Génération de métaprogramme Java à partir des contraintes architecturales OCL

Sahar Kallel[1,2], Chouki Tibermacine[2], Mohamed Reda Skay[2], Christophe Dony[2] and Ahmed Hadj Kacem[1]

[1] ReDCAD, Université de Sfax, Tunisie
sahar.kallel@lirmm.fr
ahmed.hadjkacem@fsegs.rnu.tn
[2] Lirmm, CNRS, Université Montpellier2, France
Chouki.Tibermacine@lirmm.fr
dony@lirmm.fr
reda.skay@lirmm.fr

Résumé

Afin de formaliser leurs choix de conception, les concepteurs des logiciels spécifient des contraintes sur les modèles de leurs applications. Il s'avère important de spécifier ces contraintes non seulement dans la phase de conception mais aussi dans la phase d'implémentation en les rendant vérifiables sur le code. La spécification de ces contraintes peut se faire manuellement mais celles-ci peuvent être obtenues automatiquement avec la génération de code de leurs modèles. Plusieurs travaux et outils existent permettant la génération automatique des différents modèles en code source. Cependant, la plupart de ces outils ne considèrent pas la génération des contraintes textuelles associées à ces modèles en code. Pour ceux qui le font, la transformation est appliquée sur des contraintes fonctionnelles mais non architecturales. Dans ce contexte, nous avons développé une méthode qui permet d'écrire des contraintes d'architecture d'un modèle donné en se basant sur son métamodèle et ensuite générer du code à partir de ces contraintes sous la forme d'un métaprogramme.

1 Introduction

L'architecture est la base de la conception d'une application [5]. Elle nous donne un aperçu général de l'organisation de l'application qui nous aide à vérifier certaines propriétés, comme les attributs de qualité. Dans ce contexte, des langages ont été définis pour spécifier et interpréter ces architectures. Ces langages ont pour but de décrire l'architecture d'une application et ils ne se préoccupent pas de l'implémentation des fonctionnalités de l'application. Afin de raffiner la description des architectures, Les concepteurs peuvent poser des contraintes qui vérifient leurs applications.

Selon Lionel et al. [7], OCL [17] est un langage d'expression des contraintes connu, bien outillé et facile à apprendre et à utiliser. En effet, il est le standard de l'OMG (Object Management Group) et il a pour objectif de spécifier les contraintes fonctionnelles et architecturales en phase de conception, les rendant ainsi vérifiables sur les modèles d'une application. Les contraintes OCL sont décrites relativement à un contexte. Ce contexte est un élément du diagramme de classes, le plus souvent une classe, une opération ou une association présente dans le diagramme. Nous pouvons distinguer deux catégories de contraintes OCL, les contraintes d'architecture qui sont écrites sur un métamodèle donné et les contraintes fonctionnelles qui sont appliquées sur un contexte d'un modèle précis. Par exemple, dans un diagramme de classe, où nous retrouvons

une classe Employé, ayant comme attribut age de type entier, une contrainte fonctionnelle OCL représentant un invariant sur cette classe peut forcer les valeurs de cet attribut pour qu'ils soient toujours compris dans l'intervalle 16 à 70. Cette contrainte sera vérifiée sur toutes les instances du modèle UML, et donc sur toutes les instances de la classe Employé.

Dans ce qui suit, nous allons nous concentrer sur les contraintes architecturales. Ces contraintes sont écrites non pas sur un modèle mais sur un métamodèle. Elles permettent de spécifier de façon formelle certaines décisions de conception comme le choix d'un patron ou d'un style architectural. Nous pouvons dire que ces contraintes ne relèvent pas du niveau fonctionnel (contraintes sur les valeurs des attributs), elles relèvent plutôt du niveau structurel de ces applications, et donc architectural. La spécification de ce type de contraintes en phase d'implémentation peut se faire manuellement. Cependant, celles-ci peuvent être obtenues automatiquement en transformant les contraintes spécifiées lors de la phase de conception. En effet, dans ces deux phases de développement ces contraintes sont écrites avec deux formalismes différents, qui sont interprétables indépendamment, alors qu'elles ont la même sémantique, et qu'elles peuvent être dérivées les unes des autres. Dans l'ingénierie logicielle, la plupart des générateurs de code font des transformations des modèles en code source sans prendre en compte les contraintes qu'ils imposent. Il existe un nombre limité d'outils qui ont été proposés pour transformer les contraintes OCL écrites en code dans la phase d'implémentation mais tous sont utilisés pour des contraintes fonctionnelles et non architecturales.

L'objectif de ce papier est de traduire les contraintes architecturales décrites par le langage OCL en code source sous la forme d'un métaprogramme. Nous désignons par un métaprogramme un programme qui utilise le mécanisme d'introspection du langage de programmation (ici Java) pour implémenter une contrainte d'architecture. En utilisant ce mécanisme, les contraintes d'architecture seront évaluées dynamiquement, c'est-à-dire à l'exécution des programmes sur lesquels les contraintes sont vérifiées. Ceci n'est pas nécessaire pour certaines catégories de contraintes d'architecture dans lesquelles une analyse statique de l'architecture suffit. Dans certaines cas (patron MVC, par exemple, présenté dans la section suivante), la contrainte doit vérifier les types des instances crées à l'exécution et leurs interconnexions. Elle doit donc être évaluée dynamiquement.

