

Ingénierie Dirigée par les Modèles

Méta-modélisation avec ECore et Transformation de Modèles en ATL

Marc Pantel, Xavier Crégut, Benoît Combemale, Arnaud Dieumegard

IRIT-ENSEEIH
2, rue Charles Camichel - BP 7122
F-31071 Toulouse Cedex 7
{prenom.nom}@enseeiht.fr

Septembre 2018

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Conclusion

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Modèles ?

Exemple

Intérêt des modèles

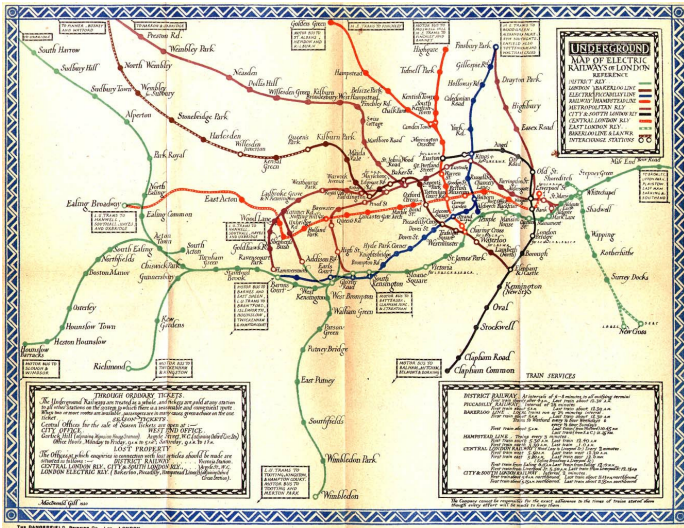
Exemple fil rouge : les processus

Rôle d'un modèle

- ▶ On utilise des modèles pour mieux comprendre un système.
Pour un observateur A, M est un modèle de l'objet O, si M aide A à répondre aux questions qu'il se pose sur O. (Minsky)
- ▶ Un modèle est une simplification, une abstraction du système.
- ▶ Exemples :
 - ▶ une carte routière
 - ▶ une partition de musique
 - ▶ un plan d'architecte
 - ▶ un diagramme UML
 - ▶ ...
- ▶ Un modèle permet :
 - ▶ de comprendre,
 - ▶ de communiquer,
 - ▶ de construire

Exemple : plan géographique du métro de Londres

Version de 1921 (<http://www.clarksbury.com/cdl/maps.html>)



Exemple : plan schématique du métro de Londres

Version schématique — Harry Beck — de 1938 (<http://www.clarksbury.com/cdl/maps.html>)



Pourquoi modéliser ?

- ▶ Mieux comprendre les systèmes complexes
- ▶ Séparation des préoccupations/aspects
- ▶ Abstraction des plateformes :
 - ▶ Architecture matérielle, Réseau
 - ▶ Architecture logicielle, Système d'exploitation
 - ▶ Langages
- ▶ Abstraction des domaines applicatifs
- ▶ Réutilisation
- ▶ Formalisation

Pourquoi de nombreux modèles ?

- ▶ Le long du cycle de vie :
 - ▶ Analyse des besoins (indépendant solution)
 - ▶ Architecture, Conception détaillée (indépendant plateforme)
 - ▶ Réalisation, Déploiement (dépendant plateforme)
- ▶ Différentes étapes de raffinement dans une même phase
- ▶ Séparation des préoccupations
 - ▶ Nombreux domaines applicatifs
 - ▶ Nombreuses plateformes (matériel, logiciel, technologique)
 - ▶ Nombreuses contraintes (service et qualité de service)

File rouge : les processus

Exemple de processus : Une version simplifiée du processus de développement du logiciel est composée de quatre activités : concevoir, programmer, tester et documenter. Programmer ne peut commencer que quand la conception est terminée. Le test peut démarrer dès que la conception est commencée. Documenter ne peut commencer que quand la programmation est commencée et ne peut s'achever que si la programmation est terminée. Le test ne peut être terminé que si la conception et la programmation sont terminées.

Exercices :

1. Représenter de manière synthétique ce processus ?
2. Quels sont les constituants d'un processus ?
3. Comment faire pour définir un nouveau processus ? Le modifier ?
4. À quoi peut servir la description d'un processus ?
5. Est-ce que le processus donné peut se terminer ? Et un processus en général ?

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Méta-modélisation

Model-Driven Engineering (MDE)

Langage métier

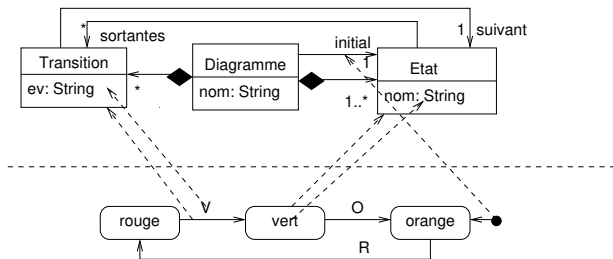
Transformations

Modèles et méta-modèles

Définition : Méta-modèle = modèle du modèle.

⇒ Il s'agit de décrire la structure du modèle.

Exemple : Structure d'un diagramme à état



Conformité : Un modèle est **conforme** à un méta-modèle si :

- ▶ tous les éléments du modèle sont instance d'un élément du méta-modèle ;
- ▶ et les contraintes exprimées sur le méta-modèle sont respectées.

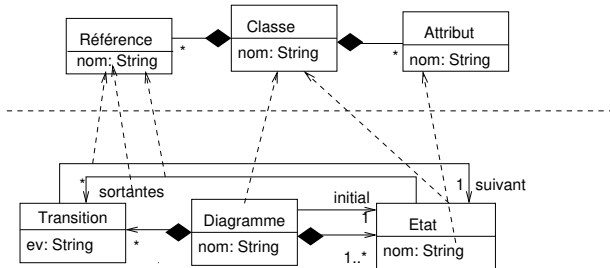
Suite : Où est décrit le méta-modèle ?

Méta-méta-modèles

Définition : Méta-méta-modèle = modèle du méta-modèle.

⇒ Il s'agit de décrire la structure du méta-modèle.

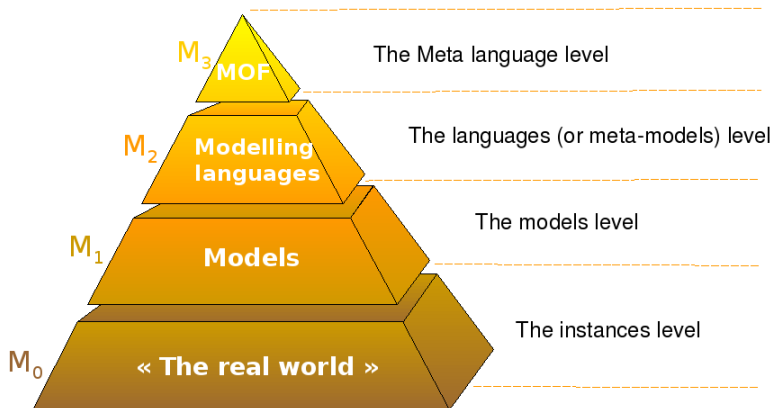
Exemple : Structure des diagrammes à état



Suite : Où est décrit le méta-méta-modèle ?

Réponse : Par lui-même ; il est **réflexif** !

Pyramide de l'OMG



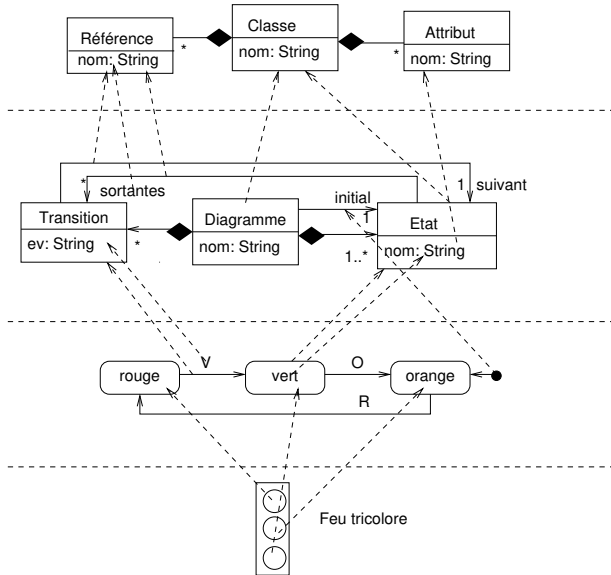
Pyramide de l'OMG

Explications

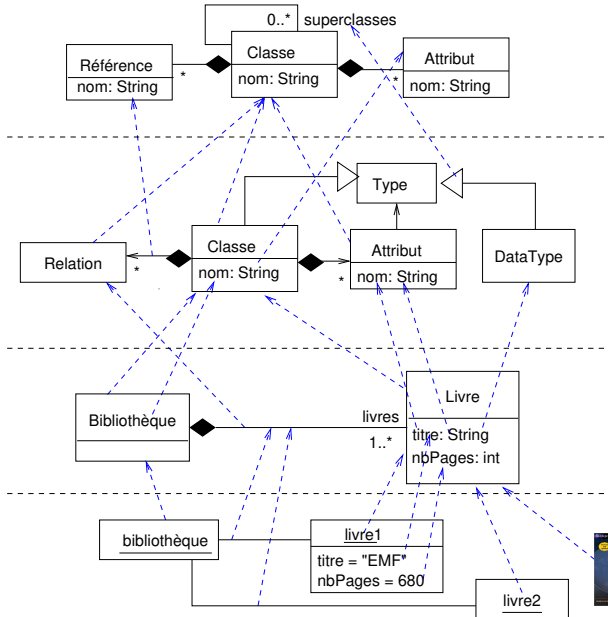
- ▶ M3 : méta-méta-modèle :
 - ▶ réflexif : se décrit en lui-même
 - ▶ pour définir des méta-modèles, langages (exemple : UML)
 - ▶ exemple MOF de l'OMG
- ▶ M2 : méta-modèle : langage de modélisation pour un domaine métier
 - ▶ Exemples : UML2, SPEM...
- ▶ M1 : modèle : un modèle du monde réel
 - ▶ Exemples : un modèle de feu tricolore, un modèle de bibliothèque...
- ▶ M0 : le monde réel
 - ▶ Exemples : un feu tricolore, une bibliothèque...

