .Net héritant de la classe ServicedComponent. Une explication détaillée de ce diagramme et de la technologie des composants .Net est disponible dans [4], [5] et [16].

Le métamodèle proposé est simple et inclue l'essentiel de la technologie des composants .Net. Il permet de décrire l'implémentation d'une application basée sur les composants .Net et aussi les règles de transformation ciblant cette technologie Il pourrait aussi servir à introduire rapidement cette technologie dans une perspective de présentation de l'architecture des composants .Net.

3.2 Les caractéristiques des transformations

Une transformation de modèles est un processus qui permet de convertir un modèle en un autre [8]. Dans une vision MDA ou IDM, une transformation est conçue pour être automatisée, elle est formée d'un ensemble de règles et doit être écrite dans un langage bien défini qu'un outil pourra compiler et exécuter. Cette automatisation sous-entend que les modèles doivent être écrits dans un langage traitable par les machines. un langage de transformation se caractérise par la manière dont ses règles s'appliquent [2], ce qui amène à spécifier, entre autre:

- L'ordre d'application sur les fragments correspondant à une règles
- L'ordre dans lequel s'appliquent les différentes règles
- La possibilité de composition de règles
- la relation entre les modèles source et cible. (même, différent)
- La traçabilité qui permet de retrouver l'élément générateur dans le modèle source, pour tout élément généré dans le modèle cible.
- La bidirectionnalité
- L'interactivité permettant la paramétrisation.

Ceci étant, nous allons examiner plus en détail l'ingrédient essentiel de toute transformation, à savoir : la règle. Elle est généralement une expression de correspondances entre un fragment du modèle source et un fragment du modèle cible; et elle se caractérise par :

- L'usage de variables, leur type et leur visibilité
- La manière de spécifier un fragment source ou cible par des patterns: graphiquement ou par du texte.
- La logique d'exécution: impérative ou déclarative
- Bidirectionnalité, traçabilité et paramétrisation au niveau de la règle.

4. La transformation

Pour chacune des deux technologies discutées (EJB et .NET), chaque élément d'un PSM est une instance d'un élément unique du métamodèle de cette technologie. Nous allons donc spécifier chaque élément mis en jeu dans une règle de transformation en se référant à son type dans le métamodèle correspondant et en imposant une condition de filtrage portant sur les propriétés de cet élément et du modèle PSM dans lequel il figure.

4.1 Langage d'expression

La transformation qui convertit un PSM spécifique aux EJB en un PSM spécifique à .Net consiste en un ensemble de règles. Pour l'écriture de ces règles, nous utilisons le langage de transformation de modèles Open QVT[9] tel qu'implémenté par la syntaxe concrète de l'outil ATL [15] qui à son tour supporte le langage normalisé *Object Constraint Language* [11].

Rappelons qu'Open QVT est une soumission de proposition en réponse à l'appel RFP émis par OMG depuis 2002. Cette proposition et son implémentation par ATL ont été élaborées par un groupe de compagnies internationales et d'universités françaises pionnières dans le domaine.

Au niveau de la syntaxe de base d'ATL, notons:

- L'analyseur d'ATL est sensible à la casse, au niveau des noms de dossiers et du code source.
- Les commentaires sont identifiés par les tirets "--".
- Chaque règle doit avoir un nom unique dans son champ de visibilité.
- Une attribution de valeur est effectuée avec l'opérateur de flèche gauche "<".
- Deux attributions de valeurs, côte à côte, sont séparées par une virgule ",".
- Le symbole # au début d'une chaîne de caractères indique que cette chaîne fait référence à un membre d'une énumération.
- Pour se diriger d'un élément vers son attribut, écrire le nom de l'élément, puis "." et le nom de l'attribut.
- OCL est utilisé pour exprimer des conditions de filtrage et des attributions de valeurs.
- La concaténation de chaînes est représentée par l'opérateur +
- Les valeurs littérales sont comprises entre guillemets: " "

La syntaxe de la définition d'une règle de transformation est de la forme suivante:

```
rule R {
  from e : nom-meta-entrée ! el-e (cond)
  to s : nom-meta-sortie ! el-s
  ( -- séquence d'attribution de valeurs pour peupler
    -- l'élément créé
    -- ex. title <- e.title, nom <- e.name + "nouveau"
  )
}
```

Avec:

R: nom de la règle.

nom-meta-entrée : nom du métamodèle du modèle d'entrée.

el-e: nom d'un élément du métamodèle. Les instances de cet élément subiront l'application de la règle R, à tour de rôle.

e: nom local de l'instance de el-e (en cours de traitement par la règle R) rencontrée dans le modèle d'entrée.

cond: condition de filtrage sur les instances d'entrée à la règle R.

nom-meta-sortie :nom du métamodèle du modèle de sortie.

s: nom local de l'instance créée en sortie de la règle.

el-s: nom de l'élément du métamodèle à instancier dans le modèle de sortie, chaque fois qu'une instance d'entrée nommée e est rencontrée.

