Métamodélisation et sémantique statique
Corrigé

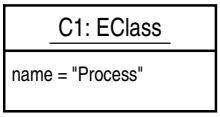
Exercice 1 : Comprendre SimplePDL

EMOF (OMG) ou Ecore (Eclipse) sont des méta-métamodèles. Un extrait d'Ecore est donné à la figure 1. Leur objectif est de permettre la définition de métamodèles. La figure 2 donne le métamodèle du langage SimplePDL, un langage très simplifié de description des procédés de développement. Ce métamodèle est conforme à EMOF/Ecore. Il a été dessiné en utilisant les conventions traditionnellement utilisées qui sont empruntées au diagramme de classe UML.

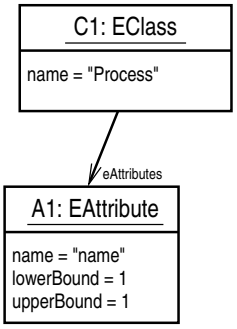
1.1. *Concepts Ecore.* Le métamodèle de SimplePDL est conforme à Ecore. Indiquer, pour chaque élément du métamodèle de SimplePDL, l'élément d'Ecore auquel il « correspond ».

Solution : Pour montrer que le métamodèle de SimplePDL est conforme à Ecore, il faut montrer que toutes éléments de SimplePDL sont des instances de concepts de Ecore. On peut le faire en construisant le diagramme d'objets ECore qui correspond à SimplePDL. Faisons le progressivement.

Commençons par le concept (formalisme d'une classe) Process. C'est une instance de EClass. Le nom (name) de cette EClass est "Process". D'après Ecore, l'attribut *name* est la seule information obligatoire.

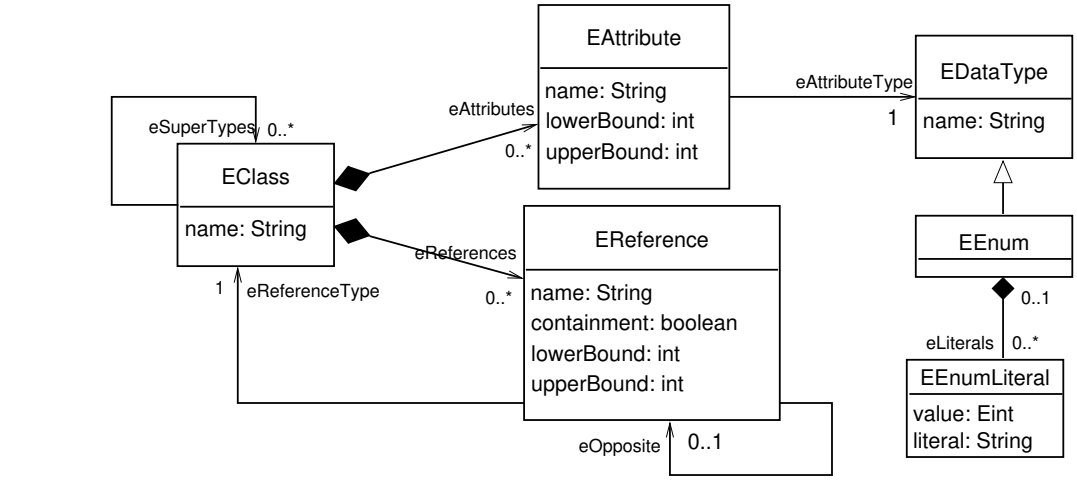


Le concept Process a un attribut *name*. C'est une instance de *EAttribute*. Pour un *EAttribute*, on doit préciser le nom (*name*) qui est ici *name* (le nom du processus) mais aussi la multiplicité (*lowerBound* et *upperBound*). Ici la multiplicité est 1 exactement, donc 1 pour les deux attributs.



On a traité *name* mais pas encore *String*. Il s'agit du type de l'attribut qui correspond à la référence EDataTYpe qui conduit à *EDataType*. Ici, le type est donc String, un EDataType (un type de base). On peut donc compléter notre diagramme d'objets.

