Transformation de Modèles

Introduction à ATL

Benoît Combemale, Xavier Crégut, Marc Pantel

† IRISA CNRS Laboratory, *University of Rennes 1*
benoit.combemale@irisa.fr

‡ IRIT CNRS Laboratory, *University of Toulouse*
{xavier.cregut, marc.pantel}@enseeiht.fr

dernière mise à jour le 24 octobre 2010

Support disponible à l'adresse (*teaching part*) :
http://perso.univ-rennes1.fr/benoit.combemale/

Sommaire

Sommaire

Origines d'ATL

- ATL (ATLAS Transformation Language) est le langage de transformation développé dans le cadre du projet ATLAS.
- ATL est développé au LINA à Nantes par l'équipe de Jean Bézivin.
- Fait partie du projet Eclipse M2M (Model-to-Model) : http://www.eclipse.org/m2m/
- ATL se compose :
  - d'un langage de transformation ;
  - d'un compilateur et d'une machine virtuelle ;
  - d'un IDE s'appuyant sur Eclipse
- Pages principales : http://www.sciences.univ-nantes.fr/lina/atl/ et http://www.eclipse.org/m2m/atl/
- Manuel utilisateur et autres documentations accessibles sur http://www.eclipse.org/m2m/atl/doc/

Sommaire

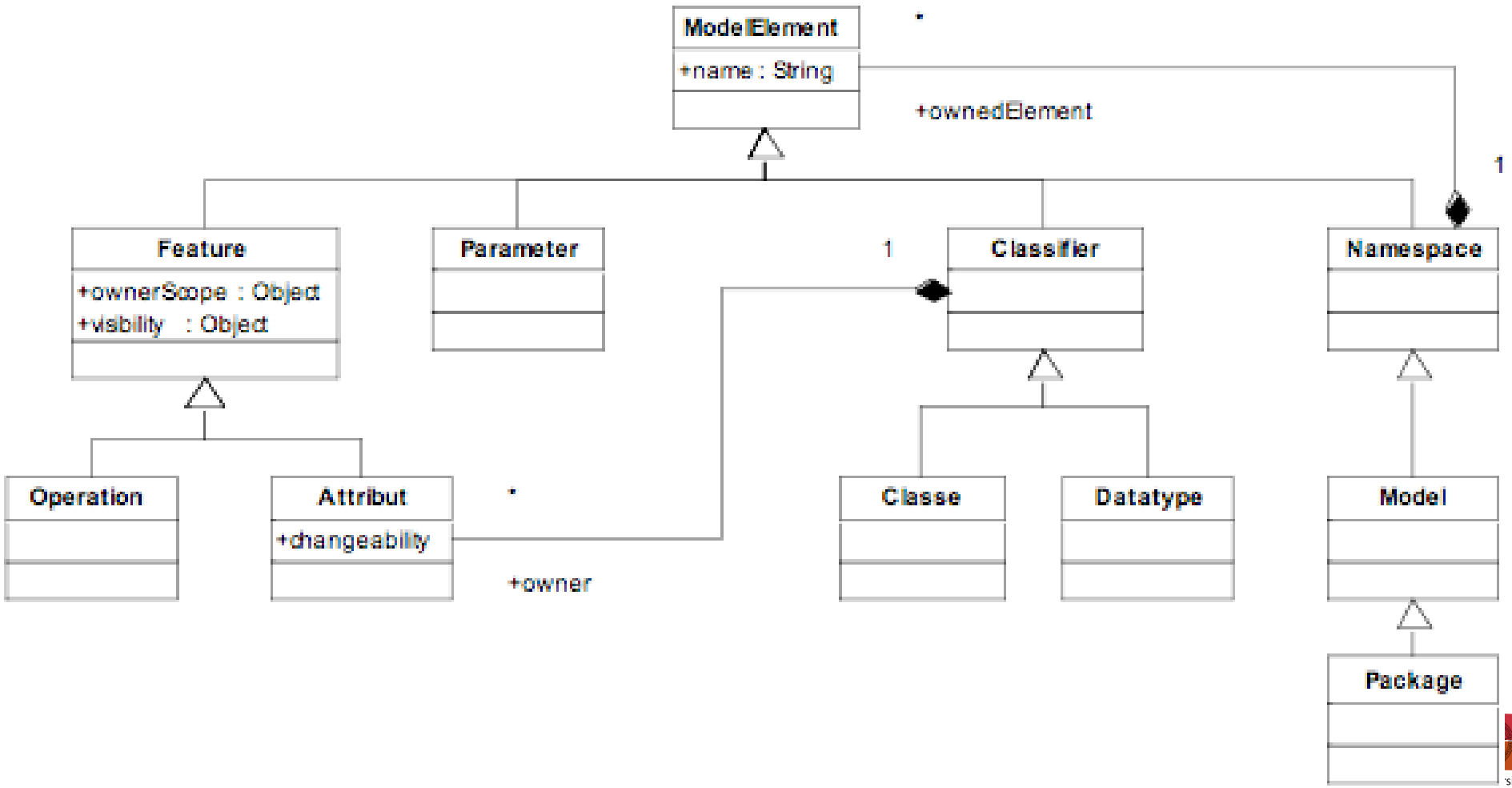
Transformation UML vers Java

Transformer notre modèle UML vers notre modèle Java consiste à :

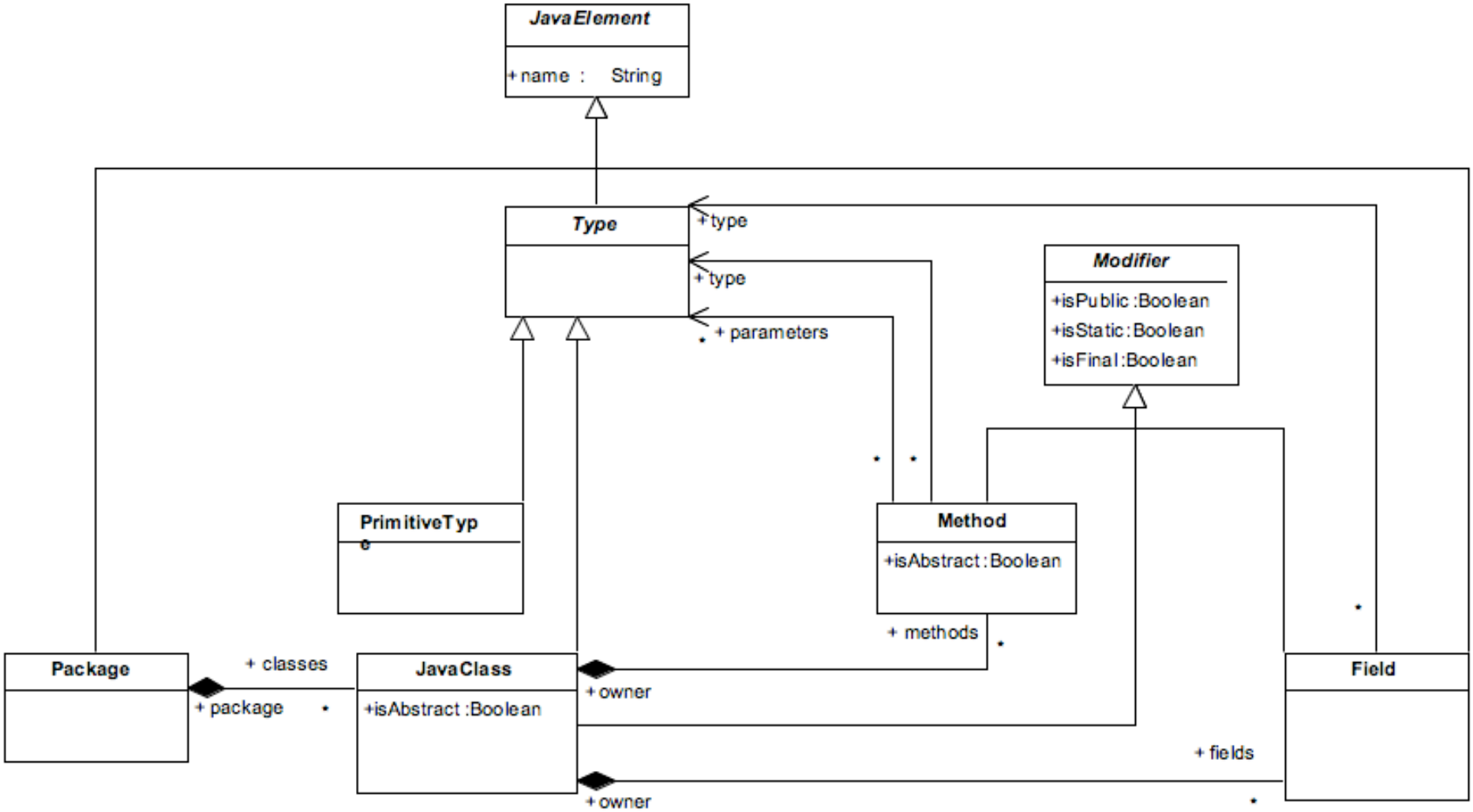
- Chaque paquetage UML donne un paquetage Java :
  - les noms sont les mêmes mais en Java, le nom est complètement qualifié.
- Chaque classe UML donne une classe Java :
  - de même nom ;
  - dans le paquetage correspondant ;
  - avec les mêmes *modifiers*.
- Chaque DataType UML donne un type primitif correspondant en Java
  - de même nom ;
  - dans le paquetage correspondant ;
- Chaque attribut UML donne un attribut Java respectant le nom, le type, la classe d'appartenance et les *modifiers*.
- Idem pour les opérations.