Remarque : Le numéro permet de préciser l'objectif du « modèle ». Dans la suite, les notions de modèle et méta-modèle sont suffisantes.

Exemple : le monde réel est un feu tricolore



Exemple : le monde réel est une bibliothèque



Conformité (vision tabulaire) M1/M2

Transition

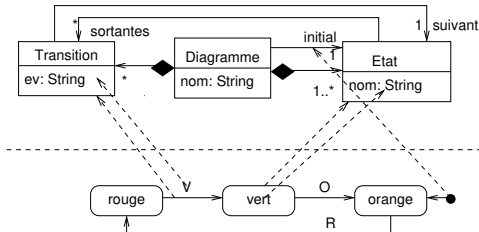
ID	ev	suivant
T1	"O"	E1
T2	"R"	E2
T3	"V"	E3

Etat

ID	nom	sortantes
E1	"orange"	T2
E2	"rouge"	T3
E3	"vert"	T1

Diagramme

ID	nom	etats	initial	transitions
D1	"Feu Tricolore"	E1, E2, E3	E1	T1, T2, T3



Conformité (vision tabulaire) M2/M3

Attribut

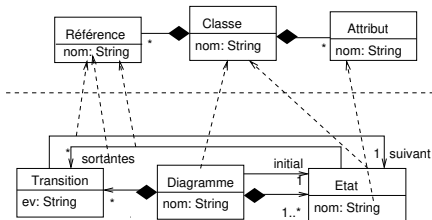
ID	nom	type
A1	"nom"	"String"
A2	"nom"	"String"
A3	"ev"	"String"

Classe

ID	nom	attributs	references
C1	"Diagramme"	A1	R1, R2, R3
C2	"Etat"	A2	R4
C3	"Transition"	A3	R5

Reference

ID	nom	cible	min	max	composition
R1	"etats"	C2	1	*	true
R2	"transitions"	C3	0	*	true
R3	"initial"	C2	1	1	false
R4	"sortantes"	C3	0	*	false
R5	"suivant"	C2	1	1	false



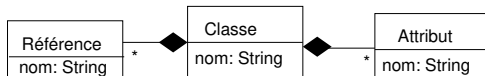
Conformité (vision tabulaire) M3/M3

Classe

ID	nom	attributs	references
C1	"Classe"	A1	R1, R2
C2	"Attribut"	A2, A3	
C3	"Reference"	A4, A5, A6, A7	R3

Attribut

ID	nom	type
A1	"nom"	"String"
A2	"nom"	"String"
A3	"type"	"String"
A4	"nom"	"String"
A5	"min"	"int"
A6	"max"	"int"
A7	"composition"	"boolean"



Reference

ID	nom	cible	min	max	composition
R1	"attributs"	C2	0	-1 (*)	true
R2	"references"	C3	0	-1 (*)	true
R3	"cible"	C1	1	1	false

Autres pyramides

On retrouve cette pyramide à 4 niveaux dans plusieurs espaces technologiques.

	grammarware	docware	relationalware
M3	EBNF	XML	Algèbre relationnelle
M2	Grammaire de Java	Schéma XML	Schéma BD
M1	un prog. Java	un document XML	une BD

Intérêt des méta-modèles

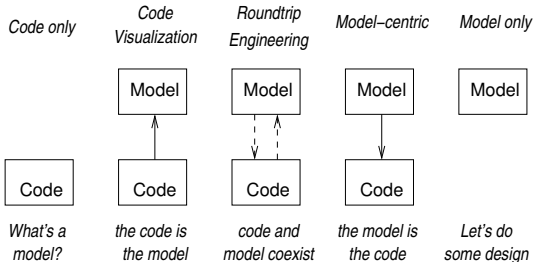
- ▶ définir les **propriétés structurelles** d'une famille de modèles :
 - ▶ capturées par la structure du méta-modèle (multiplicité, références, etc.)
 - ▶ exprimées dans un langage de contrainte.

Exemple : Exprimer que le nb de pages d'un livre est positif en OCL :

context Livre **inv**: nbPages > 0

- ▶ décider de la **conformité** d'un modèle par rapport à un métamodèle
- ▶ **transformer** le modèle (restructuration, raffinement, traduction vers un autre MM, syntaxes concrètes...)
- ▶ permettre l'**interopérabilité** entre outils grâce à une description connue (le MM) des données échangées
- ▶ plus généralement, **raisonner** et **travailler** sur les modèles
- ▶ ...

Modèle et code : différentes perspectives



(<http://www.ibm.com/developerworks/rational/library/3100.html>)

Remarque : L'évolution est à aller vers le tout modèle :

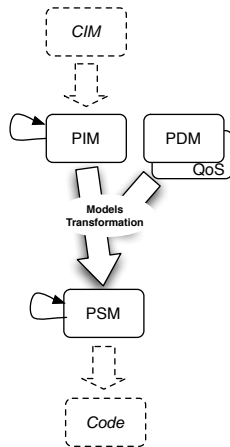
- ▶ modèles nécessaires car au début le système est trop compliqué
- ▶ besoin de vérifier/valider les modèles (coût si erreurs identifiées tardivement)
- ▶ raffiner les modèles et aller vers le code

Le modèle au centre du développement

Objectif : Tenter une interopérabilité par les modèles

- ▶ Partir de CIM (Computer Independant Model) :
 - ▶ aucune considération informatique n'apparaît
- ▶ Faire des modèles indépendants des plateformes (PIM)
 - ▶ rattaché à un paradigme informatique
 - ▶ indépendant d'une plateforme de réalisation précise
- ▶ Spécifier des règles de passage (transformation) ...
- ▶ ... en s'appuyant sur un modèle de la plateforme (PDM)
- ▶ ... vers les modèles dépendants des plateformes (PSM)
 - ▶ version modélisée du code
- ▶ Automatiser au mieux la production vers le code
PIM → PSM → Code

⇒ Processus en Y



Exemples de transformations

PIM \longrightarrow PIM :

- ▶ privatiser les attributs
- ▶ réorganiser le code (refactoring)
- ▶ introduire un patron de conception...

PIM \longrightarrow PSM :

- ▶ génération semi-automatique grâce à des marqueurs :
 - ▶ classe marquée active \Rightarrow hérite de Thread...
 - ▶ persistance
- ▶ motif de passage d'une classe UML à une classe Java
- ▶ prise en compte de l'héritage multiple (C++, Eiffel, Java...)

PSM \longrightarrow PIM :

- ▶ adaptation pour gérer l'interopérabilité entre outils
- ▶ rétroconception
- ▶ abstraction, analyse statique...

Conclusion

Ingénierie Dirigée par les Modèles

=

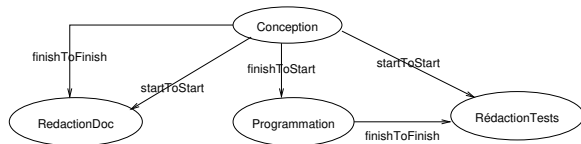
Métamodélisation

+

Transformations

Retour à nos processus

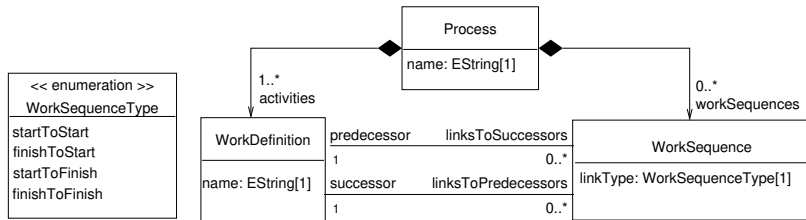
Comment représenter le (un) processus? Graphiquement et textuellement !



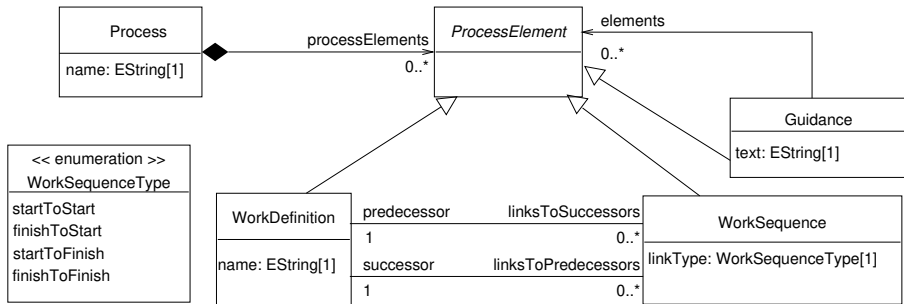
```

process ExempleProcessus {
  wd RédactionDoc
  wd Conception
  wd Développement
  wd RédactionTests
  ws Conception f2f RédactionDoc
  ws Conception s2s RédactionDoc
  ws Conception f2s Développement
  ws Conception s2s RédactionTests
  ws Développement f2f RédactionTests
}
  
```

Quels constituants ? Qu'est ce qu'un processus ?



Un autre métamodèle de SimplePDL



Attention : Toutes les propriétés ne sont pas capturées par le métamodèle.
Exemple : les activités doivent avoir des noms différents

Que faire avec un processus ?

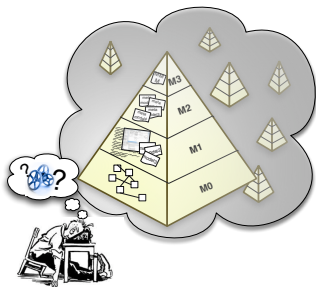
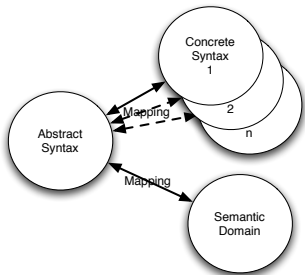
1. Le visualiser graphiquement (ci-avant).
2. En déduire un diagramme de PERT, de Gantt, etc.
3. Produire les fiches de temps des personnes impliquées dans le processus.
4. Produire le taux d'occupation des ressources utilisées.
5. Calculer le coût de la réalisation du processus...

En fonction de ce que l'on veut en faire, d'autres informations seront à modéliser : les personnes impliquées, les ressources, le temps estimé, le temps effectif, etc.