FIGURE 1 – Version très simplifiée du méta-métamodèle Ecore



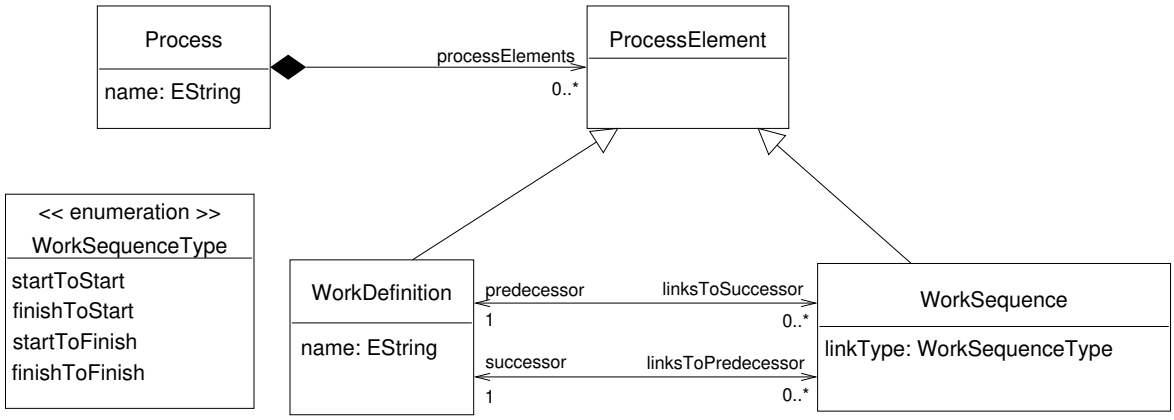
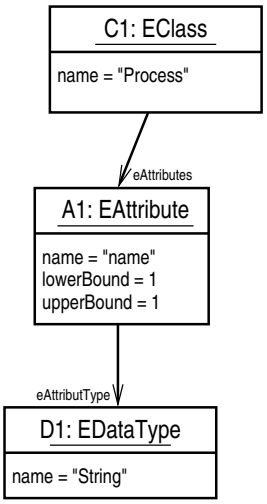
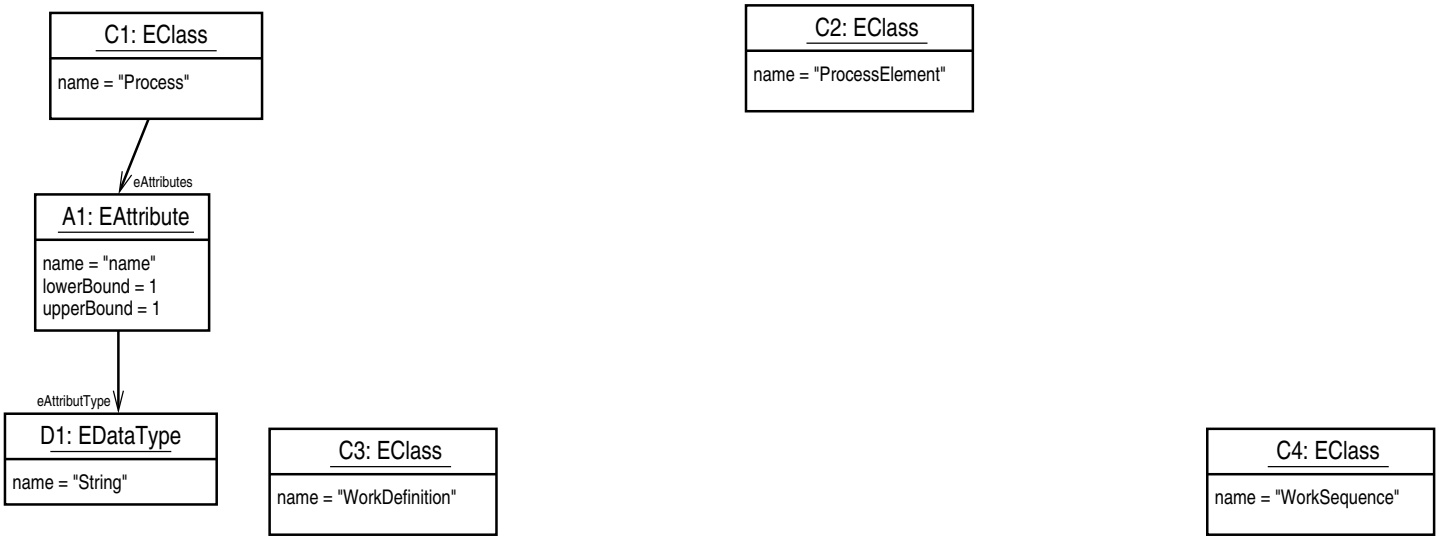


