1. ATL Transformation: path expressions to Petri nets

1.1. Introduction

The path expression to Petri nets example describes a transformation from a path expression to a Petri net. This document provides an overview of the whole transformation sequence that enables to produce an XML Petri net representation (in the PNML format [1]) from a textual definition of a path expression.

The input metamodel of this transformation sequence is the TextualPathExp metamodel. Models conforming to this metamodel are injected from a textual definition of the path expression by means of a TCS (Textual Concrete Syntax) program (this part is out of the scope of the document). A TextualPathExp model is then transformed into a PathExp model. The PathExp metamodel describes the structure of the graphical representation of the path expression. This new representation is quite similar to the structure defined by the PetriNet metamodel, and a PathExp model can easily be transformed into a PetriNet model. This PetriNet model is then transformed into a XML model providing a XML representation of the Petri net in the PNML format. As a final step, the XML model is extracted to the textual XML representation using an ATL query (this last part is not described in this document).

1.2. An example

In this section, we illustrate the transformation sequence by means of a simple example that provides a comprehensive snapshot of the transformation sequence. The initial input of this transformation sequence is the textual definition of a path expression, as illustrated in Figure 1. This path expression is composed of a simple transition (“f”), followed by a composed alternative transition (“g;h + k;m*;n”), followed by a simple alternative transition (“p+q”), and a final simple transition (“s”). The textual encoding of this path expression is injected into a corresponding TextualPathExp model (this step is not detailed in this document).

path f;(g;h + k;m*;n);(p+q);s end

Figure 1. Textual path expression example

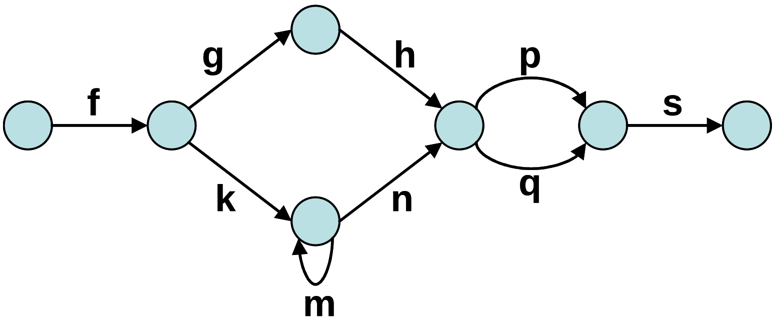

Figure 2. Graphical path expression example

From the TextualPathExp model, we build a PathExp model (by means of the TextualPathExp2PathExp transformation) that encodes the graphical representation of the path expression considered so far (see Figure 2).

Next step corresponds to the core transformation of the transformations sequence: it builds a PetriNet model from the obtained PathExp model. The PetriNet model corresponding to our PathExp model is given in Figure 3.

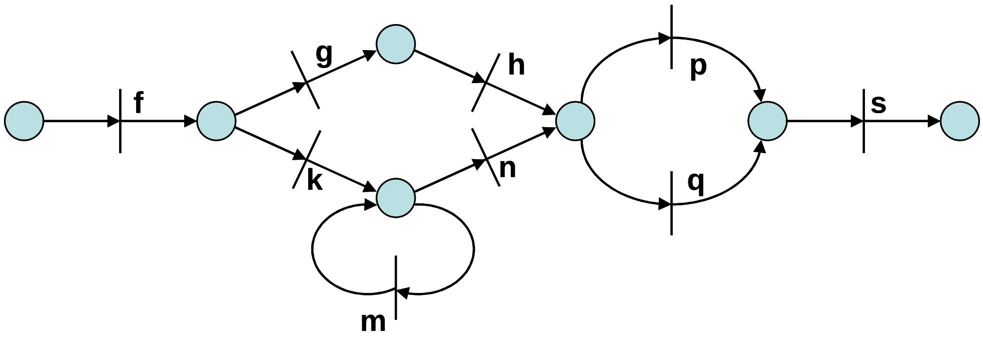

Figure 3. Petri net example

The following step of the transformations sequence aims to generate a XML model from this PetriNet model. The XML encoding of the Petri net is generated into the PNML format.

```
<pnml xmlns="http://www.example.org/pnpl">
  <net id="n1"
type="http://www.example.org/pnpl/PTNet">
    <name>
      <text></text>
    </name>
    <place id="1">
      <name>
        <text></text>
      </name>
    </place>
    <place id="2">
      <name>
        <text></text>
      </name>
    </place>
    <place id="3">
      <name>
        <text></text>
      </name>
    </place>
    <place id="4">
      <name>
        <text></text>
      </name>
    </place>
    <place id="5">
      <name>
        <text></text>
      </name>
    </place>
    <place id="6">
      <name>
        <text></text>
      </name>
    </place>
    <place id="7">
      <name>
        <text></text>
      </name>
    </place>
    <transition id="8"/>
    <transition id="9"/>
    <transition id="10"/>
    <transition id="11"/>
    <transition id="12"/>
    <transition id="13"/>
    <transition id="14"/>
    <transition id="15"/>
    <transition id="16"/>
    <arc id="17" source="3" target="9"/>
    <arc id="18" source="12" target="5"/>
    <arc id="19" source="4" target="10"/>
    <arc id="20" source="8" target="7"/>
    <arc id="21" source="13" target="7"/>
    <arc id="22" source="9" target="5"/>
    <arc id="23" source="3" target="12"/>
    <arc id="24" source="7" target="16"/>
    <arc id="25" source="7" target="13"/>
    <arc id="26" source="15" target="6"/>
    <arc id="27" source="1" target="14"/>
    <arc id="28" source="2" target="11"/>
    <arc id="29" source="14" target="3"/>
    <arc id="30" source="10" target="2"/>
    <arc id="31" source="5" target="15"/>
    <arc id="32" source="16" target="3"/>
    <arc id="33" source="11" target="1"/>
    <arc id="34" source="2" target="8"/>
  </net>
</pnml>
```

Table 1. XML example

Considering the whole transformation process, a last step would be to extract the generated XML model into a corresponding textual representation (see Table 1). This could be achieved by means of an ATL query. This last step is not detailed in this document.

1.3. Metamodels

In the scope of this example, we consider four distinct metamodels:

- The TextualPathExp metamodel, which describes the structure of a path expression in its textual form.
- The PathExp metamodel, which describes the structure of a path expression under its graphical form.
- The PetriNet metamodel, which describes the structure of a Petri net.
- The XML metamodel, which describes the generic structure of a XML file.

These metamodels are detailed in the following subsections.

1.3.1. The TextualPathExp metamodel

Figure 4 describes the TextualPathExp metamodel used in the scope of this transformation. A TextualPathExp contains a Path, which, in its turn, can contain from one to several Transitions. A Transition can be defined as a multiple or a single Transition. It is an abstract entity that can be either a PrimitiveTransition or an AlternativeTransition. A PrimitiveTransition is characterized by its name. An AlternativeTransition contains a number of alternative Paths.

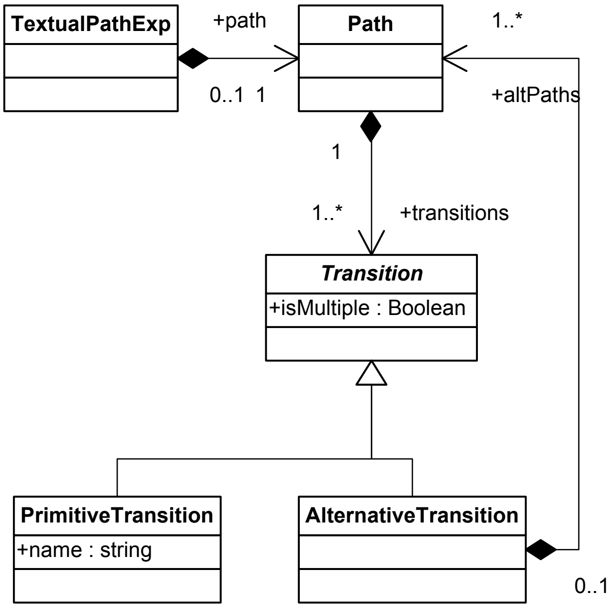

Figure 4. The TextualPathExp metamodel

1.3.2. The PathExp metamodel

The PathExp metamodel describes the different model elements that compose the graphical representation associated with path expressions, as well as the way they can be linked to each other. The considered metamodel is presented in Figure 5. It is moreover provided in KM3 format [2] in Appendix II.

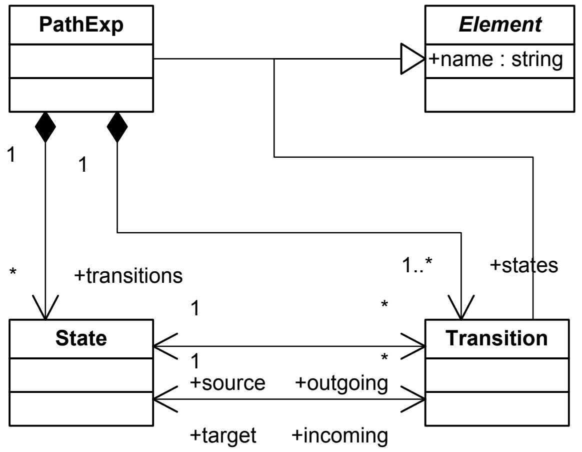

Figure 5. The PathExp metamodel

A PathExp is composed of States and Transitions. Each Transition has a State as source and a State as target. Each State can have several incoming and outgoing Transitions. Both Transition and PathExp inherits from the abstract Element entity, for which a "name" attribute is defined.

1.3.3. The PetriNet metamodel

The PetriNet metamodel describes the different model elements that compose a Petri net model, as well as the way they can be linked to each other. The considered metamodel is presented in Figure 6. It is moreover provided in KM3 format [2] in Appendix III.

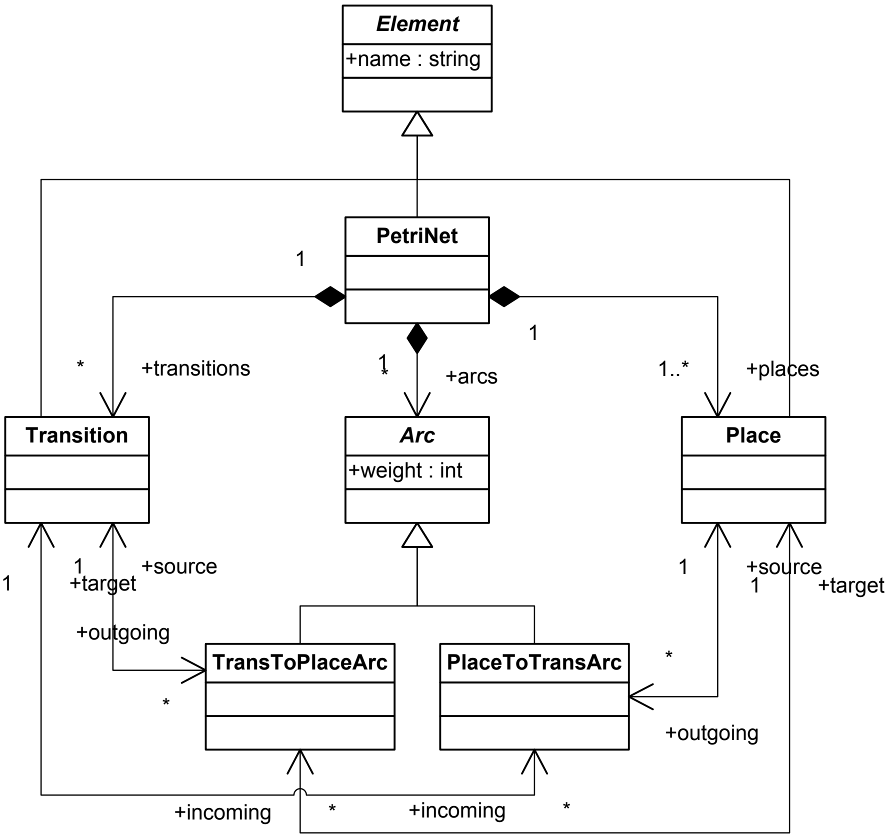

Figure 6. The PetriNet metamodel

A PetriNet model is composed of Transitions, Places and Arcs. The PetriNet entity, as well as the Transition and the Place ones, inherits from the abstract Element entity that defines a “name” attribute. An Arc is an abstract entity which is associated with a “weight” attribute. Each Arc is either of the TransToPlaceArc or PlaceToTransArc kind. A TransToPlaceArc connects a Transition to a Place, whereas a PlaceToTransArc connects a Place to a Transition.

A Place can have several outgoing PlaceToTransArcs and several incoming TransToPlaceArcs. Similarly, a Transition can have several incoming PlaceToTransArcs and several outgoing TransToPlaceArcs. Each TransToPlaceArc has a source Transition and a target Place. In the same way, each PlaceToTransArc has a source Place and a target Transition.