Dans la suite de cet article, la section 2 sera consacrée à un exemple illustratif de notre travail, la section 3 portera sur l'approche générale que nous avons adoptée pour réaliser notre objectif. La section 4 représente les métamodèles UML et Java sur lesquels nous pouvons exprimer les contraintes architecturales et connaitre le principe de raffinement et de transformation qui seront détaillés respectivement dans les sections 5 et 6. La section 7 détaille la partie génération des contraintes en métaprogramme. Enfin, nous présenterons les travaux connexes dans la section 8 et dans la section 9 nous conclurons et discuterons les perspectives.

2 Exemple

Nous prenons l'exemple des contraintes représentant l'architecture du patron MVC (Model-View-Controller) [19]. La figure 1 illustre les dépendances entre les entités de ce patron.

Ces dépendances peuvent être traduites ainsi :

- Les classes stéréotypées *Model* ne doivent pas dépendre des classes stéréotypées *View*. Ceci permet d'avoir entre autres plusieurs vues pour un même modèle, et donc de les

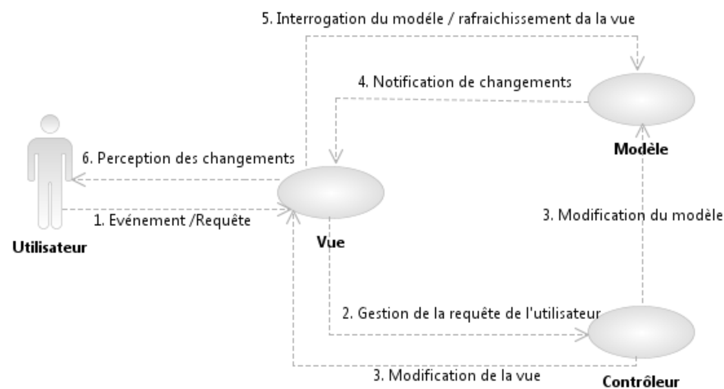

Figure 1 – Illustration du patron MVC

découpler.

- Les classes stéréotypées *Model* ne doivent pas avoir de dépendances avec les classes stéréotypées *Controller*. Ceci permet d'avoir plusieurs contrôleurs possibles pour le modèle.
- Les classes stéréotypées *View* ne doivent pas avoir de dépendances avec les classes stéréotypées *Model*. Les contrôleurs doivent relier la vue avec le modèle.

Les deux premières contraintes d'architecture peuvent être traduites en une seule contrainte OCL de la manière suivante :

```
1 context Class inv :
2 self.package.profileApplication.appliedProfile.ownedStereotype
3    -> exists(s:Stereotype | s.name='Model') implies
4 self.supplierDependency.client
5    -> forAll(t: Type | not(t.oclAsType(Class)
6       .package.profileApplication.appliedProfile.ownedStereotype
7       ->exists(s:Stereotype | s.name='View' or s.name='Controller')
8                         )
9             )
```

Listing 1 – Contrainte 1 et 2 du patron MVC en OCL

La première ligne du listing 1 déclare le contexte de la contrainte. Elle indique que la contrainte s'applique à toute classe de l'application. Les lignes 2 et 3 récupèrent l'ensemble de classes qui représente le modèle (annotées par *@Model*) en utilisant la navigation *package.profileApplication.appliedProfile.ownedStereotype*. La ligne 4 représente l'ensemble des classes ayant une dépendance directe avec notre contexte en utilisant *self.supplierDependency.client*. Le reste de la contrainte permet de parcourir ce dernier ensemble et vérifier qu'il ne comporte pas des classes annotées *View* ou *Controller*.

Idem, la troisième contrainte peut être exprimée en OCL de la façon suivante :

```
1 context Class inv :
2 self.package.profileApplication.appliedProfile.ownedStereotype
```

```
3   -> exists(s:Stereotype | s.name='View') implies
4 self.supplierDependency.client
5   -> forAll( c:Class | not(t.oclAsType(Class)
6        .package.profileApplication.appliedProfile.ownedStereotype
7          -> exists(s:Stereotype | s.name='Model')
8                           )
9           )
```

Listing 2 – Contrainte 3 du patron MVC en OCL

Notre objectif est d'obtenir un métaprogramme traduit automatiquement à partir des contraintes OCL. Les deux premières contraintes du patron MVC peuvent être exprimées en code Java de la façon suivante :

```
1 public boolean invariant(ArrayList<Class> classesMonApplication) {
2    for(Class <?> uneClasse : classesMonApplication) {
3       if(uneClasse.isAnnotationPresent(Model.class)) {
4           Field [ ] attributs = uneClasse.getDeclaredFields();
5           for(Field unAttribut : attributs) {
6               if(unAttribut.getType().isAnnotationPresent(View.class)
7               || unAttribut.getType().isAnnotationPresent(Controller.class))
8                   return false;
9          }
10      }
11   }
12   return true;
13 }
```

Listing 3 – Contrainte du patron MVC en JAVA

Le listing 3 représente un code Java qui utilise l'API Java.reflect. Nous parcourons l'ensemble des classes métiers de l'application, nous utilisons getDeclaredFields() pour collecter l'ensemble des attributs et isAnnotationPresent(..) qui vérifie si l'attribut possède une annotation particulière.

