Projet Annuel ISTY 3
Transformation BPEL en Graphe

Auteurs :
Isabelle BALOCHE
Eric DRAPERI

Encadrant :
Daniela GRIGORI

16 mars 2006

TABLE DES MATIERES

1 INTRODUCTION 3
1.1 OBJET DU DOCUMENT 3
1.2 CONTENU – STRUCTURE 3
2 INTRODUCTION AU BPEL 4
3 STRATEGIE DE TRANSFORMATION 6
3.1 TRANSFORMATION – « PICK » 6
3.2 TRANSFORMATION – « SEQUENCE » 7
3.3 TRANSFORMATION – « SWITCH » 7
3.4 TRANSFORMATION – « WHILE » 9
3.5 TRANSFORMATION – « FLOW » 9
3.6 TRANSFORMATION – « EMPTY » 12
3.7 TRANSFORMATION – « SCOPE » 14
3.7.1 Flat Graph 14
3.7.2 Overlapping graph 15
4 IMPLEMENTATION 16
4.1 BPEL META-MODEL 16
4.1.1 General Schema 16
4.1.2 Sequence 17
4.1.3 While 17
4.1.4 Switch 18
4.1.5 Flow 18
4.1.6 Pick 19
4.1.7 Scope 20
4.2 LOADING OF THE BPEL 21
4.2.1 General Schema 21
4.2.2 Configuration of IActivityReader 22
4.2.3 Utilization 22
4.3 GRAPH META MODEL 23
4.4 TRANSFORMATION EN GRAPHE .... **¡ERROR! MARCADOR NO DEFINIDO.**
*4.4.1 Schéma général .... **¡Error! Marcador no definido.***

4.4.2 Stratégie et contexte de transformation.....25
4.4.3 Transformation des liens « Link ».....27
4.4.4 Configuration des « IActivityTransform ».....28
4.4.5 Utilisation.....28
4.5 ECRITURE DU GRAPHE DANS UN FICHIER.....29
4.5.1 Schéma général.....29
4.5.2 Format de sortie.....29
4.5.3 Utilisation.....30
5 JEUX DE TESTS.....**31**
5.1 TESTS UNITAIRES.....31
5.2 TESTS D'INTEGRATION.....31
TABLE DES ILLUSTRATIONS.....**32**
ANNEXE : TRANSFORMATION COMPLEXE.....**33**

1 Introduction

1.1 Objet du document

Ce document présente l'implémentation du projet annuel de troisième année d'ISTY. Le but de ce projet est de développer un algorithme pour déduire, à partir d'un modèle BPEL, un graphe dont les nœuds sont des activités (services web) et les arcs représentent les dépendances d'ordre entre elles et les échanges de données. Ce projet correspond à la première partie d'un autre projet ayant pour objectif de comparer deux processus BPEL afin de vérifier la compatibilité des deux processus dans le contexte de collaboration entre deux entreprises

1.2 Contenu – structure

Le chapitre courant fournit l'information sur le présent document : objet ; contenu ; structure.

Le chapitre 2 donne une introduction au BPEL.

Le chapitre 3 présente la transformation du modèle BPEL en graphe.

Le chapitre 4 présente l'implémentation du projet.

Le chapitre 5 présente brièvement les jeux de tests utilisés pour valider le fonctionnement de la bibliothèque réalisée.

2 Introduction au BPEL

Un processus web peut être décrit en utilisant le langage BPEL qui décrit comment des services web sont connectés ensemble pour fournir un modèle de processus de haut-niveau. BPEL est l'acronyme de **B**usiness **P**rocess **E**xecution **L**anguage. Il s'agit d'un langage commun normalisé ayant pour objectif l'optimisation de la gestion des processus métiers de plus en plus complexes. Le BPEL est une spécification, conçue par IBM, BEA et Microsoft, relative à un langage basé sur XML qui décrit l'interaction des processus métiers basés sur les Services Web. La représentation XML de ce processus peut être déployée sur n'importe quel moteur de processus métier. Le BPEL utilise le langage WSDL pour décrire les actions d'un processus et est constitué à partir de deux standards : WSFL d'IBM et XLANG de Microsoft.

Les entreprises utilisant le langage BPEL4WS peuvent définir leurs processus métiers et en garantir l'interopérabilité à l'échelle de l'entreprise et avec leurs partenaires commerciaux, au sein d'un environnement de Services Web.

Un processus BPEL4WS est donc constitué de deux parties:

La partie *definition* : l'interface du processus, définition en WSDL des différentes opérations utilisées

La partie *process* : fichier BPEL qui comprend :

- une partie de définition de ces éléments : variables, partnerLink….;
- la description du processus et de l'enchaînement de ses actions.

L'élément premier d'un processus BPEL est une « activité », qui peut être l'envoi d'un message, la réception d'un message, l'appel d'une opération (envoi d'un message, attente d'une réponse), ou une transformation de données. L'activité est définit par la combinaison de Services Web.

Les activités sont de types basiques ou structures. Les activités basiques sont les suivantes :

- **Assign** : peut être utilisé pour mettre à jour les valeurs de la variable avec la nouvelle donnée.
- **Empty** : permet d'insérer une « non opération » dans le processus métier, cela est très utile pour la synchronisation d'activités concurrentes.
- **Invoke** : permet au processus métier d'invoquer une opération question/réponse sur un portType offert par un partenaire.
- **Receive** : permet au processus métier de se mettre en attente bloquante en attendant l'arrivée du message désiré. Le processus va donc recevoir l'invocation d'un partenaire.
- **Reply** : permet au processus métier d'envoyer un message en réponse à un message qui a été reçu avec un <receive>. La combinaison d'un <receive> et d'un <reply> forme une opération question/réponse sur le portType du WSDL pour le processus. Le processus va donc envoyer un message de réponse à l'invocation du partenaire.
- **Terminate** : permet de terminer l'instance d'un processus.

– **Throw** : génère une faute à l'intérieur du processus métier. Va donc servir Pour la détection et la gestion d'erreur.

– **Wait** : permet d'attendre pendant une période donnée ou jusqu'à ce qu'un certain temps soit passé.

Les activités structurées sont les suivantes :

– **Compensate** : utilisé pour invoquer une « compensation » sur son scope qui a déjà été effectué normalement. Il peut être invoqué seulement à partir du traitement d'un événement ou d'une autre action de compensation. Va être utilisé pour défaire une action réalisée précédemment

– **Flow** : permet de spécifier un ou plusieurs activités à effectuer de manière concurrentielle. Des liens peuvent être utilisés entre les activités concurrentes pour définir des structures arbitraires. On va donc avoir exécution des activités en parallèle.

– **Pick** : permet de mettre en attente (blocage) d'un message ou pendant la durée d'un time-out d'une alarme. Quand un de ces évènements se produit, l'activité associée est exécutée et le <pick> est terminé.

– **Scope** : permet de définir l'activité imbriquée avec ses propres variables associées, ses traitements d'évènements. On va donc avoir un contexte associée à une structure ou à un ensemble d'activités élémentaires.

– **Sequence** : permet de définir une collection d'activités qui doit être exécutée de manière séquentielle

– **Switch** : supporte les comportements conditionnels. L'activité consiste en une liste ordonnée de un plusieurs branchements conditionnels définis par des éléments case, suivi optionnellement par un autre branchement.

– **While** : permet d'indiquer une activité qui doit être répétée jusqu'à ce qu'un certain critère de succès soit atteint.

3 Stratégie de transformation

Nous n'afficherons dans le graphe de processus uniquement les éléments qui sont pertinents.

3.1 Transformation – « Pick »

The Pick activity is modelled in the following way: a connector "WAIT" is created, this one will have links towards the corresponding functions, either with the arrival of a particular message (onMessage) or after a temporal event (onAlarm). Once the function is finished the pick is leaved (with the ending OR).

```
<process name="pickProcess" …>

   <pick …>
      <onAlarm until="timeExpr" >
            <reply name="reply_3" />
      </onAlarm>
      <onMessage partnerLink="partner_1" operation="receiveOrder_1" >
            <reply name="reply_1" />
      </onMessage>
      <onMessage partnerLink="partner_2" operation="receiveOrder_2" >
            <reply name="reply_2" />
      </onMessage>
   </pick>
</process>
```