1.3.4. The XML metamodel

The XML metamodel describes the different model elements that compose a XML model, as well as the way they can be linked to each other. The considered metamodel is presented in Figure 7. It is moreover provided in KM3 format [2] in Appendix IV.

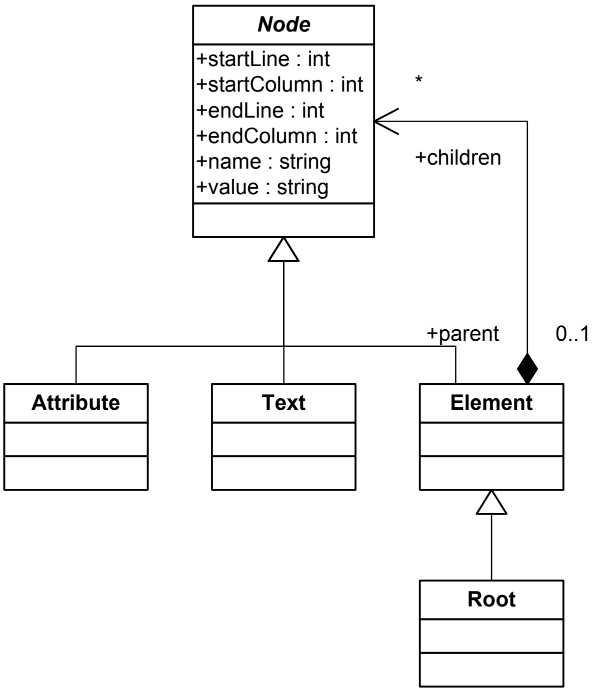

Figure 7. The XML metamodel

A XML model has a single Root element. It also contains Elements, Texts, Attributes entities. The Attribute, Text and Element elements all directly inherit from the abstract Node element, whereas Root inherits from the Element entity. The following attributes are defined for the abstract Node entity: “startLine”, “startColumn”, “endLine”, “endColumn”, “name” and “value”. In the scope of this example, we only make use of the two last attributes, “name” and “value”. In case of an Attribute entity, “name” encodes the name of the attribute, whereas “value” contains the value associated with the Attribute. In case of a Text entity, “value” contains the textual content of the Text. Finally, considering an Element entity, “name” encodes the name of the modelled XML tag.

An Element can contain several Nodes, which can be either of type Attribute, Text or Element. Inversely, a Node can be contained by zero or one Element. In fact, each Node is contained by an Element except the Root element which has no parent.

1.4. Transformations Specification

1.4.1. The TextualPathExp2PathExp transformation

The ATL code for the TextualPathExp to PathExp transformation consists of 20 helpers and 7 rules.

1.4.1.1. Assumptions

The ATL transformation described here makes a number of assumptions on the input TextualPathExp models:

- AlternativeTrans should not be “multiple” (i.e. only simple loops can be defined).

- The first and the last Transitions of a Path, including the main Path, have to be “single” Transitions.

- The first Transition of the input model must be a PrimitiveTrans.

1.4.1.2. Helpers

The first helper, **root**, is a constant helper. It provides access to the root input TextualPathExp element.

The **rootTrans** helper is a constant helper. It calculates the first Transition of the main Path of the input TextualPathExp. To this end, it returns the first Transition of the element provided by the **root** helper.

The **leafTrans** helper is a constant helper. It calculates the last Transition of the main Path of the input TextualPathExp. To this end, it returns the last Transition of the element provided by the **root** helper.

The **allPaths** helper is a constant helper. It computes a set containing all the Path elements of the input TextualPathExp model.

The **altPaths** helper is a constant helper. It calculates a set containing all the alternative Paths, that is all the Paths that are contained by an AlternativeTrans. For this purpose, the helper selects among all Paths, those that are included in an AlternativeTrans.

The **primTransitions** helper is a constant helper. It calculates the set of PrimitiveTrans that are not contained by a Path of any AlternativeTrans. To this end, the helper first gets the Paths that are not included by any AlternativeTrans, and, for each selected Path, it collects the Transition of the PrimitiveTrans type.

The **singlePrimTransitions** helper is a constant helper. It calculates the set of “single” PrimitiveTrans that are not contained by a Path of any AlternativeTrans. For this purpose, it simply selects among the **primTransitions** set, those whose *isMultiple* attribute is false.

The **multiplePrimTransitions** helper is a constant helper. It calculates the set of “multiple” PrimitiveTrans that are not contained by a Path of any AlternativeTrans. For this purpose, it simply selects among the **primTransitions** set, those whose *isMultiple* attribute is true.

The **altTransitions1** helper is a constant helper. It calculates the set of PrimitiveTrans that are contained by a Path that belongs to an AlternativeTrans, except the last Transition of each Path. To this end, the helper first gets all the Transitions contained by the each AlternativePath. It then removes the last Transition of each built Sequence of Transitions. The helper finally selects, among all Transitions, those of the PrimitiveTrans type.

The **singleAltTransitions1** helper is a constant helper. It calculates the set of “single” PrimitiveTrans that are contained by a Path of an AlternativeTrans. For this purpose, it simply selects among the **altTransitions1** set, those whose *isMultiple* attribute is false.

The **multipleAltTransitions1** helper is a constant helper. It calculates the set of “multiple” PrimitiveTrans that are contained by a Path of an AlternativeTrans. For this purpose, it simply selects among the **altTransitions1** set, those whose *isMultiple* attribute is true.

The **altTransitions2** helper is a constant helper. It calculates the set of PrimitiveTrans that are contained by a Path that belongs to an AlternativeTrans and that are the last Transition their respective Path. To this end, the helper first gets all the Transitions contained by the each AlternativePath. It then selects the last Transition of each built Sequence of Transitions. The helper finally selects, among all Transitions, those of the PrimitiveTrans type.

The **getPath()** helper returns the Path that contains the contextual Transition. To this end, it simply selects, among all Paths, the one that contains the contextual Transition.

The **isLastOfPath()** helper returns a Boolean value stating whether the contextual Transition is the last of its Path. The helper first gets the Path of the contextual Transition, and then checks whether the last Transition of this Path is equal to the contextual Transition.

The **isFirstOfPath()** helper returns a Boolean value stating whether the contextual Transition is the first of its Path. The helper first gets the Path of the contextual Transition, and then checks whether the first Transition of this Path is equal to the contextual Transition.

The **getLoopTarget()** helper returns the Transition for which is generated the target State of the loop defined by the contextual PrimitiveTrans. Since a “multiple” PrimitiveTrans only leads to the generation of a loop Transition, the target of the loop is the State generated for the previous PrimitiveTrans. As a consequence, the helper first gets the Path of the contextual PrimitiveTrans, gets its index within the Transitions Sequence, and returns the Transition that precedes it in that Sequence.

The **loopIncoming()** helper returns a boolean value stating whether the contextual PrimitiveTrans precedes a “multiple” Transition in its Path (i.e. whether the State that is going to be generated for the contextual PrimitiveTrans is the target of a loop Transition). If the contextual PrimitiveTrans is the last Transition of its Path, the helper returns false. Otherwise, the helper returns the value of the *isMultiple* attribute of the Transition that follows the contextual PrimitiveTrans in the Path.

The **getLoopIncoming()** helper returns the loop PrimitiveTrans than follows the contextual Primitivetrans in the Path. The helper should only be called on a PrimitiveTrans that precedes a “multiple” PrimitiveTrans in its Path. The helper first gets the Path of the contextual PrimitiveTrans, gets its index within the Transitions Sequence, and returns the Transition that follows it in that Sequence.

The **getOutgoing()** helper is a recursive helper that returns the set of non-loop PrimitiveTrans that follows the contextual PrimitiveTrans. Returned PrimitiveTrans are those that are going to be matched into the following States of the contextual PrimitiveTrans. To this end, the helper is based on the following rules:

- If the next Transition is a “single” PrimitiveTrans, the helper returns this next PrimitiveTrans.
- Else if the next Transition is a “multiple” PrimitiveTrans, the helper returns the result of a recursive call of **getOutgoing()** on this next PrimitiveTrans.
- Else if the next Transition is an AlternativeTrans, the helper returns a Set composed of the first Transition of each alternative Path of this AlternativeTrans.

The **getPreviousTransition()** helper is a recursive helper that returns the Transition (either primitive or alternative) that precedes the contextual PrimitiveTrans in the input TextualPathExp. This helper should not be called onto the first Transition of a TextualPathExp. The helper first checks whether the contextual PrimitiveTrans is the first one of its Path. If not, the helper then checks whether the status of the *isMultiple* attribute of the preceding transition. If this Transition is a single Transition, it returns it. Otherwise, if the preceding Transition is a multiple one, the helper returns the result of a recursive call to **getPreviousTransition()** helper onto this preceding Transition. In case the contextual helper is the first Transition of its Path, the helper first gets the AlternativeTrans this Path belongs to. It then computes the Path the AlternativeTrans is defined in, and the Transition (either primitive or alternative), that precedes the computed AlternativetTrans within this new Path. If this transition is a “single” Transition, it is returned as the result of the helper call. If the transition is a “multiple” Transition, the helper returns the result of a recursive call to **getPreviousTransition()** helper onto the calculated preceding Transition.

1.4.1.3. Rules

The **Main** rule generates both a PathExp and its initial State for the input TextualPathExp element. The generated PathExp accepts an empty string as name. Its set of States corresponds to the States generated for the input single PrimitiveTrans that are not part of an AtlernativeTrans, for those that are part of an AlternativeTrans, for the States generated for each AlternativeTrans, and the initial State generated by the rule. Its set of Transitions corresponds to Transitions generated for each PrimitiveTrans, whatever the constant helper it belongs to. The incoming Transitions of the generated State correspond to an empty set. Its outgoing Transitions correspond to the Transitions generated for the root Transition (provided by the **rootTrans** helper) of the input TextualPathExp model.

The **AlternativeTrans** rule generates a State for each input AlternativeTrans element. Matched AlternativeTrans are supposed not to be “multiple” (see assumptions). The generated State corresponds to the State that closes the alternative transition in the built PathExp model. Its set of incoming Transitions corresponds to the Transitions generated for the last PrimitiveTrans of each alternative Path of the contextual AlternativeTrans. Its set of outgoing Transitions corresponds to the Transitions generated for the outgoing TextualPathExp Transitions (computed by the **getOutgoing()** helper) of the contextual AlternativeTrans.

The **SinglePrimitiveTrans** rule generates both a State and a Transition for each input “single” PrimitiveTrans that is not defined within an AlternativeTrans. The generated Transition is the PathExp Transition that targets the State generated by the rule. The name of the generated Transition is copied from the input PrimitiveTrans. If the contextual PrimitiveTrans is the TextualPathExp **rootTrans**, the source of the generated Transition corresponds to the State generated by the **Main** rule. Otherwise, the source of the generated Transition corresponds to the State generated for the Transition that precedes the contextual PrimitiveTrans in its Path. The target of the generated Transition corresponds to the State generated by the rule. The set of incoming transitions of the State generated by the rule contains to the generated Transition and the loop Transition that is generated for a potential loop (as stated by the **loopIncoming()** helper). In this case, the **getLoopIncoming()** helper returns the input Transition for which the loop PrimitiveTrans is generated. If the contextual PrimitiveTrans is the leaf Transition of the input TextualPathExp, the set of outgoing Transitions of the generated State is empty. Otherwise, the set of outgoing Transitions of the State corresponds to the Transitions generated for the TextualPathExp Transitions returned by the **getOugoing()** helper, and the loop Transition generated for a potential loop.

The **MultiplePrimitiveTrans** rule generates a Transition for each “multiple” input PrimitiveTrans that is not defined within an AlternativeTrans. The generated Transition corresponds to a simple loop within the built PathExp model. The name of the generated Transition is copied from the input PrimitiveTrans. If the input PrimitiveTrans is the **rootTrans** of the TextualPathExp, the source and the target of the generated Transition correspond to the State generated by the Main rule. Otherwise, they correspond to the State generated for the PrimitiveTrans returned by the **getLoopTarget()** helper.

The **SingleAltTrans1** rule generates both a Transition and a State for each “single” input PrimitiveTrans that belongs to an AlternativeTrans without being the last Transition of its alternative Path. The generated Transition is the PathExp Transition that targets the State generated by the rule. The name of the generated Transition is copied from the input PrimitiveTrans. The source of the generated Transition corresponds to the State generated for the TextualPathExp Transition returned by the **getPreviousTransition()** helper. Its target corresponds to the State generated by the rule. The set of incoming transitions of the State generated by the rule contains to the generated Transition and the loop Transition that is generated for a potential loop (as stated by the **loopIncoming()** helper). In this case, the **getLoopIncoming()** helper returns the input Transition for which the loop PrimitiveTrans is generated. The set of outgoing Transitions of the State corresponds to the Transitions generated for the TextualPathExp Transitions returned by the **getOugoing()** helper, and the loop Transition generated for a potential loop.