3 Approche générale

Le but de notre travail est de proposer une méthode qui permet la génération du code à partir des contraintes architecturales écrites lorsque dans les phases de conception et d'implémentation le formalisme avec lequel les contraintes sont écrites est différent. Dans le contexte de notre travail nous avons choisi UML comme le premier formalisme et Java comme le deuxième. Des contraintes écrites en OCL sur le métamodèle UML est le départ de notre travail. Le résultat voulu est un métaprogramme Java qui spécifie la contrainte de départ et utilise l'API reflect de Java. La figure 2 décrit l'approche de notre travail.

Pour mettre en place notre approche, il faut à la fois faire une projection des abstractions de niveau conception vers des abstractions de niveau implémentation, et aussi traduire la syntaxe. Faire les deux en même temps est une tâche assez lourde et fastidieuse à réaliser, d'où l'intérêt de les séparer.

La figure 2 montre bien que notre travail englobe trois étapes. Si la contrainte possède une condition de raffinement alors la première étape consiste à la raffiner tout en restant dans le formalisme UML. Sinon une étape de transformation des contraintes dite aussi projection des concepts est mise en place afin d'arriver à la fin à la partie de la traduction syntaxique c'est à dire la génération de code.

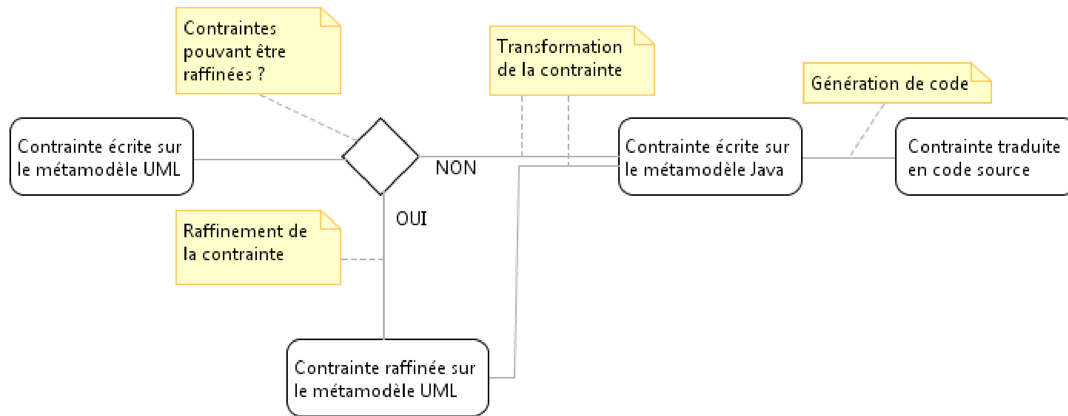

FIGURE 2 – Description du travail

4 Métamodèles

Pour effectuer le raffinement et la transformation de contraintes, nous avons besoin de décrire les métamodèles UML et Java afin de fixer par la suite les règles de transformation des contraintes d'un métamodèle à l'autre.

4.1 Métamodèle UML

La spécification UML 2.4.1 [18] décrit le métamodèle UML de façon détaillée. Cependant il existe des entités de ce métamodèle dont nous pouvons nous passer dans notre spécification de contraintes d'architecture. Ainsi nous avons décidé de prendre le métamodèle de la spécification la superstructure du langage UML, version 2.4.1 et faire en sorte de supprimer les entités dispensables dans notre travail afin de rendre les contraintes moins verbeuses. La figure 3 décrit le métamodèle simplifié d'UML.

Ce métamodèle se focalise sur la description des classes et notamment sur la description des packages, des attributs, des méthodes, des dépendances et des profils. Un package est composé par un certain nombre de *Type*. Au milieu de la figure, un *Type* peut avoir la capacité de participer dans des dépendances. A droite de la figure, il est indiqué qu'un *Type*(*Class* ou *Interface*) peut déclarer des attributs qui sont des instances de *Property* et des méthodes qui sont des instances d'*Operation*. Ces opérations peuvent avoir des paramètres et des types de retours. La partie gauche de la figure illustre que nous pouvons appliquer un *Profi*l à un *Package*, et qu'un *Profil* est constitué d'un certain nombre de stéréotypes.

4.2 Métamodèle Java

Puisque le but est de générer du code source Java, il a fallu créer un métamodèle Java. Le langage Java fournit le mécanisme d'introspection, nous nous sommes alors appuyés sur la bibliothèque Reflect [2] pour créer un métamodèle Java simplifié. Nous aurions pu définir notre métamodèle avec une autre façon (à partir de la spécification du langage Java par exemple),