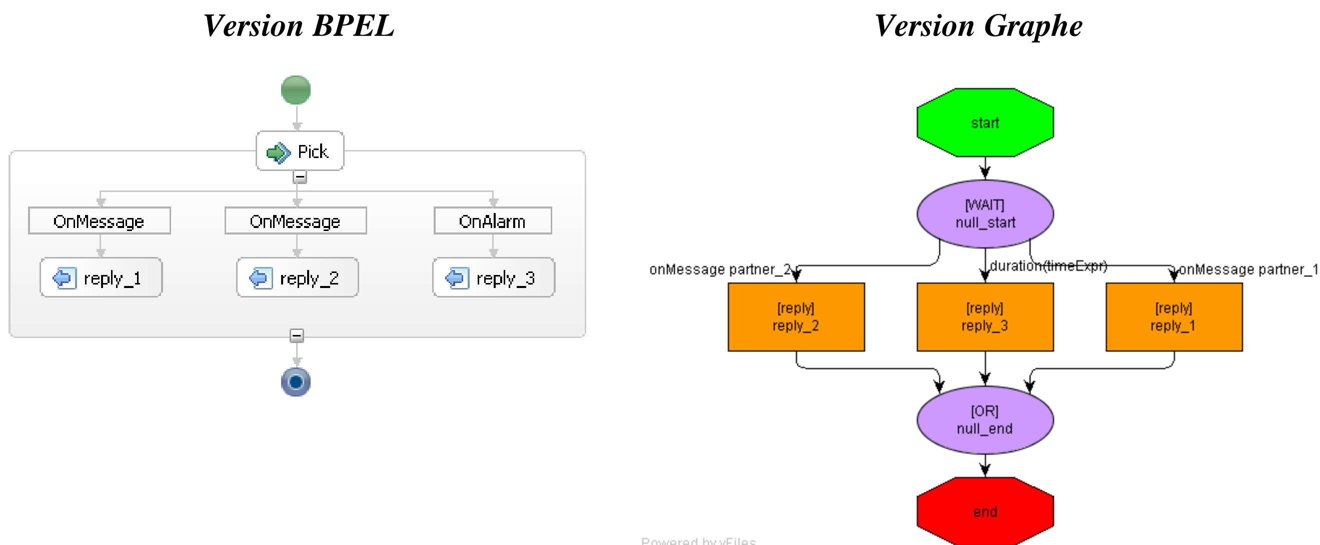


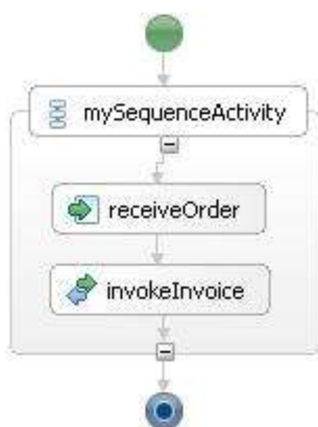
Figure 1 : Transformation – « Pick »

3.2 Transformation – « Sequence »

A sequence is transformed into a succession of functions which are followed in sequentially.

```
<process name="sequenceProcess" …>
   <sequence name="mySequenceActivity">
     <receive name="receiveOrder" />
     <invoke name="invokeInvoice" />
   </sequence>
</process>
```

Version BPEL ***Version Graphe***



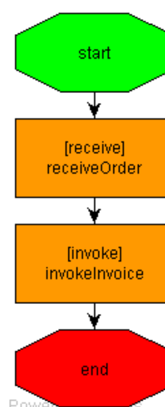


Figure 2 : Transformation – « Sequence »

3.3 Transformation – « Switch »

The Switch activity is transformed in the following way: a XOR connector is created for the switch entry, according to the conditions posted on the arcs one and only one of the functions will be executed. Once this function finished, the switch is finished (with the XOR of end)

```
<process name="swicthProcess" …>
   <switch name="mySwitchActivity">
     <case condition="[MaConditon_1]">
         <receive name="receiveOrder_1" />
     </case>
     <case condition="[MaConditon_2]">
         <receive name="receiveOrder_2" />
     </case>
     <otherwise>
         <receive name="receiveOrder_3" />
     </otherwise>
   </switch>
</process>
```

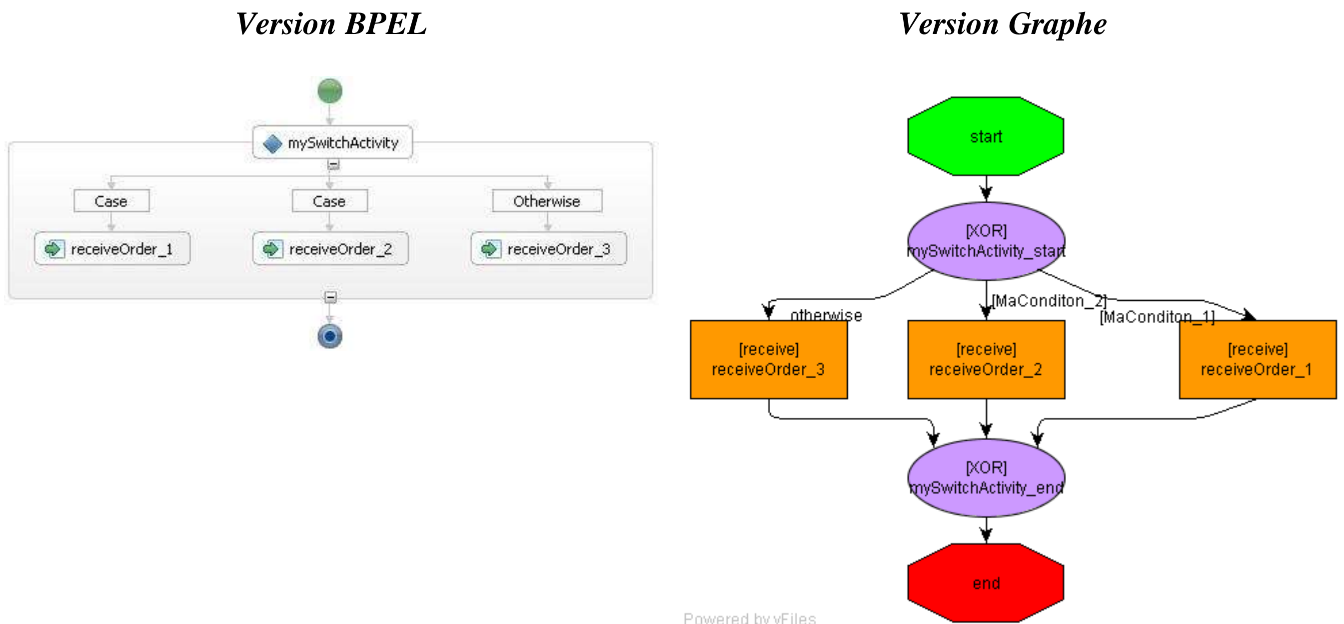


Figure 3 : Transformation – « Switch »

3.4 Transformation – « While »

The While activity is transformed in the following way : a XOR connector is created for the While entry. A second connector is created for representing the out of the cycle. The internal function of the While is placed between this two connectors and it is only executed if maCondition is true, otherwise the While finishes.

```
<process name="whileProcess" …>
   <while name="myWhileActivity" condition="[MaConditon]">
        <receive name="receiveOrder_1" />
   </while>
</process>
```

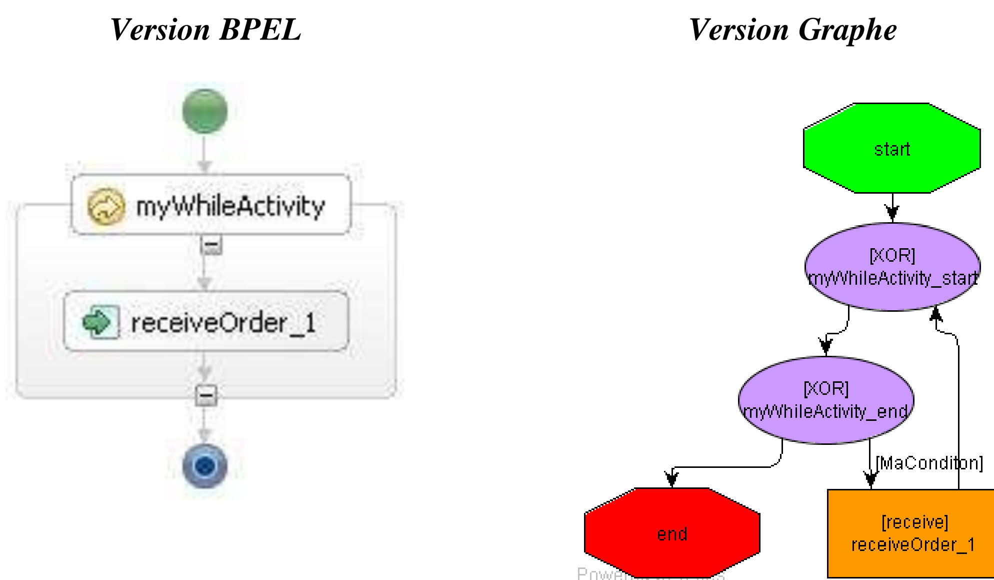


Figure 4 : Transformation – « While »

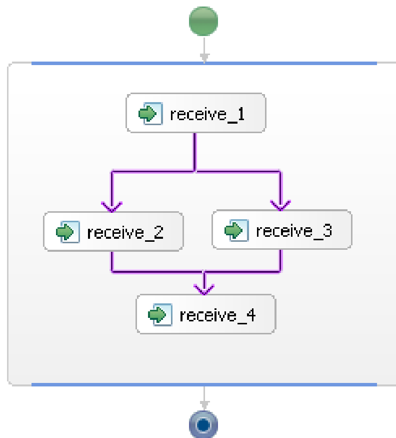
3.5 Transformation – « Flow »

The activity "Flow" is transformed in the following way: two connectors AND are created respectively to represent beginning and the end of the "Flow". Arcs are created between the first AND connector and the activities not forming part of a link relationship as a target. In a similary way, arcs are created between the second AND connector and the activities not forming part of a link relationship as a source.