En plus des règles, on peut spécifier des actions à initier au début de la transformation ou d'une règle. De telles actions sont définies dans des blocs de code avec en-tête *init*. À noter qu'*init* ainsi que d'autres éléments du métamodèle de OpenQVT et de son langage TRL ne sont présentement pas directement supportés dans ATL.

Les notations suivantes aideront à simplifier l'expression des règles:

EJB : le métamodèle des EJB.

Java : le métamodèle de Java.

Net : le métamodèle des composants .Net.

in : le modèle d'entrée.

out : le modèle de sortie.

Selon OpenQVT, la déclaration “Net::UseDefaultValue := true” implique que la création d’un élément provoque la création de tous ses éléments possédés avec une initialisation suivant les valeurs par défaut. Cette déclaration sera implicite dans la formulation des règles de notre transformation.

La navigation d’un élément a, instance de A, vers un élément b, instance de B, utilise le nom de rôle que joue B dans l’association qui le lie à A dans le métamodèle. Si l’association ne mentionne pas de nom de rôle, on utilisera le nom B. Ainsi, On aura a.B = a.roleDeB = b.

De plus, on utilisera les noms d’instances qui suivent les mots clés *from* et *to* comme noms de variables visibles dans la même règle où elles sont déclarés.

Ces deux derniers usages syntaxiques ne sont pas supportés par ATL, mais nous sont utiles pour référencer des éléments d’un modèle à l’intérieur d’une même règle et formuler efficacement nos règles de transformation.

4.2 Les règles de transformation

Les règles sont brièvement expliquées en langage naturel, puis formulées en utilisant la syntaxe ATL précédemment introduite. Dans nos explications en langage naturel, nous utilisons le symbole = pour signifier aussi bien une affectation de valeur qu’une référence à un élément du modèle source impliquant une égalité avec son correspondant par la règle de transformation. Par manque d’espace, les définitions de certaines règles ont été volontairement omises.

Règle 1. COMplusApplication - Si le modèle de départ contient au moins une instance de type EJBJar, on crée dans le modèle de sortie une instance de COMplusApplication qu’on note Comp avec :

- L’attribut name mis à “Comp”
- L’attribut activationOption de l’élément ApplicationActivationAttribute qui est indirectement associé à Comp, via Assembly, est mis à Server.
- L’élément instance de ApplicationAccessControl qui est associé à Comp sera peuplé des attributs suivants:
 - Enabled = true
 - AccessCheckLevel = ApplicationComponent
 - Authentication = Connect
 - ImpersonationLevelOption = Default

```
rule R1 {
  from jar : EJB!EJBJar ( jar = EJBJar.allInstances( ) ->asSequence( )->first( ) )
  -- le filtre utilisé permet de sélectionner seulement la première instance de EJBJar
  to as : Net ! Assembly
  to Comp : Net!COMplusApplication ( name <- “Comp”, Comp.Assembly <-as ),
  to appAC : Net!ApplicationAccessControl ( enabled <- true,
    accesCheckLevel <- # ApplicationComponent ,
    authentication <- #Connect , impersonationLevelOption <- #Default , Assembly <- as ),
  to appAct : Net!ApplicationActivationAttribute ( activationOption <- #Server, Assembly <- as )
}
```

Règle 2. Application, LifeTime, Channels, Channel et Service

Règle 3. Field - Pour chaque instance de Field de EJB (en fait, de Java), on crée une instance de Field de .Net avec correspondance de nom et de type.

Cette règle doit être munie d’une condition qui la rend non applicable à une instance de Field de EJB qui est associée à une instance de ContainerManagedEntity. Ces dernières instances seront ciblées par la règle 13 spéciale aux instances de Field représentant des champs persistants.

Règle 4. Method - Pour chaque instance de Method de Java, on crée une instance de Method de .Net avec correspondance des noms, des types, et des paramètres avec leur type.