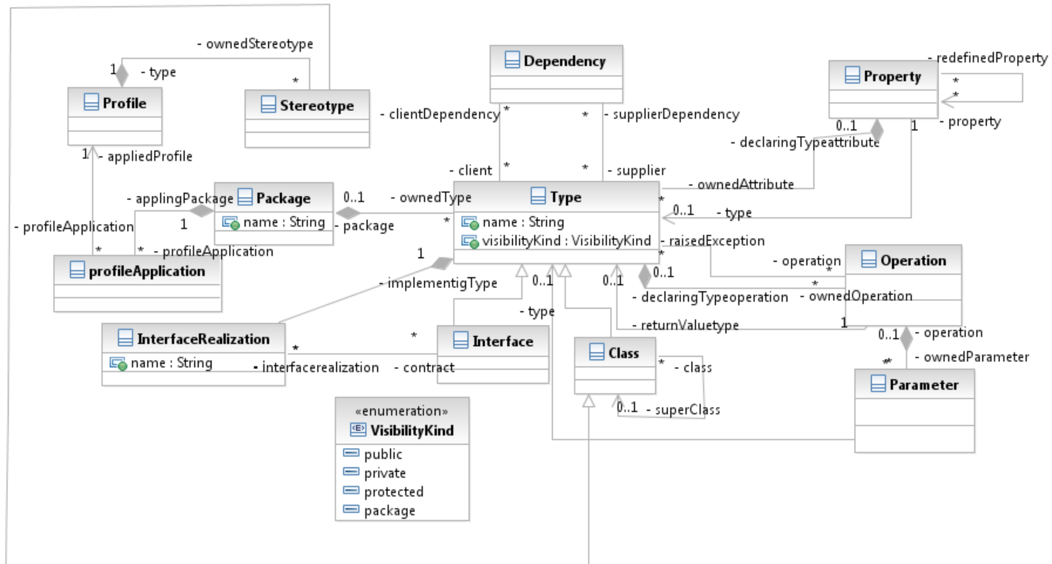

FIGURE 3 – Extrait du métamodèle UML modifié

mais nous avons volontairement choisi l'API reflect parce qu'elle nous donne accès au niveau méta du langage et aussi parce qu'elle reflète exactement ce que nous pouvons faire dans le code Java généré. Nous nous sommes contentés des entités permettant l'écriture de contraintes d'architecture. La figure 4 décrit le métamodèle Java que nous avons défini.

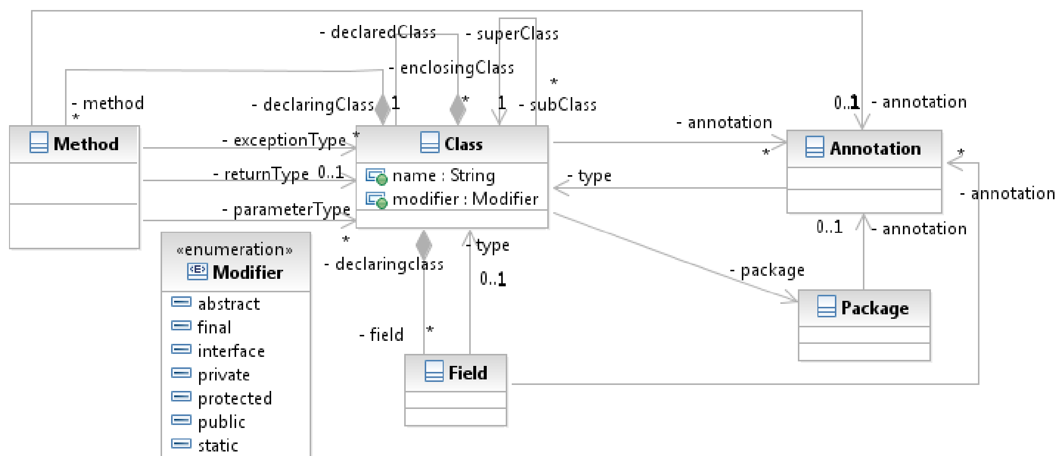

FIGURE 4 – Extrait d'un métamodèle Java

Dans la figure 4, il est indiqué qu'une *Class* peut être définie dans un *Package*. De plus, une *Class* peut contenir des attributs, qui sont des instances de la métaclasse *Field* et des méthodes qui sont des instances de la métaclasse *Method*. Nous pouvons remarquer qu'également que *Class*, *Field* et *Method* peuvent être annotés par plusieurs annotations, qui sont des instances de la métaclasse *Annotation*.

Après étudier les métaclasses UML et JAVA afin de spécifier les contraintes et leurs transformations, nous allons par la suite expliquer le principe de raffinement de la contrainte.

5 Raffinement

Dans le métamodèle UML (voir figure 3), nous pouvons récupérer tous les types (classe, interface) qui ont une dépendance à un autre type spécifique en utilisant par exemple "*supplierDependency.client*". Cette expression n'a pas d'équivalence dans Java et par conséquent, il faut raffiner la contrainte sur le métamodèle UML en exprimant les différentes catégories de dépendances pour pouvoir la transformer d'une façon claire et facile.

Souvent une dépendance entre deux classes se traduit par la déclaration dans la première classe d'au moins un attribut ayant comme type la deuxième classe. Nous pouvons également retrouver des paramètres dans les opérations de la première classe qui ont comme type la deuxième classe. Ou encore, dans une opération de la première classe, le type de retour est la deuxième classe.