Domain Specific Language (DSL) / Langage métier

Un **langage dédié** ou **langage métier** (Domain Specific Language, DSL) c'est :

- ▶ une **syntaxe abstraite** :
 - ▶ un **méta-modèle** pour décrire les concepts et leurs relations
 - ▶ une **sémantique statique** : contraintes non capturées dans le méta-modèle
- ▶ des **syntaxes concrètes**, graphiques ou textuelles
- ▶ des **domaines sémantiques** (utilisation qui en sera faite) : **transformations** !



Exemples

- ▶ HTML : Description de documents Web.
- ▶ SPEM, PBMN : modélisation de processus (OMG)
- ▶ \LaTeX : langage et système de composition de documents
- ▶ SCADE : Safety-Critical Application Development Environment (modélisation de l'application, vérification du modèle, génération de code certifié) utilisé par Airbus
- ▶ Construction (et génération) d'interfaces graphiques
- ▶ Générateurs d'applications mobiles
- ▶ Unity : moteur de jeu multi-plateforme
- ▶ Un pseudo-langage algorithmique
- ▶ On pourrait considérer tout programme qui permet à l'utilisateur de gérer des données comme offrant un langage dédié !

Topologie des transformations

Les **transformations** sont le moyen d'exploiter les modèles et donc de leur donner du sens (**sémantique**).

Transformations **texte à modèle, T2M**, pour définir des syntaxes concrètes :

- ▶ textuelles : par exemple avec Xtext
- ▶ graphiques : par exemple avec GMF ou Sirius

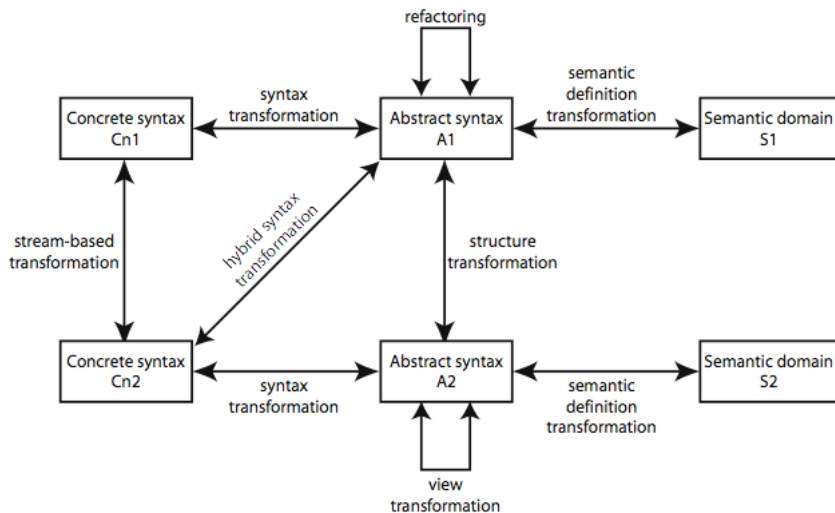
Transformations de **modèle à modèle, M2M**, :

- ▶ traduire un modèle d'un premier MM vers un modèle d'un autre MM (voir MDE)
- ▶ Moyen : avec un langage généraliste (Java...) ou dédié (ATL...)

Transformations **modèle à texte, M2T** :

- ▶ transformer un modèle vers du texte : une page Web, un programme, un langage d'entrée d'un outil, etc.
- ▶ Moyen : Acceleo, etc. (plus généralement, langage de template)

Types de transformation



(from Anneke Kleppe)

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Les langages de méta-modélisation
Le langage Ecore d'Eclipse/EMF

Les langages de méta-modélisation

Plusieurs langages proposés :

- ▶ MOF (Meta-Object Facility) proposé par l'OMG :
 - ▶ variantes EMOF (Essential MOF) et CMOF (Complete MOF)

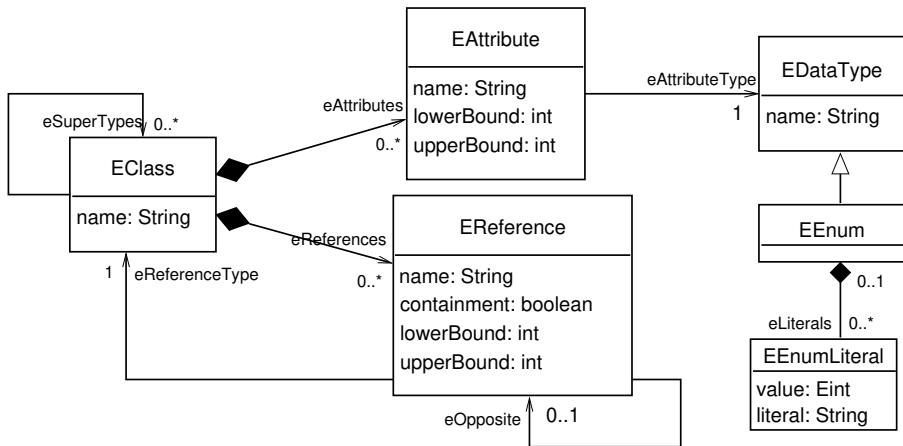
Au départ description de UML en UML

Extraction du minimum d'UML pour décrire UML \implies MOF

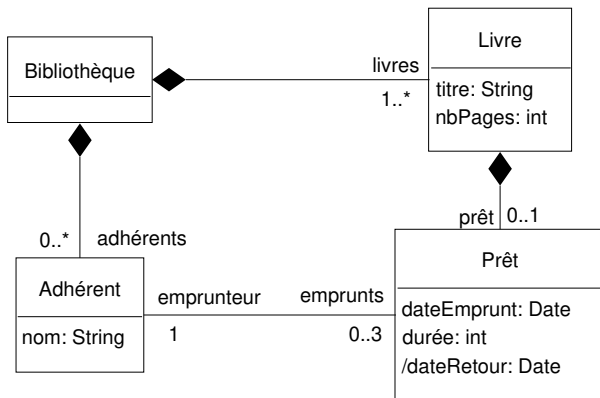
- ▶ **Ecore** : Eclipse/EMF (Eclipse Modelling Framework)
Implantation de EMOF (équivalent)
- ▶ KM3 (Kernel MetaMetaModel) : Meta-modèle de AMMA/ATL, (LINA, Nantes)
- ▶ Kermeta : (IRISA, Rennes) extension de EMOF/Ecore pour permettre de décrire le comportement d'un méta-modèle (méta-programmation).
- ▶ GME (The Generic Modeling Environment), Vanderbilt.
<http://www.isis.vanderbilt.edu/projects/gme/>
- ▶ ...

Le langage de méta-modélisation ECore (Eclipse/EMF)

Extrait du méta-modèle ECore : principales notions



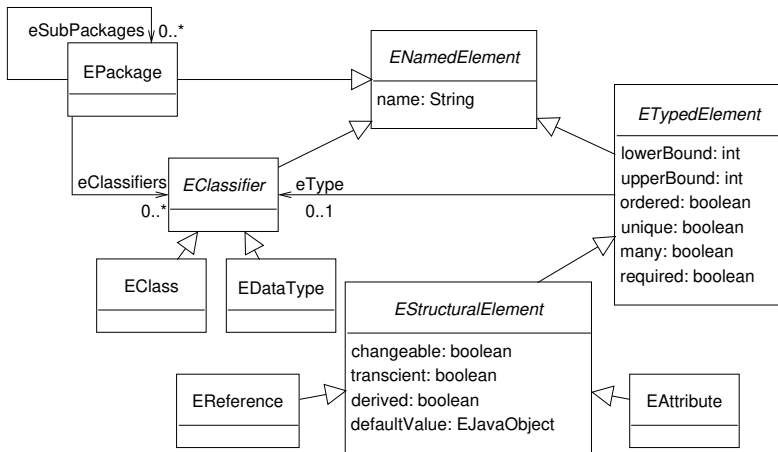
Exemple de modèle ECore : bibliothèque



Principaux constituants de ECore

- ▶ EClass : Description d'un concept caractérisé par des attributs et des références
- ▶ EAttribute : une propriété de l'objet dont le type est « élémentaire »
- ▶ EReference : une référence vers un autre concept (EClass) équivalent à une association UML avec sens de navigation
- ▶ La propriété *containment* indique s'il y a *composition* :
 - ▶ vrai : l'objet référencé est contenu (durées de vie liées)
 - ▶ faux : c'est une référence vers un objet contenu par un autre élément.
- ▶ multiplicité définie sur les attributs et les références (idem UML).
Convention : on note -1 pour indiquer * pour *upperBound*
- ▶ Héritage multiple : *eSuperTypes*
- ▶ Référence *eOpposite* pour indiquer que deux références opposées sont liées (équivalent association UML).

Extrait méta-modèle ECore : propriétés structurelles

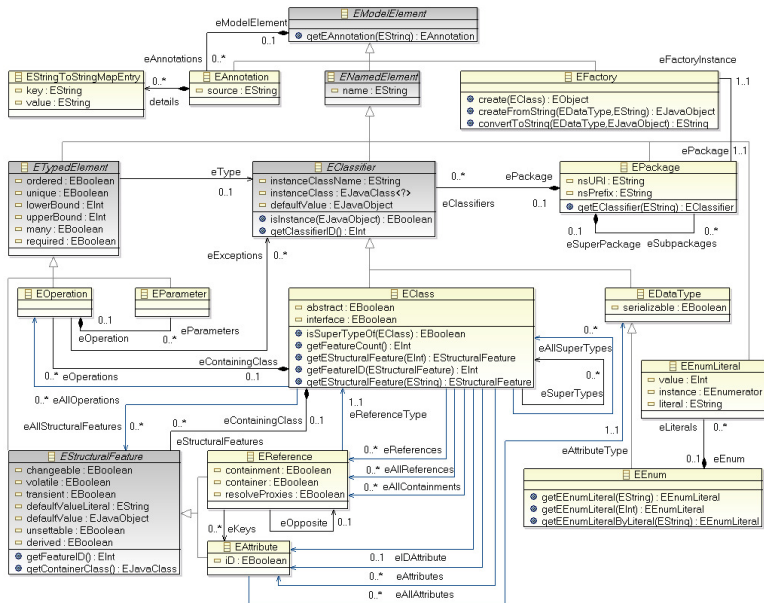


Autres caractéristiques de Ecore

- ▶ Méta-modèle : plus riche que le premier présenté.
- ▶ Éléments abstraits : `ENamedElement`, `ETypedElement`, etc.
- ▶ Paquetage : ensemble de classes et paquetages
- ▶ Caractéristiques liées à la multiplicité : `ordered`, `unique`...
- ▶ `EEnum` : énumération : lister les valeurs possibles d'un `EDataType`.
- ▶ Opération (non présentées) : décrit la signature des opérations, pas le code.