En définissant la transformation de EJB vers .Net, on se contente de toucher à un nombre minimum d’éléments du métamodèle de Java indispensables à la transformation partant du métamodèle EJB. On ne rentrera pas dans les détails de correspondance entre les éléments de base de java (Class, Field, Method, Interface, Parameter) et ceux de .Net, car les nuances à ce niveau entre ces deux technologies, sortent du cadre de notre étude. A titre d’exemple, on adopte les valeurs par défaut des *modifiers* pour Field, Method et Class de .Net, au lieu de se lancer dans une correspondance entre les *modifiers* de Java et ceux de .Net.

Par ailleurs les types primitifs de Java ont été adoptés avec les mêmes noms pour .Net, à l'exception de boolean de Java qui correspond à bool dans .Net. Par manque de place, l'implémentation de cette correspondance ne sera pas montrée.

Règle 5. JavaClass - Pour chaque instance de JavaClass de Java, on crée une instance NetClass avec correspondance des *Fields* et des *Methods*.

De plus, si l'instance de JavaClass, qu'on note intJ, est associée à une instance de EnterpriseBean avec le rôle de remoteInterface ou homeInterface, l'instance de NetClass, qu'on note intN, vérifiera les correspondances suivantes au niveau:

- dans le cas de Session, intN.NetClass = intJ.EnterpriseBean
- dans le cas de Entity, intN.implements = intJ.EnterpriseBean (avec un filtre qui retourne l'instance de ServicedComponent correspondant à Entity et laisse tomber les autres instances résultant de la transformation d'un Entity).

Règle 6. Bean Session avec état ou sans état - Pour chaque instance S de Session de EJB, on crée:

- Une instance SC de ServicedComponent de .Net, avec:
 - name de SC correspondant au S.ejbClass.name
 - Fields de SC correspondants aux S.ejbClass.Fields
 - Methods de SC correspondants à ceux de S d'après la règle 5
 - Si S.sessionType = Statefull, l'attribut CanBePooled de SC est mis à false
 - Si S.sessionType = Stateless, l'attribut CanBePooled de SC est mis à true
- Une instance opa de ObjectPoolingAttribute telle que:
 - enabled = true
 - creationTimeout = 5
 - maxPoolSize = 10
 - minPoolSize = 0
 - opa.mark = S
- Une instance act de Activated de .Net et on l'associe à l'unique instance de NetApplication déjà créée avec:
 - act.Service = l'unique instance de Service déjà créée par la règle 2
 - act.ref = "tcp://localhost:8010" avec le choix 8010 modifiable selon la convenance de celui qui utilise la transformation.
 - act.type = SC.name + ";" + SC.name
- Une instance syn de SynchronizationAttribute de .Net avec :
 - syn.synchronizationOption <- #Required
 - syn.mark = SC

Ci-dessous, la formulation de cette règle et de la fonction *pooling* qu'elle utilise:

```
-- on commence par définir une opération sur les instances de Session.
helper context EJB!Session
def : pooling () : Boolean = if sessionType = #Statless true else false;
```

```
rule R6 {
  from S : EJB!Session
  to SC : Net!ServicedComponent ( CanBePooled <- S.pooling( ), name <- S.ejbClass.name,
    Field <- S.ejbClass.Method, Method <- S.remoteInterface.Method),
  to opa : Net!ObjectPoolingAttribute ( opa.enabled <- true, opa.creationTimeout <- 5,
    opa.maxPoolSize <- 10, opa.minPoolSize <- 0, opa.mark <- S)
  to act : Net!Activated (owner <- Service.allInstances ( ), -- Une seule instance de Service est créée
    ref = "tcp://localhost:8010", type <- S.ejbClass.name + ";" + S.ejbClass.name ),
  to syn : Net!SynchronizationAttribute ( syn.synchronizationOption <- #Required )
}
```

Notons qu'on a créé une instance de ObjectPoolingAttribute pour une instance S indépendamment de son sessionType. Si ce dernier est avec état, cette création n'est pas nécessaire. Une façon d'éviter cette création systématique serait d'utiliser des opérations (helper) ou de définir deux règles différentes qui ciblent sélectivement les instances de Session selon la valeur de leur attribut sessionType.