Si nous reprenons notre exemple de la section 2, le listing 1 peut être raffiné de la façon suivante :

```
1 context Class inv :
2  self.package.profileApplication.appliedProfile.ownedStereotype
3  -> exists(s:Stereotype|s.name='Model') implies
4   self.ownedAttribute.type
5   -> forAll(t: Type | not(t.oclAsType(Class).package.profileApplication.
6     appliedProfile.ownedStereotype
7     ->exists(s:Stereotype | s.name='View' or s.name='Controller')))
8  and self.ownedOperation.returnValuetype
9   -> forAll(t: Type | not(t.oclAsType(Class).package.profileApplication.
10    appliedProfile.ownedStereotype
11    ->exists(s:Stereotype| s.name='View' or s.name='Controller')))
12 and self.ownedOperation.ownedParameter.type
13  -> forAll(t: Type | not(t.oclAsType(Class).package.profileApplication.
14    appliedProfile.ownedStereotype
15    ->exists(s:Stereotype | s.name='View' or s.name='Controller')))
```

Listing 4 – Contrainte raffinée en OCL

Nous obtenons au premier lieu un ensemble des attributs de la classe stéréotypée *Model* en utilisant *self.ownedAttribute* sur lequel nous vérifions par la suite le reste de la contrainte. Le même mécanisme sera effectué pour les ensembles des types qui représentent les retours (ligne 8) et les paramètres des méthodes (ligne 12).

Nous devons passer par cette étape si une contrainte représente une condition de dépendance, sinon nous appliquons le mécanisme de transformation qui sera détaillé dans la section suivante directement après l'écriture de la contrainte sur le métamodèle.

6 Transformation des contraintes

L'objectif de la transformation est de pouvoir remplacer dans une contrainte d'architecture le vocabulaire du métamodèle UML par le vocabulaire du métamodèle Java. Il a fallu alors faire la correspondance entre les termes du métamodèle UML et ceux du métamodèle Java en les répartissant en 3 catégories : les métaclasses, les rôles et les navigations.
Dans le tableau 1 nous avons présenté pour chaque métaclasse, rôle et navigation d'UML son équivalent en Java.

	UML	**Java**
Métaclasse	**Class**	**Class**
Rôle	ownedAttribute	field
	ownedOperation	method
	superClass	superClass
	nestedType	declaringClass
	interfaceRealization	interface
	package	package
Navigation	package.profileApplication	
	.appliedProfile.ownedStereotype	annotation
Métaclasse	**Property**	**Field**
Rôle	type	type
	declaringTypeattribute	declaringClass
Métaclasse	**Operation**	**Method**
Rôle	returnValuetype	returnType
	declaringTypeoperation	declaringClass
	ownedParameter	parameterType
	raisedException	exceptionType
Métaclasse	**Stereotype**	**Annotation**
Métaclasse	**Package**	**Package**

Table 1 – Correspondance UML-Java(Métaclasse, Rôle, Navigation)

Nous proposons la définition des projections d'abstractions sous la forme des *mappings* dans lequel nous désignons pour chaque entité du métamodèle UML sa correspondante dans le métamodèle Java. Nous aurions opté pour faire la correspondance manuellement au lieu d'utiliser un langage de transformation existant (Acceleo [3], Kermeta [15], ATL [14]) car les contraintes ne sont pas des modèles. En fait, nous aurions pu appliquer un mécanisme de transformation en modèles mais le problème avec ce genre de solution est la difficulté de mettre en place : nécessite de transformer le texte de la contrainte en modèle, transformer ensuite ce modèle et puis générer du texte de la nouvelle contrainte à partir de son modèle. Nous avons opté comme une solution simple est d'exploiter un compilateur OCL qui permet de générer un arbre syntaxique (AST) à partir du texte de la contrainte. Cette AST nous permet d'appliquer

facilement des différents traitements. Nous appliquons alors le tableau de correspondance (tableau 1) sur l'arbre AST généré afin d'obtenir une contrainte exprimée sur le métamodèle Java (ayant la même sémantique que la contrainte d'origine).

Si nous appliquons cette méthode de transformation sur notre exemple (Section 2), le listing 4 devient comme suit :

```
1 context Class inv :
2 self.annotation -> exists(s:Annotation|s.name='Model')
3 implies
4 self.field.type->forAll(t: Class | not(t.oclAsType(Class).annotation
5 ->exists(s:Annotation | s.name='View' or s.name='Controller')))
6 and
7 self.method.returnType->forAll(t: Class| not(t.oclAsType Class).annotation
8 ->exists(s:Stereotype| s.name='View' or s.name='Controller')))
9 and
10 self.method.parameterType->forAll(t: Class | not(t.oclAsType(Class).annotation
11->exists(s:Annotation | s.name='View' or s.name='Controller')))
```

Listing 5 – Contrainte 1 et 2 du patron MVC après la transformation

Comme il est indiqué dans le listing 5, nous avons remplacé, entre autres, *package.profileApplication. appliedProfile.ownedStereotype* par *annotation*, *ownedOperation* par *method* etc, en respectant les *mappings* présentés ci-dessus.

L'usage des *mappings* déclaratifs nous donne la possibilité lorsque les métamodèles évoluent de modifier facilement les entités changées. En plus ceci nous permet de proposer une méthode plus générique et qui ne dépend pas des métamodèles sur lesquels nous travaillons.