The **MultipleAltTrans2** rule generates a Transition for each “multiple” input PrimitiveTrans that is included into an AlternativeTrans without being the last Transition of its alternative Path. The generated Transition corresponds to a simple loop within the built PathExp model. The name of the generated Transition is copied from the input PrimitiveTrans. Its source and its target correspond to the State generated for the PrimitiveTrans returned by the **getLoopTarget()** helper.

The **AltTrans2** rule generates a Transition for each “single” input PrimitiveTrans which is included into an AlternativeTrans and which is the last Transition of its Path. The generated Transition corresponds to a simple loop within the built PathExp model. The name of the generated Transition is copied from the input PrimitiveTrans. The source of the generated Transition corresponds to the State generated for the TextualPathExp Transition returned by the **getPreviousTransition()** helper. Its target corresponds to the closing State generated for the AlternativeTrans that contains the contextual PrimitiveTrans.

```
module TextualPathExp2PathExp;
create OUT : PathExp from IN : TextualPathExp;


-------------------------------------------------------------------------------
-- HELPERS --------------------------------------------------------------------
-------------------------------------------------------------------------------

-- This helper returns the root TextualPathExp element of the input
-- TextualPathExp model.
-- CONTEXT: thisModule
-- RETURN:  TextualPathExp!TextualPathExp
helper def: root : TextualPathExp!TextualPathExp =
   TextualPathExp!TextualPathExp.allInstances()
      ->asSequence()->first();


-- This helper returns the 1st Transition element contained by the root
-- TextualPathExp model.
-- CONTEXT: thisModule
-- RETURN:  TextualPathExp!Transition
helper def: rootTrans : TextualPathExp!Transition =
   thisModule.root.path.transitions->first();


-- This helper returns the last Transition element contained by the root
-- TextualPathExp model.
-- CONTEXT: thisModule
-- RETURN:  TextualPathExp!Transition
helper def: leafTrans : TextualPathExp!Transition =
   thisModule.root.path.transitions->last();


-- This helper computes the Set containing all the Path elements of the input
-- TextualPathExp model.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!Path)
helper def: allPaths : Set(TextualPathExp!Path) =
   TextualPathExp!Path.allInstances();


-- This helper computes the Set of Path elements that are contained by
-- AlternativeTransition elements.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper def: altPaths : Set(TextualPathExp!Path) =
   thisModule.allPaths
      ->select(a |
         TextualPathExp!AlternativeTrans.allInstances()
            ->collect(b | b.altPaths)
            ->flatten()
            ->includes(a)
      );


-- This helper computes the Set of PrimitiveTrans that are not contained
-- by any AlternativeTransition.
-- To this end, it selects, among all Paths, those that are not contained
-- by any AlternativeTransition element. It then gets, for the selected Paths,
-- the transitions of type PrimitiveTrans.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper def: primTransitions : Set(TextualPathExp!PrimitiveTrans) =
   TextualPathExp!Path.allInstances()
      ->select(a |
         not TextualPathExp!AlternativeTrans.allInstances()
            ->collect(b | b.altPaths)
            ->flatten()
            ->includes(a)
      )
      ->collect(p | p.transitions)
      ->flatten()
      ->select(c | c.oclIsTypeOf(TextualPathExp!PrimitiveTrans));


-- This helper computes the Set of 'single' primitive transitions.
-- For this purpose, it selects in the primTransitions set, the transitions
-- whose 'isMultiple' attribute is set to false.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper def: singlePrimTransitions : Set(TextualPathExp!PrimitiveTrans) =
   thisModule.primTransitions->select(c | c.isMultiple = false);


-- This helper computes the Set of 'multiple' primitive transitions.
-- For this purpose, it selects in the primTransitions set, the transitions
-- whose 'isMultiple' attribute is set to true.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper def: multiplePrimTransitions : Set(TextualPathExp!PrimitiveTrans) =
   thisModule.primTransitions->select(c | c.isMultiple = true);


-- This helper computes the Set of PrimitiveTrans that are contained by an
-- AlternativeTransition, except those that are the last transition of their
-- Path.
-- To this end, the helper first collects the transitions contained by each
-- alternative path. For each collected sequence of transitions of size S, it
-- gets the (S-1 ) first transition. Finally, it selects in the built sequence
-- the transitions of type PrimitiveTrans.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper def: altTransitions1 : Set(TextualPathExp!PrimitiveTrans) =
   thisModule.altPaths
      ->collect(p | p.transitions)
      ->iterate(e;
              res : Sequence(Sequence(TextualPathExp!Transition)) = Set{} |
         res->including(e->subSequence(1, e->size()-1))
      )
      ->asSequence()
      ->flatten()
      ->select(c | c.oclIsTypeOf(TextualPathExp!PrimitiveTrans));


-- This helper computes the Set of 'single' alternative transitions.
-- For this purpose, it selects in the altTransitions1 set, the transitions
-- whose 'isMultiple' attribute is set to false.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper def: singleAltTransitions1 : Set(TextualPathExp!PrimitiveTrans) =
   thisModule.altTransitions1->select(c | c.isMultiple = false);


-- This helper computes the Set of 'multiple' alternative transitions.
-- For this purpose, it selects in the altTransitions1 set, the transitions
-- whose 'isMultiple' attribute is set to true.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper def: multipleAltTransitions1 : Set(TextualPathExp!PrimitiveTrans) =
   thisModule.altTransitions1->select(c | c.isMultiple = true);


-- This helper computes the Set of PrimitiveTrans that are contained by an
-- AlternativeTransition and that are the last transition of their Path (which
-- may also be the first transition if the path contains a singel transition).
-- To this end, the helper first collects the transitions contained by each
-- alternative path. For each collected sequence of transitions of size S, it
-- gets the transition number S. Finally, it selects in the built sequence
-- the transitions of type PrimitiveTrans.
-- CONTEXT: thisModule
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper def: altTransitions2 : Set(TextualPathExp!PrimitiveTrans) =
   thisModule.altPaths
      ->collect(p | p.transitions)
      ->iterate(e;
              res : Sequence(Sequence(TextualPathExp!Transition)) = Set{} |
         res->including(e->last())
      )
      ->asSequence()
      ->flatten()
      ->select(c | c.oclIsTypeOf(TextualPathExp!PrimitiveTrans));


-- This helper computes the containing Path of the contextual Transition
-- element.
-- For this purpose, it selects ammong all Paths, the one that contains the
-- contextual Transition elements.
-- CONTEXT: TextualPathExp!Transition
-- RETURN:  TextualPathExp!Path
helper context TextualPathExp!Transition
   def: getPath() :  TextualPathExp!Path =
   thisModule.allPaths
      ->select(a | a.transitions->includes(self))
      ->first();


-- This helper computes a boolean value assessing whether or not the contextual
-- PrimitiveTrans is the last transition of its Path.
-- To this end, the helper first gets the path of the contextual transition (by
-- means of the 'getPath' helper) and then compares the contextual transition
-- to the last transition of the path.
-- CONTEXT: TextualPathExp!PrimitiveTrans
-- RETURN:  TextualPathExp!Transition
helper context TextualPathExp!PrimitiveTrans
   def: isLastOfPath() : Boolean =
   let p : TextualPathExp!Path = self.getPath()
   in self = p.transitions->last();


-- This helper computes a boolean value assessing whether or not the contextual
-- PrimitiveTrans is the first transition of its Path.
-- To this end, the helper first gets the path of the contextual transition (by
-- means of the 'getPath' helper) and then compares the contextual transition
-- to the first transition of the path.
-- CONTEXT: TextualPathExp!PrimitiveTrans
-- RETURN:  TextualPathExp!Transition
```

```
helper context TextualPathExp!PrimitiveTrans
   def: isFirstOfPath() : Boolean =
   let p : TextualPathExp!Path = self.getPath()
   in self = p.transitions->first();


-- This helper computes the Transition for which is generated the target state
-- of the loop defined by the contextual PrimitiveTrans. A multiple primitive
-- transition only leads to the generation of a loop transition. As a
-- consequence, the computed Transition is the one preceding the contextual
-- primitive transition in their path. The contextual primitrive transition
-- should therefore not be the first of its path.
-- CONTEXT: TextualPathExp!PrimitiveTrans
-- RETURN:  TextualPathExp!Transition
helper context TextualPathExp!PrimitiveTrans
   def: getLoopTarget() : TextualPathExp!Transition =
   let p : TextualPathExp!Path = self.getPath()
   in let i : Integer = p.transitions->indexOf(self)
   in p.transitions->at(i-1);


-- This helper computes a boolean value assessing whether or not the contextual
-- PrimitiveTrans is preceding a multiple transition in its Path.
-- If the contextual PrimitiveTrans is the last transition of its Path, the
-- helper returns false. Otherwise, it returns the value of the 'isMultiple'
-- attribute of the next transition in the path.
-- CONTEXT: TextualPathExp!PrimitiveTrans
-- RETURN:  Boolean
helper context TextualPathExp!PrimitiveTrans
   def: loopIncoming() : Boolean =
   let p : TextualPathExp!Path = self.getPath()
   in let i : Integer = p.transitions->indexOf(self)
   in if self = p.transitions->last() then
         false
      else
         p.transitions->at(i+1).isMultiple
      endif;


-- This helper computes the incoming/outgoing loop Transition of the contextual
-- multiple PrimitiveTrans.
-- For this purpose, it returns the next transition in the path.
-- PRECOND: this helper should only be called from a PrimTransition that
-- precedes a multiple PrimitiveTrans.
-- CONTEXT: TextualPathExp!PrimitiveTrans
-- RETURN:  TextualPathExp!Transition
helper context TextualPathExp!PrimitiveTrans
   def: getLoopIncoming() : TextualPathExp!Transition =
   let p : TextualPathExp!Path = self.getPath()
   in let i : Integer = p.transitions->indexOf(self)
   in p.transitions->at(i + 1);


-- This helper computes the set of primitive transitions (except loop
-- transitions) that follow the contextual transition.
-- For this purpose, the helper first gets the transition next to the
-- contextual transition in the same path.
-- If this following transition is a PrimitiveTrans and is not multiple, the
-- helper returns the transition. If the following transition is a multiple
-- PrimitiveTrans, then the helper looks for the transitions that follow this
-- next transition by means of a recursive call onto this "next transition".
-- If the following transition is an AlternativeTrans, the helper collects the
-- first transition of each alternative path of the AlternativeTrans, and
-- returns the calculated set.
-- CONTEXT: TextualPathExp!Transition
-- IN:      Integer
-- RETURN:  Set(TextualPathExp!PrimitiveTrans)
helper context TextualPathExp!Transition
```

```
   def: getOutgoing() : Set(TextualPathExp!PrimitiveTrans) =
   let p : TextualPathExp!Path = self.getPath()
   in let i : Integer = p.transitions->indexOf(self)
   in let t : TextualPathExp!Transition = p.transitions->at(i + 1)
   in if t.oclIsTypeOf(TextualPathExp!PrimitiveTrans) then
         if not t.isMultiple then
            Set{t}
         else
            t.getOutgoing()
         endif
      else
      t.altPaths
         ->iterate(e; res : Set(TextualPathExp!PrimitiveTrans) = Set{} |
            res->including(e.transitions->first())
         )
      endif;


-- This helper computes the Transition (primitive or alternative) that precedes
-- the contextual PrimitiveTrans in the input TextualPathExp model.
-- To this end, the helper first checks whether or not the contextual
-- PrimitiveTrans is the first transition of its Path.
-- If the contextual transition is the first of its path, the helper first gets
-- the AlternativeTrans the contextual transition belongs to. It then gets the
-- Path in which this AlternativeTrans is defined, and the rank of the
-- AlternativeTrans within this Path. From then, it gets the transition that
-- precedes the computed AlternativeTrans. The helper returns this preceding
-- transition if it is not multiple. If the preceding transition is multiple,
-- the helper returns the transition that precedes this preceding transition
-- by means of a recursive call of the helper onto the transition that precedes
-- the AlternativeTrans.
-- If the contextual transition is not the first of its path, the helper
-- returns its preceding transition if this last is not multiple. If the
-- preceding transition is multiple, the helper returns the preceding
-- transition of the preceding transition by means of a recursive call of the
-- helper onto the transition preceding the contextual transition.
-- PRECOND: this helper should not be called on the root Transition of the
-- input model.
-- CONTEXT: TextualPathExp!PrimitiveTrans
-- RETURN:  TextualPathExp!Transition
helper context TextualPathExp!PrimitiveTrans
   def: getPreviousTransition() : TextualPathExp!Transition =
