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

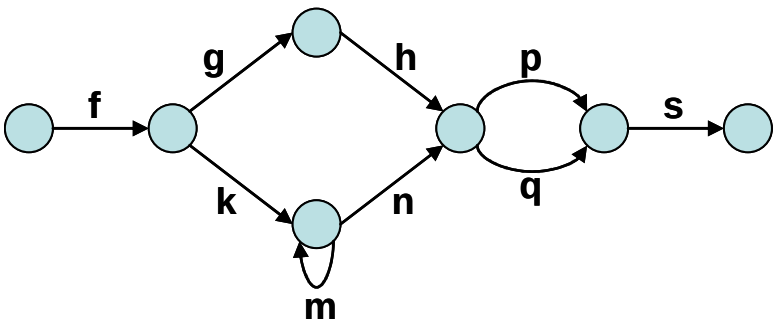

Figure 2. Graphical path expression example

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.

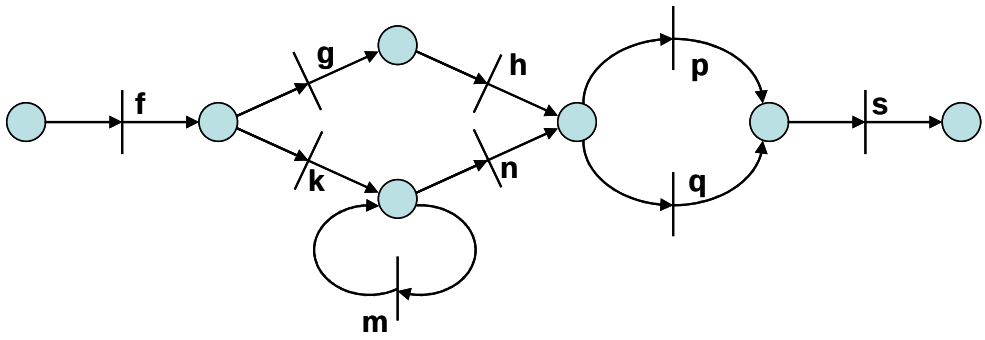

Figure 3. Petri net example

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.

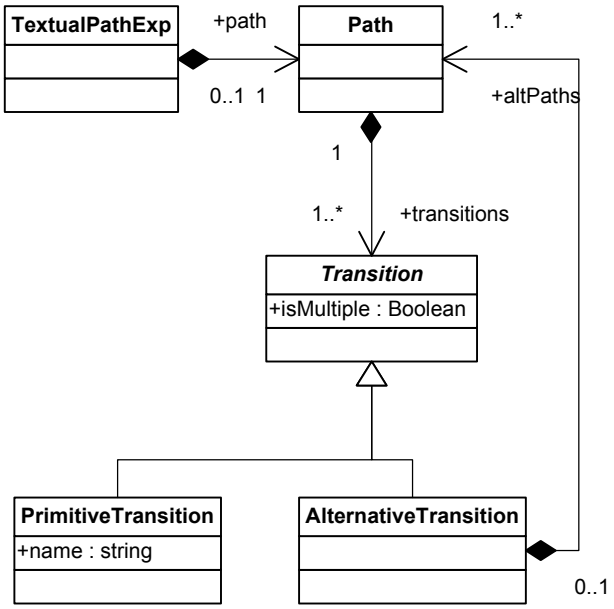

Figure 4. The TextualPathExp metamodel

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.

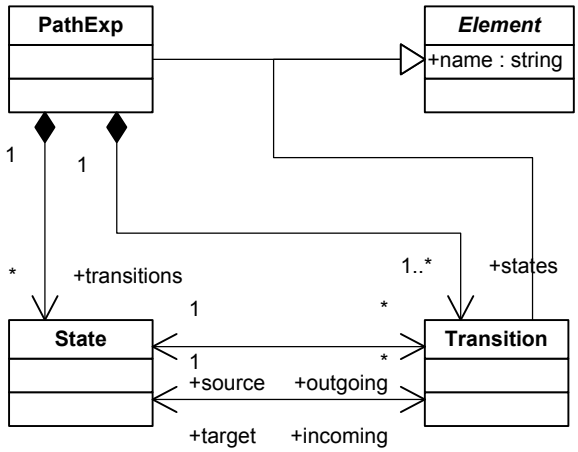

Figure 5. The PathExp metamodel

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.