Then, the arcs between the internal activities in the "Flow" are established at the time of the transformation of the "Scope" (or "BpelDefinition") to which the "Flow" belongs. Each one of these arcs has associated the corresponding transition condition.

```
<process name="flowProcess" …>
   <flow name="myFlowActivity">
     <links>
        <link name="Link1" />
        <link name="Link2" />
        <link name="Link3" />
        <link name="Link4" />
     </links>
     <receive name="receive_1">
        <sources>
           <source linkName="Link1" transitionCondition="cond_1"/>
           <source linkName="Link2" transitionCondition="cond_2"/>
        </sources>
     </receive>
     <receive name="receive_2">
        <targets>
           <target linkName="Link2"/>
        </targets>
        <sources>
           <source linkName="Link3" transitionCondition="cond_3"/>
        </sources>
     </receive>
     <receive name="receive_3">
        <targets>
           <target linkName="Link1" />
        </targets>
        <sources>
           <source linkName="Link4" transitionCondition="cond_4"/>
        </sources>
     </receive>
     <receive name="receive_4">
        <targets>
           <target linkName="Link3" />
           <target linkName="Link4" />
        </targets>
     </receive>
   </flow>
</process>
```

Version BPEL ***Version Graphe***



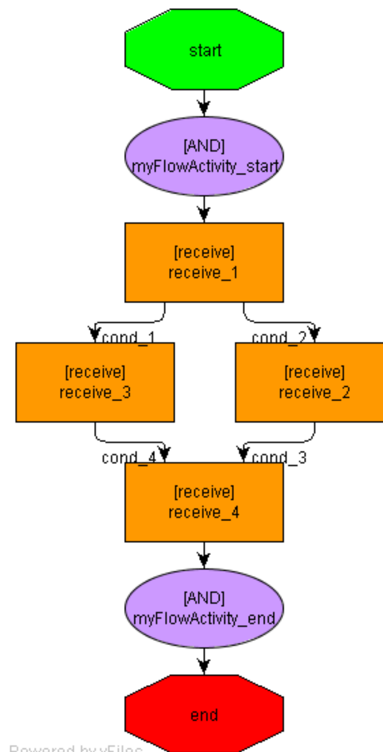


Figure 5 : Transformation – « Flow »

3.6 Transformation – « Empty »

The activity "Empty" is simply ignored at the time of its transformation and thus does not appear in the graph resulting from the transformation. However in general, this type of activity is used in the "Flow" as connectors. When it is the case, the activity is transformed as a connector of the type "OR".

```
<process name="flowProcess" …>
   <flow name="flow_1">
     <links>
        <link name="emptyLink" />
        <link name="choice_1" />
        <link name="choice_2" />
     </links>

     <receive name="receiveOrder">
        <source linkName="emptyLink"
                    transitionCondition="cond_1"/>
     </receive>

     <empty name="empty">
        <target linkName="emptyLink" />
        <source linkName="choice_1"
                    transitionCondition="cond_2"/>
        <source linkName="choice_2"
                    transitionCondition="cond_3"/>
     </empty>

     <invoke name="invokeInvoice_1">
        <target linkName="choice_1"/>
     </invoke>

     <invoke name="invokeInvoice_2">
        <target linkName="choice_2" />
     </invoke>
   </flow>
</process>
```

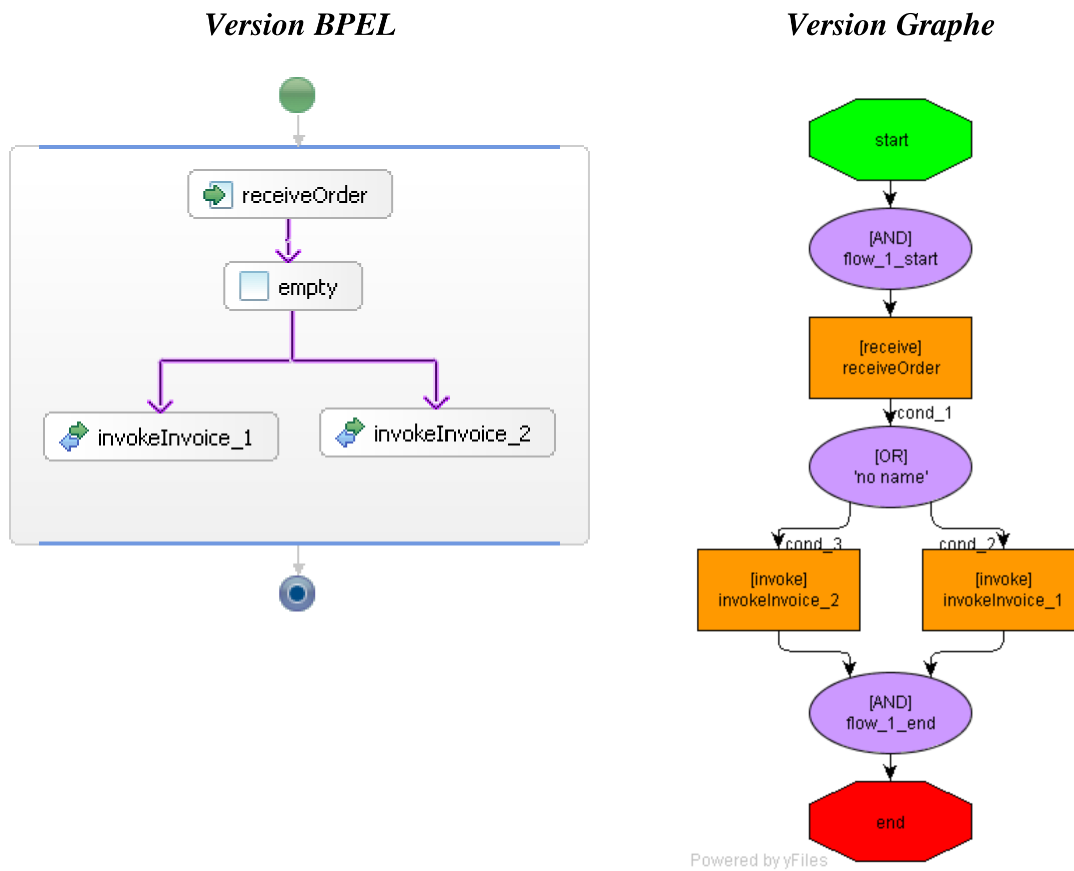


Figure 6 : Transformation – « Empty »

3.7 Transformation – « Scope »

For the activity "Scope" two strategies of transformation are considered: flat graph or overlapping graph.

```
<process name="flowProcess" …>
   <sequence name="my_sequence">
     <receive name="receive_1" />
     <scope name="my_scope" >
         <receive name="receive_2" />
     </scope>
     <receive name="receive_3" />
   </sequence>
</process>
```

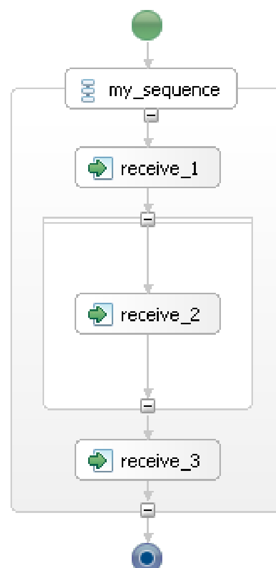


Figure 7 : BPEL – « Scope »

3.7.1 Flat Graph

The strategy of transformation of a flat graph consists in removing the overlap related to the "scope" in the final graph. The transformation resulting from this strategy is represented on the figure below.

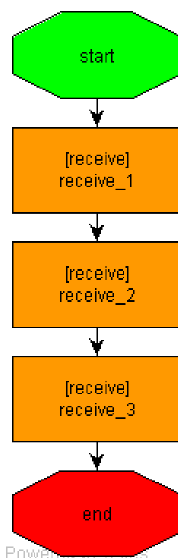
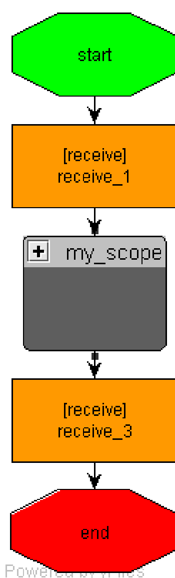


Figure 8 : Transformation « Scope » – Stratégie de mise à plat

3.7.2 Overlapping graph

The strategy of transformation with overlapping graph consists in creating a specific node in the graph which is itself a graph for each activity "Scope" found. In this way a hierarchy of graph can be obtained. The figure below exposes the graph obtained by using this strategy.

Graphe principal ***Sous-graphe***



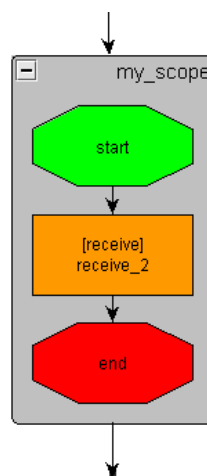


Figure 9 : Transformation « Scope » – Stratégie de graphe imbriqué

4 Implémentation

4.1 BPEL Meta-Model

4.1.1 General Schema

The general outline of the metamodel BPEL such as it is defined in the project is represented on the diagram of classes below.