Remarque : Héritage multiple et classes abstraites favorisent la factorisation et la réutilisation (ex : `ENamedElement`, `ETypedElement`).

refcardz.dzone.com/refcardz/essential-emf



Intérêt de définir un modèle ECore

EMF permet d'engendrer :

- ▶ Le modèle Java correspondant :
 - ▶ chaque EClass donne une interface et une réalisation. Justification :
 - ▶ Bonne pratique que de définir des interfaces !
 - ▶ Permet de gérer l'héritage multiple
 - ▶ équipée d'observateurs (changement d'attribut ou de référence).
- ▶ Un schéma XML correspondant et les opérations de sérialisation/désérialisation associées.
- ▶ Un éditeur arborescent pour saisir un modèle.

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Motivation

Présentation générale d'OCL

Syntaxe du langage

Conseils

Objectif général

Objectif : OCL est avant tout un **langage de requête** pour calculer une *expression sur un modèle en s'appuyant sur sa syntaxe* (son méta-modèle).

⇒ Une expression exprimée une fois, pourra être évaluée sur tout modèle conforme au méta-modèle correspondant.

Exemple : pour une bibliothèque particulière on peut vouloir demander :

- ▶ Livres possédés par la bibliothèque ? Combien y en a-t-il ?
- ▶ Auteurs dont au moins un titre est possédé par la bibliothèque ?
- ▶ Titres dans la bibliothèque écrits par Martin Fowler ?
- ▶ Nombre de pages du plus petit ouvrage ?
- ▶ Nombre moyen de pages des ouvrages ?
- ▶ Ouvrages de plus 100 pages écrits par au moins trois auteurs ?
- ▶ ...

Programmation par contrat

Principe : Établir formellement les responsabilités d'une classe et de ses méthodes.

Moyen : définition de propriétés (expressions booléennes) appelées :

- ▶ **invariant** : propriété définie sur une **classe** qui doit toujours être vraie, de la création à la disparition d'un objet.

Un invariant lie les requêtes d'une classe (état externe).

- ▶ **précondition** : propriété sur une **méthode** qui :
 - ▶ doit être vérifiée par l'appelant pour que l'appel à cette méthode soit possible ;
 - ▶ peut donc être supposée vraie dans le code de la méthode.

postconditions : propriété sur une **méthode** qui définit l'effet de la méthode, c'est-à-dire :

- ▶ spécification de ce que doit écrire le programmeur de la méthode ;
- ▶ caractérisation du résultat que l'appelant obtiendra.

Exercice : Invariant pour une Fraction (état = numérateur et dénominateur) ?

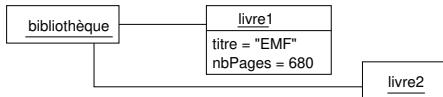
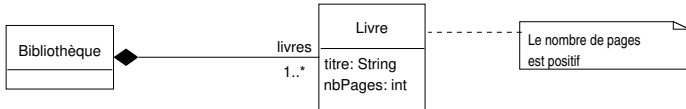
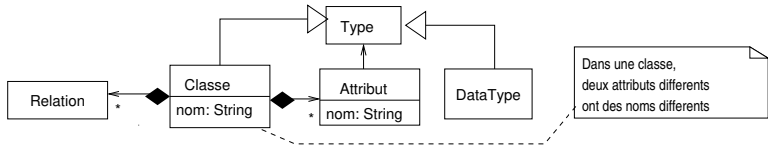
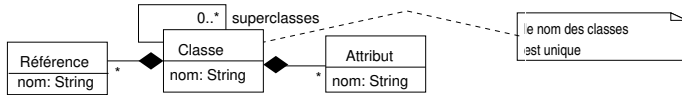
Exercice : Pré- et postconditions de racine carrée et de pgcd ?

OCL et Diagrammes d'UML

OCL peut être utilisé sur différents diagrammes d'UML :

- ▶ diagramme de classe :
 - ▶ définir des préconditions, postconditions et invariants :
Stéréotypes prédéfinis : «precondition», «postcondition» et «invariant»
 - ▶ caractérisation d'un **attribut dérivé** (p.ex. le salaire est fonction de l'âge)
 - ▶ spécifier la **valeur initiale** d'un attribut (p.ex. l'attribut *salaire* d'un employé)
 - ▶ spécifier le **code d'une opération** (p.ex. le salaire annuel est 12 fois le salaire mensuel)
- ▶ diagramme d'état :
 - ▶ spécifier une garde sur une transition
 - ▶ exprimer une expression dans une activité (affectation, etc.)
 - ▶ ...
- ▶ diagramme de séquence :
 - ▶ spécifier une garde sur un envoi de message
- ▶ ...

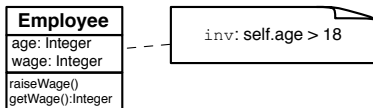
OCL et Méta-modélisation : préciser la sémantique statique d'un modèle



The *Object Constraint Language*

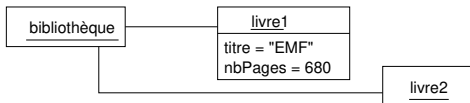
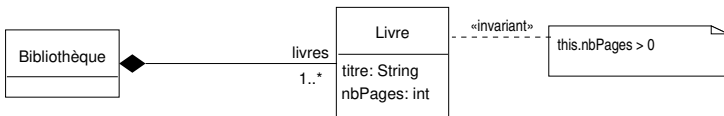
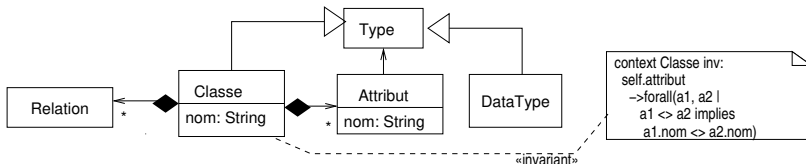
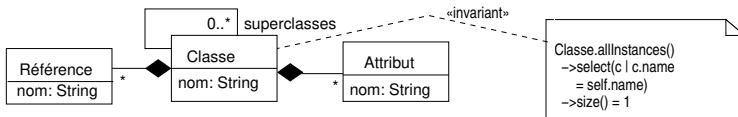
Objectifs initiaux

- ▶ Les langages formels traditionnels (e.g. Z) requièrent de la part des utilisateurs une bonne compréhension des fondements mathématiques.
- ▶ *Object Constraint Language (OCL)* a été développé dans le but d'être :
 - ▶ formel, précis et non ambigu,



- ▶ utilisable par un large nombre d'utilisateurs,
 - ▶ un langage de spécification (et non de programmation !),
 - ▶ supporté par des outils.
- ▶ Historique :
 - ▶ Développé en 1995 par IBM,
 - ▶ Inclut dans le standard UML jusqu'à la version 1.1 (1997),
 - ▶ OCL 2.0 *Final Adopted Specification* (ptc/06-05-01), May 2006.
 - ▶ Version actuelle : OCL 2.4 (formal/2014-02-03), January 2012.

Préciser la sémantique statique d'un modèle



The *Object Constraint Language*

Propriétés du langage

► **Langage de spécification sans effet de bord**

- une expression OCL calcule une valeur... **et** laisse le modèle inchangé !
 - ⇒ l'état d'un objet ne peut pas être modifié **par** l'évaluation d'une expression OCL
- l'évaluation d'une expression OCL est instantanée
 - ⇒ l'état des objets ne peut donc pas être modifié **pendant** l'évaluation d'une expression OCL
- OCL n'est pas un langage de programmation !

► OCL est un **langage typé** :

- Chaque expression OCL a un type
- OCL définit des types primitifs : **Boolean**, **Integer**, **Real** et **String**
- Chaque *Classifier* du modèle est un nouveau type OCL
- *Intérêt* : vérifier la cohérence des expressions
exemple : il est interdit de comparer un String et un Integer

Les types OCL de base

Les types de base (*Primitive*) sont **Integer**, **Real**, **Boolean** et **String**. Les opérateurs suivants s'appliquent sur ces types :

Opérateurs relationnels	<code>=, <>, >, <, >=, <=</code>
Opérateurs logiques	<code>and, or, xor, not, if ... then ... else ... endif</code>
Opérateurs mathématiques	<code>+, -, /, *, min(), max()...</code>
Opérateurs pour les chaînes de caractères	<code>concat, toUpper, substring...</code>

Attention : Concernant l'opérateur **if ... then ... else ... endif** :

- ▶ la clause **else** est nécessaire et,
- ▶ les expressions du **then** et du **else** doivent être de même type.

Attention : **and**, **or**, **xor** ne sont pas évalués en court-circuit !

Priorité des opérateurs

Liste des opérateurs dans l'ordre de priorité décroissante :

- 1 **@pre**
- 2 . --> -- *notation pointée et fléchée*
- 3 **not** -- -- *opérateurs unaires*
- 4 * /
- 5 + -- -- *opérateurs binaires*
- 6 **if** -- **then** -- **else** -- **endif**
- 7 < > <= >=
- 8 = <>
- 9 **and** **or** **xor**
- 10 **implies** -- *implication*

Remarque : Les parenthèses peuvent être utilisées pour changer la priorité.

Les autres types OCL

- ▶ Tous les éléments du modèle sont des types (*OclModelElementType*),
 - ▶ y compris les énumérations : *Gender :: male*,
- ▶ Type *Tuple* : enregistrement (produit cartésien de plusieurs types)
Tuple {a : Collection(Integer) = Set{1, 3, 4}, b : String = 'foo'}
- ▶ *OclMessageType* :
 - ▶ utilisé pour accéder aux messages d'une opération ou d'un signal,
 - ▶ offre un rapport sur la possibilité d'envoyer/recevoir une opération/un signal.
- ▶ *VoidType* :
 - ▶ a seulement une instance *oclUndefined*,
 - ▶ est conforme à tous les types.