   let p : TextualPathExp!Path = self.getPath() in
   if self.isFirstOfPath() then
      let alt : TextualPathExp!AlternativeTrans =
         TextualPathExp!AlternativeTrans.allInstances()
            ->select(a | a.altPaths->includes(p))
            ->first()
      in let p2 : TextualPathExp!Path =
         thisModule.allPaths
            ->select(a | a.transitions->includes(alt))
            ->first()
      in let i : Integer = p2.transitions->indexOf(alt)
      in let t : TextualPathExp!Transition =
         p2.transitions->at(i-1) in
      if t.isMultiple then
         t.getPreviousTransition()
      else
         t
      endif
   else
      let i : Integer = p.transitions->indexOf(self)
      in let t : TextualPathExp!Transition =
         p.transitions->at(i-1) in
      if t.isMultiple then
         t.getPreviousTransition()
      else
         t
      endif
   endif;


-----------------------------------------------------------------------------
-- RULES --------------------------------------------------------------------
-----------------------------------------------------------------------------

-- Rule 'Main'
-- This rule generates both a PathExp element and its initial State element
-- from the input root TextualPathExp element.
-- The generated PathExp element accepts an empty string as name. Its set of
-- states corresponds to the 'pe_s' elements generated for the input elements
-- of the singlePrimTransitions, singleAltTransitions sets, plus the 'pe_s'
-- elements generated for AlternativeTransition, plus the initial State
-- generated by the current rule. Its set of transitions corresponds to the
-- 'pe_t' elements generated for the input elements in the primTransitions,
-- altTransitions1, and altTransitions2 sets.
-- The generated initial State has an empty set of incoming transitions. Its
-- set of outgoing transitions corresponds to the 'pe_t' elements that are
-- generated for the outgoing transitions computed by the getOutgoing(0) call.
rule Main {
   from
        tpe : TextualPathExp!TextualPathExp
   to
      pe : PathExp!PathExp (
         name <- '',
         states <- Set{
                  thisModule.singlePrimTransitions
                     ->collect(e | thisModule.resolveTemp(e, 'pe_s')),
                  thisModule.singleAltTransitions1
                     ->collect(e | thisModule.resolveTemp(e, 'pe_s')),
                  TextualPathExp!AlternativeTrans.allInstances()
                     ->collect(e | thisModule.resolveTemp(e, 'pe_s')),
                  pe_s
                  },
         transitions <- Set{
                  thisModule.primTransitions
                     ->collect(e | thisModule.resolveTemp(e, 'pe_t')),
                  thisModule.altTransitions1
                     ->collect(e | thisModule.resolveTemp(e, 'pe_t')),
                  thisModule.altTransitions2
                     ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
                     }
      ),

      pe_s : PathExp!State (
         incoming <- Set{},
         outgoing <- Set{thisModule.rootTrans}
                     ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
      )
}


-- Rule 'AlternativeTrans'
-- This rule generates the State element that closes an input
-- AlternativeTransition element. The generated State is the one at which the
-- different alternative paths of the AlternativeTransition join.
-- Incoming transitions of the generated state correspond to the elements
-- generated for the last alternative transitions of the input
-- AlternativeTransition element.
-- Outgoing transitions of the generated state correspond to the 'pe_t'
-- elements generated for the set of transitions returned by the call of
-- getOutgoing(1).
rule AlternativeTrans {
   from
      tpe_at : TextualPathExp!AlternativeTrans (
--       tpe_at.isMultiple = false
```

```
         true
      )
   to
      pe_s : PathExp!State (
         incoming <- thisModule.altTransitions2
                     ->select(a | tpe_at.altPaths
                        ->collect(b | b.transitions)
                        ->flatten()
                        ->includes(a)
                     ),
         outgoing <- tpe_at.getOutgoing()
                     ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
      )
}


-- Rule ''
-- This rule generates ...
--rule MultipleAlternativeTrans {
-- from
--    tpe_at : TextualPathExp!AlternativeTrans (
--       tpe_at.isMultiple = true
--    )
-- to
--    pe_s : PathExp!State (
--       outgoing <- Set{
--                tpe_at.getOutgoing()
--                   ->collect(e | thisModule.resolveTemp(e, 'pe_t')),
--                pe_t}
--    ),
--
--    pe_t : PathExp!Transition (
--       name <- ''--,
--       target <- pe_s
--    )
--}


-- Rule 'SinglePrimitiveTrans'
-- This rule generates both a Transition and a State for each PrimitiveTrans
-- element that belongs to the 'singlePrimTransitions' set.
-- The generated transition accepts as name the name of the input
-- PrimitiveTrans. If the input PrimitiveTrans is the root transition of the
-- input model, its source corresponds to the 'pe_s' initial state generated
-- for the input TextualPathExp element by rule 'Main'. Otherwise, the source
-- element corresponds to the 'pe_s' element generated for the transition that
-- precedes the input PrimitiveTrans in the current Path. Its target is the
-- State generated by the rule.
-- Incoming transitions for the generated State include the Transition
-- generated by the rule and, when the input Transition precedes a multiple
-- transition, the 'pe_t' element generated for this next transition.
-- If the input PrimitiveTrans is the leaf transition of the input model, the
-- generated State has no outgoing transitions. Otherwise, its outgoing
-- transition corresponds to the 'pe_t' element generated for the input
-- transition returned by the call of getOugoing(). Moreover, if the input
-- Transition precedes a multiple transition, the 'pe_t' element generated for
-- this next transition is added to the set ougoing transitions of the
-- generated State.
rule SinglePrimitiveTrans {
   from
       tpe_pt : TextualPathExp!PrimitiveTrans (
         thisModule.singlePrimTransitions->includes(tpe_pt)
         )
   to
      pe_t : PathExp!Transition (
         name <- tpe_pt.name,
         source <-
               if tpe_pt = thisModule.rootTrans then
```

```
                  thisModule.resolveTemp(thisModule.root, 'pe_s')
               else
                  let p : TextualPathExp!Path = tpe_pt.getPath()
                  in let i : Integer = p.transitions->indexOf(tpe_pt)
                  in let t : TextualPathExp!Transition =
                     p.transitions->at(i-1)
                  in thisModule.resolveTemp(t, 'pe_s')
               endif,
         target <- pe_s
      ),

      pe_s : PathExp!State (
         incoming <- Set{pe_t}->union(
                     if tpe_pt.loopIncoming() then
                        Set{thisModule.resolveTemp(tpe_pt.getLoopIncoming(), 'pe_t')}
                     else
                        Set{}
                     endif
                  ),
         outgoing <- if tpe_pt = thisModule.leafTrans then
                     Set{}
                  else
                     tpe_pt.getOutgoing()
                        ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
                        ->union(
                        if tpe_pt.loopIncoming() then
                           Set{thisModule.resolveTemp(tpe_pt.getLoopIncoming(), 'pe_t')}
                        else
                           Set{}
                        endif
                        )
                  endif
      )
}


-- Rule 'MultiplePrimitiveTrans'
-- This rule generates a loop transition for each transition that belongs
-- to the 'multiplePrimTransitions' set. The generated transition is a
-- transition from and to the state generated for the previous input
-- transition.
-- The generated loop transition accepts the name of the input Transition as
-- name.
-- If the input PrimitiveTrans is the root transition of the input model, its
-- source is the initial State generated by the 'Main' rule. Otherwise, the
-- source is computed by the getLoopTarget() helper.
-- If the input PrimitiveTrans is the root transition of the input model, its
-- target is the initial State generated by the 'Main' rule. Otherwise, the
-- target is computed by the getLoopTarget() helper.
rule MultiplePrimitiveTrans {
   from
       tpe_pt : TextualPathExp!PrimitiveTrans (
         thisModule.multiplePrimTransitions->includes(tpe_pt)
         )
   to
      pe_t : PathExp!Transition (
         name <- tpe_pt.name,
         source <-
               if tpe_pt = thisModule.rootTrans then
                  thisModule.resolveTemp(thisModule.root, 'pe_s')
               else
                  tpe_pt.getLoopTarget()
               endif,
         target <-
               if tpe_pt = thisModule.rootTrans then
                  thisModule.resolveTemp(thisModule.root, 'pe_s')
               else
                  tpe_pt.getLoopTarget()
               endif
      )
}


-- Rule 'SingleAltTrans1'
-- This rule generates both a Transition and a State for each PrimitiveTrans
-- element that belongs to the 'singleAltTransitions1' set.
-- The generated transition accepts as name the name of the input
-- PrimitiveTrans. Its source element corresponds to the 'pe_s' element
-- generated for the transition returned by the call of the
-- 'getPrevioustransition' helper. Its target is the State generated by the
-- rule.
-- Incoming transitions for the generated State include the Transition
-- generated by the rule and, when the input Transition precedes a multiple
-- transition, the 'pe_t' element generated for this next transition.
-- Outgoing transitions for the generated State include to the 'pe_t' element
-- generated for the input transition returned by the call of getOugoing(1).
-- Moreover, if the input Transition precedes a multiple transition, the 'pe_t'
-- element generated for this next transition is added to the set ougoing
-- transitions of the generated State.
rule SingleAltTrans1 {
   from
       tpe_pt : TextualPathExp!PrimitiveTrans (
         thisModule.singleAltTransitions1->includes(tpe_pt)
         )
   to
      pe_t : PathExp!Transition (
         name <- tpe_pt.name,
         source <-
               thisModule.resolveTemp(
                  tpe_pt.getPreviousTransition(),
                  'pe_s'
               ),
         target <- pe_s
      ),

      pe_s : PathExp!State (
         incoming <- Set{pe_t}->union(
                     if tpe_pt.loopIncoming() then
                        Set{thisModule.resolveTemp(tpe_pt.getLoopIncoming(), 'pe_t')}
                     else
                        Set{}
                     endif
                  ),
         outgoing <- tpe_pt.getOutgoing()
                     ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
                     ->union(
                     if tpe_pt.loopIncoming() then
                        Set{thisModule.resolveTemp(tpe_pt.getLoopIncoming(), 'pe_t')}
                     else
                        Set{}
                     endif
                     )
      )
}


-- Rule 'MultipleAltTrans1'
-- This rule generates a loop transition for each transition that belongs
-- to the 'multipleAltTransitions1' set. The generated transition is a
-- transition from and to the state generated for the previous input
-- transition.
-- The generated loop transition accepts the name of the input Transition as
-- name.
-- Its source corresponds to the 'pe_s' element generated for the input State
-- returned by the call to the getLoopTarget() helper.
-- Its target corresponds to the 'pe_s' element generated for the input State
-- returned by the call to the getLoopTarget() helper.
rule MultipleAltTrans1 {
   from
       tpe_pt : TextualPathExp!PrimitiveTrans (
         thisModule.multipleAltTransitions1->includes(tpe_pt)
         )
   to
      pe_t : PathExp!Transition (
         name <- tpe_pt.name,
         source <- thisModule.resolveTemp(tpe_pt.getLoopTarget(), 'pe_s'),
         target <- thisModule.resolveTemp(tpe_pt.getLoopTarget(), 'pe_s')
      )
}


-- Rule 'AltTrans2'
-- This rule generates a Transition from the last Transition of a Path
-- contained by an AlternativeTransition. The generated transition goes from
-- the state generated for the previous transition to the final state generated
-- for the current AlternativeTransition by the 'AlternativeTrans' helper.
-- The generated loop transition accepts the name of the input Transition as
-- name.
-- Its source corresponds to the 'pe_s' element generated for the input element
-- returned by the call of the 'getPreviousTransition()' helper.
-- Its target corresponds to the 'pe_s' element generated for the
-- AlternativeTransition element that contains the rule input PrimitiveTrans
-- element in one of its alternative pathes.
rule AltTrans2 {
   from
       tpe_pt : TextualPathExp!PrimitiveTrans (
         thisModule.altTransitions2->includes(tpe_pt)
         )
   to
      pe_t : PathExp!Transition (
         name <- tpe_pt.name,
         source <- thisModule.resolveTemp(
                  tpe_pt.getPreviousTransition(),
                  'pe_s'),
         target <- thisModule.resolveTemp(
                  TextualPathExp!AlternativeTrans.allInstances()
                     ->select(a | a.altPaths
                        ->collect(b | b.transitions)
                        ->flatten()
                        ->includes(tpe_pt)
                     )->asSequence()
                     ->first(),
                  'pe_s')
      )
}
```