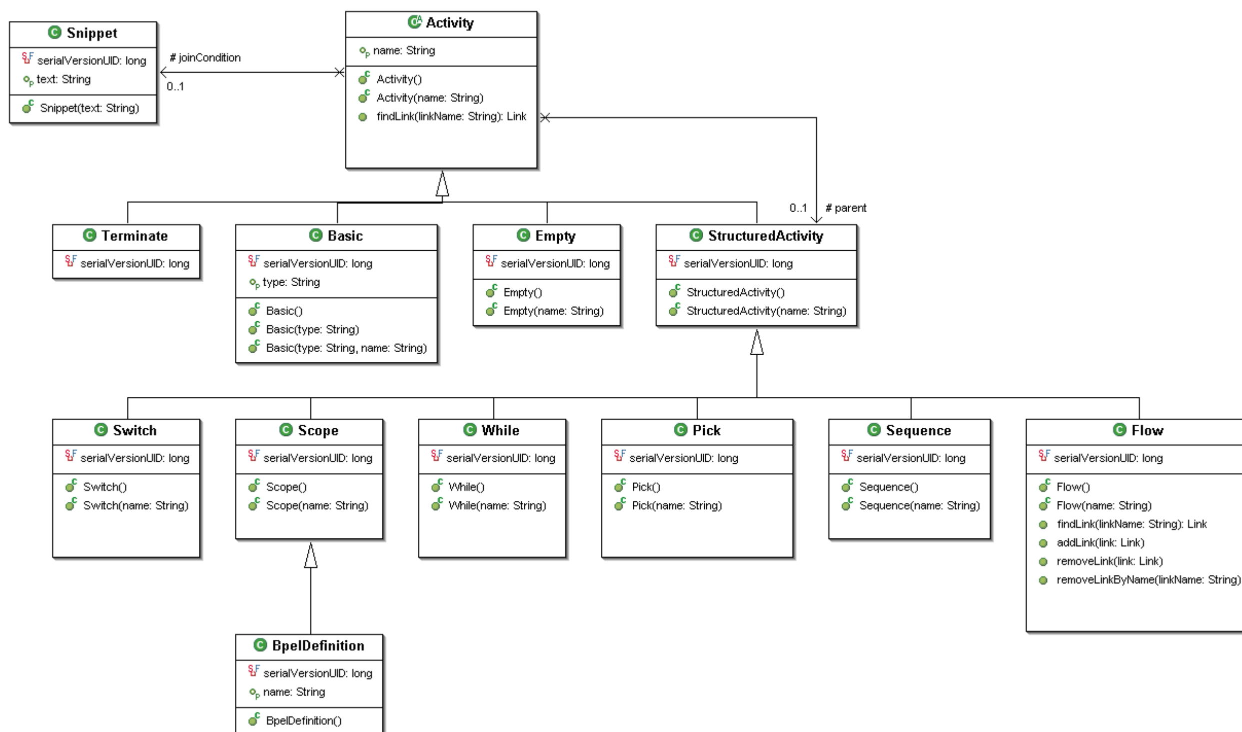


Figure 10 : Diagramme de classes – Méta modèle BPEL

Two types of activities are distinguished: structured and not structured. An activity is known as structured when it is made up of nested activities. The other activities are not structured.

In the implementation suggested, all the basic activities are gathered in the form of class called Basic. A possible extension to supplement the metamodel would consist in specializing this class according to the type of the activity (Receive, Invoke, Assign, etc.)

Note that the definition of BPEL process is also defined as a specialization of the Scope activity. All the activities, excluded the definition of the process "BpelDefinition", point on the structured activity to which they belong.

4.1.2 Sequence

The following class diagram represents the Sequence activity.

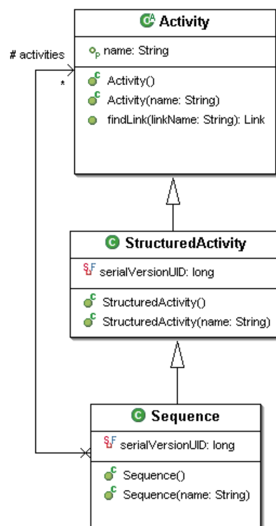


Figure 11 : Sequence class diagram

The sequence is defined like a structured activity and contains a list of the activities which composes it. These activities are able themselves to be composite.

4.1.3 While

The following class diagram represents the While activity.

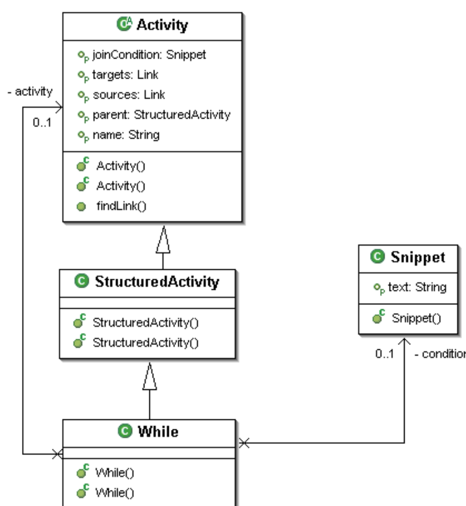


Figure 12 : Diagramme de classes « While »

The loop is defined as a structured activity and contains a link towards the activity defining the body of the loop. It also contains an object of the Snippet type defining the condition associated with the loop.

4.1.4 Switch

The following class diagram represents the Switch activity.

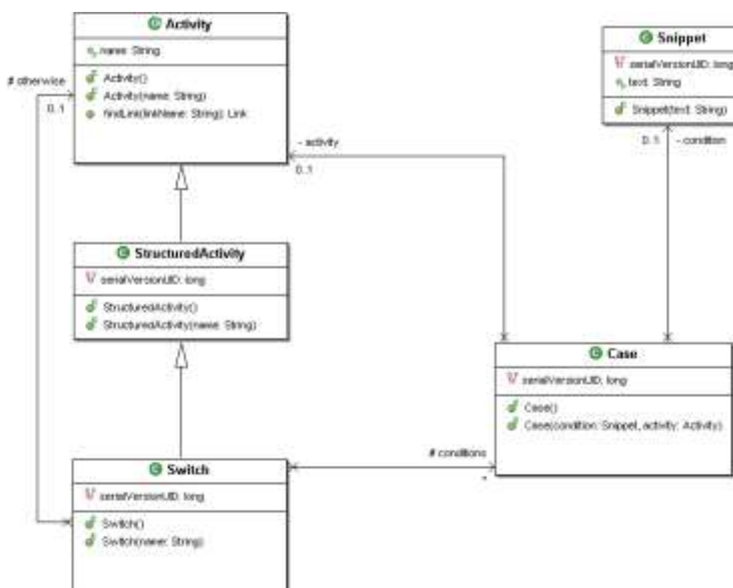


Figure 13 : Diagramme de classes « Switch »

The activity of conditional connection "Switch" is defined as a structured activity. It contains a group of connection "Case". With each case a condition Snippet object is associated and the activity that must be executed when its condition is valid. The activity "Switch" also contains a link towards the activity that must be executed when none the case conditions is valid.

4.1.5 Flow

The following class diagram represents the Flow activity.

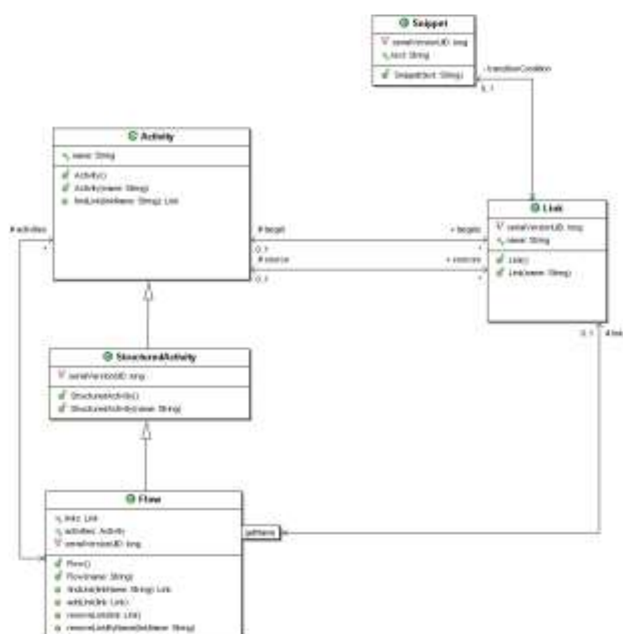


Figure 14 : Diagramme de classes « Flow »

The activity "Flow" is defined as an structured activity. It contains the list of the links defined in the graph representing the flow of activity indexed by their name. It also contains the list of the activities defined as entrance points in the flow. An activity is defined as an entrance point when it is not the target in a link i.e.: it does not have a predecessor in the flow.

The links between the activities and the sources/targets of the links are bidirectional. In this manner each activity contains a list of the links in which it takes part as a source and a list of the links in which it takes part as a target.

Each link has a Snippet as a transition condition.

4.1.6 Pick

The following class diagram represents the Pick activity.

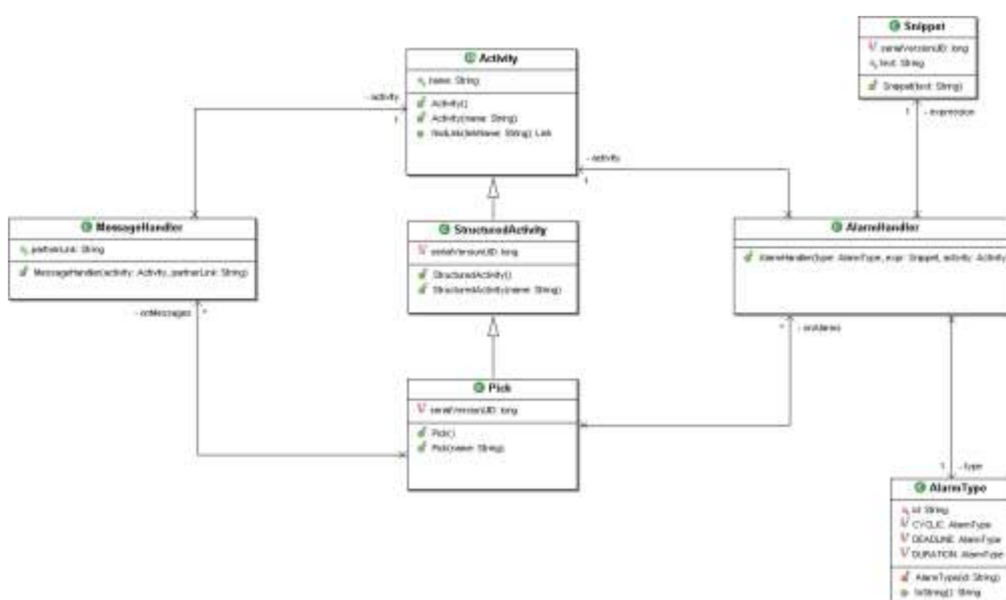


Figure 15 : Diagramme de classes « Pick »

The activity Pick is defined as a structured activity. It contains a list of the events it is wating to happen. Two types of events are distinguished : arrival of a message and time event.