7 Génération de code

La génération de code consiste à traduire la contrainte transformée sur le métamodèle Java en code Java sous la forme d'un métaprogramme. Le code généré doit respecter ce qui a été exprimé dans la contrainte puisque la contrainte transformée est écrite avec l'API Reflect de Java. Cette API donne un accès au niveau méta du langage Java et offre aussi le mécanisme d'introspection. Ce dernier consiste en la découverte dynamique des informations propres à un objet et à sa description(sa classe).

Pour générer un code source à partir des contraintes architecturales exprimées sur le métamodèle Java, nous avons suivi les étapes suivantes. Premièrement, nous générons l'arbre syntaxique AST à partir de la contrainte écrite sur le métamodèle Java. Ensuite, nous effectuons un parcours hiérarchique sur cet arbre (de haut en bas et de gauche à la droite) et nous générons du code Java en se basant sur les règles ci-dessous. Il faut mentionner tout d'abord que la première règle est appliquée une seule fois dans la génération de code d'une contrainte. Les autres règles sont appliquées le long de l'analyse de la contrainte selon le type du nœud de l'AST. En fait, si c'est un rôle ou une navigation alors nous devons appliquer la règle 2. S'il s'agit d'un quantificateur, la règle 3 est alors appliquée et ainsi de suite.

1. Il faut considérer tout d'abord qu'une contrainte est représentée par une méthode Java qui retourne un booléen et qui prend en paramètre une collection d'objets. Cette collection contient des instances de la métaclasse sur laquelle la contrainte s'applique. Cette

méthode se trouve dans une classe Java et fait appel si nécessaire à d'autres méthodes implémentées lors de la génération du code.

2. Chaque rôle et navigation dans le métamodèle Java sera remplacé par la méthode accesseur dans Java Reflect (*getter*) de Java. Par exemple si nous naviguons vers *Field* nous appliquons getDeclaredFields() [1]. Si nous voulons accéder au type de retour d'une méthode nous appelons getReturnType().

3. Concernant les quantificateurs et les opérations ensemblistes, nous avons défini pour chacune un squelette de code Java. Des exemples sont représentés dans le tableau 2. La méthode select(..) présentée dans la dernière ligne du tableau 2 peut être appliquée sur des différents types OCL comme Set, Collection ou Sequence. Lors de la génération de code Java, il faut distinguer ce type OCL pour savoir implémenter son équivalent en Java (List, ArrayList ou Set..).

<table>
<tr><th>OCL</th><th>Java</th></tr>
<tr><td><code>forAll(ex:OclExpression): Boolean</code></td><td><pre>private boolean forAll(Collection c) {
    for(Iterator i = c.iterator(); c.hasNext();) {
        if(!oclExpressionInJava) return false;
    }
    return true;
}</pre></td></tr>
<tr><td><code>exists(ex:OclExpression):Boolean</code></td><td><pre>private boolean exists(Collection c) {
    for (Iterator i = c.iterator(); c.hasNext();) {
        ElementType e = (ElementType) i.next();
        if(OclExpressionInJava) return true;
    }
    return false;
}</pre></td></tr>
<tr><td><code>includes(o:Object):Boolean</code></td><td><code>contains(o:Object):Boolean</code></td></tr>
<tr><td><code>includesAll(o:Object):Boolean</code></td><td><code>containsall(o:Object):Boolean</code></td></tr>
<tr><td><code>select(ex:OclExpression):Sequence</code></td><td><pre>List result = new ...();
private Set select(Collection c) {
    for (Iterator i = c.iterator(); c.hasNext();){
        ElementType e = (ElementType) i.next();
        if (OclExprInJava) {
              result.add(e);
        }
     }
return result;
}</pre></td></tr>
</table>

TABLE 2 – Génération de code des quantificateurs et opérations ensemblistes d'OCL en Java

4. Dans chaque quantificateur ou opération ensembliste, nous parcourons de façon récursive l'expression évaluée comme s'il s'agit d'une sous contrainte et nous générons de nouveau les correspondances : si nous rencontrons un rôle ou une navigation dans le métamodèle Java nous re-appliquons la règle 2. Dans le cas ou le quantificateur est imbriqué, nous

1. Nous utilisons getDeclaredField() au lieu de getFields() pour récupérer tous les attributs (privé et publique). Pour ceux que nous héritons, nous pouvons les spécifier en utilisant superClass.

re-appliquons la règle 3.

5. Pour les opérateurs logiques (and, not ..), nous avons défini aussi des méthodes au lieu de mettre || ou && .. dans le code. Si nous rencontrons par exemple not lors de l'analyse de la contrainte nous devons appeler la méthode not qui prend un booléen et retourne sa négation. Ces méthodes sont implémentées dans une classe nommée *OpLogiques*. Si la contrainte contient un opérateur logique alors cette classe devra être une classe mère de notre classe générée dans laquelle se trouve la méthode Invariant afin de rendre notre code plus optimale.

6. Les opérations arithmétiques (>, <, =) et les types (Integer, Real, String) sont les mêmes dans le code généré de la contrainte.

7. Les opérations sur les chaînes de caractères comme *substring(), concat()* sont changées par les méthodes Java équivalents.

Pour mieux comprendre le déroulement de la génération de code, le tableau 3 illustre un exemple de génération de code d'une contrainte OCL exprimée dans le métamodèle Java. Cette contrainte vérifie la contrainte 3 du patron MVC présenté dans la section 2. Pour simplifier nous raffinons ici la dépendance entre deux classes par la déclaration dans la première classe d'au moins un retour de méthodes ayant comme type la deuxième classe.