Contexte d'une expression OCL

Une expression est définie sur un **contexte** qui identifie :

- une **cible** : l'élément du modèle sur lequel porte l'expression OCL

T	Type (Classifier : Interface, Classe...)	context Employee
M	Opération/Méthode	context Employee::raiseWage(inc:Int)
A	Attribut ou extrémité d'association	context Employee::job : Job

- le **rôle** : indique la **signification** de cette expression (pré, post, invariant...) et donc contraint sa **cible** et son **évaluation**.

rôle	cible	signification	évaluation
inv	T	invariant	toujours vraie
pre	M	précondition	avant tout appel de M
post	M	postcondition	après tout appel de M
body	M	résultat d'une requête	appel de M
init	A	valeur initiale de A	création
derive	A	valeur de A	utilisation de A
def	T	définir une méthode ou un attribut	

Syntaxe d'OCL

inv (invariant) doit toujours être vrai (avant et après chaque appel de méthode)

context Employee

inv: self.age > 18

context e : Employee

inv age_18: e.age > 18

pre (precondition) doit être vraie avant l'exécution d'une opération

post (postcondition) doit être vraie après l'exécution d'une opération

context Employee::raiseWage(increment : **Integer**)

pre: increment > 0

post my_post: self.wage = self.wage@**pre** + increment

context Employee::getWage() : **Integer**

post: result = self.wage

Remarques : result et @**pre** : utilisables seulement dans une postcondition

- ▶ **exp@pre** correspond à la valeur de expr avant l'appel de l'opération.
- ▶ result est une variable prédéfinie qui désigne le résultat de l'opération.

Syntaxe d'OCL

- ▶ **body** spécifie le résultat d'une opération

context Employee::getWage() : **Integer**
body: self.wage

- ▶ **init** spécifie la valeur initiale d'un attribut ou d'une association

context Employee::wage : **Integer**
init: 900

- ▶ **derive** spécifie la règle de dérivation d'un attribut ou d'une association

context Employee::wage : **Integer**
derive: self.age * 50

- ▶ **def** définition d'opérations (ou variables) qui pourront être (ré)utilisées dans des expressions OCL.

context Employee
def: annualIncome : **Integer** = 12 * wage

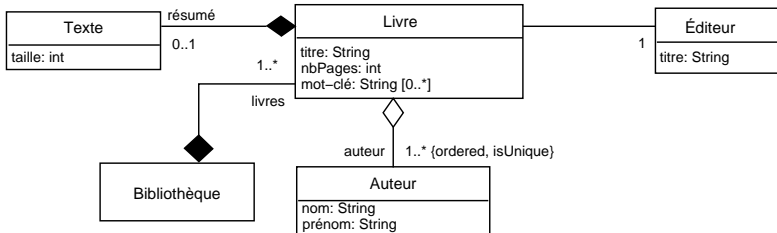
La navigation dans le modèle

Accès aux informations de la classe

- ▶ Une expression OCL est définie dans le contexte d'une classe
 - ▶ en fait : un type, une interface, une classe, etc.
 - ▶ Elle s'applique sur un objet, instance de cette classe :
 - ⇒ cet objet est désigné par le mot-clé **self**.
 - ▶ Étant donné un accès à un objet (p.ex. `self`), une expression OCL peut :
 - ▶ accéder à la valeur des attributs :
 - ▶ `self.nbPages`
 - ▶ `unLivre.nbPages`
 - ▶ appeler toute requête définie sur l'objet :
 - ▶ `self.getNbPages()`
 - ▶ `unLivre.getNbPages()`
- Rappel : Une requête (notée `{isQuery}` en UML) est une opération :*
- ▶ qui a un type de retour (calcule une expression) ;
 - ▶ et n'a pas d'effet de bord (ne modifie pas l'état du système).
- ▶ parcourir les associations...

Correspondance entre association et OCL

► Exemple de diagramme de classe



► pour atteindre l'autre extrémité d'une association, on utilise :

- le rôle, p.ex. : `unLivre.résumé`
- à défaut le nom de la classe en minuscule : `unLivre.editeur`

► La manière dont une association est vue en OCL dépend :

- de sa multiplicité : un exactement (`1`), optionnel (`0..1`), ≥ 2
- de ses qualificatifs : `{ isUnique }`, `{ isOrdered }`

Correspondance entre association et OCL

association avec multiplicité ≤ 1

- ▶ multiplicité 1 : nécessairement un objet à l'extrémité (invariant implicite)
 - ▶ `unLivre.editeur`
- ▶ multiplicité 0..1 (optionnel) :
 - ▶ utiliser l'opération **`oclIsUndefined()`**
 - ▶ `unLivre.résumé.oclIsUndefined()` est :
 - ▶ vraie si pas de résumé,
 - ▶ faux sinon
 - ▶ Exemple d'utilisation :

```
if unLivre.résumé.oclIsUndefined() then
    true
else
    unLivre.résumé.taille >= 60
endif
```

Correspondance entre association et OCL

association avec multiplicité ≥ 2

- ▶ les éléments à l'extrémité d'une association sont accessibles par une collection
- ▶ OCL définit quatre types de collection :
 - ▶ **Set** : pas de double, pas d'ordre
 - ▶ **Bag** : doubles possibles, pas d'ordre
 - ▶ **OrderedSet** : pas de double, ordre
 - ▶ **Sequence** : doubles possibles, ordre
- ▶ Lien entre associations UML et collections OCL

UML	Ecore	OCL
		Bag
isUnique	Unique	Set
isOrdered	Ordered	Sequence
isUnique, isOrdered	Unique, Ordered	OrderedSet

- ▶ Exemple : `unLivre.auteur` : la collection des auteurs de `unLivre`

Les collections OCL

- **Set** : ensemble d'éléments *sans* doublon et *sans* ordre

Set {7, 54, 22, 98, 9, 54, 20..25}

— *Set*{7,54,22,98,9,20,21,23,24,25} : *Set(Integer)*

— *ou Set*{7,9,20,21,22,23,24,25,54,98} : *Set(Integer)*, *ou...*

- **OrderedSet** : ensemble d'éléments *sans* doublon et *avec* ordre

OrderedSet {7, 54, 22, 98, 9, 54, 20..25}

— *OrderedSet*{7,9,20,21,22,23,24,25,54,98} : *OrderedSet(Integer)*

- **Bag** : ensemble d'éléments *avec* doublons possibles et *sans* ordre

Bag {7, 54, 22, 98, 9, 54, 20..25}

— *p.ex.* : *Bag*{7,9,20,21,22,22,23,24,25,54,54,98} : *Bag(Integer)*

- **Sequence** : ensemble d'éléments *avec* doublons possibles et *avec* ordre

Sequence {7, 54, 22, 98, 9, 54, 20..25}

— *Sequence*{7,54,22,98,9,54,20,21,22,23,24,25} : *Sequence(Integer)*

Les collections sont génériques : *Bag(Integer)*, *Set(String)*, *Bag(Set(Livre))*

Opérations sur les collections (bibliothèque standard)

Pour tous les types de Collection

size(): Integer *-- nombre d'éléments dans la collection self*

includes(object: T): Boolean *-- est-ce que object est dans self ?*

excludes(object: T): Boolean *-- est-ce que object n'est pas dans self ?*

count(object: T): Integer *-- nombre d'occurrences de object dans self*

includesAll(c2: Collection(T)): Boolean
 -- est-ce que self contient tous les éléments de c2 ?

excludesAll(c2: Collection(T)): Boolean
 -- est-ce que self ne contient aucun des éléments de c2 ?

isEmpty(): Boolean *-- est-ce que self est vide ?*

notEmpty(): Boolean *-- est-ce que self est non vide ?*

sum(): T *-- la somme (+) des éléments de self*
 -- l'opérateur + doit être défini sur le type des éléments de self

product(c2: Collection(T2)): Set(Tuple(premier: T, second: T2))
 -- le produit () des éléments de self*

Opérations de la bibliothèque standard pour les collections

En fonction du sous-type de *Collection*, d'autres opérations sont disponibles :

- ▶ union
- ▶ intersection
- ▶ append
- ▶ flatten
- ▶ =
- ▶ ...

Une liste exhaustive des opérations de la bibliothèque standard pour les collections est disponible dans [OMG OCL 2.3.1, §11.7].

Opérations de la bibliothèque standard pour tous les objets

OCL définit des opérations qui peuvent être appliquées à tous les objets

- ▶ *oclIsTypeOf*($t : \text{OclType}$) : *Boolean*

Le résultat est vrai si le type de *self* et *t* sont identiques.

context Employee

inv: self. **oclIsTypeOf**(Employee) — *is true*

inv: self. **oclIsTypeOf**(Company) — *is false*

- ▶ *oclIsKindOf*($t : \text{OclType}$) : *Boolean*

vrai si *t* est le type de *self* ou un super-type de *self*.

- ▶ *oclIsNew*() : *Boolean*

Uniquement dans les post-conditions

vrai si le récepteur a été créé au cours de l'exécution de l'opération.

- ▶ *oclIsInState*($t : \text{OclState}$) : *Boolean*

Le résultat est vrai si l'objet est dans l'état *t*.

Opérations de la bibliothèque standard pour tous les objets

OCL définit des opérations qui peuvent être appliquées à tous les objets

► *oclAsType*(*t* : *OclType*) : *T*

Retourne le même objet mais du type *t*

Nécessite que *oclIsKindOf*(*t*) = *true*

► *allInstances*()

► prédéfinie pour les classes, les interfaces et les énumérations,

► le résultat est la collection de toutes les instances du type au moment de l'évaluation.

context Employee

inv: Employee. **allInstances**() → **forAll**(*p1*, *p2*
| *p1* <> *p2* **implies** *p1.name* <> *p2.name*)

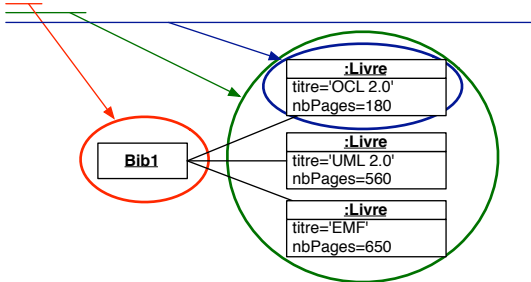
Opérateur *select* (resp. *reject*)

Permet de spécifier le sous-ensemble de tous les éléments de *collection* pour lesquels l'expression est vraie (resp. fausse pour *reject*).