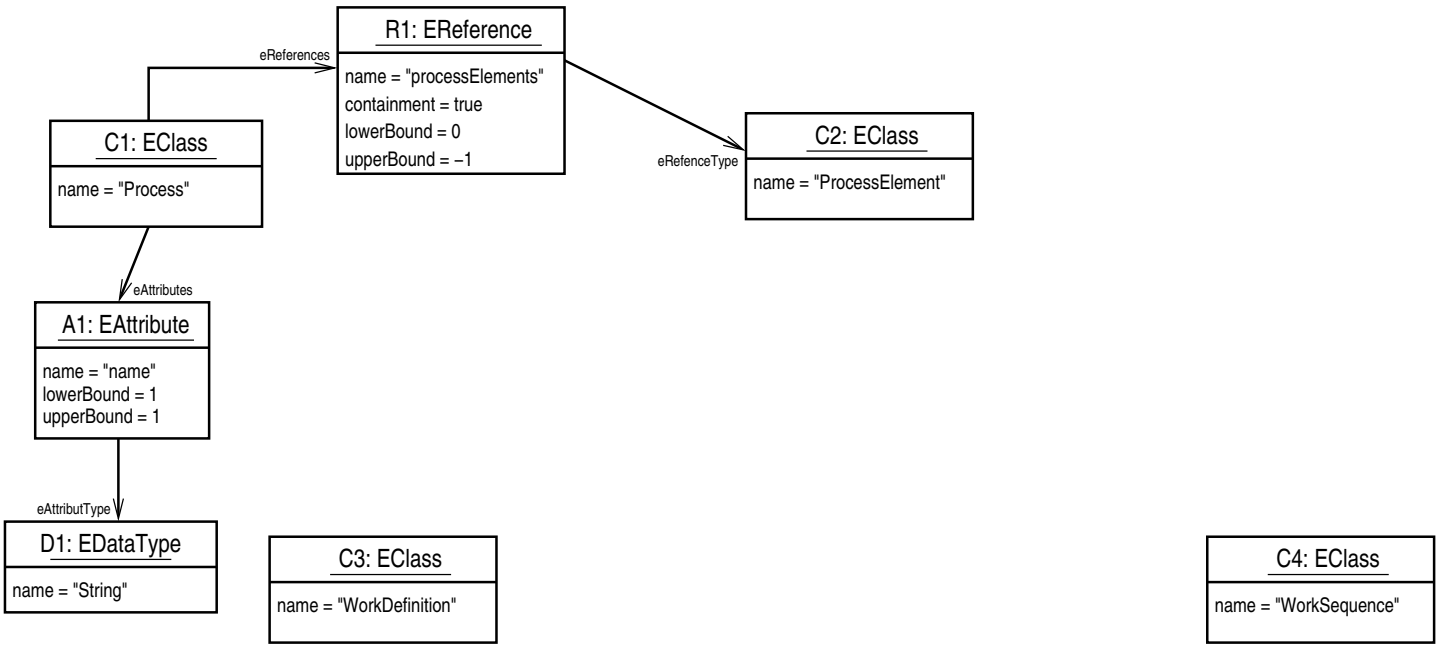
FIGURE 2 – Méta-modèle de SimplePDL conforme à EMOF/Ecore



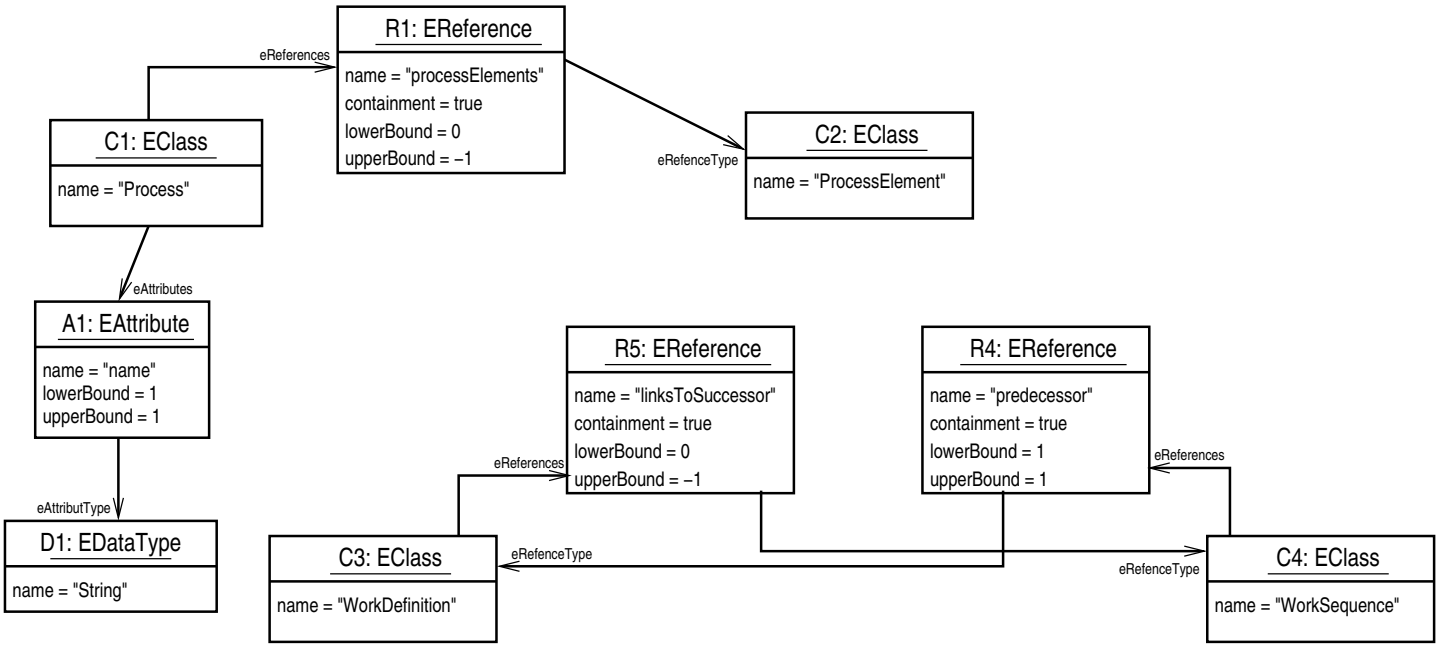
On pourrait faire pareil pour les autres classes (ProcessElement, WorkDefinition et WorkSequence). Nous ne détaillons pas les attributs pour éviter d'alourdir le diagramme. Notons que ProcessElement devrait être déclaré abstrait ou comme une interface mais ces notions ne sont pas dans l'extrait d'Ecore présenté dans ce sujet.