Voir http://www.eclipse.org/m2m/atl/atlTransformations/#UML2Java.

Méta-modèle UML (diagramme de classe simplifié)



Méta-modèle Java (simplifié)



Exemple de transformation d'un modèle vers du texte

- Exemple 1 : Transformer le modèle Java vers du code Java.
- Exemple 2 : Contrôler la cohérence du modèle UML en écrivant une transformation qui affiche un diagnostic :
  - Tous les éléments d'un Namespace doivent avoir des noms différents.
  - Un Namespace ne peut contenir que des Classifier et des Namespace.

Remarque : Dans ce dernier cas, il serait possible d'aller vers un modèle d'erreur plutôt que simplement vers du texte !
Génération vers un modèle d'erreur puis affichage du modèle d'erreur.

Sommaire

Entête d'une transformation modèle à modèle

module UML2JAVA;
create OUT : JAVA **from** IN : UML;

- La convention veut qu'une transformation d'un modèle à un autre, soit nommée d'après les méta-modèles avec un 2 (to) ajouté entre les deux !
- OUT et IN sont les noms données aux modèles. Ils ne sont pas utilisés par la suite (sauf dans le cas d'une utilisation avec MDR) !
- Les modèles sources et cibles sont typés, ici UML et JAVA.
  Ils doivent donc être conformes au méta-modèle définissant leur type.

Helpers : méthodes auxiliaires

```
helper context UML!ModelElement def: isPublic(): Boolean =
        self.visibility = #vk_public;

helper context UML!Feature def: isStatic(): Boolean =
        self.ownerScope = #sk_static;

helper context UML!Attribute def: isFinal(): Boolean =
        self.changeability = #ck_frozen;

helper context UML!Namespace def: getExtendedName(): String =
        if self.namespace.oclIsUndefined() then
                ''
        else if self.namespace.oclIsKindOf(UML!Model) then
                ''
        else
                self.namespace.getExtendedName() + '.'
        endif endif + self.name;
```

Helpers : définition

- Un helper est l'équivalent d'une méthode auxiliaire ;
- Il est défini sur un contexte et pourra être appliqué sur toute expression ayant pour type ce contexte (comme une méthode dans une classe en Java)
- Le contexte peut être celui du module :

  helper def : carre(x: **Real**): **Real** = x * x;

- Un helper peut prendre des paramètres et possède un type de retour
- Le code d'un helper est une expression OCL.
- **Remarque :** Il existe des helpers sans paramètre (et donc sans les parenthèses), appelés *attribut helper*.

Matched rule : règle déclenchée sur un élément du modèle

```
rule P2P {
        from e: UML!Package (e. oclIsTypeOf(UML!Package))
        to out: JAVA!Package (
                name <- e.getExtendedName()
        )
}
```

Une règle est caractérisée par deux éléments obligatoires :

- un motif sur le modèle source (**from**) avec une éventuelle contrainte ;
- un ou plusieurs motifs sur le modèle cible (**to**) qui expliquent comment les éléments cibles sont initialisés à partir des éléments sources correspondant.