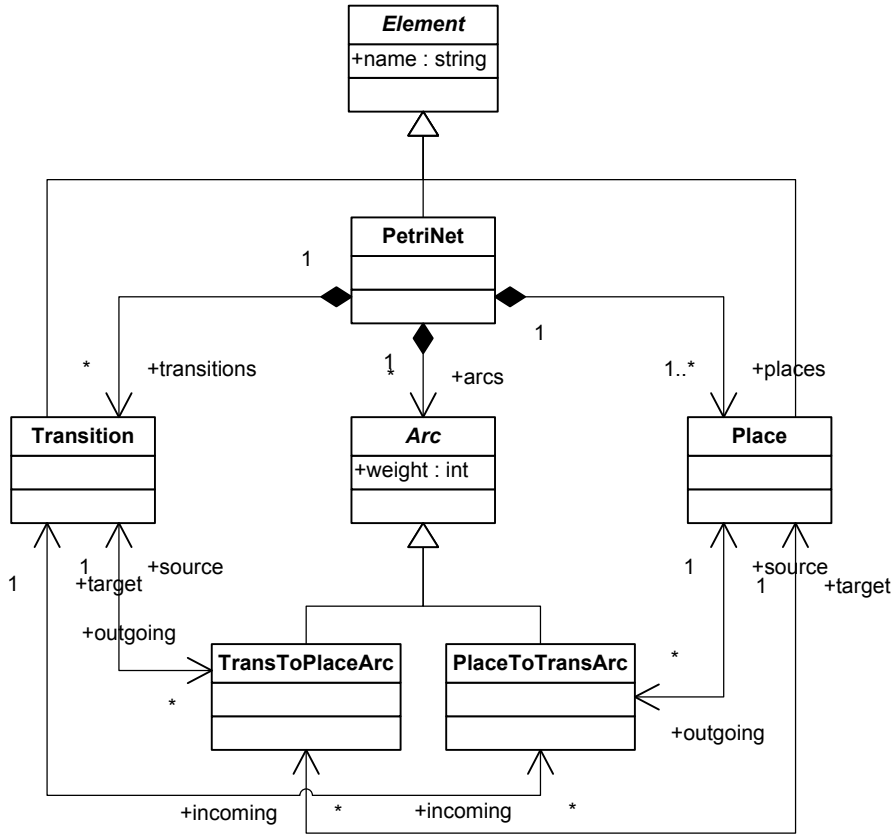

Figure 6. The PetriNet metamodel

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.