The MessageHandler class knows the activity to be launched when a message arrives.

The AlarmHandler class knows the activity to be launched in response to a time event. This class has asociated a Snippet object defining the event or the alarm type.

4.1.7 Scope

The following class diagram represents the Scope activity.

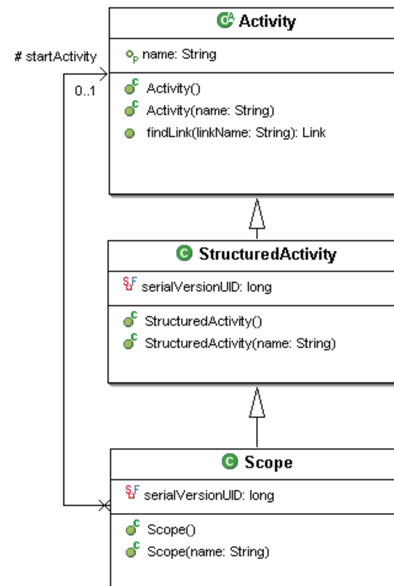


Figure 16 : Diagramme de classes « Scope »

The activity Scope is defined as a structured activity. It contains a link towards the activity to be launched at the time of the entry in the Scope.

This class could be supplemented in future versions to take into account the declaration of the variables and the actions of compensation for example.

4.2 Loading of the BPEL

4.2.1 General Schema

The following class diagram represents the classes of the bpm.bpel.input package. This classes alow to read a BPEL file.

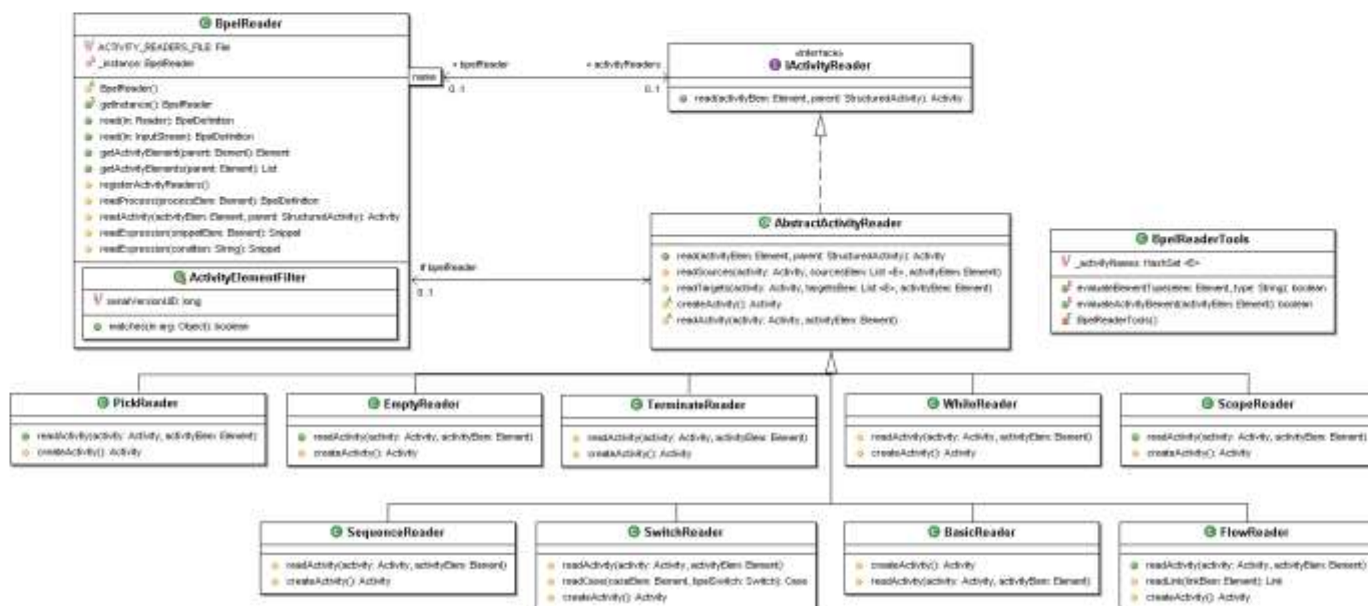


Figure 17 : Diagramme de classes – Chargement du méta modèle BPEL

The BpelReader class is the entry point to allow the loading of a BPEL file. (See 4.2.3).

The interface "IActivityReader" is a common interface having to be implemented by all the classes specialized in the reading of a particular activity. Thus one finds a "Reader" for each activity defined in the BPEL metamodel.

The BpelReader class contains the whole of the specialized readers of activities and is responsible of chosing the appropriate reader for the XML tag being analysed.

For easying the evolution, the list of the "IActivityReader" used by the "BpelReader" is initialized using a XML configuration file. (see 4.2.2)

4.2.2 Configuration of IActivityReader

The specialized activity reades used during the loading of the BPEL reader are configured in the XML file conf/bpm.bpel.input.readers.xml.

An excerpt of that file is presented bellow:

```
<?xml version="1.0" encoding="UTF-8"?>
<activityReaders>

  <activityReader name="empty" class="bpm.bpel.input.EmptyReader"/>
  <activityReader name="sequence"
class="bpm.bpel.input.SequenceReader"/>
  […]
</activityReaders>
```

The activityReader tag contains two attributes. corresponding to the class (class) that contains the implementation of IActivityReader that must be used with the activity of name name. The name of BPEL activity is the name of the tag of the activity in a BPEL file.

It is thus very simple to extend or modify the model by simply modifying the mapping between the tags and the classes allowing their loading.

4.2.3 Utilization

The use of this packet to charge a BPEL file consists in recovering an instance of BpelReader and to call the method "read" on flow of reading of the file. An object of the BpelDefinition type will then be returned.

```
// Chargement du fichier Bpel
BpelReader reader = BpelReader.getInstance();
BpelDefinition bpel = reader.read(new FileInputStream("MonFichier");
```

4.3 Graph Meta Model

The general outline of the meta model of the process graph as it is defined in the project is represented on the diagram of classes below.

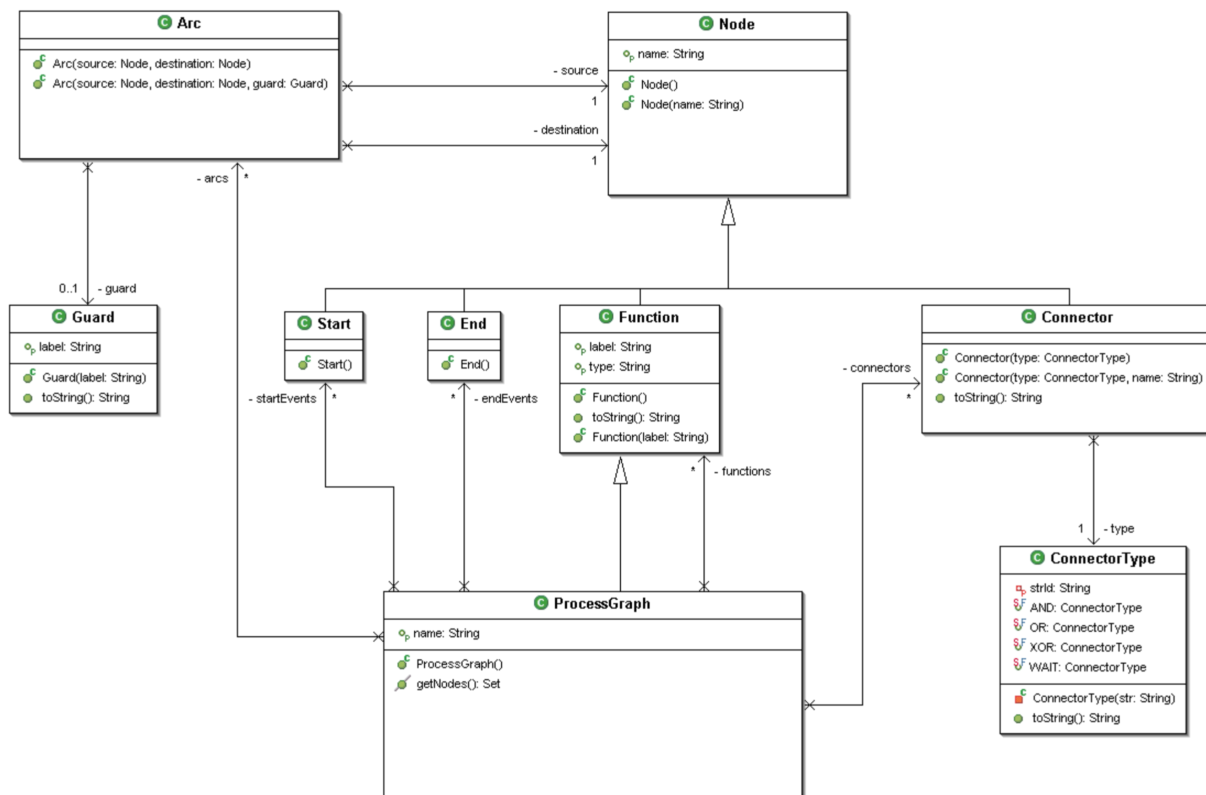


Figure 18 : Diagramme de classes – Méta Modèle Graphe

The ProcessGraph class represents a graph. This graph is represented by a group of arc objects, function objects, connector objects and Start/End objects.

The functions are the nodes of the graph that represent particular actions. A possible evolution of the model would involve specializing this class and allow it to define different types of functions (receive, assign, etc.)