Intéressons nous maintenant à la relation entre Process et ProcessElement. Ceci correspond à une *EReference* en Ecore. Une *EReference* se distingue d'un *EAttribute* par son type, une *EDataType* pour le second, une *EClass* pour le premier. Notons qu'on a un attribut supplémentaire qui est *containment*. Il correspond à une composition en UML et se dessine traditionnellement avec un losange plein.

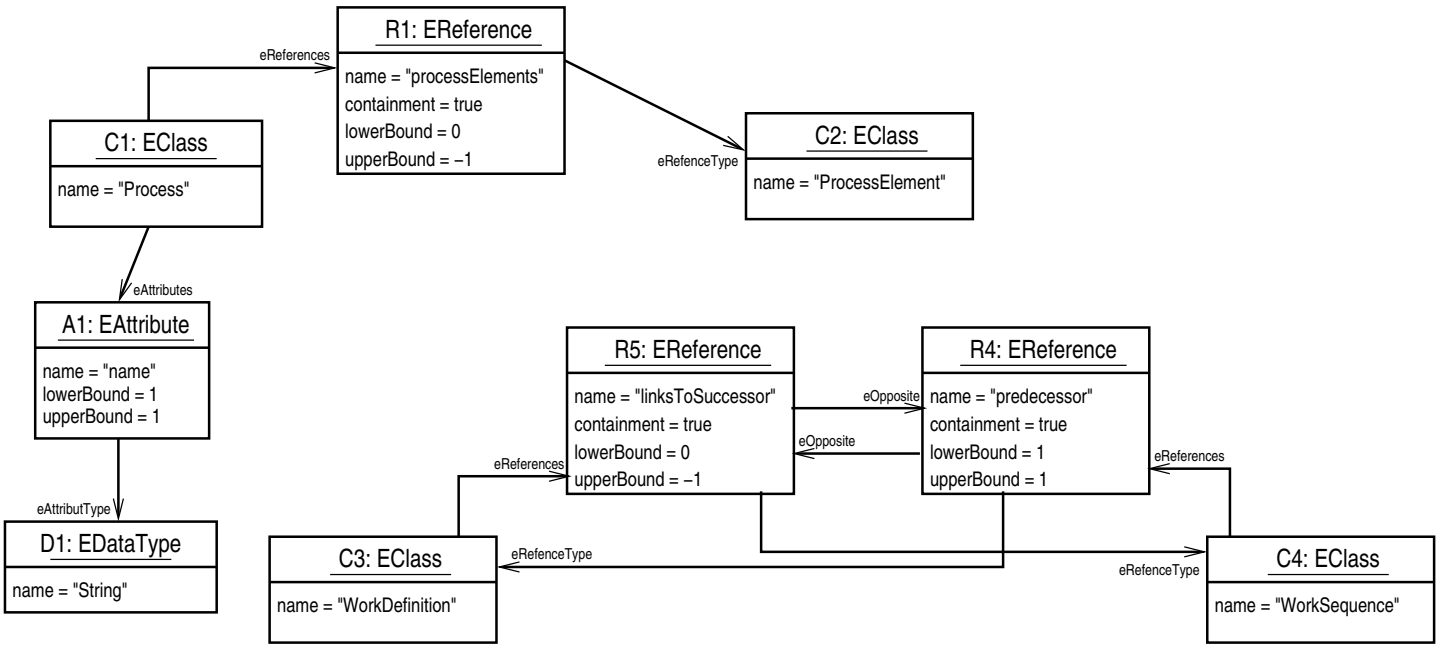


Considèeons maintenant la relation entre *WorkDefinition* et *WorkSequence*. On remarque qu'elle est dans les deux sens (un nom à chaque extrémité). Ceci correspond à deux références que l'on pourrait représenter ainsi.