- ▶ $collection \rightarrow select(elem : T | expr)$
- ▶ $collection \rightarrow select(elem | expr)$
- ▶ $collection \rightarrow select(expr)$

context Bibliothèque inv:

self.livres->select(name = 'OCL 2.0')->notEmpty()



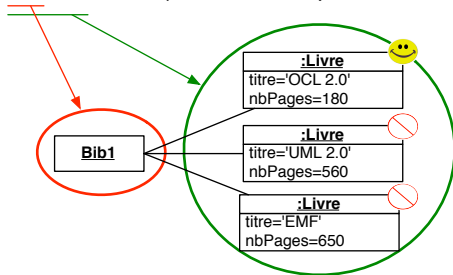
Opérateur *exists*

Retourne vrai si l'expression est vraie pour au moins un élément de la collection.

- ▶ $collection \rightarrow exists(elem : T|expr)$
- ▶ $collection \rightarrow exists(elem|expr)$
- ▶ $collection \rightarrow exists(expr)$

context Bibliothèque inv:

self.livres->exists(name = 'OCL 2.0')



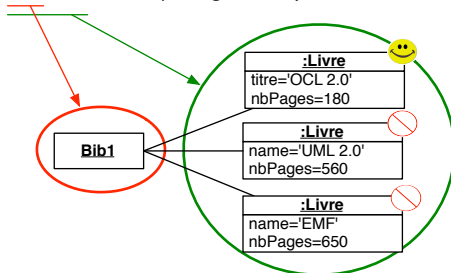
Opérateur *forAll*

Retourne vrai si l'expression est vraie pour tous les éléments de la collection.

- ▶ *collection* → *forAll*(*elem* : *T* | *expr*)
- ▶ *collection* → *forAll*(*elem* | *expr*)
- ▶ *collection* → *forAll*(*expr*)

context Bibliothèque inv:

self.livres->forAll(nbPages < 200)

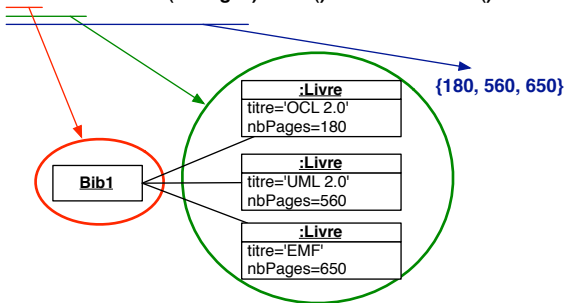


Opérateur *collect*

Retourne la collection des valeurs (*Bag*) résultant de l'évaluation de l'expression appliquée à tous les éléments de collection.

- ▶ *collection* → *collect(elem : T|expr)*
- ▶ *collection* → *collect(elem|expr)*
- ▶ *collection* → *collect(expr)*

```
context Bibliothèque def moyenneDesPages : Real =  
  self.livres->collect(nbPages)->sum() / self.livres->size()
```



Opérateur *iterate*

Forme générale d'une itération sur une collection et permet de redéfinir les précédents opérateurs.

```
collection—> iterate(elem : Type;
    answer : Type = <value>
    | <expression_with_elem_and_answer>)
```

```
context Bibliothèque def moyenneDesPages : Real =
    self.livres—> collect(nbPages)—> sum() / self.livres—> size()
```

— est identique à :

```
context Bibliothèque def moyenneDesPages : Real =
    self.livres—> iterate(l : Livre;
        lesPages : Bag{Integer} = Bag{ }
        | lesPages—> including(l.nbPages))
    —> sum() / self.livres—> size()
```

Plusieurs itérateurs pour un même opérateur

Remarque : les opérateurs *forAll*, *exist* et *iterate* acceptent plusieurs itérateurs :

```
Auteur. allInstances() -> forAll(a1, a2 |  
    a1 <> a2 implies  
    a1.nom <> a2.nom or a1.prénom <> a2.prénom)
```

Bien sûr, dans ce cas il faut nommer tous les itérateurs !

Conseils

OCL ne remplace pas les explications en langage naturel.

- ▶ Les deux sont *complémentaires* !
- ▶ *Comprendre* (informel)
- ▶ *Lever les ambiguïtés* (OCL)

Éviter les expressions OCL trop compliquées

- ▶ éviter les navigations complexes (utiliser **let** ou **def**)
- ▶ bien choisir le contexte (associer l'invariant au bon type !)
- ▶ éviter d'utiliser **allInstances()** :
 - ▶ rend souvent les invariants plus complexes
 - ▶ souvent difficile d'obtenir toutes les instances dans un système (sauf BD !)
- ▶ dissocier une conjonction de contraintes en plusieurs (inv, post, pre)
- ▶ Toujours nommer les extrémités des associations (rôle des objets)

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Syntaxe concrète textuelle (avec Xtext)

Objectif : définir une syntaxe concrète textuelle pour un méta-modèle

Il s'agit d'une **transformation texte à modèle (T2M)**

Intérêt : pouvoir utiliser tous les outils classiques pour les textes :

- ▶ éditeurs de texte
- ▶ outils de gestion de version et configuration
- ▶ recherche de texte, remplacement de texte
- ▶ ...

Xtext : <http://www.eclipse.org/Xtext>

- ▶ définit un pont entre modelware et grammarware
- ▶ syntaxe concrète définie par une grammaire LL(k)
- ▶ engendre un outil de transformation à la demande des fichiers texte
- ▶ engendre un éditeur syntaxique pour Eclipse

Syntaxe concrète textuelle pour SimplePDL

- Un exemple de syntaxe concrète textuelle pour SimplePDL :

```
process ExempleProcessus {  
    wd RedactionDoc  
    wd Conception  
    wd Developpement  
    wd RedactionTests  
    ws Conception f2f RedactionDoc  
    ws Conception s2s RedactionDoc  
    ws Conception f2s Developpement  
    ws Conception s2s RedactionTests  
    ws Developpement f2f RedactionTests  
}
```

- De nombreuses autres syntaxes sont possibles !

Description de la syntaxe concrète avec Xtext

```
1 grammar fr.n7.PDL1 with org.eclipse.xtext.common.Terminals
2 generate pDL1 "http://www.n7.fr/PDL1"
3
4 Process : 'process' name=ID '{'
5         processElements+=ProcessElement*
6         '}' ;
7
8 ProcessElement : WorkDefinition | WorkSequence | Guidance ;
9
10 WorkDefinition : 'wd' name=ID ;
11
12 WorkSequence : 'ws' linkType=WorkSequenceType
13              'from' predecessor=[WorkDefinition]
14              'to' successor=[WorkDefinition] ;
15
16 enum WorkSequenceType : start2start = 's2s'
17                        | finish2start = 'f2s'
18                        | start2finish = 's2f'
19                        | finish2finish = 'f2f' ;
20
21 Guidance : 'note' texte=STRING ;
```

Principales caractéristiques

Xtext d'appuie sur une **grammaire décrite en EBNF** :

```
Process : 'process' ID '{' ProcessElement* '}' ;  
ProcessElement : WorkDefinition | WorkSequence | Guidance ;  
WorkDefinition : 'wd' ID ;  
WorkSequence : 'ws' WorkSequenceType 'from' [WorkDefinition] 'to' [WorkDefinition] ;  
enum WorkSequenceType : 's2s' | 'f2s' | 's2f' | 'f2f' ;
```

- ▶ les chaînes OCL (entre apostrophes) sont les terminaux.
- ▶ ? : optionnel, + : au moins une fois, * : zéro ou plusieurs fois
- ▶ ID est une chaîne qui doit être unique (identifiant)
- ▶ un non terminal entre [] est un ID correspondant à son identifiant

Chaque **non terminal correspond à un objet** dont le type est une classe qui porte le même nom (si le méta-modèle est engendré par Xtext).

Des **actions sémantiques** sont ajoutées sous la forme `S : ... x=Y ...` ; où x est une propriété de S initialisée avec l'objet créé lors de l'analyse de Y.

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

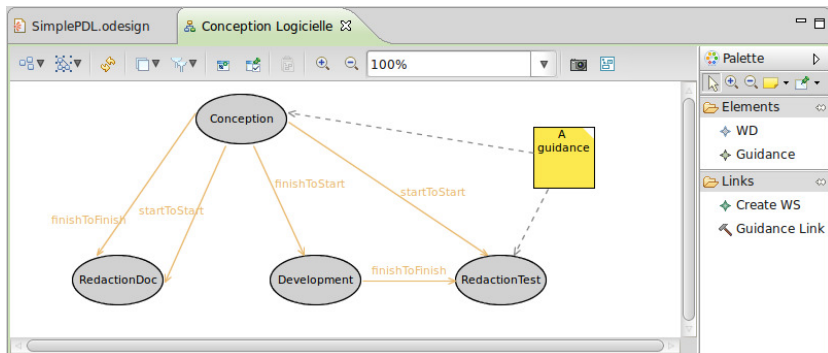
Syntaxe concrète graphique avec Sirius

Objectifs :

- ▶ Présenter graphiquement (une vue d')un modèle
- ▶ Éditer graphiquement un modèle

Principe : Passerelle entre Modelware et composants graphiques.

Exemple : avec Sirius d'Obéo (<https://www.eclipse.org/sirius/>)



Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Transformation modèle à texte

Objectif : Engendrer un texte à partir d'un modèle.