The Start and End objects represent the initial and final nodes of the graph.

The connectors are nodes of the graph that make it possible to connect the elements and to carry out various connection in the process. There are different types of connectors defined in the ConnectorType class.

The arcs make it possible to bind the nodes between them. Moreover it is possible for each arc to associate a condition of transition in the graph represented by the class "Guard".

It will be noted that the class "ProcessGraph" is itself derived from the class "Function". That can make it possible to create in a graph of the nodes being themselves of the graphs.

4.4 Transformation into Graph

4.4.1 General Schema

The class diagram below shows the classes of the bpm.graph.transform package, wich is responsible of transform a meta model into a process graph.

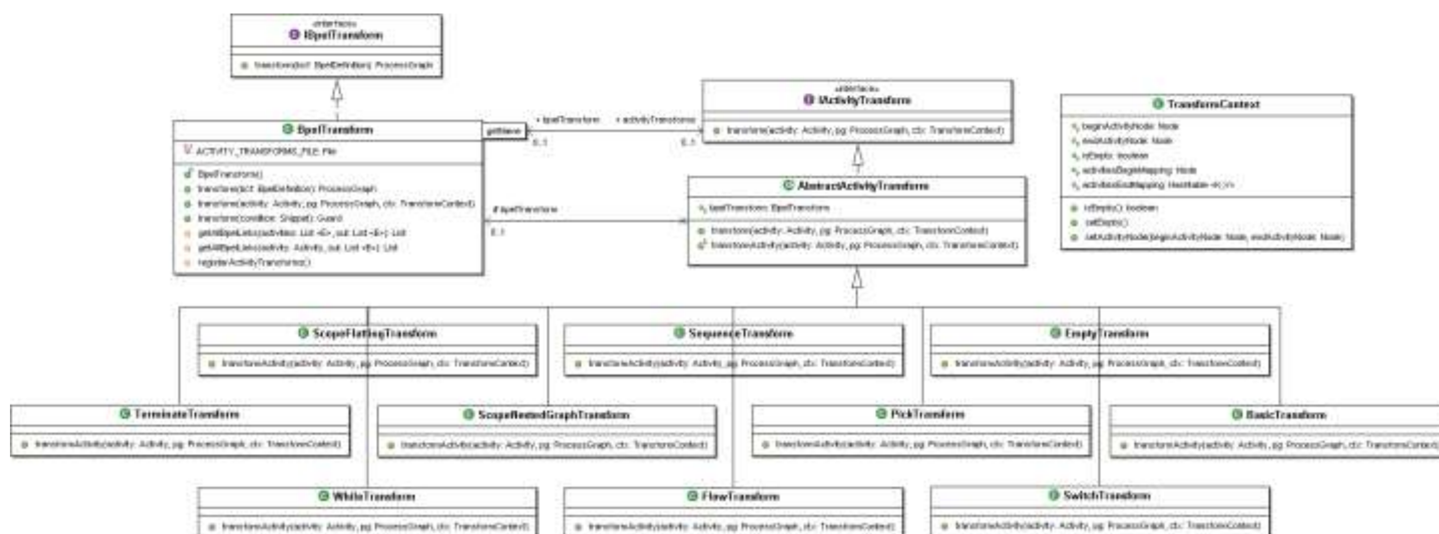


Figure 19 : Class diagram – Transformation of BPEL into Graph

The class *BpelTransform* is the entrance point for transforming a BPEL process into a Graph. (see 4.4.5).

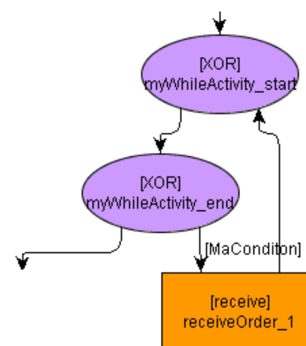
The *IActivityTransform* interface must be implemented by all the specialized classes responsible of transforming a particular activity. Thus there is an *ActivityTransform* for each activity defined in the BPEL meta model.

The *BpelTransform* class contains the strategies of transformation of BPEL activities. It is thus possible to define several strategies of transformation of an BPEL activity into graph. It is the case for example with the classes *ScopeFlattingTransform* and *ScopeNestedGraphTransform* which respectively make it possible to transform a BPEL activity of the type Scope either into flat graph or by using overlapping graphs. The list of the strategies used by default is configurable by means of a XML file. (see 4.4.4).

4.4.2 Strategy and context of transformation

As mentioned in part 4.4.1 the transformation of an BPEL activity is done by a specialized class implementing the transformation strategy. The class *BpelTransform* determines the strategy to use according to the type of BPEL activity. It calls the method "*transform(Activity activity, ProcessGraph pg, TransformContext ctx)*" of the class implementing the strategy. This method is given the responsability to transform the activity and all of its nested activities if it is a structured one.

It returns in the context (passed as a parameter) the node of entry and exit in the activity after transformation.



For example for the transformation of an activity of the type "While", the context will point to the connectors: "myWhileActivity_start" and "myWhileActivity_end".

In the case of a Basic activity the context will point to the function corresponding to the activity in the graph, and to the beginning node and ending node.

It is then with called to establish the arcs connecting the functions and connectors of the graph between them.

The sequence diagram presented proposes a scenario of transformation of a Sequence activity into graph. It considers that BPEL sequence contains two unspecified sub activities.

At the time of the call of the transformation on the class *BpelTransform*, the strategy adapted to the transformation of the activity is selected from the activity name. In this scenario the recovered class is *SequenceTransform*.

The transformation method is then called on the selected strategy class. This function is defined in the class *AbsractActivityTransform* wich contains the transformation strategies common to all the activities. This method calls immediately the specialized class *transformActivity* method. This one is given the responsability of transforming the sub activities of the sequence into graph and to establish the arcs and links between them.

In our scenario, the sequence contains two activities. The method "transform(...)" of the class "BpelTransform" is called recursively 2 times on each one of them. An arc is created to connect the end of the 1st activity and the beginning of the second. Then the context of transformation is updated to return the nodes respectively representing the beginning and the end of the sequence in the graph. The beginning of the sequence corresponds to the node of entry in the 1st activity, and the end the node of exit of the 2nd activity.

In our scenario, it is considered that the sequence belongs to an activity of the type "Flow" and that it is the source of another unspecified activity. That is translated in the meta model BPEL by the fact that the list of the sources of the activity is not empty. To establish the missing arcs related to the activity "Flow", a mapping between the activity and the node of beginning of the sequence is added in the context. This fact during the establishment of the bonds, one will be able to recover the starting node to which to connect the arc missing in the graph (see part 4.4.3).

Note: If our sequence were the target of another activity, the same method with the list of the targets would be used.