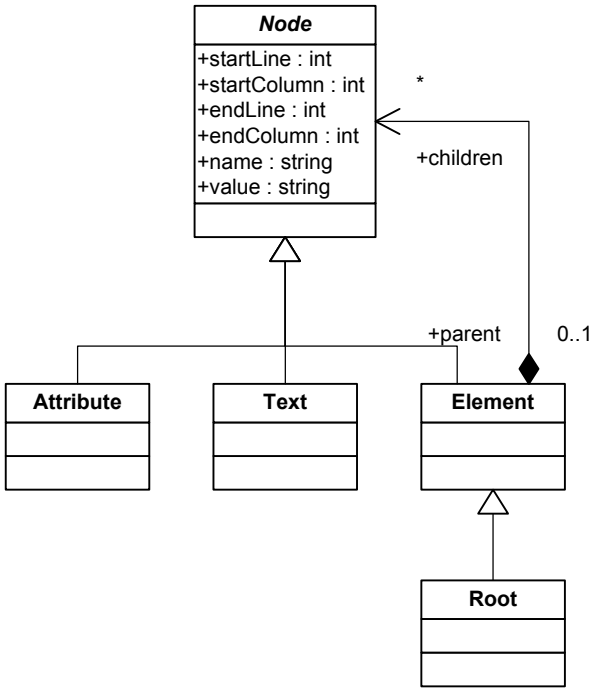

Figure 7. The XML metamodel

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.

```
 1  module TextualPathExp2PathExp;
 2  create OUT : PathExp from IN : TextualPathExp;
 3
 4
 5  ------------------------------------------------------------------------------
 6  -- HELPERS ---------------------------------------------------------------------
 7  ------------------------------------------------------------------------------
 8
 9  -- This helper returns the root TextualPathExp element of the input
10  -- TextualPathExp model.
11  -- CONTEXT: thisModule
12  -- RETURN:  TextualPathExp!TextualPathExp
13  helper def: root : TextualPathExp!TextualPathExp =
14     TextualPathExp!TextualPathExp.allInstances()
15        ->asSequence()->first();
16
17
18  -- This helper returns the 1st Transition element contained by the root
19  -- TextualPathExp model.
20  -- CONTEXT: thisModule
21  -- RETURN:  TextualPathExp!Transition
22  helper def: rootTrans : TextualPathExp!Transition =
23     thisModule.root.path.transitions->first();
24
25
26  -- This helper returns the last Transition element contained by the root
27  -- TextualPathExp model.
28  -- CONTEXT: thisModule
29  -- RETURN:  TextualPathExp!Transition
30  helper def: leafTrans : TextualPathExp!Transition =
31     thisModule.root.path.transitions->last();
32
33
34  -- This helper computes the Set containing all the Path elements of the input
35  -- TextualPathExp model.
36  -- CONTEXT: thisModule
37  -- RETURN:  Set(TextualPathExp!Path)
38  helper def: allPaths : Set(TextualPathExp!Path) =
39     TextualPathExp!Path.allInstances();
40
41
42  -- This helper computes the Set of Path elements that are contained by
43  -- AlternativeTransition elements.
44  -- CONTEXT: thisModule
45  -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
46  helper def: altPaths : Set(TextualPathExp!Path) =
47     thisModule.allPaths
48        ->select(a |
49           TextualPathExp!AlternativeTrans.allInstances()
50              ->collect(b | b.altPaths)
```

```
51              ->flatten()
52              ->includes(a)
53        );
54
55
56    -- This helper computes the Set of PrimitiveTrans that are not contained
57    -- by any AlternativeTransition.
58    -- To this end, it selects, among all Paths, those that are not contained
59    -- by any AlternativeTransition element. It then gets, for the selected Paths,
60    -- the transitions of type PrimitiveTrans.
61    -- CONTEXT: thisModule
62    -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
63    helper def: primTransitions : Set(TextualPathExp!PrimitiveTrans) =
64       TextualPathExp!Path.allInstances()
65          ->select(a |
66             not TextualPathExp!AlternativeTrans.allInstances()
67                ->collect(b | b.altPaths)
68                ->flatten()
69                ->includes(a)
70          )
71          ->collect(p | p.transitions)
72          ->flatten()
73          ->select(c | c.oclIsTypeOf(TextualPathExp!PrimitiveTrans));
74
75
76    -- This helper computes the Set of 'single' primitive transitions.
77    -- For this purpose, it selects in the primTransitions set, the transitions
78    -- whose 'isMultiple' attribute is set to false.
79    -- CONTEXT: thisModule
80    -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
81    helper def: singlePrimTransitions : Set(TextualPathExp!PrimitiveTrans) =
82       thisModule.primTransitions->select(c | c.isMultiple = false);
83
84
85    -- This helper computes the Set of 'multiple' primitive transitions.
86    -- For this purpose, it selects in the primTransitions set, the transitions
87    -- whose 'isMultiple' attribute is set to true.
88    -- CONTEXT: thisModule
89    -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
90    helper def: multiplePrimTransitions : Set(TextualPathExp!PrimitiveTrans) =
91       thisModule.primTransitions->select(c | c.isMultiple = true);
92
93
94    -- This helper computes the Set of PrimitiveTrans that are contained by an
95    -- AlternativeTransition, except those that are the last transition of their
96    -- Path.
97    -- To this end, the helper first collects the transitions contained by each
98    -- alternative path. For each collected sequence of transitions of size S, it
99    -- gets the (S-1 ) first transition. Finally, it selects in the built sequence
100   -- the transitions of type PrimitiveTrans.
101   -- CONTEXT: thisModule
102   -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
103   helper def: altTransitions1 : Set(TextualPathExp!PrimitiveTrans) =
104      thisModule.altPaths
105         ->collect(p | p.transitions)
106         ->iterate(e;
107                res : Sequence(Sequence(TextualPathExp!Transition)) = Set{} |
108            res->including(e->subSequence(1, e->size()-1))
109         )