1.4.2. The PathExp2PetriNet transformation

The ATL code for the PathExp to PetriNet transformation consists of 1 helper and 3 rules.

1.4.2.1. Helpers

The **allTransitions** helper is a constant helper. It calculates a Set that contains all the Transition model elements of the input PetriNet model.

1.4.2.2. Rules

The **Main** rule generates a PetriNet element from the input PathExp element. Name of the generated PetriNet element is copied from the one the PathExp. Its set of Places corresponds to the Places generated for the input State elements. Its set of Transitions corresponds to output Transitions generated for the input Transition elements. Finally, its set of Arcs corresponds to the PlaceToTransArc and TransToPlaceArcs elements generated for the input Transition elements.

The **State** rule generates a Place element for each PathExp State input element. Generated Place accepts an empty string as name. Its set of incoming arcs corresponds to the TransToPlaceArcs generated for the incoming Transitions of the input PathExp State. Its set of outgoing arcs corresponds to the PlaceToTransArcs generated for the outgoing Transitions of the input PathExp State.

The **Transition** rule generates a PetriNet Transition, a PlaceToTransArc and a TransToPlaceArc for each input PathExp Transition. The generated Transition accepts an empty string as name. Its set of incoming arcs corresponds to the generated PlaceToTransArc (“pn_ia”). Its set of outgoing arcs corresponds to the generated TransToPlaceArc (“pn_oa”). The generated PlaceToTransArc weight is set to 1. Its source corresponds to the Place generated for the source of the input PathExp Transition. Its target corresponds to the generated Transition (“pn_t”). The generated TransToPlaceArc weight is set to 1. Its source corresponds to the generated Transition (“pn_t”). Its target corresponds to the Place generated for the target of the input PathExp Transition.

```
module PathExp2PetriNet;
create OUT : PetriNet from IN : PathExp;


-------------------------------------------------------------------------------
-- HELPERS --------------------------------------------------------------------
-------------------------------------------------------------------------------


-- This helper computes the Set containing all the Transitions of the input
-- PathExp model.
-- CONTEXT: thisModule
-- RETURN:  Set(PathExp!Transition)
helper def: allTransitions : Set(PathExp!Transition) =
   PathExp!Transition.allInstances();


-------------------------------------------------------------------------------
-- RULES ----------------------------------------------------------------------
-------------------------------------------------------------------------------


-- Rule 'Main'
-- This rule generates a PetriNet element from the input PathExp element.
-- The name of the generated PetriNet is copied from the input PathExp element.
-- Its set of places and its set of transitions respectively correspond to the
-- elements generated for states and the transitions of the input PathExp.
-- Its set of arcs correspond to the 'pn_ia' and 'pn_oa' elements generated for
-- the input Transition elements.
rule Main {
   from
        pe : PathExp!PathExp
   to
      pn : PetriNet!PetriNet (
         name <- pe.name,
         places <- pe.states,
         transitions <- pe.transitions,
         arcs <- thisModule.allTransitions
                  ->collect(e | thisModule.resolveTemp(e, 'pn_ia'))
                  ->union(
                     thisModule.allTransitions
                     ->collect(e | thisModule.resolveTemp(e, 'pn_oa'))
                  )
      )
}


-- Rule 'State'
-- This rule generates a Place element from an input State element.
-- Generated Place accepts an empty string as name.
-- Its set of incoming arcs correspond to 'pn_oa' elements that are generated
-- for the incoming Transitions of the input State.
-- Its set of outgoing arcs correspond to 'pn_ia' elements that are generated
-- for the outgoing Transitions of the input State.
rule State {
   from
       pe_s : PathExp!State
   to
      pn_p : PetriNet!Place (
         name <- '',
         incoming <- pe_s.incoming
                     ->collect(e | thisModule.resolveTemp(e, 'pn_oa')),
         outgoing <- pe_s.outgoing
                     ->collect(e | thisModule.resolveTemp(e, 'pn_ia'))
      )
}


-- Rule 'Transition'
-- From an input PathExp Transition, this rule generates:
--   * a PetriNet Transition
--   * a PlaceToTransArc
--   * a TransToPlaceArc
-- The generated Transition accepts an empty string as name, the generated
-- 'pn_ia' PlaceToTransArc as incoming arc, and the generated 'pn_oa'
-- TransToPlaceArc as outgoing arc.
-- The generated PlaceToTransArc accepts the element generated for the source
-- of the input PathExpTransition as source, and the generated PetriNet
-- Transition as target.
-- The generated TransToPlaceArc accepts the generated PetriNet Transition as
-- source, and the element generated for the target of the input
-- PathExpTransition as target.
rule Transition {
   from
       pe_t : PathExp!Transition
   to
      pn_t : PetriNet!Transition (
         name <- '',
         incoming <- pn_ia,
         outgoing <- pn_oa
      ),

      pn_ia : PetriNet!PlaceToTransArc (
         source <- pe_t.source,
         target <- pn_t,
         weight <- 1
      ),

      pn_oa : PetriNet!TransToPlaceArc (
         source <- pn_t,
         target <- pe_t.target,
         weight <- 1
      )
}
```

1.4.3. The PetriNet2XML transformation

The ATL code for the PetriNet to XML transformation consists of 3 helpers and 5 rules.

1.4.3.1. Helpers

The first helper, **allPlaces**, is a constant helper. It calculates a Sequence that contains all the Place model elements of the input PetriNet model.

The **allTransitions** helper is a constant helper. It calculates a Sequence that contains all the Transition model elements of the input PetriNet model.

The **allArcs** helper is a constant helper. It calculates a Sequence that contains all the Arc (PlaceToTransArc and TransToPlaceArc ones) model elements of the input PetriNet model.

1.4.3.2. Rules

Besides helpers, the UML to Amble transformation is composed of 5 rules.

The **Main** rule generates the XML Root element as well as a collection of 3 Attributes, 3 Elements and a Text node from the PetriNet input element. The generated Root element is a “pnml” tag that has an “xmlns” Attribute and a “net” Element as children. Value of the “xmlns” attribute is the “http://www.example.org/pnpl” constant string. The “net” Element has an “id” and a “type” Attribute, a “name” sub-Element, as well as the Elements generated for each input element of the **allPlaces**, **allTransitions** and **allArcs** Sequences. The “id” attribute corresponds to a constant value (not used here), whereas the “type” attribute contains the “http://www.example.org/pnpl/PTNet” constant string. Finally, the “name” Element contains a “text” Element, which itself contains a Text node whose value corresponds to the name of the input PetriNet element.

The **Place** rule generates three XML Elements, one XML Attribute and one XML Text for each PetriNet Place input element. The first generated Element, “xml_place”, is a “place” tag which accepts an “id” Attribute as well as a child “name” Element. The value of the “id” attribute corresponds to the index of the input Place in the **allPlaces** Sequence.

The **Transition** rule generates both a XML Element and a XML Attribute for each PetriNet Transition input element. The generated element is a “transition” tag that accepts the generated “id” Attribute as attribute. The value of this generated attribute corresponds to the size of the **allPlaces** Sequence plus the index of the input Transition in the **allTransitions** Sequence. The generated “name” Element accepts a “text” Element as child. This last one has a child which is a Text node. Its value corresponds to the name of the input Place.

The **PlaceToTransArc** rule generates a XML Element with three XML Attributes for each PetriNet PlaceToTransArc. The generated Element is an “arc” tag that has three Attribute children: “id”, “source” and “target”. The value of the “id” attribute corresponds to the size of the **allPlaces** Sequence plus the size of the **allTransitions** Sequence plus the index of the input PlaceToTransArc in the **allArcs** Sequence. The value of the “source” attribute corresponds to the index of the source of the input PlaceToTransArc in the **allPlaces** Sequence. Finally, the value of the “target” attribute corresponds to the size of the **allPlaces** Sequence plus the index of the target of the input PlaceToTransArc in the **allTransitions** Sequence.

The **TransToPlaceArc** rule generates a XML Element with three XML Attributes for each PetriNet TransToPlaceArc. The generated Element is an “arc” tag that has three Attribute children: “id”, “source” and “target”. The value of the “id” attribute corresponds to the size of the **allPlaces** Sequence plus the size of the **allTransitions** Sequence plus the index of the input PlaceToTransArc in the **allArcs** Sequence. The value of the “source” attribute corresponds to the size of the **allPlaces** Sequence plus the index of the source of the input TransToPlaceArc in the **allTransitions** Sequence. Finally, the value of the “target” attribute corresponds to the index of the target of the input TransToPlaceArc in the **allPlaces** Sequence.

```
module PetriNet2XML;
create OUT : XML from IN : PetriNet;


-------------------------------------------------------------------------------
-- HELPERS --------------------------------------------------------------------
-------------------------------------------------------------------------------


-- This helper computes a Sequence that contains all the Places of the input
-- PetriNet model.
-- CONTEXT: thisModule
-- RETURN:  Sequence(PetriNet!Place)
helper def: allPlaces : Sequence(PetriNet!Place) =
   PetriNet!Place.allInstances()->asSequence();


-- This helper computes a Sequence that contains all the Transitions of the
-- input PetriNet model.
-- CONTEXT: thisModule
-- RETURN:  Sequence(PetriNet!Transition)
helper def: allTransitions : Sequence(PetriNet!Transition) =
   PetriNet!Transition.allInstances()->asSequence();


-- This helper computes a Sequence that contains all the Arcs of the input
-- PetriNet model.
-- CONTEXT: thisModule
-- RETURN:  Sequence(PetriNet!Arc)
helper def: allArcs : Sequence(PetriNet!Arc) =
   PetriNet!Arc.allInstances()->asSequence();


-------------------------------------------------------------------------------
-- RULES ----------------------------------------------------------------------
-------------------------------------------------------------------------------


-- Rule 'Main'
-- This rule generates the "pnml" root tag from the input PetriNet element.
-- This tag has an "xmlns" attribute and a "net" element as child element.
-- The "net" tag has an "id", a "type" and a "name" attributes, and the
-- following children elements:
--   * a "place" element for each Place of the input PetriNet model
--   * a "transition" element for each Transition of the input PetriNet model
--   * an "arc" element for each Arc of the input PetriNet model.
rule Main {
   from
        pn : PetriNet!PetriNet
   to
      root : XML!Root (
         name <- 'pnml',
         children <- Sequence{xmlns, net}
      ),
      xmlns: XML!Attribute (
         name <- 'xmlns',
         value <- 'http://www.example.org/pnpl'
      ),

      net : XML!Element (
         name <- 'net',
         children <- Sequence{
                     id,
                     type,
                     name,
                     thisModule.allPlaces,
                     thisModule.allTransitions,
                     thisModule.allArcs
                  }
      ),
      id : XML!Attribute (
         name <- 'id',
         value <- 'n1'
      ),
      type : XML!Attribute (
         name <- 'type',
         value <- 'http://www.example.org/pnpl/PTNet'
      ),

      name : XML!Element (
         name <- 'name',
         children <- Sequence{text}
      ),
      text : XML!Element (
         name <- 'text',
         children <- Sequence{val}
      ),
      val : XML!Text (
         value <- pn.name
      )
}


-- Rule 'Place'
-- This rule generates a "place" tag from an input Place element.
-- This tag has an "id" attribute which value corresponds to the Place rank
-- within the allPlaces sequence.
-- The "place" tag also has a "name" child element, which has itself a "text"
-- child element that contains the name of the place (copied from the input
-- Place element).
rule Place {
   from
        pn_s : PetriNet!Place
   to
      xml_place : XML!Element (
         name <- 'place',
         children <- Sequence{id, name}
      ),
      id : XML!Attribute (
         name <- 'id',
         value <- thisModule.allPlaces->indexOf(pn_s).toString()
      ),

      name : XML!Element (
         name <- 'name',
         children <- Sequence{text}
      ),
      text : XML!Element (
         name <- 'text',
         children <- Sequence{val}
      ),
      val : XML!Text (
         value <- pn_s.name
      )
}
```

```
-- Rule 'Transition'
-- This rule generates a "transition" tag from an input Transition element.
-- This tag has an "id" attribute which value corresponds to (the size of the
-- allPlaces sequence + the Transition rank within the allTransitions
-- sequence).
rule Transition {
   from
       pn_t : PetriNet!Transition
   to
      xml_trans : XML!Element (
         name <- 'transition',
         children <- Sequence{trans_id}
      ),
      trans_id : XML!Attribute (
         name <- 'id',
         value <- (thisModule.allPlaces->size() +
                 thisModule.allTransitions->indexOf(pn_t)).toString()
      )
}


-- Rule 'PlaceToTransArc'
-- This rule generates an "arc" tag from an input PlaceToTransArc element.
-- This tag has an "id", a "source" and a "target" attributes.
-- Value of the "id" attribute corresponds to (the size of the allPlaces
-- sequence + the size of the allTransitions sequence + the Arc rank within
-- the allArcs sequence).
-- Value of the "source" attribute corresponds to the source Place rank
-- within the allPlaces sequence.
-- Value of the "target" attribute corresponds to (the size of the allPlaces
-- sequence + the target Transition rank within the allTransitions sequence).
rule PlaceToTransArc {
   from
      pn_a : PetriNet!PlaceToTransArc
   to
      xml_arc : XML!Element (
         name <- 'arc',
         children <- Sequence{id, source, target}
      ),
      id : XML!Attribute (
         name <- 'id',
         value <- (thisModule.allPlaces->size() +
                 thisModule.allTransitions->size() +
                 thisModule.allArcs->indexOf(pn_a)).toString()
      ),
      source : XML!Attribute (
         name <- 'source',
         value <- thisModule.allPlaces
                  ->indexOf(pn_a.source).toString()
      ),
      target : XML!Attribute (
         name <- 'target',
         value <- (thisModule.allPlaces->size() +
                 thisModule.allTransitions
                  ->indexOf(pn_a.target)).toString()
      )
}


-- Rule 'TransToPlaceArc'
-- This rule generates an "arc" tag from an input TransToPlaceArc element.
-- This tag has an "id", a "source" and a "target" attributes.
-- Value of the "id" attribute corresponds to (the size of the allPlaces
-- sequence + the size of the allTransitions sequence + the Arc rank within
-- the allArcs sequence).
-- Value of the "source" attribute corresponds to (the size of the allPlaces
-- sequence + the source Transition rank within the allTransitions sequence).
-- Value of the "target" attribute corresponds to the target Place rank
-- within the allPlaces sequence.
rule TransToPlaceArc {
   from
      pn_a : PetriNet!TransToPlaceArc
   to
      xml_arc : XML!Element (
         name <- 'arc',
         children <- Sequence{id, source, target}
      ),
      id : XML!Attribute (
         name <- 'id',
         value <- (thisModule.allPlaces->size() +
                 thisModule.allTransitions->size() +
                 thisModule.allArcs->indexOf(pn_a)).toString()
      ),
      source : XML!Attribute (
         name <- 'source',
         value <- (thisModule.allPlaces->size() +
                 thisModule.allTransitions
                  ->indexOf(pn_a.source)).toString()
      ),
      target : XML!Attribute (
         name <- 'target',
         value <- thisModule.allPlaces
                  ->indexOf(pn_a.target).toString()
      )
}
```