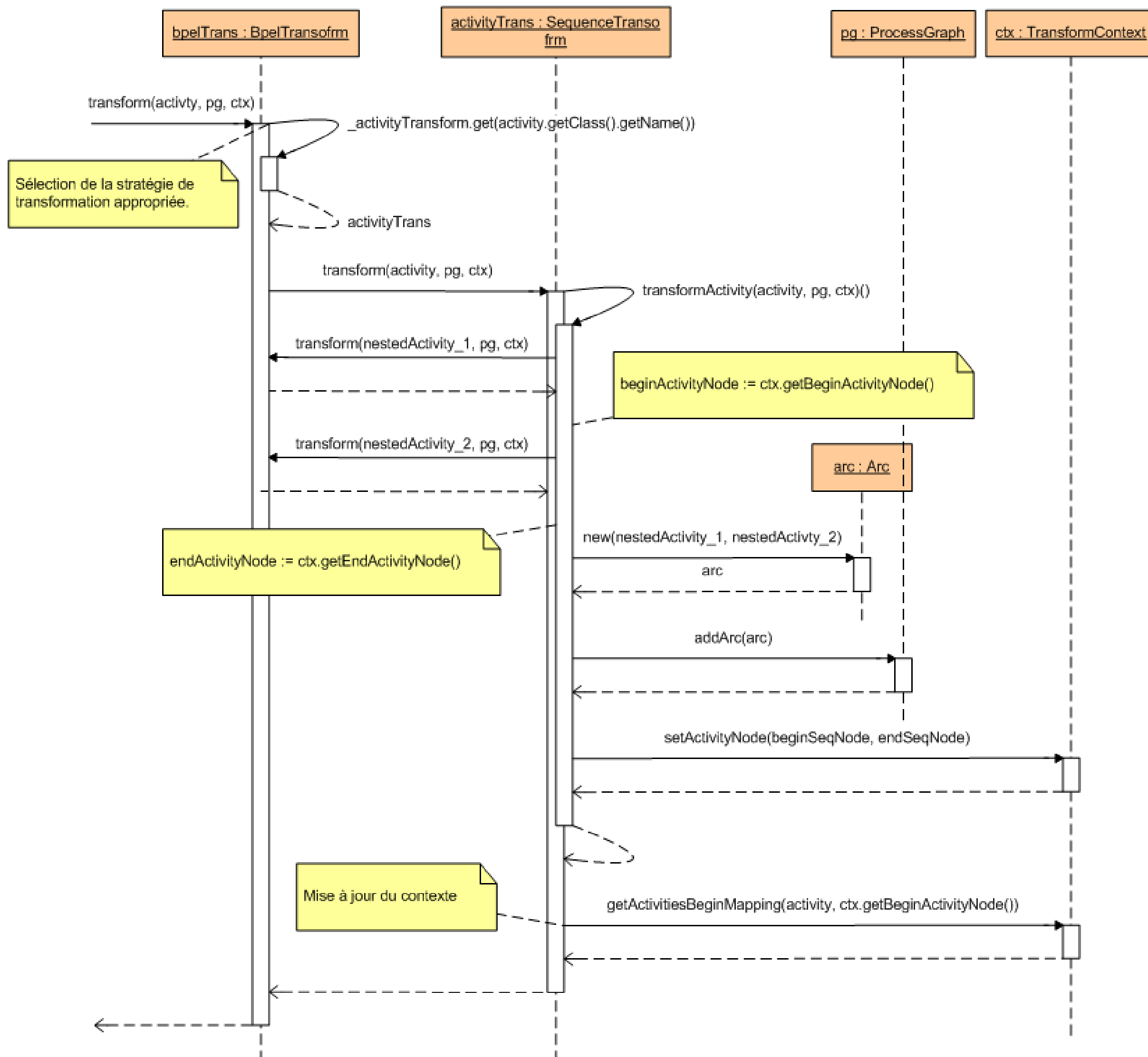


Figure 20 : Scénario de transformation

4.4.3 Transformation des liens « Link »

The Flow activities in BPEL make it possible to create concurrent executions represented in the form of graph. The transitions between these activities are defined by Links. The transformation of these *Links* into arcs in the graph takes place on the level of the basic function of the creation of the graph. The pseudo code allowing this transformation is provided bellow.

```
                       -- Functions definition --

// Transform a BPEL transition condition into a graph transition
condition
Let Transform(TransitionCondition tc)

                       -- Variables definition --
Let links       // Group of BPEL Links of the current scope
Let ctx         // Transformation context
Let srcNode     // Source node
Let dstNode     // Destination node
Let pg          // Graph being created

                          -- Pseudo code --

For each link in Links

   // Recherche du nœud source dans la table de hachage de mapping
   // des activités BPEL / nœud de début dans le graphe.
   srcNode := ctx.activiesBeginMapping.get(link.getSource())
   // Recherche du nœud destination dans la table de hachage de mapping
   // des activités BPEL / nœud de fin dans le graphe.
   dstNode := ctx.activitiesEndMapping.get(link.getTarget())

   // Création de l'arc correspondant au lien BPEL entre le nœud source
   // et le nœud destination.
   arc := (srcNode, dstNode)
   arc.guard := transformCondition(link.transitionCondition)

   // Ajout de l'arc dans le graphe
   pg.arcs := pg.arcs U arc
End for
```

4.4.4 Configuration des « IActivityTransform »

Les stratégies de transformation à utiliser par défaut pour chaque activité BPEL sont définies dans le fichier XML « conf/bpm.graph.transforms.xml ».

Un extrait de ce fichier est présenté ci-dessous :

```
<activityTransforms>
  <activityTransform bpelClass="bpm.bpel.model.Empty"
                     transClass="bpm.graph.transform.EmptyTransform"/>
  <activityTransform bpelClass="bpm.bpel.model.Sequence"
                     transClass="bpm.graph.transform.SequenceTransform"/>
  […]
</activityTransforms>
```

La balise « activityTransform » contient deux attributs correspondant à la classe BPEL de l'activité à transformer « bpelClass » et à la classe utilisée pour sa transformation en graphe « transClass ».

4.4.5 Utilisation

L'utilisation de ce paquetage pour transformer un fichier BPEL en graphe consiste à créer une instance de la classe permettant la transformation « BpelTransform » puis appeler la méthode « transform » en lui passant la définition BPEL « BpelDefinition » de notre processus à transformer.

```
// Transformation en graphe
BpelTransform trans = new BpelTransform();
ProcessGraph graph = trans.transform(bpelDefinition);
```

4.5 Ecriture du graphe dans un fichier

Escritura de un grafo en un archivo

4.5.1 Schéma général

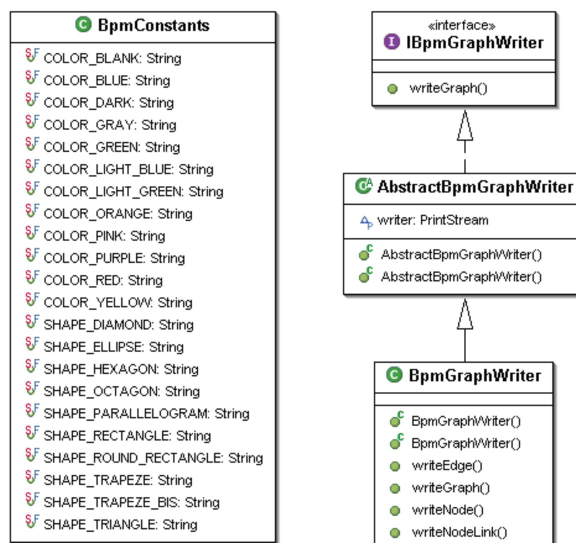


Figure 21 : Diagramme de classes – Ecriture d'un fichier GML

La classe « BpmGraphWriter » est le point d'entrée permettant d'écrire un Graph dans un fichier (voir partie 4.5.3). Le format du fichier généré en sortie est spécifié dans la partie 4.5.2)

4.5.2 Format de sortie

Les fichiers générés par le BpmGraphWriter sont au format GML. Pour éditer ces fichiers il est possible d'utiliser le logiciel gratuit yEd – Java[tm] Graph Editor. Il est téléchargeable à l'adresse suivante : http://www.yworks.com/en/products_yed_about.htm#download. Ce logiciel permet en autre de disposer l'ensemble des nœuds du graphe correctement. Pour cela il suffit de passer par le menu : Layout → Hierarchical → Interactive. Le logiciel réorganise alors tous les nœuds du graphe.

Le fichier à produire est de la forme suivante :

```
// Description du graphe à afficher
graph
[
   hierarchic   1
   label        ""
   directed     1
   // Descriptions des noeuds
   node
   [
     // Identifiant du noeud, label …
       id 0
       label  "n1"
     // Positionnement du noeud, forme, couleur…
       graphics
       [
```

```
        x   0
        y   0
        w   116.0
        h   58.0
        type        "roundrectangle"
        fill        "#FFFF99"
        outline     "#000000"
      ]
      // Caractéristique du texte affiché dans le noeud
      LabelGraphics
      [
        text        "n1"
        fontSize    12
        fontName    "Dialog"
        anchor      "c"
      ]
    ]
    // Descriptions des liens
    edge
    [
      source 0        // id du nœud course
      target 1        // id du noeud destination
      label  ""
     // Spécification concernant la flèche
      graphics
      [
        smoothBends 1
        fill        "#000000"
        targetArrow "standard"
      ]
      edgeAnchor
      [
        ySource     1.0
        yTarget     -1.0
      ]
     // Spécification concernant le texte sur la flèche
      LabelGraphics
      [
        text        ""
        fontSize    12
        fontName    "Dialog"
        model       "six_pos" position "tail"
      ]
    ]
]
```

Lors de la génération, à chaque type de nœud (connecteur, fonction, start, end, etc.) est associé une forme et une couleur particulière afin de les différencier facilement.

4.5.3 Utilisation

L'utilisation de ce paquetage pour générer un fichier GML à partir d'un graphe consiste à créer une instance de la classe « BpmGraphWriter » en lui passant en paramètre le fichier destination dans lequel écrire le graphe. Il suffit après d'appeler la méthode « writeGraph » en lui passant en paramètre le graphe à générer.

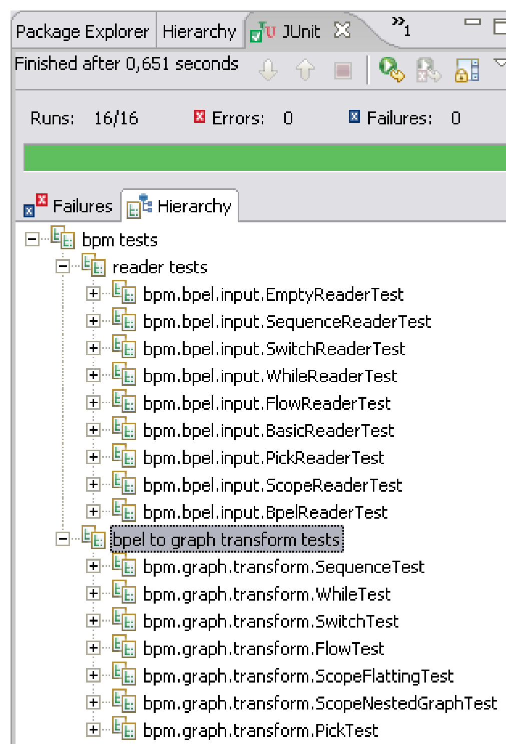
```
// Ecriture du fichier de sortie
BpmGraphWriter writer = new BpmGraphWriter(new File("outFile.gml"));
writer.writeGraph(processGraph);
```

5 Jeux de tests

5.1 Tests unitaires

Les paquetages « bpm.bpel.input » et « bpm.graph.transfom » permettant respectivement de charger un fichier BPEL et de le transformer en graphe font l'objet de tests unitaires. Ces tests sont réalisés en utilisant le framework « JUnit ».

Toutes les implémentations des tests se trouvent dans le répertoire « srcTests » et sont organisés par paquetage. A chaque paquetage correspond une suite de test. Toutes les suites de test peuvent être lancées automatiquement en utilisant la classe « bpm.UnitTests ».