Intérêt :

- ▶ engendrer du code (ex : code Java à partir d'un diagramme UML)
- ▶ engendrer de la documentation
- ▶ engendrer un texte d'entrée d'un outil
ex : engendrer la représentation Tina d'un réseau de Petri

Outils :

- ▶ Langage de programmation généraliste
- ▶ Langage de transformation (plus précisément de requête)
- ▶ Langage de template, en particulier ceux de <http://www.eclipse.org/modeling/m2t/>
 - ▶ le texte de la page intègre des morceaux de scripts.
- ▶ Notre choix : **Acceleo** : <http://www.eclipse.org/acceleo/>

Sérialisation d'un modèle SimplePDL avec Acceleo I

```

1  [comment encoding = UTF-8 /]
2  [module toPDL('http://simplepdl') /]
3
4  [comment Generation de la syntaxe PDL1 à partir d'un modèle de processus/]
5
6  [template public toPDL(proc : Process)]
7  [comment @main/]
8  [file (proc.name.concat('.pdl1'), false, 'UTF-8')]
9  process [proc.name/]{
10 [for (wd : WorkDefinition | proc.processElements->getWDs())
11     wd [wd.name/]
12 [/for]
13 [for (ws : WorkSequence | proc.processElements->getWSs())
14     ws [ws.predecessor.name/] [ws.getWSType()/] [ws.successor.name/]
15 [/for]
16 }
17 [/file]
18 [/template]
19
20 [query public getWDs(elements : OrderedSet(ProcessElement)) : OrderedSet(WorkDefinition) =
21     elements->select( e | e.ocIsTypeOf(WorkDefinition) )
22     ->collect( e | e.ocIsType(WorkDefinition) )
23     ->asOrderedSet()
24 /]
25

```

Sérialisation d'un modèle SimplePDL avec Acceleo II

```
26 [query public getWSS(elements : OrderedSet(ProcessElement)) : OrderedSet(WorkSequence) =
27     elements->select( e | e.ocllsTypeOf(WorkSequence) )
28     ->collect( e | e.oclAsType(WorkSequence) )
29     ->asOrderedSet()
30 /]
31
32 [template public getWSType(ws : WorkSequence)]
33 [if (ws.linkType = WorkSequenceType::startToStart)]
34 s2s[elseif (ws.linkType = WorkSequenceType::startToFinish)]
35 s2f[elseif (ws.linkType = WorkSequenceType::finishToStart)]
36 f2s[elseif (ws.linkType = WorkSequenceType::finishToFinish)]
37 f2f[/if]
38 [/template]
```

Principales caractéristiques d'Acceleo

- ▶ Acceleo est un langage à balises (à la XML, [] au lieu de <>)
- ▶ Acceleo utilise OCL pour requêter le modèle
- ▶ Les commentaires se sont avec [comment].
 - ▶ Ils peuvent contenir des directives exploitées par Acceleo (main, encoding...)
- ▶ Une requête (**query**) est une méthode qui renvoie une donnée
- ▶ Un **template** contient le texte à produire qui peut inclure des structures de contrôle (**for**, **if**...) ou des expressions Acceleo ([expression/])
 - ▶ Attention : les espaces sont significatifs (template oblige) !
 - ▶ Comment faire un [? [' [' /]
 - ▶ **file** indique que le résultat du *template* sera enregistré dans le fichier précisé
- ▶ Attention, pas de liaison dynamique lors de l'appel de requêtes et templates !
- ▶ **Question** : Est-ce qu'il était judicieux d'utiliser un template pour getWSType ?

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Transformation modèle à modèle

Objectif : définir une transformation d'un modèle M_A conforme à un métamodèle MM_A à un modèle M_B conforme à un métamodèle MM_B

Exemples :

- ▶ Transformer des données d'un formalisme à un autre
- ▶ Modifier un modèle
- ▶ Extraire une vue d'un modèle...

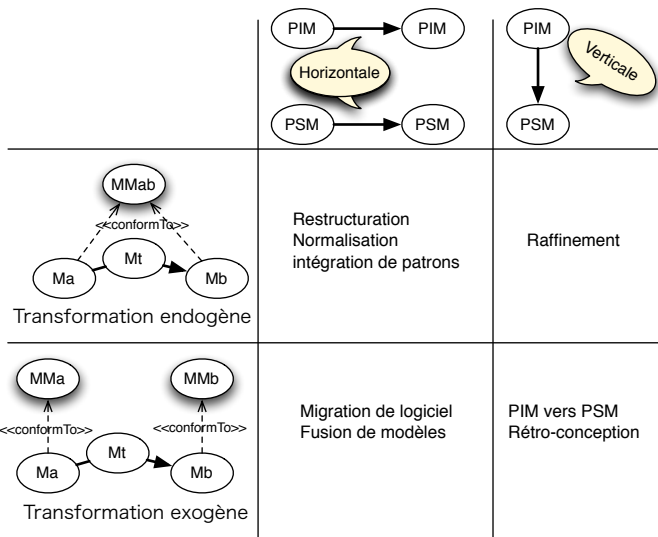
Moyens :

- ▶ Langage généraliste (e.g. Java) : tout est à la charge du programmeur !
- ▶ Langage de transformation déclaratif :
 - ▶ ensemble de règles
 - ▶ une règle transforme un motif sur M_A en éléments de M_B
 - ▶ l'outil de transformation identifie les motifs, applique les règles dans le bon ordre, collecte les résultats et construit M_B
- ▶ Dans tous les cas : importance d'avoir un langage de requête efficace (type OCL)
- ▶ Un standard publié par l'OMG : QVT (Query/View/Transformation)

Propriétés d'une transformation modèle à modèle

- ▶ Transformations :
 - ▶ **endogènes** : mêmes méta-modèles source et cible,
 - ▶ **exogènes** : méta-modèles source et cible différents
- ▶ Transformations **unidirectionnelles** ou **bidirectionnelles**
- ▶ **Traçabilité** : garder un lien (une trace) entre les éléments cibles et les éléments sources correspondants.
- ▶ **Incrémentalité** : une modification du modèle source sera répercutée immédiatement sur le modèle cible.
- ▶ **Réutilisabilité** : mécanismes de structuration, capitalisation et réutilisation de transformation.
- ▶ **Remarque** : Il peut y avoir plusieurs modèles d'entrée et/ou plusieurs modèles de sortie.

Classes de transformation M2M



Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Introduction

Exemples de transformation

Module

Requête (Query)

Bibliothèques (libraries)

Langage de requête d'ATL

Origines d'ATL

- ▶ ATL (ATLAS Transformation Language) est le langage de transformation développé dans le cadre du projet ATLAS.
- ▶ ATL a été développé au LINA à Nantes par l'équipe de Jean Bézivin.
- ▶ Fait désormais partie du projet EMP (Eclipse Modeling Project) : <http://www.eclipse.org/modeling/>
- ▶ ATL se compose :
 - ▶ d'un langage de transformation (déclaratif et impératif) ;
 - ▶ d'un compilateur et d'une machine virtuelle ;
 - ▶ d'un IDE s'appuyant sur Eclipse
- ▶ Pages principales : <http://www.eclipse.org/atl/>
- ▶ Manuel utilisateur et autres documentations accessibles sur <http://www.eclipse.org/atl/documentation/>

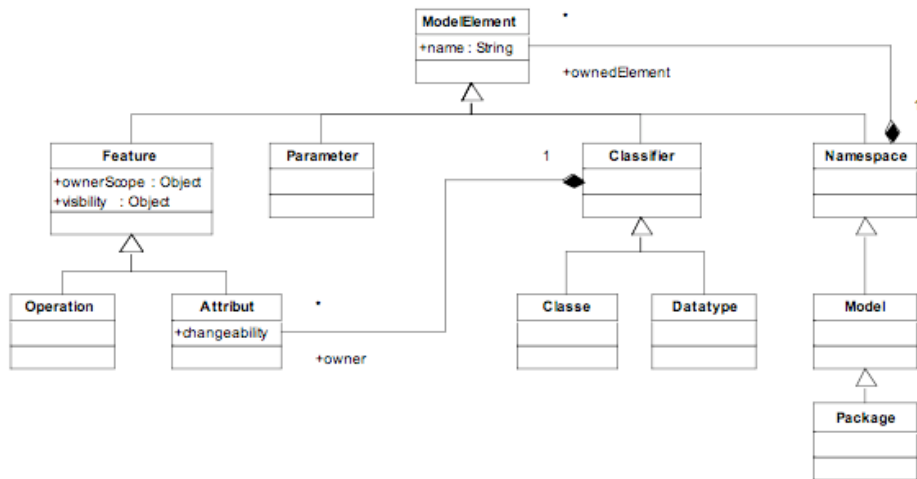
Transformation UML vers Java

Transformer notre modèle UML vers notre modèle Java consiste à :

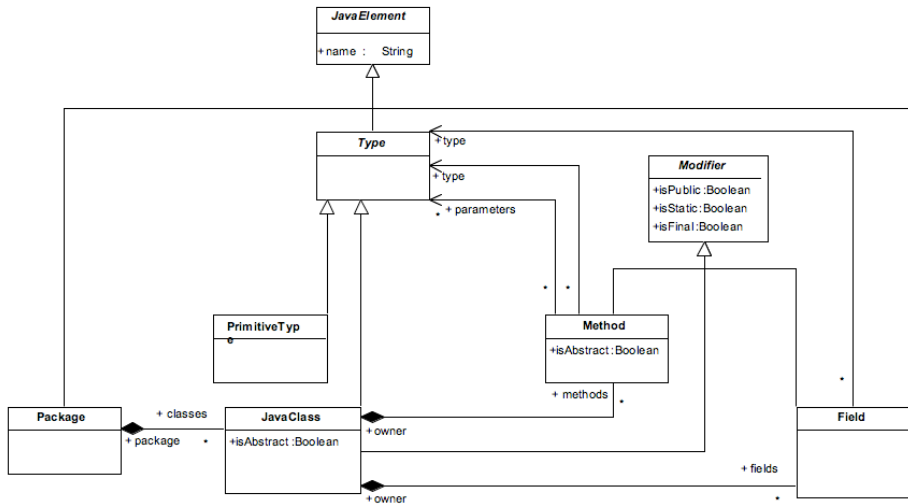
- ▶ Chaque paquetage UML donne un paquetage Java :
 - ▶ les noms sont les mêmes mais en Java, le nom est complètement qualifié.
- ▶ Chaque classe UML donne une classe Java :
 - ▶ de même nom ;
 - ▶ dans le paquetage correspondant ;
 - ▶ avec les mêmes *modifiers*.
- ▶ Chaque DataType UML donne un type 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/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.

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 explique comment les éléments cibles sont initialisés à partir des éléments sources correspondant.