Une règles peut aussi définir :

- une contrainte sur les éléments correspondant au motif source ;
- une partie impérative
- des variables locales

Matched rule : lien entre éléments cibles et sources

```
rule C2C {
        from e: UML!Class
        to out: JAVA!JavaClass (
                 name <- e.name,
                isAbstract <- e.isAbstract,
                isPublic <- e.isPublic(),
                package <- e.namespace
        )
}
```

- Lors de la création d'un élément cible à partir d'un élément source, ATL conserve un lien de traçabilité entre les deux.
- Ce lien est utilisé pour initialiser un élément cible dans la partie **to**.
- le package d'une JavaClass est initialisé avec l'élément du modèle cible construit pour l'élément e.namespace.
  $\Longrightarrow$ Il doit y avoir une *match rule* qui porte sur UML!Package et qui crée un JAVA!Package

Matched rule : avec condition sur l'élément filtré

Supposons que l'on veut traduire différemment un attribut UML suivant qu'il est déclaré public ou non. On peut alors écrire deux règles.

```
rule A2F {
        from e : UML!Attribute ( not e.isPublic() )
        to out : JAVA!Field (
                  name <- e.name,
                  isStatic <- e.isStatic(),
                  isPublic <- e.isPublic(),
                  isFinal <- e.isFinal(),
                  owner <- e.owner,
                  type <- e.type
        )
}
```

Attention : Pour un même élément source, on ne peut avoir au plus que deux règles, l'une filtrée sur *condition* et l'autre sur **not** *condition*.

Matched rule : avec condition sur l'élément filtré (suite)

```
rule A2F {
  from e : UML!Attribute
           ( e.isPublic() )
  to out : JAVA!Field (
     name <- e.name,
     isStatic <- e.isStatic(),
     isPublic <- e.false,
     isFinal <- e.isFinal(),
    owner <- e.owner,
    type <- e.type
  ),
  accesseur : JAVA!Method (
     name <- 'get'
       + e.name.toCapitalize(),
     isStatic <- e.isStatic(),
     isPublic <- true,
    owner <- e.owner,
    parameters <- Sequence{}
  )
  modifieur : JAVA!Method(...)
}
```

- Si l'attribut est déclaré public en UML, il est transformé en attribut privé en Java avec un accesseur et un modifieur.

$\Rightarrow$ Cette règle crée donc plusieurs éléments cibles.

Matched rule : partie impérative (do)

```
rule C2C {
        from e: UML!Class
        to out: JAVA!JavaClass (
                 name <- e.name,
                isAbstract <- e.isAbstract,
                isPublic <- e.isPublic(),
        )
        do {
                package <- e.namespace
        }
}
```

- la clause do est optionnelle ;
- do contient des instructions (partie impérative d'une règle) ;
- ces instructions sont exécutées après l'initialisation des éléments cibles.

Matched rule : variables locales (using)

```
from
   c : GeometricElement!Circle
using {
   pi : Real = 3.14;
   area : Real = pi * c.radius.square();
}
```

- la clause using est optionnelle ;
- elle permet de déclarer des variables locales à la règle ;
- les variables peuvent utilisées dans les clauses using, **to** et do ;
- une variable est caractérisée par son nom, son type et doit être initialisée avec une expression OCL.

Called rules

```
rule newPackage(qualifiedName: String) {
    to
        p: JAVA!Package (
            name <- qualifiedName
        )
}
```

- équivalent d'un helper qui peut créer des éléments dans le modèle cible
- doit être appelée depuis une *matched rule* ou une autre *called rule*
- ne peut pas avoir de partie **from**
- peut avoir des paramètres

Exécution d'un module ATL

L'exécution d'un module se fait en trois étapes :

1. initialisation du module
   - initialisation des attributs définis dans le contexte du module ;
2. mise en correspondance des éléments sources des *matched rules* :
   - quand une règle correspond à un élément du modèle source, les éléments cibles correspondants sont créés (mais pas encore initialisés)
3. initialisation des éléments du modèle cible.
   Le code impératif des règles (do) est exécuté après l'initialisation de la règle correspondante. Il peut appeler des *called rules*.

Sommaire

Requête (Query)

```
query JAVA2String_query = JAVA!JavaClass.allInstances()->
    select(e | e.oclIsTypeOf(JAVA!JavaClass))->
        collect(x | x.toString().writeTo('/tmp/'
                + x.package.name.replaceAll('.', '/')
                + '/' + x.name + '.java'));
...
```

- une requête (query) est une tranformation d'un modèle vers un type primitif
- Exemple classique : construire une chaîne de caractères.
- Une requête a :
  - un nom ;
  - un type ;
  - une expression ATL qui calcule la valeur de la requête.
- Comme un module, une requête peut définir des helpers et des attributs.