110         ->asSequence()
111         ->flatten()
112         ->select(c | c.oclIsTypeOf(TextualPathExp!PrimitiveTrans));
113
114
115   -- This helper computes the Set of 'single' alternative transitions.
116   -- For this purpose, it selects in the altTransitions1 set, the transitions
117   -- whose 'isMultiple' attribute is set to false.
118   -- CONTEXT: thisModule
```

```
119    -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
120    helper def: singleAltTransitions1 : Set(TextualPathExp!PrimitiveTrans) =
121       thisModule.altTransitions1->select(c | c.isMultiple = false);
122
123
124    -- This helper computes the Set of 'multiple' alternative transitions.
125    -- For this purpose, it selects in the altTransitions1 set, the transitions
126    -- whose 'isMultiple' attribute is set to true.
127    -- CONTEXT: thisModule
128    -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
129    helper def: multipleAltTransitions1 : Set(TextualPathExp!PrimitiveTrans) =
130       thisModule.altTransitions1->select(c | c.isMultiple = true);
131
132
133    -- This helper computes the Set of PrimitiveTrans that are contained by an
134    -- AlternativeTransition and that are the last transition of their Path (which
135    -- may also be the first transition if the path contains a singel transition).
136    -- To this end, the helper first collects the transitions contained by each
137    -- alternative path. For each collected sequence of transitions of size S, it
138    -- gets the transition number S. Finally, it selects in the built sequence
139    -- the transitions of type PrimitiveTrans.
140    -- CONTEXT: thisModule
141    -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
142    helper def: altTransitions2 : Set(TextualPathExp!PrimitiveTrans) =
143       thisModule.altPaths
144          ->collect(p | p.transitions)
145          ->iterate(e;
146                  res : Sequence(Sequence(TextualPathExp!Transition)) = Set{} |
147             res->including(e->last())
148          )
149          ->asSequence()
150          ->flatten()
151          ->select(c | c.oclIsTypeOf(TextualPathExp!PrimitiveTrans));
152
153
154    -- This helper computes the containing Path of the contextual Transition
155    -- element.
156    -- For this purpose, it selects ammong all Paths, the one that contains the
157    -- contextual Transition elements.
158    -- CONTEXT: TextualPathExp!Transition
159    -- RETURN:  TextualPathExp!Path
160    helper context TextualPathExp!Transition
161       def: getPath() :  TextualPathExp!Path =
162       thisModule.allPaths
163          ->select(a | a.transitions->includes(self))
164          ->first();
165
166
167    -- This helper computes a boolean value assessing whether or not the contextual
168    -- PrimitiveTrans is the last transition of its Path.
169    -- To this end, the helper first gets the path of the contextual transition (by
170    -- means of the 'getPath' helper) and then compares the contextual transition
171    -- to the last transition of the path.
172    -- CONTEXT: TextualPathExp!PrimitiveTrans
173    -- RETURN:  TextualPathExp!Transition
174    helper context TextualPathExp!PrimitiveTrans
175       def: isLastOfPath() : Boolean =
176       let p : TextualPathExp!Path = self.getPath()
177       in self = p.transitions->last();
178
179
180    -- This helper computes a boolean value assessing whether or not the contextual
181    -- PrimitiveTrans is the first transition of its Path.
182    -- To this end, the helper first gets the path of the contextual transition (by
183    -- means of the 'getPath' helper) and then compares the contextual transition
184    -- to the first transition of the path.
185    -- CONTEXT: TextualPathExp!PrimitiveTrans
186    -- RETURN:  TextualPathExp!Transition
```

```
187   helper context TextualPathExp!PrimitiveTrans
188      def: isFirstOfPath() : Boolean =
189      let p : TextualPathExp!Path = self.getPath()
190      in self = p.transitions->first();
191
192
193   -- This helper computes the Transition for which is generated the target state
194   -- of the loop defined by the contextual PrimitiveTrans. A multiple primitive
195   -- transition only leads to the generation of a loop transition. As a
196   -- consequence, the computed Transition is the one preceding the contextual
197   -- primitive transition in their path. The contextual primitrive transition
198   -- should therefore not be the first of its path.
199   -- CONTEXT: TextualPathExp!PrimitiveTrans
200   -- RETURN:  TextualPathExp!Transition
201   helper context TextualPathExp!PrimitiveTrans
202      def: getLoopTarget() : TextualPathExp!Transition =
203      let p : TextualPathExp!Path = self.getPath()
204      in let i : Integer = p.transitions->indexOf(self)
205      in p.transitions->at(i-1);
206
207