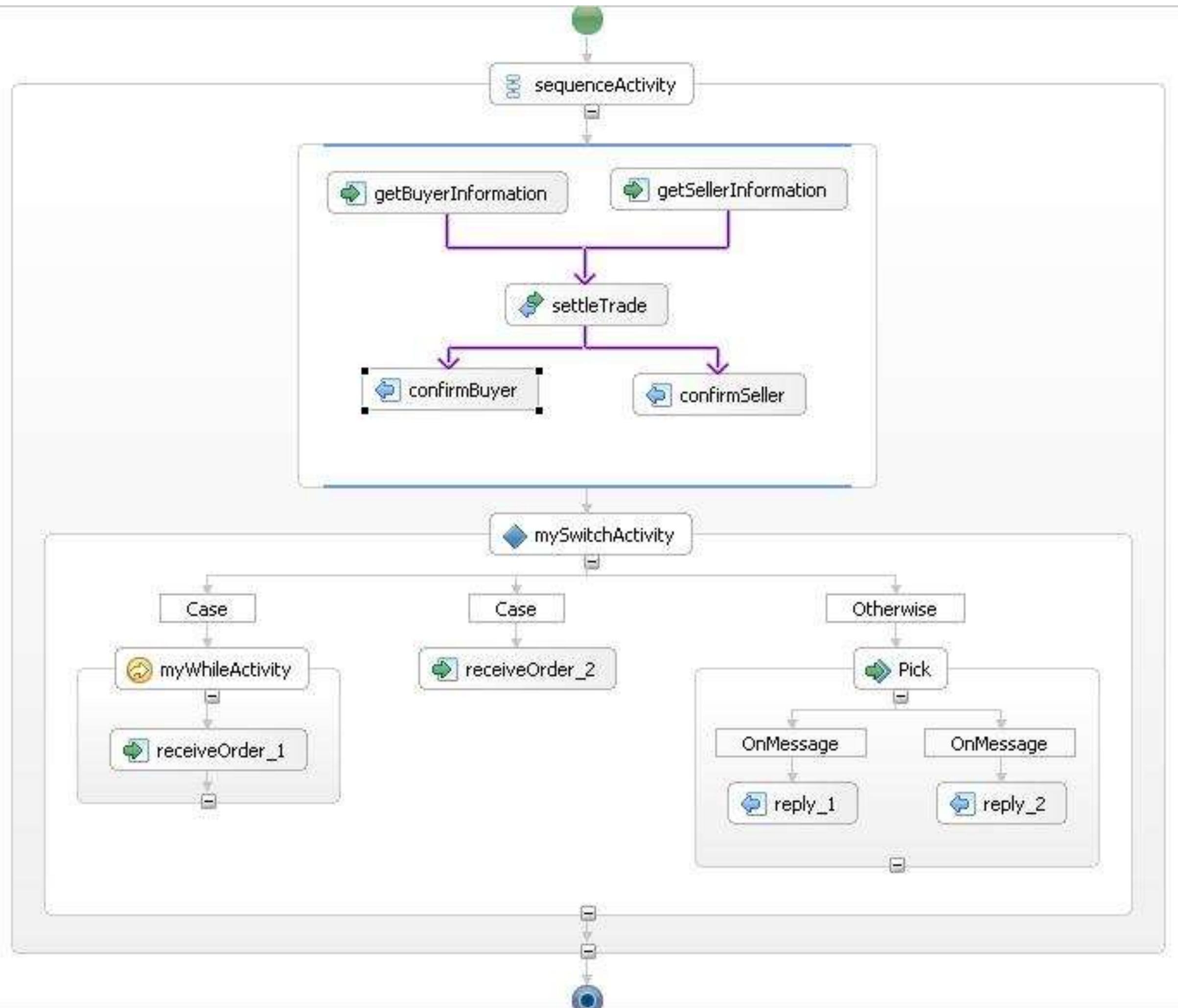
5.2 Tests d'intégration

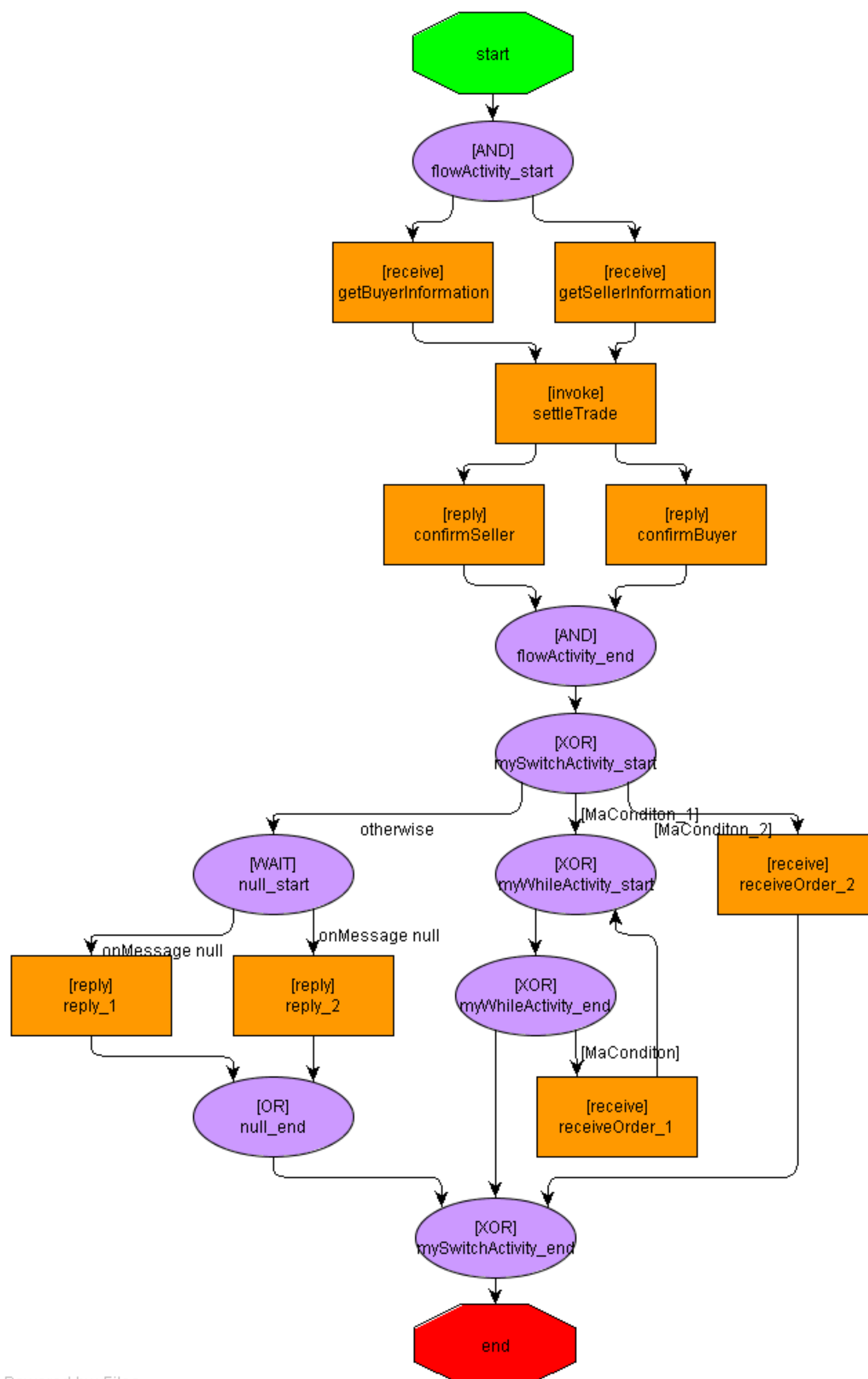
Le répertoire « tests » du projet contient un ensemble de jeux de tests BPEL. Lors du lancement de la commande « bpm.IntegrationTests » tout le fichier BPEL de ce répertoire sont transformés en graphe. Les fichiers contenant les graphes (au format GML) issus de la transformation sont générés dans le répertoire « tests_out ».

TABLE DES ILLUSTRATIONS

Figure 1 : Transformation – « Pick »....6
Figure 2 : Transformation – « Sequence »....7
Figure 3 : Transformation – « Switch »....8
Figure 4 : Transformation – « While »....9
Figure 5 : Transformation – « Flow »....11
Figure 6 : Transformation – « Empty »....13
Figure 7 : BPEL – « Scope »....14
Figure 8 : Transformation « Scope » – Stratégie de mise à plat....15
Figure 9 : Transformation « Scope » – Stratégie de graphe imbriqué....15
Figure 10 : Diagramme de classes – Méta modèle BPEL....16
Figure 11 : Diagramme de classes « Sequence »....17
Figure 12 : Diagramme de classes « While »....17
Figure 13 : Diagramme de classes « Switch »....18
Figure 14 : Diagramme de classes « Flow »....18
Figure 15 : Diagramme de classes « Pick »....19
Figure 16 : Diagramme de classes « Scope »....20
Figure 17 : Diagramme de classes – Chargement du méta modèle BPEL....21
Figure 18 : Diagramme de classes – Méta Modèle Graphe....23
Figure 19 : Diagramme de classes – Transformation de BPEL vers Graphe....24
Figure 20 : Scénario de transformation....26
Figure 21 : Diagramme de classes – Ecriture d'un fichier GML....29

ANNEXE : TRANSFORMATION COMPLEXE




start
[AND]
flowActivity_start
[receive]
getBuyerInformation
[receive]
getSellerInformation
[invoke]
settleTrade
[reply]
confirmSeller
[reply]
confirmBuyer
[AND]
flowActivity_end
[XOR]
mySwitchActivity_start
otherwise
[MaConditon_1]
[MaConditon_2]
[WAIT]
null_start
[XOR]
myWhileActivity_start
[receive]
receiveOrder_2
onMessage null
onMessage null
[reply]
reply_1
[reply]
reply_2
[XOR]
myWhileActivity_end
[MaConditon]
[OR]
null_end
[receive]
receiveOrder_1
[XOR]
mySwitchActivity_end
end
Powered by yFiles