Une règle peut aussi définir :

- ▶ une contrainte sur les éléments correspondant au motif source
- ▶ une partie impérative (T. 101)
- ▶ des variables locales (T. 102)

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`.
⇒ 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 deux règles s'appliquant à une même instance d'un objet. Les filtres de règles doivent être exclusifs.

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.
- ⇒ 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** and **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

Utilisation du module

- Pour appeler un **Helper** ou une **Called rule**, il faut utiliser :

`thisModule.<nom de l'élément_à_appeler>[(<parametres_eventuels>)]`

Operation particulière : **resolveTemp**

- Récupérer un élément crée dans une règle de transformation différente
- Deux arguments
 - Un objet : cibler une règle de transformation
 - Une chaîne de caractère : spécifier l'élément de la règle ciblée à retourner

```
rule A2B {
  from a : MMA!TypeA
  to
    b : MMB!TypeB (
      [...]
    ),
    obj: MMB!TypeObj
      (
        [...]
      )
}
```

```
rule C2D {
  from c : MMA!TypeC
  to
    d : MMB!TypeD (
      attr1 <- thisModule.resolveTemp(
        c.attributeA, "obj")
    )
}
```


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*.

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 transformation 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.

Bibliothèques (libraries)

library JAVA2String;

— *definition of helpers*

...

- ▶ Une bibliothèque permet de définir des helpers qui pourront être (ré)utilisés 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;

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-type (équivalent instanceof de Java) ?
 - ▶ **oclIsTypeOf(t: oclType)** : *self* 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**, $\text{implies}(b : \text{Boolean}) \equiv \text{self} \Rightarrow b$
- ▶ 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()**
- ▶ 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
- ▶ Opérations sur les collections :
 - ▶ **size()**
 - ▶ **includes(o : oclAny), excludes(o : oclAny)**
 - ▶ **count(o : oclAny)**
 - ▶ **includesAll(c : Collections), excludesAll(c : Collections)**
 - ▶ **isEmpty(), nonEmpty()**
 - ▶ **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

<i>exists</i>	body vraie pour au moins un élément de source
<i>forAll</i>	body vraie pour tous les éléments de source
<i>isUnique</i>	body a une valeur différente pour chaque élément de source
<i>any</i>	un élément de source qui satisfait body (OclUndefined sinon)
<i>one</i>	un seul élément de source satisfait body
<i>collect</i>	collection des éléments résultant de l'application de body sur chaque élément de source
<i>select</i>	collection des éléments de source satisfaisant body
<i>reject</i>	collection des éléments de source NE satisfaisant PAS body
<i>sortedBy</i>	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 = anAutohor,
editor: **String** = 'ATL_Eds.'}

Tuple{a = anAutohor, editor = 'ATL_Manual', editor = 'ATL_Eds.'}

Autres types

Map

- ▶ **Map**(type_clé, type_élément) : définit un tableau associative 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, val) 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**

- expression **else** obligatoire
- mot clé **endif** obligatoire

```
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) {  
    instructions1;  
} else {  
    instructions2;  
}
```

► instruction **for**

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

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Le problème

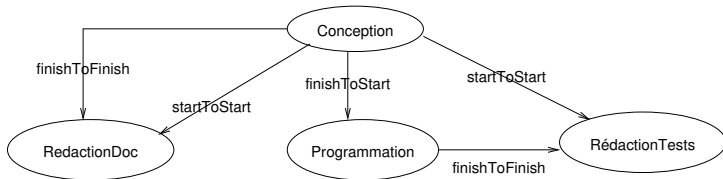
Une solution

Les réseaux de Petri

Schéma général de l'approche

Étude de cas : Vérifier la terminaison de processus

- ▶ **Définition** : Un processus (Process) est composé de plusieurs éléments :
 - ▶ activités (WorkDefinition)
 - ▶ dépendances (WorkSequence) entre activités
 - ▶ ressources (Resource)
 - ▶ et des notes (Guidance)
- ▶ **Exemple** de processus (sans ressources)



- ▶ **Question** : Est-ce qu'un processus (quelconque) peut se terminer ?

Problèmes posés

Pour répondre à la question, il faut :

- ▶ savoir comment sera exécuté un processus (sémantique d'exécution)
- ▶ en particulier, tenir compte des contraintes :
 - ▶ dépendances (*WorkSequence*) : vérifier l'état des activités
 - ▶ ressources (*Resource*) : il faut gérer les allocations et les libérations
- ▶ examiner (toutes) les exécutions possibles pour voir si une au moins termine
- ▶ être efficace, etc.

⇒ **Ceci est difficile !**

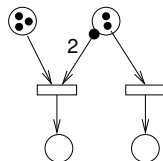
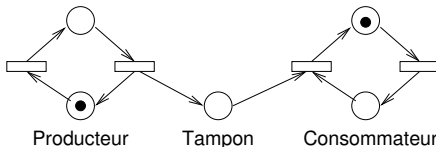
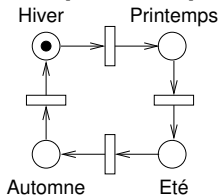
Solution choisie :

- ▶ Définir une **sémantique par traduction**
 - ▶ exprimer la sémantique de SimplePDL en s'appuyant sur un langage formel (ex. les réseaux de Petri).
- ▶ et **s'appuyer sur les outils existants** (ex. le *model-checker* de Tina)

Contrainte : On veut que ça fonctionne pour tout processus ⇒ s'appuyer sur le MM

Les réseaux de Petri

► Quelques exemples :



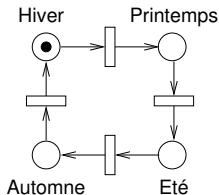
► **Vocabulaire** : place, transition, arc, read_arc, jeton

- Une transition est **franchissable** si toutes les places entrantes contiennent au moins le nombre de jetons indiqué sur l'arc

- **Tirer une transition** : enlever des places entrantes le nombre de jetons correspondant au poids de l'arc et placer dans les places de sorties le nombre de jetons indiqué sur les arcs sortants

Cas du read_arc : les jetons sont laissés dans la place d'entrée.

Syntaxe concrète des réseaux de Petri pour Tina



pl Hiver (1)

tr h2p Hiver → Printemps

tr p2e Printemps → Eté

tr e2a Eté → Automne

tr a2h Automne → Hiver

Expression de propriétés

► Propriétés exprimées en LTL (Logique Temporelle Linéaire) :

```
[ ] <> Ete;    # Toujours il y aura un été
- <> Ete;      # Il n'y aura pas d'été
```

► Pour vérifier ces propriétés, il suffit de taper :

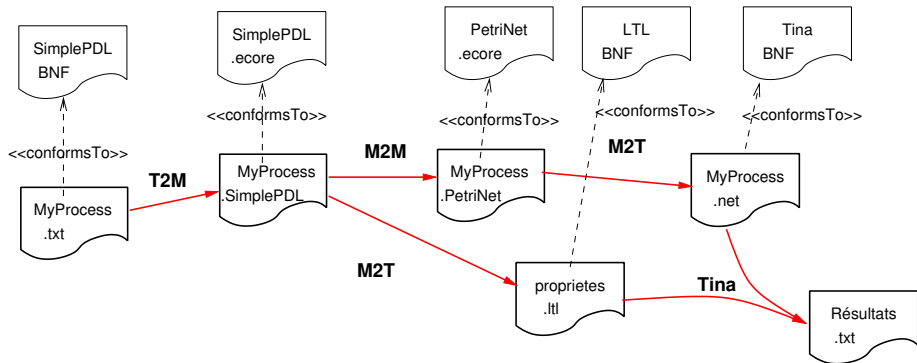
```
tina -s 3 saisons.net saisons.ktz
selt -S saisons.scn saisons.ktz -prelude saisons.ltl
```

► Le résultat est :

```
1 Selt version 2.9.4 -- 11/15/08 -- LAAS/CNRS
2 ktz loaded, 4 states, 4 transitions
3 0.000s
4
5 - source saisons.ltl;
6 TRUE
7 FALSE
8   state 0: Hiver
9   -h2p ... (preserving T)->
10  state 2: Ete
11  -e2a ... (preserving Ete)->
12  state 4: Automne
13  [accepting all]
14 0.000s
```

Propriété 1 vraie
Propriété 2 fausse
un contre-exemple fourni

Schéma général



Des processus aux réseaux de Petri

1. Comment traduire un modèle de processus en réseau de Petri ?
On ne tiendra pas compte des ressources.
2. Quelle est la propriété LTL qui indique qu'un processus se termine ?
3. Comment obtenir un ordonnancement des activités qui termine le processus ?
4. Comment prendre en compte les ressources ?

Sommaire

Motivation

De L'IDM aux DSL

Syntaxe abstraite (avec Ecore)

Sémantique statique (avec OCL)

Syntaxe concrète textuelle (avec Xtext)

Syntaxe concrète graphique (avec Sirius)

Transformations Modèle à Texte (avec Acceleo)

Transformations Modèle à Modèle

Le langage ATL

Étude de cas : vérification de processus

Conclusion : Vers les langages dédiés (DSL)

- ▶ DSL : (Domain Specific Language) Langage dédié à un domaine métier
 - ▶ Augmenter le niveau d'abstraction
 - ▶ Permettre à un utilisateur d'exprimer directement son besoin !
 - ▶ Plus d'autonomie, moins de dépendance vis à vis des « informaticiens »
- ▶ Définir le méta-modèle qui capture les concepts du métier.
 - ▶ Utilisation de langage de méta-modélisation : Ecore, EMOF, etc.
- ▶ Exprimer les propriétés non capturées par le méta-modèle.
 - ▶ Utilisation d'OCL ou autre (transfo vers modèle d'erreur, ...)
- ▶ Définir une syntaxe concrète textuelle et/ou graphique (ou plusieurs !).
- ▶ Ajouter des outils autour de ce DSL :
 - ▶ outils de simulation : animer le modèle pour le valider, mesurer ses performances...
 - ▶ outils de vérification : vérifier des propriétés sur le modèle métier
 - ▶ outils de génération (vers code ou autres modèles)
- ▶ Une solution consiste à réutiliser des outils existants
 - ▶ transformer le modèle métier en un modèle exploitable par l'outil cible