208   -- This helper computes a boolean value assessing whether or not the contextual
209   -- PrimitiveTrans is preceding a multiple transition in its Path.
210   -- If the contextual PrimitiveTrans is the last transition of its Path, the
211   -- helper returns false. Otherwise, it returns the value of the 'isMultiple'
212   -- attribute of the next transition in the path.
213   -- CONTEXT: TextualPathExp!PrimitiveTrans
214   -- RETURN:  Boolean
215   helper context TextualPathExp!PrimitiveTrans
216      def: loopIncoming() : Boolean =
217      let p : TextualPathExp!Path = self.getPath()
218      in let i : Integer = p.transitions->indexOf(self)
219      in if self = p.transitions->last() then
220            false
221         else
222            p.transitions->at(i+1).isMultiple
223         endif;
224
225
226   -- This helper computes the incoming/outgoing loop Transition of the contextual
227   -- multiple PrimitiveTrans.
228   -- For this purpose, it returns the next transition in the path.
229   -- PRECOND: this helper should only be called from a PrimTransition that
230   -- precedes a multiple PrimitiveTrans.
231   -- CONTEXT: TextualPathExp!PrimitiveTrans
232   -- RETURN:  TextualPathExp!Transition
233   helper context TextualPathExp!PrimitiveTrans
234      def: getLoopIncoming() : TextualPathExp!Transition =
235      let p : TextualPathExp!Path = self.getPath()
236      in let i : Integer = p.transitions->indexOf(self)
237      in p.transitions->at(i + 1);
238
239
240   -- This helper computes the set of primitive transitions (except loop
241   -- transitions) that follow the contextual transition.
242   -- For this purpose, the helper first gets the transition next to the
243   -- contextual transition in the same path.
244   -- If this following transition is a PrimitiveTrans and is not multiple, the
245   -- helper returns the transition. If the following transition is a multiple
246   -- PrimitiveTrans, then the helper looks for the transitions that follow this
247   -- next transition by means of a recursive call onto this "next transition".
248   -- If the following transition is an AlternativeTrans, the helper collects the
249   -- first transition of each alternative path of the AlternativeTrans, and
250   -- returns the calculated set.
251   -- CONTEXT: TextualPathExp!Transition
252   -- IN:      Integer
253   -- RETURN:  Set(TextualPathExp!PrimitiveTrans)
254   helper context TextualPathExp!Transition
```

```
255       def: getOutgoing() : Set(TextualPathExp!PrimitiveTrans) =
256       let p : TextualPathExp!Path = self.getPath()
257       in let i : Integer = p.transitions->indexOf(self)
258       in let t : TextualPathExp!Transition = p.transitions->at(i + 1)
259       in if t.oclIsTypeOf(TextualPathExp!PrimitiveTrans) then
260             if not t.isMultiple then
261                Set{t}
262             else
263                t.getOutgoing()
264             endif
265          else
266          t.altPaths
267             ->iterate(e; res : Set(TextualPathExp!PrimitiveTrans) = Set{} |
268                res->including(e.transitions->first())
269             )
270          endif;
271
272
273    -- This helper computes the Transition (primitive or alternative) that precedes
274    -- the contextual PrimitiveTrans in the input TextualPathExp model.
275    -- To this end, the helper first checks whether or not the contextual
276    -- PrimitiveTrans is the first transition of its Path.
277    -- If the contextual transition is the first of its path, the helper first gets
278    -- the AtlernativeTrans the contextual transition belongs to. It then gets the
279    -- Path in which this AlternativeTrans is defined, and the rank of the
280    -- AlternativeTrans within this Path. From then, it gets the transition that
281    -- precedes the computed AlternativeTrans. The helper returns this preceding
282    -- transition if it is not multiple. If the preceding transition is multiple,
283    -- the helper returns the transition that precedes this preceding transition
284    -- by means of a recursive call of the helper onto the transition that precedes
285    -- the AlternativeTrans.
286    -- If the contextual transition is not the first of its path, the helper
287    -- returns its preceding transition if this last is not multiple. If the
288    -- preceding transition is multiple, the helper returns the preceding
289    -- transition of the preceding transition by means of a recursive call of the
290    -- helper onto the transition preceding the contextual transition.
291    -- PRECOND: this helper should not be called on the root Transition of the
292    -- input model.
293    -- CONTEXT: TextualPathExp!PrimitiveTrans
294    -- RETURN:  TextualPathExp!Transition
295    helper context TextualPathExp!PrimitiveTrans
296       def: getPreviousTransition() : TextualPathExp!Transition =