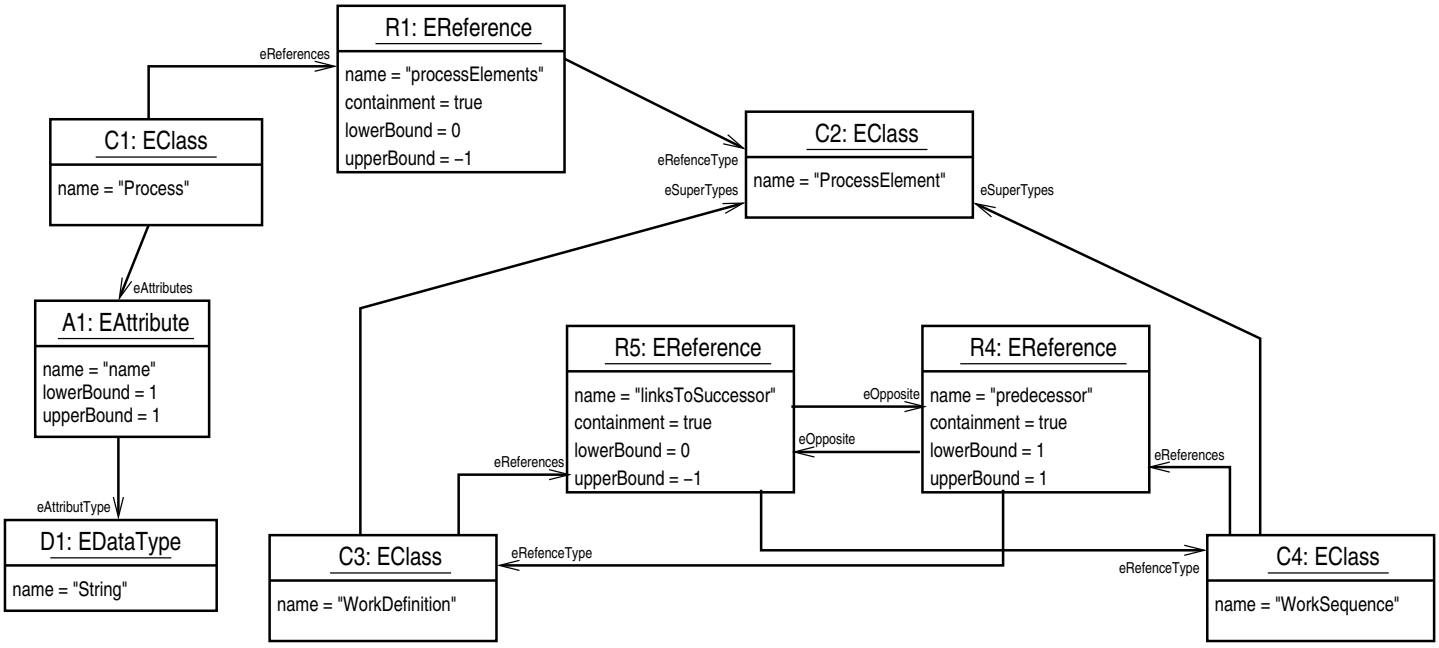


Mais ce n'est pas suffisant. Considérons une *WorkSequence* ws1 qui a comme *predecessor* wd3. Alors on devrait avoir dans les *linksToSuccessor* de wd3 la *WorkSequence* ws1. Avoir deux références indépendantes ne le permet pas. Il faut les lier. C'est le sens de la référence (*EReference*) *eOpposite* qui doit être définie sur les deux références avec comme cible l'autre référence.

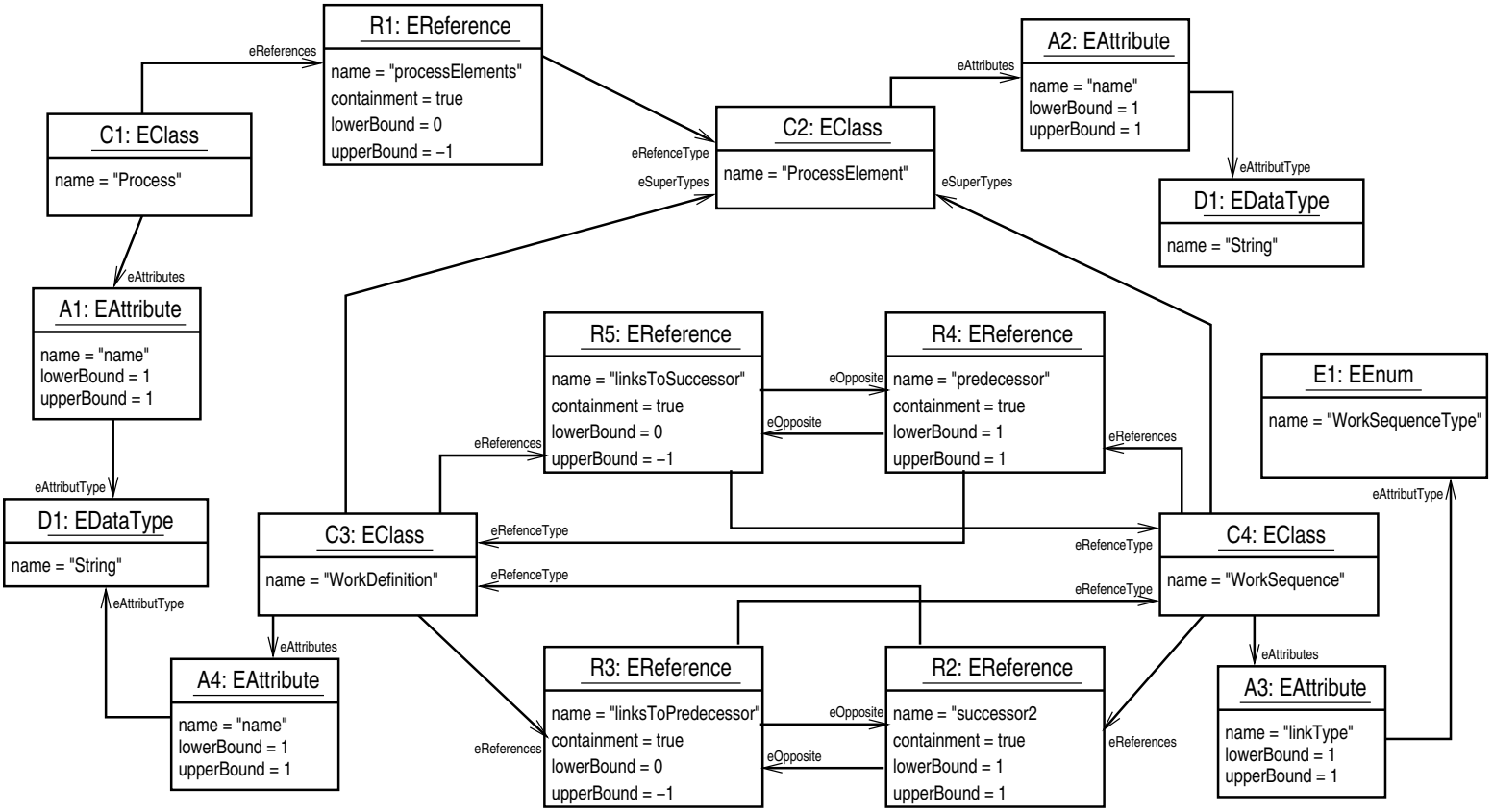
Voici le diagramme d'objet qui inclut ces deux nouvelles références.