Règle 7. Gestion de transaction - Pour toute instance bean de EntrepriseBean dont la valeur de l'attribut transactionType est Container, on crée une instance ta de TransactionAttribute de .Net selon la règle suivante.

```
rule R7 {
  from bean : EntrepriseBean!EJB ( transactionType = #Container )
  to tr : TransactionAttribute!Net ( value <-
    bean.MethodElement->asSequence()->first().MethodTransaction.transactionAttribute.f( ),
    -- on cible par cette attribution la première instance de l'ensemble des instances
    -- de MethodTransaction reliées à un même bean
    -- f( ) est une fonction (un helper de ATL) de transactionAttribute
    tr.class <- bean ,
    -- pour associer tr à la bonne instance de SercicedComponent correspondant au bean
    transactionIsolationLevel <- #Serializable, -- c'est un choix conservateur
    timeOut <- 2000 -- 2 secondes
  )
}
```

Cette règle s'applique au cas où transactionType = Container et où les différentes méthodes du même EntrepriseBean partagent le même attribut de transaction.

La valeur de tr.value est donnée par la fonction f (utilisée dans la règle ci haut) qui est définie suivant le tableau de correspondance suivant:

EJB	.Net
TransactionAttribute	TransactionAttribute.value
NotSupported	NotSupported
Required	Required
Supports	Supported
RequiresNew	RequiresNew
Mandatory	Dans ce cas, f() n'est pas définie et il faut ajouter une annotation au modèle cible signalant qu'il faut programmer le lancement d'une exception si un client invoque une méthode de SC sans que le client soit dans un contexte transactionnel.
Never	Dans ce cas, f() n'est pas définie et il faut ajouter une annotation au modèle cible signalant qu'il faut programmer le lancement d'une exception si un client invoque une méthode de SC tout en étant dans un contexte transactionnel.

Règle 8. Security Role et MethodPermission

Règle 9. Références à d'autres composants

Règle 10. Références à des ressources

Règle 11. EnvEntry

Règle 12. Les types

Règle 13 et 14. ContainerManagedEntity - Afin de créer un patron dans .Net qui représente ce qu'est un bean entité dont la persistance est gérée par le *container* de EJB, on va appliquer une suite de règles qui créent pour chaque instance de ContainerManagedEntity, un composant SC de type SercicedComponent qui est au service des clients appelants, et qui a visibilité sur une classe sérialisable (ayant son nom avec le suffixe“_Serializable”) contenant l'ensemble des champs dont on a à gérer la persistance. Une instance att de Attribute de Net contiendra les informations sur les champs clés primaires, et sera associée au composant SC pour lui fournir ces informations.

La gestion de la persistance est alors aisée en utilisant un SGBD (ex. : SQLServer) supportant la gestion de persistance des objets de classes sérialisables au format XML. Les règles sont:

Règle 13. Pour chaque instance `jf` de `Field from Java`, associée à une instance de `ContainerManagedEntity`, on fait correspondre:

- Une instance `f` de `Field` de `Net` avec:
 - `f.name = jf.name`
 - `f.type = jf.type` - Une correspondance des types est supposée être implicite.
 - `f.owner = jf.ContainerManagedEntity`

Cette dernière expression fait référence à l'instance résultant de la première règle de correspondance définie sur un `ContainerManagedEntity` noté `ESerial` ci-dessous.

Règle 14. Pour chaque instance `ent` de `ContainerManagedEntity` de `EJB` on fait correspondre:

- Une instance `ESerial` de `Net` classe avec
 - `NetDefinedAttribute.name = "Serializable"`
 - `name = ContainerManagedEntity.ejbClass.name + "_Serializable"`
- Une instance `SC` de `ServicedComponent` avec
 - `name = ContainerManagedEntity.ejbClass.name`
 - `CanBePooled <- true`
 - On crée une instance `act` de `Activated` de `.Net` avec :
 - `act.Service = l'unique instance de Service déjà créée par la règle 2`
 - `act.ref = tcp://localhost:8010`
 - `act.type = SC.name + "," + SC.name`
- Une instance `opa` de `ObjectPoolingAttribute` avec
 - `enabled: true`
 - `creationTimeout = 5`
 - `maxPoolSize = 10`
 - `minPoolSize = 0`
 - `opa.mark = SC`
- Une instance `kNames` de `Attribute` de `Net` avec:
 - Création d'une instance `f` de `Field` de `Net`
 - `f.name = "KeyfieldsNames"`
 - `f.type = String`
 - Le champ créé `f` sera utilisé pour stocker l'ensemble des noms des champs clés primaires de l'entité. Cette ensemble de noms peut être représenté par la valeur:
`ent.keyFor -> iterate(att : Field, acc : String = "" | acc + att.name + ",")`
 - `kNames.mark = SC`
On a créé ainsi un `Attribute` qui marque `SC` et qui lui fournit les noms des champs clés primaires séparés par des virgules.
- Une instance `syn` de `SynchronizationAttribute` de `Net` avec :
 - `syn.synchronizationOption = Disabled` si `syn.isreentrant = true` ; autrement elle sera égale à `#Required`.
 - `syn.mark = SC`