297       let p : TextualPathExp!Path = self.getPath() in
298       if self.isFirstOfPath() then
299          let alt : TextualPathExp!AlternativeTrans =
300             TextualPathExp!AlternativeTrans.allInstances()
301                ->select(a | a.altPaths->includes(p))
302                ->first()
303          in let p2 : TextualPathExp!Path =
304             thisModule.allPaths
305                ->select(a | a.transitions->includes(alt))
306                ->first()
307          in let i : Integer = p2.transitions->indexOf(alt)
308          in let t : TextualPathExp!Transition =
309             p2.transitions->at(i-1) in
310          if t.isMultiple then
311             t.getPreviousTransition()
312          else
313             t
314          endif
315       else
316          let i : Integer = p.transitions->indexOf(self)
317          in let t : TextualPathExp!Transition =
318             p.transitions->at(i-1) in
319          if t.isMultiple then
320             t.getPreviousTransition()
321          else
322             t
```

```
323         endif
324      endif;
325
326
327   ----------------------------------------------------------------------------
328   -- RULES -------------------------------------------------------------------
329   ----------------------------------------------------------------------------
330
331   -- Rule 'Main'
332   -- This rule generates both a PathExp element and its initial State element
333   -- from the input root TextualPathExp element.
334   -- The generated PathExp element accepts an empty string as name. Its set of
335   -- states corresponds to the 'pe_s' elements generated for the input elements
336   -- of the singlePrimTransitions, singleAltTransitions sets, plus the 'pe_s'
337   -- elements generated for AlternativeTransition, plus the initial State
338   -- generated by the current rule. Its set of transitions corresponds to the
339   -- 'pe_t' elements generated for the input elements in the primTransitions,
340   -- altTransitions1, and altTransitions2 sets.
341   -- The generated initial State has an empty set of incoming transitions. Its
342   -- set of outgoing transitions corresponds to the 'pe_t' elements that are
343   -- generated for the outgoing transitions computed by the getOutgoing(0) call.
344   rule Main {
345      from
346           tpe : TextualPathExp!TextualPathExp
347      to
348         pe : PathExp!PathExp (
349            name <- '',
350            states <- Set{
351                     thisModule.singlePrimTransitions
352                        ->collect(e | thisModule.resolveTemp(e, 'pe_s')),
353                     thisModule.singleAltTransitions1
354                        ->collect(e | thisModule.resolveTemp(e, 'pe_s')),
355                     TextualPathExp!AlternativeTrans.allInstances()
356                        ->collect(e | thisModule.resolveTemp(e, 'pe_s')),
357                     pe_s
358                     },
359            transitions <- Set{
360                     thisModule.primTransitions
361                        ->collect(e | thisModule.resolveTemp(e, 'pe_t')),
362                     thisModule.altTransitions1
363                        ->collect(e | thisModule.resolveTemp(e, 'pe_t')),
364                     thisModule.altTransitions2
365                        ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
366                        }
367         ),
368
369         pe_s : PathExp!State (
370            incoming <- Set{},
371            outgoing <- Set{thisModule.rootTrans}
372                        ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
373         )
374   }
375
376
377   -- Rule 'AlternativeTrans'
378   -- This rule generates the State element that closes an input
379   -- AlternativeTransition element. The generated State is the one at which the
380   -- different alternative paths of the AlternativeTransition join.
381   -- Incoming transitions of the generated state correspond to the elements
382   -- generated for the last alternative transitions of the input
383   -- AlternativeTransition element.
384   -- Outgoing transitions of the generated state correspond to the 'pe_t'
385   -- elements generated for the set of transitions returned by the call of
386   -- getOutgoing(1).
387   rule AlternativeTrans {
388      from
389         tpe_at : TextualPathExp!AlternativeTrans (
390   --       tpe_at.isMultiple = false
```

```
391             true
392         )
393      to
394         pe_s : PathExp!State (
395             incoming <- thisModule.altTransitions2
396                         ->select(a | tpe_at.altPaths
397                            ->collect(b | b.transitions)
398                            ->flatten()
399                            ->includes(a)
400                         ),
401             outgoing <- tpe_at.getOutgoing()