Il nous reste à considérer l'héritage entre *WorkDefinition* (ou *WorkSequence*) et *ProcessElement*. Elle se traduit par la référence *eSuperTypes* de *EClass*. On obtient alors le diagramme suivant.



Enfin, voici le diagramme d'objet complet ou presque car il manque la définition des littéraux du type énuméré *WorkSequenceType*.



Notons que ce diagramme d'objet n'est pas très lisible. Il est donc souhaitable d'utiliser une notation plus agréable à lire comme par exemple le diagramme de classe utilisé traditionnellement comme à la figure 2. Le principe est de dire comment nous allons représenter les éléments de ce diagramme d'objet. Naturellement, on le fait en s'appuyant sur son métamodèle : les objets de même type seront représentés de la même manière :

- Une EClass comme une classe.
- Un *EAttribute* comme un attribut de la classe qui le contient ;
- les *EReference* comme des relations orientées de l'EClass qui contient la référence vers la EClass référencée par *eReferenceType*,
- la référence *eSuperTypes* d'une *EClasse* comme une relation d'héritage,
- l'attribut *containment* d'une *EReference* à vrai comme une composition, etc.

Nous verrons l'outil Sirius qui permet de définir une telle *syntaxe concrète graphique*.

Une alternative consiste à adopter une approche de type base de données. Chaque concept (boîtes) de Ecore donne une table dont les colonnes sont ses *eAttributes* et *eReferences*. Chaque objet du diagramme d'objet est une ligne dans l'une de ses tables.

EClass				
ID	name	eSuperType	eAttributes	eReferences
C1	Process		A1	R1
C2	ProcessElement		A2	
C3	WorkDefinition	C2	A4	R2, R4
C4	WorkSequence	C2	A3	R3, R5

EAttribute				
ID	name	lowerBound	upperBound	eAttributeType
A1	name	1	1	D1
A2	name	1	1	D1
A3	linkType	1	1	E1
A4	name	1	1	D1

EDataType	
ID	name
D1	String

EEnum		
ID	name	eLiterals
E1	LinkType	L1, L2, L3, L4

EEnumLiteral	
ID	name
L1	startToStart
L2	finishToStart
L3	startToFinish
L4	finishToFinish

EReference					
ID	name	lowerBound	upperBound	eReferenceType	eOpposite
R1	processElements	0	$\star$ (-1)	C2	
R2	successor	1	1	C4	R3
R3	linksToPredecessor	0	$\star$ (-1)	C3	R2
R4	predecessor	1	1	C4	R5
R5	linksToSuccessor	0	$\star$ (-1)	C3	R4

1.2. *Signification de SimplePDL.* Expliquer ce que décrit le métamodèle SimplePDL.
Solution : Il y les concepts de Process, WorkDefinition, WorkSequence....

Ici on ne peut qu'être très descriptif. Le sens des éléments n'est que faiblement capturer par leur nom. Il est donc nécessaire de les expliquer, soit en langage naturel, soit plus formellement.