<table>
<tr><th>Contrainte</th><th>Métaprogramme Java</th><th>Explication</th></tr>
<tr><td>context Class inv :</td><td>Boolean result=true;
public Boolean Invariant
   (Class[] listClass){
  for(Class c:listClass){
   //code
   }
   //code
  return result;
}</td><td>La méthode Invariant se trouve dans une classe Java. Si nous obtenons le contexte de la contrainte, nous pouvons préparer notre classe Java et la méthode Invariant comme il est indiqué dans la colonne métaprogramme Java.</td></tr>
<tr><td>self.annotation</td><td>Annotation[] annotations
  = c.getAnnotations();</td><td></td></tr>
<tr><td>->exists
   (a:Annotation|
    a.name='Model'
)</td><td>Boolean resultexists1
  = exists1(annotations);</td><td>exists1(..) est une méthode qui a comme corps le squelette de code proposée dans le tableau 2. Nous suivons une démarche récursive pour implémenter le corps de la méthode exists1().</td></tr>
<tr><td>implies</td><td>if(resultexists1){
}</td><td></td></tr>
<tr><td>self.method</td><td>Method[] methods
 =c.getDeclaredMethods();</td><td></td></tr>
</table>

<table>
<tr>
<td>->
forAll(m:method|not
(
m.returnType.annotation
->exists(a:Annotation|
a.name='View')
))</td>
<td>Boolean resultforAll1
= forAll1(methods);</td>
<td>Nous appliquons la récursivité pour implémenter la méthode forAll1(). forAll et exists sont imbriqués. forAll1() appelle la méthode exists2()

public class Contrainte
extends OpLogiques{
//...
public Boolean Invariant(Class[]
listClass) {
for(Class c: listClass){
Annotation [] annotations
= c.getAnnotations();
resultexists1
= exists1(annotations);
if(resultexists1){
Method [] methods
=c.getDeclaredMethods();
Boolean resultforAll1
=forAll1(methods);
}
}
//..
}
private Boolean exists1(..){..}
private Boolean forAll1(Method[]
methods){
for(Method m : methods){
Type type = m.getReturnType();
Annotation[] annotations
= type.getAnnotations();
resultexist2
= not(exists2(annotations));
if(!resultexist2)
return false;
}
return true;
}
private Boolean exists2(..){..}
}</td>
</tr>
</table>

TABLE 3 – Exemple de génération de code de la contrainte 3 du patron MVC

La troisième colonne du tableau 3 explique la constitution progressive du code Java de la contrainte. Ce code utilise le mécanisme d'introspection. Comme nous voyons, ce code est différent syntaxiquement au code optimal voulu (voir listing 3) mais ils ont la même sémantique. Bien sûr, la traduction automatique n'est pas optimale en termes de complexité mais pour les développeurs elle est meilleure que la traduction manuelle pour vérifier dynamiquement leurs choix de conception sans implémenter un code externe au code de l'application.

8 Travaux connexes

Notre travail repose sur deux grandes parties : la transformation des contraintes OCL d'un métamodèle à un autre et la génération de code à partir de ces contraintes.

Hassam et al. [11] ont proposé une méthode de transformation de contraintes OCL lors le refactoring du modèle UML en utilisant la transformation des modèles. leur approche consiste à préparer un ensemble ordonné des opérations pour le refactoring des contraintes OCL après le refactoring du modèle UML et pour cela ils doivent premièrement utiliser une méthode d'annotation du modèle UML source pour obtenir un modèle cible aussi annoté en utilisant une transformation de modèles, par la suite ils prennent les deux annotations pour former une table de *mapping* qui sera utilisée finalement avec Roclet [13] pour transformer les contraintes OCL du modèle source à des contraintes OCL conforme au modèle cible obtenu. Nous trouvons que notre idée d'utiliser un compilateur OCL existant est plus simple. Leur solution de transformation des contraintes est difficile à mettre en place comme nous avons mentionné dans la section 6 et nécessite des connaissances sur les outils et les langages de transformation de modèles. D'ailleurs Roclet avec lequel ils font la transformation utilise des résultats de Dresden OCL comme analyseur des contraintes [13].

Cabot et al. [8] ont travaillé sur la transformation du modèle uml/OCL en CSP afin de vérifier des propriétés de qualité sur le modèle. Cette approche consiste en premier lieu de transformer le modèle en CSP (Constraint Satisfaction Problem), les propriétés à vérifier sont translatées comme des contraintes à CSP et par la suite utiliser ECLiPSe [4] pour résoudre le CSP. le résultat est une instance du diagramme de classe du modèle utilisé qui approuve ou désapprouve les propriétés à vérifier. La première partie de cette approche est similaire à notre transformation du système parce que la transformation en CSP se fait à l'aide de l'outil Dresden OCL comme un compilateur OCL pour pouvoir analyser les contraintes à transformer et appliquer les règles qui conviennent. Cependant, nous avons implémenté en plus de la transformation une méthode pour générer du code source interprétable directement sur le code de l'application et dans le même environnement d'exécution sans besoin d'utiliser un outil d'interprétation des contraintes sur l'application.