Sommaire

Bibliothèques (libraries)

```
library JAVA2String;
-- definition of helpers
...
```

- Une bibliothèque permet de définir des helpers qui pourront être (ré)utilisés dans dans des modules, requêtes ou autres bibliothèques (clause uses).
- Tout helper doit être attaché à un contexte car pas de contexte par défaut.
- Une bibliothèque peut utiliser une autre bibliothèque (clause uses)

```
query JAVA2String_query = JAVA!JavaClass.allInstances()->
    select(e | e.oclIsTypeOf(JAVA!JavaClass))->
        collect(x | x.toString().writeTo('/tmp/'
                + x.package.name.replaceAll('.', '/')
                + '/' + x.name + '.java'));
uses JAVA2String;
```

Sommaire

Types primitifs

- **OclAny** décrit le type le plus général
- Ses opérations sont :
  - comparaisons : = (égalité) <> (différence)
  - oclIsUndefined () : *self* est-il défini ?
  - **oclIsKindOf**(t: oclType) : *self* est-il une instance de t ou d'un de ses sous-types (équivalent instanceof de Java) ?
  - **oclIsTypeOf**(t: oclType) : *selft* est-il une instance de t ?
  - **Remarque : oclIsNew()** et **oclAsType()** ne sont pas définies en ATL
  - Autres opérations :
    - toString
    - oclType() : Le type de *self*
    - output(s : String) : affiche s sur la console Eclipse
    - debug(s : String) : affiche s + ' : ' + self.toString() dans la console

Types primitifs

- Boolean : **true** et **false**
  - opérateurs : **and**, **or**, **not**, **xor**, implies(b : Boolean) ($\equiv$ self $\Rightarrow$ b)
- Number : Integer (0, 10, -412) ou Real (3.14)
  - binaire : $*$ $+$ $-$ $/$ **div**(), **max**() , **min**()
  - unaire : **abs**()
  - Integer : **mod**() ;
  - Real : **floor**(), **round**()
  - cos(), sin(), tan(), acos(), asin(), toDegrees(), toRadians(), exp(), log(), sqrt()

Types primitifs

- String : 'bonjour', 'aujourd\'hui'
  - les caractères sont numérotés de 1 à size()
  - opérations : **size**(), **concat**(s : String) (ou +), **substring**(lower : Integer, upper : Integer), toInteger(), toReal()
  - **toUpper**(), **toLower**(), toSequence() (of char), trim(), startsWith(s : String), indexOf(s : String), lastIndexOf(s : String), split(regexp : String), replaceAll(c1 : String, c2 : String), regexReplaceAll(c1 : String, c2 : String)
  - writeTo(fileName : String)
  - println() : écrire la chaîne dans la console d'Eclipse

Collections

- Quatre types de collection :
  - **Set** : sans ordre, ni double
  - OrderedSet : avec ordre, sans double
  - **Bag** : sans ordre, doubles possibles
  - **Sequence** : avec ordre, doubles possibles
- Les collections sont génériques :
  - **Sequence**(**Integer**) : déclarer une séquence d'entiers
  - **Sequence**$\{1,\ 2,\ 3\}$: Sequence d'entiers contenant $\{1,\ 2,\ 3\}$
  - **Set**(**Sequence**(**String**)) : un ensemble de séquences de String

Collections

- Opérations sur les collections :
  - **size**()
  - **includes**(o : oclAny), **excludes**(o : oclAny)
  - **count**(o : oclAny)
  - **includesAll**(c : Collections), excludesAll(c : Collections)
  - **isEmpty**(), **notEmpty**()
  - **sum**() : pour les types qui définissent l'opérateur +
  - asSequence(), asSet(), asBag() : obtenir une collection du type précisé
- pour les opérations spécifiques d'un type de collection, voir ATL User Guide

Collections

Itérer sur les collections

source−>operation_name(iterators | body)

- source : collection itérées ;
- iterators : variables qui prennent leur valeur dans source
- body : l'expression fournie à l'opération d'itération
- operation_name : l'opération d'itération utilisée

exists	body vraie pour au moins un élément de source
forAll	body vraie pour tous les éléments de source
isUnique	body a une valeur différente pour chaque élément de source
any	un élément de source qui satisfait body (OclUndefined sinon)
one	un seul élément de source satisfait body
collect	collection des éléments résultant de l'application de body sur chaque élément de source
select	collection des éléments de source satisfaisant body
reject	collection des éléments de source NE satisfaisant PAS body
sortedBy	collection d'origine ordonnée suivant la valeur de body. Les éléments doivent posséder l'opérateur $<$.