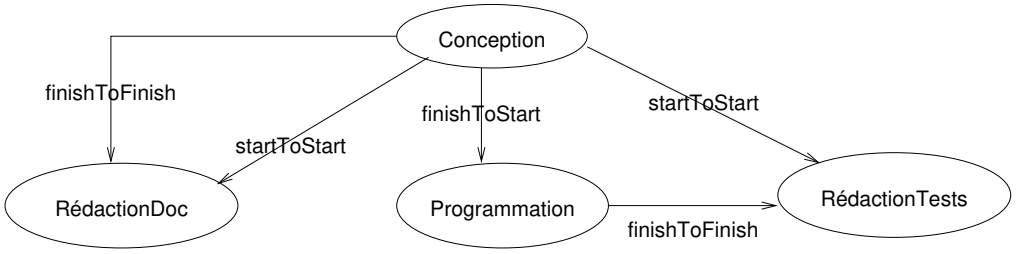
1.3. *Description d'un procédé particulier.* On s'intéresse à un procédé simple composé de quatre activités : concevoir, programmer, tester et documenter. Programmer ne peut commencer que

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 peut s'achever que si la programmation est terminée. Le test ne peut être terminé que si la conception et la programmation sont terminées.

1.3.1. Dessiner le modèle de ce procédé. On utilisera une boîte pour représenter une activité et une flèche pour les relations de précédence.

Solution :

Ici, on choisit une syntaxe concrète graphique pour représenter ce processus. On a forcément besoin d'une syntaxe concrète pour échanger sur des concepts. Elle peut être textuelle, graphique, tabulaire, etc.



Remarque : Ici n'est pas représenté l'objet *Process* qui correspond à ce processus. Il est essentiel car on commence par instancier un premier objet, ici un objet de type *Process*. À partir de cet objet nous pourrons créer d'autres éléments s'ils sont accessibles au moyen d'une relation de composition. Ici, on peut donc créer des *ProcessElement*, donc des *WorkDefinition* et des *WorkSequence*.

Si on avait créé au début une *WorkDefinition*, on aurait pas pu créer d'autres objets car sa métaclasse ne possède pas de référence de type composition.

1.3.2. Montrer que le modèle de procédé ainsi construit est bien conforme à SimplePDL.

Solution : Il s'agit de montrer que tous les éléments sont instances d'un élément de SimplePDL et que les contraintes sur les attributs et les références sont respectées.

On peut adopter la même démarche que tout à l'heure et construire un diagramme d'objet qui doit être conforme au diagramme de classe correspondant au métamodèle SimplePDL.

1.4. Expliquer les contraintes OCL portant sur SimplePDL données ci-dessous.

```
context ProcessElement
def: process(): Process =
      Process.allInstances()
            ->select(p | p.processElements->includes(self))
            ->asSequence()->first()

context WorkSequence
inv previousWDinSameProcess: self.predecessor.process() = self.process()
inv nextWDinSameProcess: self.successor.process() = self.process()
```

Solution : On définit une nouvelle méthode sur l'élément ProcessElement. Elle s'appelle *process*, ne prend pas de paramètres et retourne un objet de type Process.

allInstances() est une méthode qui permet de récupérer tous les objets d'un modèle qui sont du type qui précède (ici *Process*). Il est préférable d'éviter d'utiliser cette méthode car elle est

coûteuse, d'autant plus que le modèle est gros.

allInstances() retourne une collection d'objet. On doit donc ensuite utiliser le sélecteur `->`. L'opérateur *select* permet de conserver de la collection cible que les éléments qui respectent la propriété, ici les *ProcessElement* p qui sont inclus dans les *processElements* de ce *Process*, celui sur lequel *process()* est appelé. *self* est l'équivalent de *this* en Java.

asSequence() permet de transformer la collection cible en liste puis *first()* retourne le premier élément de cette séquence. Il correspond au processus qui contient ce *ProcessElement*.

On vient donc de définir, sous forme d'une méthode, l'équivalent de la référence opposée à *processElements*. Ceci permettra de retrouver le processus d'une *WorkDefinition* ou d'une *WorkSequence*.

Nous avons ensuite deux invariants qui sont définis dans le contexte d'une *WorkSequence*. Ils expriment que l'activité précédente (resp. suivante) doivent appartenir au même processus que la dépendance.

Les contraintes OCL sont en particulier utilisées pour définir des invariants sur une modèle ou un métamodèle. Ceci permet de capturer les contraintes qui n'ont pas pu l'être au niveau du métamodèle ou du méta-métamodèle.

D'un point de vue pratique, il est souhaitable que les propriétés soient définies dans le contexte le plus précis possible. En effet, les outils de vérification OCL vont évaluer les invariants sur tous les éléments du modèle correspondant au contexte de la clause OCL. Si l'invariant n'est pas satisfait, l'élément correspondant sera signalé comme faux (avec une croix rouge par exemple pour l'éditeur graphique). Ainsi, on aurait pu se placer dans le contexte d'un processus pour vérifier que les *WorkDefinition* avaient même processus que les *WorkSequence* qui leur sont liées. Cependant, en cas d'erreur, c'est le process qui serait signalé comme faux sans plus de détail. Tel que défini ci-dessus, ce sont les *WorkSequence* qui seront marquées comme fausses. Ce sera beaucoup plus précis pour le concepteur.