2. ATL Transformation: Petri nets to path expressions

2.1. Introduction

The Petri nets to path expression example describes the reverse transformation of the one described in Section 1. This section provides an overview of the whole transformation sequence that enables to produce a textual definition of a path expression from a XML Petri net representation (in the PNML format [1]).

The input metamodel of this transformation sequence is the XML metamodel. Indeed, the PNML XML textual representation of the Petri net is first injected into a XML model (this part is out of the scope of the document). The XML model is then transformed into a PetriNet model that describes the structure of the encoded Petri net. The PetriNet model can then be transformed into a PathExp model, which defines the structure of a path expression as it is expressed in a graphical way. The PathExp model is then transformed into a TextualPathExp that encodes the same path expression according to the semantics of its textual representation. Finally, the TextualPathExp model is extracted to a textual representation of the path expression by means of a TCS (Textual Concrete Syntax) program. This last step is not documented in this document.

2.2. Metamodels

This transformation sequence is based on the same four metamodels that the path expression to Petri nets transformation sequence: XML, PetriNet, PathExp, and TextualPathExp. Description of these metamodels can be found in Section 1.3.

2.3. Transformations Specification

2.3.1. The XML2PetriNet transformation

The ATL code for the XML to PetriNet transformation consists of 8 helpers and 5 rules.

2.3.1.1. Helpers

The first helper, **allPlaces**, is a constant helper. It calculates a Set that contains all the XML Elements named “place”.

The **allTransitions** helper is a constant helper. It calculates a Set that contains all the XML Elements named “transition”.

The **allArcs** helper is a constant helper. It calculates a Set that contains all the XML Elements named “arc”.

The **getAttributeValue()** helper returns the value of an attribute (identified by its name, passed as a parameter) of the contextual XML Element. For this purpose, its collects, among the children of this contextual Element, the Attribute whose name matches the name passed in parameter. The helper returns the value of the first matched attribute.

The **getName()** helper returns the name of a “net” or a “place” XML Element. To this end, it first gets, among its Element children, the one named “name”. It then gets the “text” XML Element child of this new node, and finally returns the value associated with it.

The **getId()** helper returns the value of the “id” attribute of the contextual XML Element. For this purpose, it returns the value provided by the **getAttributeValue()** helper called with “id” as parameter.

The **getTarget()** helper returns the value of the “target” attribute of the contextual XML Element. For this purpose, it returns the value provided by the **getAttributeValue()** helper called with “target” as parameter.

The **getSource()** helper returns the value of the “source” attribute of the contextual XML Element. For this purpose, it returns the value provided by the **getAttributeValue()** helper called with “source” as parameter.

2.3.1.2. Rules

The **Main** rule generates a PetriNet from each “net” XML Element input element. Name of the generated PetriNet is computed by calling the **getName()** helper. Its set of Places corresponds to the Places generated for the “place” XML Elements. Its set of Transitions corresponds to the Transitions generated for the “transition” XML Elements. Finally, its set of Arcs corresponds to TransToPlaceArcs and PlaceToTransArcs generated for the “arc” XML Elements.

The **Place** rule generates a PetriNet Place for each “place” XML Element. Name of the generated Place is computed by a call to the **getName()** helper. Its set of incoming arcs contains the TransToPlaceArcs generated for the XML Elements whose target (computed by the **getTarget()** helper) is equal to the input “place” XML Element id (returned by the **getId()** helper). Similarly, its set of outgoing arcs contains the PlaceToTransArcs generated for the XML Elements whose source (computed by the **getSource()** helper) is equal to the input “place” XML Element id (returned by the **getId()** helper).

The **Transition** rule generates a PetriNet Transition for each “transition” XML Element. Generated Transition accepts an empty string as name. Its set of incoming arcs contains the PlaceToTransArcs generated for the XML Elements whose target (computed by the **getTarget()** helper) is equal to the input “transition” XML Element id (returned by the **getId()** helper). Similarly, its set of outgoing arcs contains the TransToPlaceArcs generated for the XML Elements whose source (computed by the **getSource()** helper) is equal to the input “transition” XML Element id (returned by the **getId()** helper).

The **PlaceToTransArc** rule generates a PlaceToTransArc for each “arc” XML Element whose source (obtained by means of the **getSource()** helper) corresponds to the id of a “place” XML Element. Weight of the generated PlaceToTransArc is set to 1. Its source corresponds to the Place generated for the “place” XML Element whose id (obtained with **getId()**) is equal to the source of the input “arc” XML Element. Its target corresponds to the Transition generated for the “transition” XML Element whose id (obtained with **getId()**) is equal to the target of the input “arc” XML Element.

The **TransToPlaceArc** rule generates a TransToPlaceArc for each “arc” XML Element whose source (obtained by means of the **getSource()** helper) corresponds to the id of a “transition” XML Element. Weight of the generated TransToPlaceArc is set to 1. Its source corresponds to the Transition generated for the “transition” XML Element whose id (obtained with **getId()**) is equal to the source of the input “arc” XML Element. Its target corresponds to the Place generated for the “place” XML Element whose id (obtained with **getId()**) is equal to the target of the input “arc” XML Element.

```
module XML2PetriNet;
create OUT : PetriNet from IN : XML;

```

```

--------------------------------------------------------------------------------
-- HELPERS ---------------------------------------------------------------------
--------------------------------------------------------------------------------


-- This helper computes the Set containing all the XML!Element of the input
-- XML model that are named 'place'.
-- CONTEXT: thisModule
-- RETURN:  Set(XML!Element)
helper def: allPlaces : Set(XML!Element) =
   XML!Element.allInstances()
      ->select(e | e.name = 'place');


-- This helper computes the Set containing all the XML!Element of the input
-- XML model that are named 'transition'.
-- CONTEXT: thisModule
-- RETURN:  Set(XML!Element)
helper def: allTransitions : Set(XML!Element) =
   XML!Element.allInstances()
      ->select(e | e.name = 'transition');


-- This helper computes the Set containing all the XML!Element of the input
-- XML model that are named 'arc'.
-- CONTEXT: thisModule
-- RETURN:  Set(XML!Element)
helper def: allArcs : Set(XML!Element) =
   XML!Element.allInstances()
      ->select(e | e.name = 'arc');


-- This helper computes the name value of an input XML!Element.
-- For this purpose, it first selects among its elements children the one
-- named 'name'. It then selects, among children of this new element, the one
-- named 'text'. It then selects the XML!Text child of this last element and
-- returns its value.
-- CONTEXT: XML!Element
-- RETURN:  String
helper context XML!Element def : getName() : String =
   self.children
      ->select(c | c.oclIsTypeOf(XML!Element) and c.name = 'name')
      ->first().children
      ->select(c | c.oclIsTypeOf(XML!Element) and c.name = 'text')
      ->first().children
      ->first().value;


-- This helper calculates the value of a given attribute (identified by the
-- name provided as a parameter) of the contextual XML!Element.
-- To this end, it selects among its attribute children the one which has the
-- name provided in parameter, and returns its value.
-- CONTEXT: XML!Element
-- IN:      String
-- RETURN:  String
helper context XML!Element def : getAttributeValue(name : String) : String =
   self.children
      ->select(c | c.oclIsTypeOf(XML!Attribute) and c.name = name)
      ->first().value;


-- This helper calculates the value of the 'id' attribute of the contextual
-- XML!Element. For this purpose, it calls the 'getAttributeValue' with 'id'
-- as parameter.
-- CONTEXT: XML!Element
-- RETURN:  String
helper context XML!Element def : getId() : String =
   self.getAttributeValue('id');


-- This helper calculates the value of the 'target' attribute of the contextual
-- XML!Element. For this purpose, it calls the 'getAttributeValue' with
-- 'target' as parameter.
-- CONTEXT: XML!Element
-- RETURN:  String
helper context XML!Element def : getTarget() : String =
   self.getAttributeValue('target');


-- This helper calculates the value of the 'source' attribute of the contextual
-- XML!Element. For this purpose, it calls the 'getAttributeValue' with
-- 'source' as parameter.
-- CONTEXT: XML!Element
-- RETURN:  String
helper context XML!Element def : getSource() : String =
   self.getAttributeValue('source');


-------------------------------------------------------------------------------
-- RULES ----------------------------------------------------------------------
-------------------------------------------------------------------------------


-- Rule 'Main'
-- This rule generates a PetriNet element from the XML!Element called 'net'.
-- Name of the generated PetriNet is computed by the 'getName' helper.
-- Its places, transitions and arcs respectively correspond to the elements
-- generated for the XML!Elements named 'place', 'transition', and 'arc'.
rule Main {
   from
        xml_net : XML!Element (
         xml_net.name = 'net'
        )
   to
      pn : PetriNet!PetriNet (
         name <- xml_net.getName(),
         places <- thisModule.allPlaces,
         transitions <- thisModule.allTransitions,
         arcs <- thisModule.allArcs
      )
}


-- Rule 'State'
-- This rule generates a Place element for each XML!Element called 'place'.
-- Name of the generated Place is computed by the 'getName' helper.
-- Its incoming arcs correspond to the elements generated for the XML!Element
-- named 'arc' whose target is the input 'place' XML!Element.
-- Its outgoing arcs correspond to the elements generated for the XML!Element
-- named 'arc' whose source is the input 'place' XML!Element.
rule Place {
   from
        xml_place :  XML!Element (
         xml_place.name = 'place'
        )
   to
      pn_p : PetriNet!Place (
         name <- xml_place.getName(),
         incoming <- thisModule.allArcs
                     ->select(a | a.getTarget() = xml_place.getId()),
         outgoing <- thisModule.allArcs
                     ->select(a | a.getSource() = xml_place.getId())
      )
}
```

```

-- Rule 'Transition'
-- This rule generates a Transition element for each XML!Element called
-- 'transition'.
-- Generated Place accepts an empty string as name.
-- Its incoming arcs correspond to the elements generated for the XML!Element
-- named 'arc' whose target is the input 'transition' XML!Element.
-- Its outgoing arcs correspond to the elements generated for the XML!Element
-- named 'arc' whose source is the input 'transition' XML!Element.
rule Transition {
   from
        xml_trans :  XML!Element (
         xml_trans.name = 'transition'
        )
   to
      pn_t : PetriNet!Transition (
         name <- '',
         incoming <- thisModule.allArcs
                     ->select(a | a.getTarget() = xml_trans.getId()),
         outgoing <- thisModule.allArcs
                     ->select(a | a.getSource() = xml_trans.getId())
      )
}


-- Rule 'PlaceToTransArc'
-- This rule generates a PlaceToTransArc element for each XML!Element called
-- 'arc' whose source is a 'place' XML!Element.
-- The source of the generated PlaceToTransArc corresponds to the element
-- generated for the 'place' XML!Element whose 'id' is equal to the source of
-- the input 'arc' XML!Element.
-- The target of the generated PlaceToTransArc corresponds to the element
-- generated for the 'transition' XML!Element whose 'id' is equal to the target
-- of the input 'arc' XML!Element.
rule PlaceToTransArc {
   from
        xml_arc :  XML!Element (
         if xml_arc.name = 'arc' then
            thisModule.allPlaces
               ->collect(p | p.getId())
               ->includes(xml_arc.getSource())
         else
            false
         endif
        )
   to
      pn_a : PetriNet!PlaceToTransArc (
         weight <- 1,
         source <- thisModule.allPlaces
                  ->select(b | b.getId() = xml_arc.getSource())
                  ->first(),
         target <- thisModule.allTransitions
                  ->select(b | b.getId() = xml_arc.getTarget())
                  ->first()
      )
}


-- Rule 'TransToPlaceArc'
-- This rule generates a TransToPlaceArc element for each XML!Element called
-- 'arc' whose source is an 'transition' XML!Element.
-- The source of the generated TransToPlaceArc corresponds to the element
-- generated for the 'transition' XML!Element whose 'id' is equal to the source
-- of the input 'arc' XML!Element.
-- The target of the generated TransToPlaceArc corresponds to the element
-- generated for the 'place' XML!Element whose 'id' is equal to the target of
-- the input 'arc' XML!Element.
rule TransToPlaceArc {
   from
        xml_arc :  XML!Element (
         if xml_arc.name = 'arc' then
            thisModule.allTransitions
               ->collect(p | p.getId())
               ->includes(xml_arc.getSource())
         else
            false
         endif
        )
   to
      pn_a : PetriNet!TransToPlaceArc (
         weight <- 1,
         source <- thisModule.allTransitions
                  ->select(b | b.getId() = xml_arc.getSource())
                  ->first(),
         target <- thisModule.allPlaces
                  ->select(b | b.getId() = xml_arc.getTarget())
                  ->first()
      )
}
```

2.3.2. The PetriNet2PathExp transformation

The ATL code for the PetriNet to PathExp transformation consists of 3 rules (no helpers).