Autres types

- Le type **énumération** doit être défini dans les méta-modèles.
  La notation ATL pour accéder à une valeur du type est #female au lieu de la forme OCL qui est Gender::female
- Tuple
  - Un tuple définit un produit cartésien (un enregistrement) sous la forme de plusieurs couples (nom, type).
  - Déclaration d'un Tuple

    Tuple{a: MMAuthor!Author, title: **String**, editor : **String**}

  - Instanciation d'un tuple (les deux sont équivalentes) :

    Tuple{ editor : **String** = 'ATL Manual', a: MMAuthor!Author = anAuthor,
    editor : **String** = 'ATL Eds.'}
    Tuple{a = anAutohor, editor = 'ATL Manual', editor = 'ATL Eds.'}

Autres types
Map

- **Map**(type_clé, type_élément) : définit un tableau associatif muni des opérations :
  - *get(clé : oclAny)* : la valeur associées à la clé (sinon OclUndefined)
  - *including(clé : oclAny, val : oclAny)* : copie de *self* avec le couple (clé, cal) ajouté
  - *union(m : Map)* : l'union de *self* et m
  - *getKeys()* : l'ensemble des clés de *self*
  - *getValues()* : le sac (bag) des valeurs de *self*
- Ne fait pas partie de la spécification d'OCL

Types issus des méta-modèles cibles et sources

- Tout type défini dans le méta-modèle source ou cible est aussi un type
- **Notation :** metamodel !class
  Exemples : JAVA !Class, UML !Package...
- Un tel type a des caractéristiques (attributs ou références) accessibles avec la notation pointée : self.name, self.package, etc.
- oclIsUndefined () : permet pour une caractéristique de multiplicité [0..1] de savoir si elle n'est pas définie.
  Ne marche pas pour multiplicité > 1 car représentée par une collection.
- **allInstances** () : obtenir toutes les instances de la méta-classe self

Expressions d'ATL déclaratif (issues d'OCL)

- Expression **if**

```
if condition then
    exp1
else
    exp2
endif
```

- Expression **let**

```
let var : Type = init in exp
```

```
let x: Real =
    if aNumber > 0 then
        aNumber.sqrt()
    else
        aNumber.square()
    endif
in let y: Real = 2 in X/y
```

Code impératif d'ATL

- **Affectation**

  target <− expr;

- instruction **if**

```
if (condition) {
    instructions
}
```

```
if (condition) {
    instructions 1
} else {
    instructions 2
}
```

- instruction **for**

```
for ( iterator in collection ) {
    instructions
}
```

Sommaire

ATL 2006 - Dernière version du compilateur

- Compilateur plus rapide, plus simple, entièrement bootstrappé
  - $\Rightarrow$ pris en compte en positionnant sur la première ligne du module :

    `-- @atlcompiler atl2006`

- Possibilité d'avoir plusieurs itérateurs dans *iterate*, *exists*, et *forAll*,
- Amélioration de la partie déclarative :
  - héritage de règle, règle abstraite,
  - plusieurs éléments de filtrage,
- Appel de *super helpers* (à la Java),
- Amélioration de la partie impérative : **endpoint** *called rules*

Héritage de règles

- Syntaxe :

```
abstract rule R1 {
-- ...
}
rule R2 extends R1 {
-- ...
}
```

- Quand l'utiliser ?
  - pour factoriser une partie commune à plusieurs règles,
  - pour spécifier le nom d'éléments d'entrés et de sorties à un ensemble de règles.
- Comment l'utiliser ?
  - une règle fille doit s'appliquer à un sous ensemble des éléments de sa règle parente,
  - une règle fille spécialise les éléments de sortie de la règle parente,
  - une seule règle fille peut s'appliquer sur un élément.

Héritage de règles - Sémantique d'exécution

1. sélection des règles principales (i.e. sans règle parent),
2. pour chaque élément pris en compte par la règle, test de la garde de chaque sous-règle,
3. sélection, si il y en a une, de celle qui correspond,
4. si aucune ne correspond, sélection, si il y en a une, de la règle par défaut [1].
5. si la règle sélectionnée n'est pas une feuille (i.e. si elle a des règles filles), on recommence en 2.
6. création des éléments cible en utilisant les types les plus spécifiques.

1. i.e. règle fille sans garde ou règle parente si elle n'est pas abstraite