1.5. *Compléter les contraintes de SimplePDL.* Exprimer les contraintes suivantes sur SimplePDL et les évaluer sur des exemples de modèles de procédé :

1. une dépendance ne peut pas être réflexive.

   Solution :

   ```
   context WorkSequence
   inv notReflexive: self.predecessor <> self.successor;
   ```

2. deux sous-activités différentes d'un même processus ne peuvent pas avoir le même nom.

   Solution :

   ```
   context Process
   inv uniqNames: self.processElements
         ->select(pe | pe.oclIsKindOf(WorkDefinition))
         ->collect(pe | pe.oclAsType(WorkDefinition))
         ->forAll(w1, w2 | w1 = w2 or w1.name <> w2.name);

   context WorkDefinition
   inv uniqNames: self.process.processElements
   ```

```
    ->select(pe | pe.oclIsKindOf(WorkDefinition))
    ->collect(pe | pe.oclAsType(WorkDefinition))
    ->forAll(w | self = w or self.name <> w.name);
```

La deuxième a un contexte plus précis et est préférable.

3. le nom d'une activité doit être composé d'au moins un caractère.
   Solution :

```
context Process
inv nameIsDefined: -- is the name of process correct?
   if self.name.oclIsUndefined() then
      false
   else
      self.name <> ''
   endif
```

Attention : Le **or** d'OCL n'a pas d'évaluation en court-circuit. Elle est toujours complètement évaluée. Il faudra recourir à un **if** si une évaluation partielle est nécessaire comme ici : on ne pourra accéder au nom que si il est défini.

Remarque : Le métamodèle peut imposer que l'attribut *name* doit être défini. Si le modèle est conforme, on est donc sûr que l'attribut *name* est défini. On peut cependant demander l'évaluation de règles OCL sur un modèle non conforme. La solution avec **if** est donc plus robuste et préférable.

4. les dépendances du modèle de processus ne provoquent pas de blocage.
   Solution : C'est compliqué à exprimer, en particulier si on ajoute les ressources. Il est donc préférable d'utiliser d'autres techniques, par exemple la traduction d'un modèle de processus en réseau de Petri pour utiliser les outils de model checking tels que Tina.

Exercice 2 : Mise en œuvre avec OCLinEcore

OCLinEcore propose un éditeur qui offre une syntaxe concrète textuelle pour un métamodèle Ecore. Il permet d'ajouter des éléments OCL directement sur ce métamodèle. Ils sont ensuite sauvegardés dans des éléments `EAnnotation` dans le `.ecore`. Le listing 1 présente un exemple avec SimplePDL.

2.1. Expliquer les différents éléments présents sur le listing 1.

2.2. Comparer les approches OCL et OCLinEcore.

Solution : OCL : les contraintes sont définies à côté du MM (d'où le contexte qui doit être explicite). Ceci permet par exemple d'avoir plusieurs jeux de contraintes sur un même MM. On peut ensuite choisir d'utiliser l'un ou l'autre.

OCLinEcore propose une syntaxe concrète qui peut être utile (éditeur habituelle, gestion de version, etc).

Mettre les éléments OCL directement sur les éléments concernés peut être plus pratique.

Exercice 3 : Méta-modèle des réseaux de Petri

L'objectif de cet exercice est de construire un métamodèle des réseaux de Petri.

Solution : Pas de solution proposée car ce travail sera à faire en TP et pendant le mini-projet.

Listing 1 – Le métamodèle SimplePDL en OCLinEcore avec des éléments OCL

```
package simplepdl : simplepdl = 'http://simplepdl'
{
  enum WorkSequenceType { serializable }
  {
    literal startToStart;
    literal finishToStart = 1;
    literal startToFinish = 2;
    literal finishToFinish = 3;
  }
  class Process
  {
    attribute name : String;
    property processElements : ProcessElement[*] { ordered composes };
  }
  abstract class ProcessElement
  {
    property process : Process { derived readonly transient volatile !resolve }
    {
      derivation: Process.allInstances()
          ->select(p | p.processElements->includes(self))
          ->asSequence()->first();
    }
  }
  class WorkDefinition extends ProcessElement
  {
    property linksToPredecessors#successor : WorkSequence[*] { ordered };
    property linksToSuccessors#predecessor : WorkSequence[*] { ordered };
    attribute name : String;
  }
  class WorkSequence extends ProcessElement
  {
    attribute linkType : WorkSequenceType;
    property predecessor#linksToSuccessors : WorkDefinition;
    property successor#linksToPredecessors : WorkDefinition;
    invariant previousWDinSameProcess: self.process = self.predecessor.process;
    invariant nextWDinSameProcess: self.process = self.successor.process;
  }
  class Guidance extends ProcessElement
  {
    property element : ProcessElement[*] { ordered };
    attribute text : String;
  }
}
```

3.1. Proposer un métamodèle des réseaux de Petri. On utilisera Ecore.
3.2. Dessiner quelques modèles de réseau de Petri qui sont conformes au métamodèle défini mais non valides.
3.3. Définir des contraintes OCL pour exprimer les propriétés qui n'ont pas été capturées par le métamodèle ECore.