2.3.2.1. Rules

The **Main** rule generates a PathExp from the input PetriNet. Name of the generated PathExp is copied from the name of the PetriNet. Its set of States corresponds to the States generated for the Places of the input PetriNet. Its set of Transitions corresponds to the Transitions generated for the Transitions of the input PetriNet.

The **Place** rule generates a State for each input Place. The set of incoming Transitions of the generated State corresponds to the Transitions generated for the PetriNet Transitions that are source of the incoming arcs of the input Place. Its set of outgoing Transitions corresponds to the Transitions generated for the PetriNet Transitions that are target of the outgoing arcs of the input Place.

The **Transition** rule generates a Transition for each input PetriNet Transition. The generated Transition accepts an empty string as name. The source State of the generated Transition corresponds to the State generated for the PetriNet Place that is source of the first incoming arc of the input Transition. Its target State corresponds to the State generated for the PetriNet Place that is target of the first outgoing arc of the input Transition.

```
module PetriNet2PathExp;
create OUT : PathExp from IN : PetriNet;


-------------------------------------------------------------------------------
-- RULES ----------------------------------------------------------------------
-------------------------------------------------------------------------------

-- Rule 'Main'
-- This rule generates a PathExp from the input PetriNet element.
-- Name of the generated PathExp is copied from the PetriNet one.
-- Its set of states and transitions respectively correspond to the elements
-- that are generated for the input Places and Transitions.
rule Main {
   from
        pn : PetriNet!PetriNet
   to
      pe : PathExp!PathExp (
         name <- pn.name,
         states <- pn.places,
         transitions <- pn.transitions
      )
}


-- Rule 'Place'
-- This rule generates State for each input Place element.
-- The set of incoming transitions of the generated Place corresponds to the
-- elements generated for Transitions that are source of the incoming
-- PetriNet!Arc.
-- The set of outgoing transitions of the generated Place corresponds to the
-- elements generated for Transitions that are tagret of the outgoing
-- PetriNet!Arc.
rule Place {
   from
        pn_p : PetriNet!Place
   to
      pe_s : PathExp!State (
         incoming <- pn_p.incoming
                     ->collect(e | e.source)
                     ->flatten(),
         outgoing <- pn_p.outgoing
                     ->collect(e | e.target)
                     ->flatten()
      )
}


-- Rule 'Transition'
-- This rule generates a PathExp!Transition for each PetriNet!Transition.
-- Source of the generated Transition corresponds to the element generated for
-- the Place that is the source of the incoming PetriNet!Arc.
-- Target of the generated Transition corresponds to the element generated for
-- the Place that is the target of the outgoing PetriNet!Arc.
rule Transition {
   from
        pn_t : PetriNet!Transition
   to
      pe_t : PathExp!Transition (
         name <- '',
         source <- pn_t.incoming
                  ->collect(e | e.source)
                  ->flatten()
                  ->first(),
         target <- pn_t.outgoing
                  ->collect(e | e.target)
                  ->flatten()
                  ->first()
      )
}
```

2.3.3. The PathExp2TextualPathExp transformation

The ATL code for the PathExp to TextualPathExp transformation consists of 10 helpers and 5 rules.

2.3.3.1. Assumptions

The ATL transformation described here is based on the following assumption on the input PathExp models:

- The PathExp input model includes only “simple” (single transition) loops (i.e. the transformation is not able to produce composed multiple Transitions).

2.3.3.2. Helpers

The first helper, **rootState**, is a constant helper. It calculates the root State of the input PathExp model. For this purpose, it selects among all State instances, the one that has no incoming Transitions.

The **existLoop()** helper returns a Boolean value stating whether the contextual State is targeted by a simple loop Transition. To this end, it checks if there exists a Transition, among the incoming Transitions of the State, whose source is the State itself.

The **getLoop()** helper returns the simple loop Transition of the contextual State. This contextual State must have a simple loop Transition. The helper returns the first Transition, among incoming ones of the State, whose source is the State itself.

The **getInT()** helper computes a Sequence of all the non-loop incoming Transitions of the contextual State. For this purpose, it collects all the State incoming Transitions whose source is different from the contextual State.

The **getOutT()** helper computes a Sequence of all the non-loop outgoing Transitions of the contextual State. For this purpose, it collects all the State outgoing Transitions whose target is different from the contextual State.

The **getPrevStates()** helper computes the Sequence of the States that precede the contextual State in the input PathExp model. Note that the contextual State is excluded from the result when it has a simple loop transition. The helper simply collects the source State of the Transitions returned by a call to the **getInT()** helper on the contextual State.

The **getNextStates()** helper computes the Sequence of the States that follow the contextual State in the input PathExp model. Note that the contextual State is excluded from the result when it has a simple loop transition. The helper simply collects the target State of the Transitions returned by a call to the **getOutT()** helper on the contextual State.

The **findNextState(n:Integer)** helper is a recursive helper that returns the State that closes the alternative Transition that is initiated by the contextual State of the initial call. The helper accepts an integer parameter *n*, 0 at the initial call, which encodes the number of successive nested alternative Transition currently opened. The helper is based on the following rules:

- If the current contextual State has more than one previous State (computed by the **getPrevStates()** helper), and its parameter is 0, the closing State has been found and the helper returns the current contextual State.

- Else if the current contextual State has more than one previous State and more than one next State (computed by the **getNextStates()** helper), a nested alternative transition is closed and a new one is opened. The helper then returns the result of the recursive call of **findNextState(0)** on one of the next States of the current contextual State.

- Else if the current contextual State has more than one previous State and a single next State, a nested alternative transition is closed. The helper then returns the result of the recursive call of **findNextState(n-1)** on the next State of the current contextual State.

- Else if the current contextual State has a single previous State and more than one next State, a new alternative transition is initiated. The helper then returns the result of the recursive call of **findNextState(n+1)** on one of the next States of the current contextual State.

- Else if the current contextual State has a single previous State and a single next State, the helper then returns the result of the recursive call of **findNextState(n)** on the next State of the current contextual State.

The **getTransitionsFromStates(Boolean)** helper computes the Sequence of oclAny elements (that are either State or Transition elements) that are going to be matched into the Transitions of the Path initiated by the contextual State. The helper is a recursive helper that accepts a Boolean parameter that encodes the fact that a nested alternative transition has just been parsed. **getTransitionsFromStates(Boolean)** is initially called with false as parameter. The helper is base on the following rules:

- If the contextual State has more than one previous State (computed by the **getPrevStates()** helper) and the Boolean parameter is false, the helper returns an empty Sequence. This rule handles the State that corresponds to the end of the Path currently being parsed.

- Else if the contextual State has more then one next State (computed by the **getNextStates()** helper), a new alternative is opened. The helper then returns a Sequence composed of a potential loop Transition, the contextual State, and the result of the recursive call of **getTransitionsFromStates(true)** on the closing State of the opened alternative Transition (this State is obtained by means of the **findNextState()** helper).

- Else if the contextual State has a single next State, it returns a Sequence composed of a potential loop Transition, its outgoing Transition, and the result of the recursive call of **getTransitionsFromStates(false)** on the next State of the contextual State.

- Else if the contextual State has no next States, it returns an empty Sequence. This rule handles the case of the end of the PathExp, which also corresponds to the end of the Path currently being parsed.

The **getTransitionsFromTrans()** helper computes the Sequence of oclAny elements (that are either State or Transition elements) that are going to be matched into the Transitions of the Path initiated by the contextual Trans. To this end, it returns a Sequence composed by the contextual Transition and the result of the call of the **getTransitionsFromStates(Boolean)** helper onto the target of the contextual Transition.

2.3.3.3. Rules

The **Main** rule generates a TextualPathExp and its main Path element from the input PathExp. The generated TextualPathExp takes the generated Path as path. The transitions sequence of the generated Path corresponds to the Transition Sequence returned by the call of **getTransitionsFromStates(flase)** on the root State of the input PathExp.

The **Loop** rule generates a PrimitiveTrans from each input PathExp Transition that has the same State as source and target. Generated PrimitiveTrans accepts an empty string as name. Its *isMultiple* attribute is set to true.

The **STransition** rule generates a PrimitiveTrans from each input PathExp Transition whose target is different from source, and whose source State has a single non-loop outgoing Transition. Generated PrimitiveTrans accepts an empty string as name. Its *isMultiple* attribute is set to false.

The **MTransition** rule generates a PrimitiveTrans from each input PathExp Transition whose target is different from source, and whose source State has more than one non-loop outgoing Transition. Generated PrimitiveTrans accepts an empty string as name. Its *isMultiple* attribute is set to false.