Les auteurs dans [10, 20] ont proposé une méthode pour simplifier les contraintes OCL. En fait, ils sont contentés seulement par la simplification des contraintes qui peut être une sous étape de la transformation des contraintes.

Dans la littérature, des outils comme Eclipse OCL [1] et Dresden OCL [9] ont contribué dans le projet de traduction des contraintes OCL en Java mais les contraintes sont des contraintes fonctionnelles et non architecturales. Le code généré par Dresden OCL [9, 12, 16] est difficilement compréhensible. En fait, Il est vrai que Dresden OCL est le premier outil développé dans ce domaine mais il utilise du vocabulaire proposé uniquement dans l'API Dresden OCL. Ce code est normalement adressé aux développeurs qui ont déjà vu Dresden OCL contrairement à notre objectif qui est de s'adresser à tout développeur voulant transformer des contraintes et ayant des connaissances en Java mais pas obligatoirement connaissant l'API Dresden OCL. En plus, il faut obligatoirement créer au préalable les classes du modèle pour pouvoir générer la contrainte. Nous remarquons ainsi que cet outil est pratique uniquement pour les contraintes fonctionnelles et non pas pour les contraintes architecturales. Ces raisons expliquent pourquoi nous avons choisi de ne pas travailler avec le générateur de code proposé par Dresden OCL.

Un autre outil existe est intitulé OCL2j [6] permettant la génération des contraintes fonctionnelles en Java en utilisant AOP (Aspect oriented Programming). Cet outil ne supporte pas la transformation des contraintes d'architecture et n'utilise pas une librairie standard pour Java comme l'API Java.Reflect. L'indisponibilité de cet outil restreint son examination.

9 Conclusion et perspectives

Nous avons présenté dans cet article une méthodologie adoptée par notre prototype pour automatiquement transformer des contraintes d'architecture OCL en métaprogramme Java en utilisant l'introspection fourni par ce langage. Notre méthode consiste tout d'abord à écrire les contraintes sur un métamodèle UML, les raffiner si nécessaire, les transformer par la suite en contraintes exprimées sur un métamodèle Java et enfin générer du code source Java en se basant sur des règles d'implémentation.

Nous pouvons énoncer comme perspectives autour de ce concept de contraintes d'architecture : la mise en œuvre de ces contraintes dans un environnement de développement intégré. Nous pouvons encore penser à un niveau d'abstraction plus élevé de façon que la spécification de contraintes devient indépendante d'un paradigme donné en utilisant un métamodèle de graphes (une description d'architecture étant considérée comme un graphe avec des nœuds et arrêtes), puis leur transformation vers les différents paradigmes.

Références

[1] Eclipse ocl. `http://www.eclipse.org/modeling/mdt/?project=ocl`.

[2] Java reflect. `http://docs.oracle.com/javase/7/docs/api/java/lang/reflect/packagesummary.html`.

[3] Acceleo. Implementation of mof to text language. `http://www.omg.org/news/meetings/tc/mn/specialevents/ecl/Juliot-Acceleo.pdf`.

[4] Krzysztof R Apt and Mark Wallace. *Constraint logic programming using ECLiPSe*. Cambridge University Press, 2007.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, 2012.

[6] Lionel C Briand, Wojciech Dzidek, and Yvan Labiche. Using aspect-oriented programming to instrument ocl contracts in java. *Technischer Report, Carlton University, Kanada*, 2004.

[7] Lionel C. Briand, Yvan Labiche, Massimiliano Di Penta, and Han (Daphne) Yan-Bondoc. An experimental investigation of formality in uml-based development, 2005.

[8] Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp : a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548. ACM, 2007.

[9] Birgit Demuth. The dresden ocl toolkit and its role in information systems development. In *Proc. of the 13th International Conference on Information Systems Development (ISD'2004)*, 2004.

[10] Martin Giese and Daniel Larsson. Simplifying transformations of ocl constraints. In *Model Driven Engineering Languages and Systems*, pages 309–323. Springer, 2005.

[11] Kahina Hassam, Salah Sadou, Régis Fleurquin, et al. Adapting ocl constraints after a refactoring of their model using an mde process. In *BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, pages 16–27, 2010.

[12] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting ocl. In *UML 2000—The Unified Modeling Language*, pages 278–293. Springer, 2000.

[13] Cédric Jeanneret, Leander Eyer, S Markovic, and Thomas Baar. Roclet : Refactoring ocl expressions by transformations. In *Software & Systems Engineering and their Applications, 19th International Conference, ICSSEA*, volume 2006, 2006.

[14] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[15] Kermeta. Kermeta. `http://www.kermeta.org`.

[16] LCI. Object constraint language environnement. `http://lci.cs.ubbcluj.ro/ocle/`.

[17] OMG. Object constraint language, version 2.3.1, document formal/2012-01-01. `http://www.omg.org/spec/OCL/2.3.1/PDF/`.

[18] OMG. Unified modeling language superstructure, version 2.4.1, specification document formal/2011-08-06. `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/`.

[19] T Reenskaug. Thing-model-view editor an example from a planning system, xerox parc technical note (may 1979).

[20] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Simplifying transformation of software architecture constraints. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1240–1244, New York, NY, USA, 2006. ACM.