5. Discussion de la transformation

En écrivant nos règles de transformation, nous étions conscients du fait qu'un `EJB` ne peut jamais devenir un composant `.Net`. Nous avons plutôt essayé de produire un `PSM` selon `.Net` qui permet de répondre aux besoins fonctionnels et non fonctionnels du système représenté par `PSM` source. Les divergences entre source et cible doivent être confinées aux détails dont les modèles font abstraction. Nous nous sommes fixés les deux critères suivants pour nous guider dans l'élaboration des règles de transformation.

- Le premier critère est de ne pas altérer la logique métier (*business logic*) contenue dans le `PSM` source. En effet il est aisé de remarquer que le métamodèle de `EJB` porte essentiellement sur la partie structurelle du modèle métier, laissant de côté les détails d'implémentation. Ainsi, notre transformation fournit à un `PSM` qui conserve pour chaque composant : les *Fields*, les interfaces et les références aux autres composants ou ressources.
- Le deuxième critère consiste à s'assurer que le modèle cible représente des composants `.Net` qui jouissent des services de middleware comparables à ceux représentés par le modèle source de `EJB`.

Table 1. Principaux concepts technologiques et éléments correspondants dans .Net

Concepts EJB		Éléments correspondants dans le métamodèle des composants .Net
Accès à distance	Un composant passe une référence à un appelant distant qui, à son tour, disposera d'un proxy qu'il traite comme une instance locale du composant	MarshalByRefObject
	Un composant possède un nom JNDI	Service, Channel, Activated, WellKnown
Sécurité	Sécurité basée sur les rôles avec un checking au niveau de : l'application, le composant, les méthodes, les interfaces	SecurityRoleAttribute associable à différents éléments
Gestion des instances	Pooling, passivation des instances inactives, durée de vie.	ObjectPoolingAttribute, JITA, ILeaseObject, LifeTime, SingleCall, Singleton.
Gestion du comportement transactionnel	Il y a plusieurs choix déterminant la conciliation entre le contexte transactionnel du client et celui du composant	TransactionAttribute et ses attributs
Synchronisation et <i>reentrance</i>	Le container garantit une mise en file d'attente (en série) des appels ciblant une instance de session. La <i>reentrance</i> n'est pas recommandée mais elle est toujours permise.	SynchronizationAttribute et les options permises via son attribut synchronizationOption. La <i>reentrance</i> peut être permise en mettant la valeur de synchronizationOption à Disabled. On obtient alors une absence de synchronisation, ce qui est pire que la <i>reentrance</i> par rapport à la cohérence des données qui est contrôlable au niveau du SGBD.
Persistence	ContainerManagedEntity permet de stocker ses attributs sur un support permanent en utilisant un service offert par le container et transparent au développeur	Le DefinedNetAttribute dont le nom est Serializable représente un attribut permettant la sérialisation des objets d'une classe au format XML. De l'autre côté, SQLServer permet des requêtes ou mises à jour en manipulant des objets sérialisés au format XML. Ainsi on aboutit toujours à une persistance gérée par programmation mais qui traite directement le stockage des instances de la classe sérialisable associée au ServicedComponent.

Par ailleurs, il importe de souligner que le métamodèle des EJB se base sur celui de Java et que de ce fait, on doit définir les correspondances dans .NET, des éléments essentiels de Java (Class, Method, Interface, PrimitiveType, etc) . Dans le cadre limité de cet article, on se contentera de signaler que la quasi-coïncidence entre ces éléments est implicite à la définition de la transformation. À titre d'exemple, les types primitifs de Java ont été adoptés avec les mêmes noms dans .Net, à l'exception de boolean de Java qui correspond à bool dans .Net.

De plus, on s'est limité à un modèle cible comprenant une seule application .Net et une seule application COM+ et ce par souci de se concentrer sur les caractéristiques d'un composant indépendamment du site sur lequel il roule et du nombre d'applications qui peuvent le réutiliser.

Le modèle cible ne conserve pas les informations de gestion de transaction de EJB si les instances de MethodTransaction d'un EntrepriseBean ne partagent pas la même valeur de transactionAttribute, ce qui veut dire que les méthodes d'un même bean ont différentes gestions du comportement transactionnel. Ceci est dû au fait que la gestion des transactions selon le métamodèle de .Net est configurable au niveau d'un composant plutôt qu'au niveau des méthodes.