The **State** rule generated an AlternativeTrans, along with its multiple alternative Paths, for each PathExp State that has more than one non-loop outgoing Transition. To this end, the rule first computes the Sequence **transitions2**, which is a Sequence of Sequence of oclAny. For each non-loop outgoing Transition of the input State, transitions2 contains the Sequence of Transition/State that are going to be matched into TextualPathExp Transitions (each of these Sequences is computed by a call of the **getTransitionsFromTrans()** helper on an outgoing Transition). The set of paths of the generated AlternativeTrans corresponds to the different paths generated by the rule execution. The AlternativeTrans *isMultiple* attribute is set to false. The Sequence of Transitions of each generated Path corresponds to the Transitions generated for the corresponding (i.e. same rank) Sequence of State/Transition in **transitions2**.

```
module PathExp2TextualPathExp;
create OUT : TextualPathExp from IN : PathExp;


-------------------------------------------------------------------------------
-- HELPERS --------------------------------------------------------------------
-------------------------------------------------------------------------------

-- This helper computes the root State of the input PathExp model.
-- To this end, it selects among all State instances the one that has no
-- incoming transition.
-- CONTEXT: thisModule
-- RETURN:  PathExp!State
helper def: rootState : PathExp!State =
   PathExp!State.allInstances()
      ->select(s | s.incoming->isEmpty())
      ->asSequence()
      ->first();


-- This helper computes a boolean value stating whether a loop transition is
-- defined for the contextual State.
-- For this purpose, the helper checks if there exists an incoming transition
-- whose source is the contextual State.
-- CONTEXT: PathExp!State
-- RETURN:  Boolean
helper context PathExp!State def: existLoop() : Boolean =
   self.incoming
      ->select(e | e.source = self)
      ->notEmpty();


-- This helper returns the loop Transition defined for the contextual State.
-- To this end, it returns the first incoming transition that has the
-- contextual State as source.
-- PRECOND: a loop transition must be defined for the contextual State.
-- CONTEXT: PathExp!State
-- RETURN:  PathExp!Transition
helper context PathExp!State def: getLoop() : PathExp!Transition =
   self.incoming
      ->select(e | e.source = self)
      ->asSequence()
      ->first();


-- This helper computes the set of non-loop incoming transitions of the
-- contextual State.
-- To this end, it selects among incoming transitions the ones that do not
-- target the contextual State.
-- CONTEXT: PathExp!State
-- RETURN:  Sequence(PathExp!Transition)
helper context PathExp!State def: getInT() : Sequence(PathExp!Transition) =
   self.incoming
      ->select(e | e.source <> self)
      ->asSequence();


-- This helper computes the set of non-loop outgoing transitions of the
-- contextual State.
-- To this end, it selects among outgoing transitions the ones that do not
-- target the contextual State.
-- CONTEXT: PathExp!State
-- RETURN:  Sequence(PathExp!Transition)
helper context PathExp!State def: getOutT() : Sequence(PathExp!Transition) =
   self.outgoing
      ->select(e | e.target <> self)
      ->asSequence();


-- This helper computes the set of States whose transitions lead to the
-- contextual State (except the contextual State if it is reachable from itself
-- by means of a loop transitions).
-- For this purpose, the helper simply collects the source of the transitions
-- returned by the call of the 'getInT' helper onto the contextual State.
-- CONTEXT: PathExp!State
-- RETURN:  Sequence(PathExp!State)
helper context PathExp!State def: getPrevStates() : Sequence(PathExp!State) =
   self.getInT()->collect(e | e.source);


-- This helper computes the set of States that can be reached by means of
-- outgoing transitions of the contextual State (except the contextual State
-- if it is reachable from itself through a loop transitions).
-- For this purpose, the helper simply collects the target of the transitions
-- returned by the call of the 'getOutT' helper onto the contextual State.
-- CONTEXT: PathExp!State
-- RETURN:  Sequence(PathExp!State)
helper context PathExp!State def: getNextStates() : Sequence(PathExp!State) =
   self.getOutT()->collect(e | e.target);


-- This helper computes the sequence of both Path!State and Path!Transition
-- input elements that correspond to the transitions of the Path initiated by
-- the contextual State.
-- The helper accepts a Boolean parameter that encodes the fact that what
-- cooresponds to a nested alternative transition has just been parsed. The
-- helper is initially called with false as parameter.
--   * A contextual State with several non-loop incoming transitions along with
--     a false nested parameter, identifies the end of the current Path. The
--     helper therefore returns an empty sequence.
--   * If the contextual State has several non-loop outgoing transitions with a
--     true along with a nested parameter, this identifies the beginning of a
--     new nested alternative transition within the current Path. The helper
--     then returns a sequence made of 1) the loop transition of the contextual
--     State, if it is defined, 2) the contextual State itself, and 3) the
--     sequence returned by a recursive call of 'getTransitionsFromState' on the
--     state that closes the new alternative transition (it is computed by the
--    'findNextState' helper), with the nested parameter set to true.
--  * If the contextual State has a single non-loop outgoing transition, the
--    helper returns a sequence made of 1) the loop transition of the
--    contextual State, if it is defined, 2) the outgoing transition of the
--    contextual State, and 3) the sequence returned by a recursive call of
--    'getTransitionsFromState' onto the the next state of the contextual
--    State, with the nested parameter set to false.
--  * Finally, a contextual State with no outgoing Transitions indicates the
--    end of the input PathExp and (also) of the current Path. The  helper
--    therefore returns an empty sequence.
--
-- NOTE: the result type of the helper is currently encoded as a sequence of
-- strings since 1) the oclAny type is not implemented yet 2) and no type
-- verifications are performed by the current atl version.
--
-- CONTEXT: PathExp!State
-- IN:      Boolean
-- RETURN:  Sequence(oclAny)
helper context PathExp!State
   def: getTransitionsFromState(nested : Boolean) : Sequence(String) =
   let nextStates : Sequence(PathExp!State) = self.getNextStates()
   in let prevStates : Sequence(PathExp!State) = self.getPrevStates()
   in let loop : Sequence(PathExp!Transition) =
      if self.existLoop() then
         self.getLoop()
      else
         Sequence{}
      endif
   in

   if prevStates->size() > 1 and not nested then
      Sequence{}
   else
      if nextStates->size() > 1 then
         let state : PathExp!State = nextStates->first().findNextState(0)
         in Sequence{
               loop,
               self,
               state.getTransitionsFromState(true)
            }->flatten()
      else
         if nextStates->size() = 1 then
            Sequence{
               loop,
               self.getOutT()->first(),
               nextStates->first().getTransitionsFromState(false)
            }->flatten()
         else
            Sequence{}
         endif
      endif
   endif;


-- This helper computes the sequence of both Path!State and Path!Transition
-- input elements that correspond to the transitions of the Path initiated by
-- the contextual Transition.
-- The returned sequence is composed of the contextual transition followed by
-- the result of the call of the 'getTransitionsFromState' helper onto the
-- target of this contextual transition.
--
-- NOTE: the result type of the helper is currently encoded as a sequence of
-- strings since 1) the oclAny type is not implemented yet 2) and no type
-- verifications are performed by the current atl version.
--
-- CONTEXT: PathExp!State
-- RETURN:  Sequence(oclAny)
helper context PathExp!Transition
```

```
   def: getTransitionsFromTrans() : Sequence(String) =
   Sequence{self, self.target.getTransitionsFromState(false)}->flatten();


-- This helper aims to compute the State that closes the alternative transition
-- that is started at the contextual State of the initial call.
-- It accepts an Integer as parameter which indicates the number of opened
-- nested alternative transitions. It is initially called with n = 0.
-- In order to compute its closing State, the helper recursively parses the
-- Path:
--  * if the contextual State has more than one incoming transition and no
--    nested alternative trans. are opened (n=0), the helper returns the
--    contextualState.
--  * if the contextual State has more than one incoming transition and more
--    than one outgoing transition, the helper returns the value provided by
--    the recursive call of 'findNextState(n)' onto one of the next states of
--    the contextual state.
--  * if the contextual State has more than one incoming transition but a
--    single outgoing transition, the helper returns the value provided by the
--    recursive call of 'findNextState(n-1)' onto the next state of the
--    contextual state.
--  * if the contextual State has a single incoming transition and more than
--    one outgoing transition, the helper returns the value provided by the
--    'findNextState(n+1)' onto one of the next states of the contextual state.
--  * finally, if the contextual State has a single incoming transition and a
--    single outgoing transition, the helper returns the value provided by the
--    recursive call of 'findNextState(n+)' onto the next state of the
--    contextual state.
-- CONTEXT: PathExp!State
-- IN:      Integer
-- RETURN:  PathExp!State
helper context PathExp!State def: findNextState(n : Integer) : PathExp!State =
   let prevStates : Sequence(PathExp!State) = self.getPrevStates() in
   let nextStates : Sequence(PathExp!State) = self.getNextStates() in
   if prevStates->size() > 1 and n = 0 then
      self
   else
      if prevStates->size() > 1 then
         if nextStates->size() > 1 then
            nextStates->first().findNextState(n)
         else
            nextStates->first().findNextState(n-1)
         endif
      else
         if nextStates->size() > 1 then
            nextStates->first().findNextState(n+1)
         else
            nextStates->first().findNextState(n)
         endif
      endif
   endif;


--------------------------------------------------------------------------------
-- RULES -----------------------------------------------------------------------
--------------------------------------------------------------------------------

-- Rule 'Main'
-- This rule generates both a TextualPathExp and its main Path from the root
-- PathExp input element.
-- The generated TextualPathExp accepts the Path generated by the rule as path.
-- The sequence of transitions contained by the generated Path is returned by
-- the call of the 'getTransitionsFromState' helper onto the root State element
-- of the input model.
rule Main {
   from
        pe : PathExp!PathExp
   to
```

```
      tpe : TextualPathExp!TextualPathExp (
         path <- p
      ),

      p : TextualPathExp!Path (
         transitions <- thisModule.rootState.getTransitionsFromState(false)
      )
}


-- Rule 'Loop'
-- This rule generates a multiple PrimitiveTrans from a loop Transition.
-- The generated PrimitiveTrans accepts an empty string as name. Its
-- 'isMultiple' attribute is set to 'true'.
rule Loop {
   from
       t : PathExp!Transition (
         t.source = t.target
       )
   to
      pt : TextualPathExp!PrimitiveTrans (
         name <- '',
         isMultiple <- true
      )
}


-- Rule 'STransition'
-- This rule generates a simple PrimitiveTrans from a non-loop transition
-- which is the only outgoing transition of its source State.
-- The generated PrimitiveTrans accepts an empty string as name. Its
-- 'isMultiple' attribute is set to 'false'.
rule STransition {
   from
       t : PathExp!Transition (
         t.source <> t.target and
         t.source.getOutT()->size() = 1
       )
   to
      pt : TextualPathExp!PrimitiveTrans (
         name <- '',
         isMultiple <- false
      )
}


-- Rule 'MTransition'
-- This rule generates a simple PrimitiveTrans from a non-loop transition
-- which is NOT the only outgoing transition of its source State.
-- The generated PrimitiveTrans accepts an empty string as name. Its
-- 'isMultiple' attribute is set to 'false'.
rule MTransition {
   from
       t : PathExp!Transition (
         t.source <> t.target and
         t.source.getOutT()->size() > 1
       )
   to
      pt : TextualPathExp!PrimitiveTrans (
         name <- '',
         isMultiple <- false
      )
}


-- Rule 'State'
-- This rule generates both an AlternativeTransition and the different Paths
-- that compose that compose this alternative transition from an input State
-- that has multiple non-loop outgoing Transitions.
-- Paths of the generated AlternativeTransition are those that are generated
-- by the rule. Its 'isMultiple' attribute is set to 'false'.
-- A distinct Path is generated for each non-loop outgoing Transition of the
-- input State. The sequence of transitions that is assigned to a generated
-- Path is the corresponding (ie. at same rank) sequence of model elements in
-- the 'transitions2' sequence (calculated in the using clause).
rule State {
   from
        s : PathExp!State (
         s.getOutT()->size() > 1
        )
   using {
      transitions2 : Sequence(String) =
         s.getOutT()->collect(e | e.getTransitionsFromTrans());
   }
   to
      at : TextualPathExp!AlternativeTrans (
         altPaths <- paths,
         isMultiple <- false
      ),

      paths : distinct TextualPathExp!Path foreach(e in transitions2) (
         transitions <- transitions2
      )
}
```

I. TextualPathExp metamodel in KM3 format

```
package TextualPathExp {

  class TextualPathExp {
    reference path container : Path;
  }

  class Path {
    reference transitions [1-*] ordered container : Transition;
  }

  abstract class Transition {
    attribute isMultiple : Boolean;
  }

  class AlternativeTrans extends Transition {
    reference altPaths [1-*] ordered container : Path;
  }

  class PrimitiveTrans extends Transition {
    attribute name : String;
  }

}

package PrimitiveTypes {
  datatype String;
  datatype Boolean;
}
```

II. PathExp metamodel in KM3 format

```
package PathExp {

  abstract class Element {
    attribute name : String;
  }

  class PathExp extends Element {
    reference states [1-*] container : State;
    reference transitions [*] container : Transition;
  }

  class State {
    reference incoming [*] : Transition oppositeOf target;
    reference outgoing [*] : Transition oppositeOf source;
  }

  class Transition extends Element {
    reference source : State oppositeOf outgoing;
    reference target : State oppositeOf incoming;
  }
}

package PrimitiveTypes {
  datatype String;
}
```

III. PetriNet metamodel in KM3 format

```
package PetriNet {

  abstract class Element {
    attribute name : String;
  }

  class PetriNet extends Element {
    reference places[1-*] container : Place;
    reference transitions[*] container : Transition;
    reference arcs [*] container : Arc;
  }

  class Place extends Element {
    reference incoming [*] : TransToPlaceArc oppositeOf target;
    reference outgoing [*] : PlaceToTransArc oppositeOf source;
  }

  class Transition extends Element {
    reference incoming [1-*] : PlaceToTransArc oppositeOf target;
    reference outgoing [1-*] : TransToPlaceArc oppositeOf source;
  }

  abstact class Arc {
    attribute weight : Integer;
  }

  class PlaceToTransArc extends Arc {
    reference source : Place oppositeOf outgoing;
    reference target : Transition oppositeOf incoming;
  }

  class TransToPlaceArc extends Arc {
    reference source : Transition oppositeOf outgoing;
    reference target : Place oppositeOf incoming;
  }
}

package PrimitiveTypes {
  datatype String;
  datatype Integer;
}
```

IV. XML metamodel in KM3 format

```
package XML {

  abstract class Node {
    attribute startLine[0-1] : Integer;
    attribute startColumn[0-1] : Integer;
    attribute endLine[0-1] : Integer;
    attribute endColumn[0-1] : Integer;
    attribute name : String;
    attribute value : String;
    reference parent[0-1] : Element oppositeOf children;
  }

  class Attribute extends Node {
  }

  class Text extends Node {
  }

  class Element extends Node {
    reference children[*] ordered container : Node oppositeOf parent;
  }

  class Root extends Element {
  }
}

package PrimitiveTypes {
  datatype Boolean;
  datatype Integer;
  datatype String;
}
```

References

[1] The Petri Net Markup Language (PNML). Documentation and tools available at http://www.informatik.hu-berlin.de/top/pnml/about.html.

[2] KM3: Kernel MetaMetaModel. Available at http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/doc/atl/index.html.