402                         ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
403         )
404   }
405
406
407   -- Rule ''
408   -- This rule generates ...
409   --rule MultipleAlternativeTrans {
410   -- from
411   --    tpe_at : TextualPathExp!AlternativeTrans (
412   --       tpe_at.isMultiple = true
413   --    )
414   -- to
415   --    pe_s : PathExp!State (
416   --       outgoing <- Set{
417   --                tpe_at.getOutgoing()
418   --                   ->collect(e | thisModule.resolveTemp(e, 'pe_t')),
419   --                pe_t}
420   --    ),
421   --
422   --    pe_t : PathExp!Transition (
423   --       name <- ''--,
424   --       target <- pe_s
425   --    )
426   --}
427
428
429   -- Rule 'SinglePrimitiveTrans'
430   -- This rule generates both a Transition and a State for each PrimitiveTrans
431   -- element that belongs to the 'singlePrimTransitions' set.
432   -- The generated transition accepts as name the name of the input
433   -- PrimitiveTrans. If the input PrimitiveTrans is the root transition of the
434   -- input model, its source corresponds to the 'pe_s' initial state generated
435   -- for the input TextualPathExp element by rule 'Main'. Otherwise, the source
436   -- element corresponds to the 'pe_s' element generated for the transition that
437   -- precedes the input PrimitiveTrans in the current Path. Its target is the
438   -- State generated by the rule.
439   -- Incoming transitions for the generated State include the Transition
440   -- generated by the rule and, when the input Transition precedes a multiple
441   -- transition, the 'pe_t' element generated for this next transition.
442   -- If the input PrimitiveTrans is the leaf transition of the input model, the
443   -- generated State has no outgoing transitions. Otherwise, its outgoing
444   -- transition corresponds to the 'pe_t' element generated for the input
445   -- transition returned by the call of getOugoing(). Moreover, if the input
446   -- Transition precedes a multiple transition, the 'pe_t' element generated for
447   -- this next transition is added to the set ougoing transitions of the
448   -- generated State.
449   rule SinglePrimitiveTrans {
450      from
451          tpe_pt : TextualPathExp!PrimitiveTrans (
452            thisModule.singlePrimTransitions->includes(tpe_pt)
453            )
454      to
455         pe_t : PathExp!Transition (
456            name <- tpe_pt.name,
457            source <-
458                  if tpe_pt = thisModule.rootTrans then
```

```
459                     thisModule.resolveTemp(thisModule.root, 'pe_s')
460                  else
461                     let p : TextualPathExp!Path = tpe_pt.getPath()
462                     in let i : Integer = p.transitions->indexOf(tpe_pt)
463                     in let t : TextualPathExp!Transition =
464                        p.transitions->at(i-1)
465                     in thisModule.resolveTemp(t, 'pe_s')
466                  endif,
467            target <- pe_s
468         ),
469
470         pe_s : PathExp!State (
471            incoming <- Set{pe_t}->union(
472                        if tpe_pt.loopIncoming() then
473                           Set{thisModule.resolveTemp(tpe_pt.getLoopIncoming(), 'pe_t')}
474                        else
475                           Set{}
476                        endif
477                     ),
478            outgoing <- if tpe_pt = thisModule.leafTrans then
479                        Set{}
480                     else
481                        tpe_pt.getOutgoing()
482                           ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
483                           ->union(
484                           if tpe_pt.loopIncoming() then
485                              Set{thisModule.resolveTemp(tpe_pt.getLoopIncoming(), 'pe_t')}
486                           else
487                              Set{}
488                           endif
489                           )
490                     endif
491         )
492   }
493
494
495   -- Rule 'MultiplePrimitiveTrans'
496   -- This rule generates a loop transition for each transition that belongs
497   -- to the 'multiplePrimTransitions' set. The generated transition is a
498   -- transition from and to the state generated for the previous input
499   -- transition.
500   -- The generated loop transition accepts the name of the input Transition as
501   -- name.
502   -- If the input PrimitiveTrans is the root transition of the input model, its
503   -- source is the initial State generated by the 'Main' rule. Otherwise, the
504   -- source is computed by the getLoopTarget() helper.
505   -- If the input PrimitiveTrans is the root transition of the input model, its
506   -- target is the initial State generated by the 'Main' rule. Otherwise, the