À prendre en charge aussi par le développeur, les astuces à utiliser pour remédier au fait qu'un constructeur d'une classe dérivée de `ServiceComponent` ne supporte pas de constructeur avec paramètres. Une méthode `create` de l'interface `home` d'un bean `Entity` de EJB pourra être transformée en un constructeur suivi d'une méthode acceptant des paramètres.

Donc on voit bien que malgré les restrictions dans notre définition de la transformation, il y a toujours un moyen pour les contourner ou les prendre en charge par un outil qui supporte une paramétrisation des règles de transformation et qui offre une interaction conviviale avec l'utilisateur.

6 Conclusion

Le métamodèle proposé pour la technologie des composants `.Net` peut être vu comme un premier pas vers un métamodèle normalisé comparable à celui des Enterprise JavaBeans. Cette normalisation de métamodèle est essentielle pour tirer profit de l'ingénierie dirigée par les modèles, car c'est cette dernière qui fournit le langage d'écriture des PSMs indispensable à la définition des transformations de modèles.

La définition de la transformation horizontale de PSMs entre les technologies EJB et `.NET`, devrait ouvrir la porte à d'autres initiatives de métamodélisation et de définition de transformations ciblant d'autres technologies. Une continuation logique de ce travail sera l'élaboration d'un outil de génération de code qui se basera sur les métamodèles dont on dispose. À ce stade, et même en l'absence de normalisation, notre travail pourra s'avérer utile à tout développeur désirant une diversification de sa technologie ou une migration vers une autre technologie à un coût moindre.

Une autre extension de ce travail sera la définition des deux transformations verticales qui permettront de convertir un PIM écrit dans le profil UML pour EDOC [12] en PSMs spécifiques à EJB et `.Net`. Notons enfin que la contribution de la modélisation dans le processus de développement de logiciel, est toujours loin de son plein potentiel et ce par manque d'outils et de standards. On espère que des travaux futurs fourniront les métamodèles des technologies ainsi qu'un environnement de développement intégré ouvert aux métamodèles, qui permet l'édition des modèles et supporte l'automatisation de transformation de modèles.

Références

- [1] Abd-Ali, J., K. El Guemhioui. "An MDA-Oriented `.Net` Metamodel" soumis à EDOC 2005. Disponible à <http://w3.uqo.ca/karim/publications/edoc-2005.pdf>
- [2] Czarnecki, K., S. Helsen. "Classification of Model Transformation Approaches". OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture.
- [3] EJB Specs from Sun <http://java.sun.com/products/ejb/docs.html>
- [4] Lowy, Juval. *COM and .NET Component Services*. O'Reilly, 2001. 384 p.
- [5] Lowy, Juval. *Programming .NET Components*. O'Reilly, 2003. 459 p.
- [6] Metamodel and UML Profile for Java and EJB Specification. February 2004. Version 1.0, formal/04-02-02. An Adopted Specification of the Object Management Group, Inc.
- [7] Object Management Group. Request for Proposal: MOF 2.0 Query / View / Transformations RFP. OMG 2002. <http://www.omg.org/docs/ad/02-04-10.pdf>
- [8] OMG: MDA GUIDE Version 1.0.1 document number omg/2003-06-01 available at <http://www.omg.org/docs/omg/03-06-01.pdf>
- [9] OMG / MOF 2.0, Query / Views / Transformation. ad/2002-04-10, Revised Submission, Version 1.0, 2003/08/18, OpenQVT, disponible à <http://www.omg.org/docs/ad/03-08-05.pdf>
- [10] OMG :Object Management Group. www.omg.org
- [11] OMG : OCL Response to the UML 2.0 OCL RfP (ad/2000-09-03). Revised Submission, Version 1.6. January 6, 2003. OMG Document ad/2003-01-07, disponible à <http://www.omg.org/docs/ad/03-01-07.pdf>
- [12] UML Profile for EDOC. Disponible à http://www.omg.org/technology/documents/modeling_spec_catalog.htm#UML_for_EDOC
- [13] QVT Partners. QVT: The high level scope, QVT-Partners, 2003. <http://qvtp.org/downloads/qvtscope.pdf>
- [14] Sun Microsystems, Enterprise JavaBeans. <http://java.sun.com/products/ejb/>
- [15] The ATL language definition page web disponible à <http://www.sciences.univ-nantes.fr/lina/atl/atlProject/languageDefinition/>
- [16] The Microsoft Developer Network (MSDN). <http://msdn.microsoft.com/library/default.asp>