507   -- target is computed by the getLoopTarget() helper.
508   rule MultiplePrimitiveTrans {
509      from
510          tpe_pt : TextualPathExp!PrimitiveTrans (
511            thisModule.multiplePrimTransitions->includes(tpe_pt)
512            )
513      to
514         pe_t : PathExp!Transition (
515            name <- tpe_pt.name,
516            source <-
517                  if tpe_pt = thisModule.rootTrans then
518                     thisModule.resolveTemp(thisModule.root, 'pe_s')
519                  else
520                     tpe_pt.getLoopTarget()
521                  endif,
522            target <-
523                  if tpe_pt = thisModule.rootTrans then
524                     thisModule.resolveTemp(thisModule.root, 'pe_s')
525                  else
526                     tpe_pt.getLoopTarget()
```

```
527                 endif
528         )
529     }
530
531
532     -- Rule 'SingleAltTrans1'
533     -- This rule generates both a Transition and a State for each PrimitiveTrans
534     -- element that belongs to the 'singleAltTransitions1' set.
535     -- The generated transition accepts as name the name of the input
536     -- PrimitiveTrans. Its source element corresponds to the 'pe_s' element
537     -- generated for the transition returned by the call of the
538     -- 'getPrevioustransition' helper. Its target is the State generated by the
539     -- rule.
540     -- Incoming transitions for the generated State include the Transition
541     -- generated by the rule and, when the input Transition precedes a multiple
542     -- transition, the 'pe_t' element generated for this next transition.
543     -- Outgoing transitions for the generated State include to the 'pe_t' element
544     -- generated for the input transition returned by the call of getOugoing(1).
545     -- Moreover, if the input Transition precedes a multiple transition, the 'pe_t'
546     -- element generated for this next transition is added to the set ougoing
547     -- transitions of the generated State.
548     rule SingleAltTrans1 {
549        from
550            tpe_pt : TextualPathExp!PrimitiveTrans (
551              thisModule.singleAltTransitions1->includes(tpe_pt)
552              )
553        to
554           pe_t : PathExp!Transition (
555              name <- tpe_pt.name,
556              source <-
557                    thisModule.resolveTemp(
558                       tpe_pt.getPreviousTransition(),
559                       'pe_s'
560                    ),
561              target <- pe_s
562           ),
563
564           pe_s : PathExp!State (
565              incoming <- Set{pe_t}->union(
566                          if tpe_pt.loopIncoming() then
567                             Set{thisModule.resolveTemp(tpe_pt.getLoopIncoming(), 'pe_t')}
568                          else
569                             Set{}
570                          endif
571                       ),
572              outgoing <- tpe_pt.getOutgoing()
573                          ->collect(e | thisModule.resolveTemp(e, 'pe_t'))
574                          ->union(
575                          if tpe_pt.loopIncoming() then
576                             Set{thisModule.resolveTemp(tpe_pt.getLoopIncoming(), 'pe_t')}
577                          else
578                             Set{}
579                          endif
580                          )
581        )
582     }
583
584
585     -- Rule 'MultipleAltTrans1'
586     -- This rule generates a loop transition for each transition that belongs
587     -- to the 'multipleAltTransitions1' set. The generated transition is a
588     -- transition from and to the state generated for the previous input
589     -- transition.
590     -- The generated loop transition accepts the name of the input Transition as
591     -- name.
592     -- Its source corresponds to the 'pe_s' element generated for the input State
593     -- returned by the call to the getLoopTarget() helper.
594     -- Its target corresponds to the 'pe_s' element generated for the input State
```

```
595    -- returned by the call to the getLoopTarget() helper.
596    rule MultipleAltTrans1 {
597       from
598           tpe_pt : TextualPathExp!PrimitiveTrans (
599             thisModule.multipleAltTransitions1->includes(tpe_pt)
600             )
601       to
602          pe_t : PathExp!Transition (
603             name <- tpe_pt.name,
604             source <- thisModule.resolveTemp(tpe_pt.getLoopTarget(), 'pe_s'),
605             target <- thisModule.resolveTemp(tpe_pt.getLoopTarget(), 'pe_s')
606          )
607    }
608
609
610    -- Rule 'AltTrans2'
611    -- This rule generates a Transition from the last Transition of a Path
612    -- contained by an AlternativeTransition. The generated transition goes from
613    -- the state generated for the previous transition to the final state generated
614    -- for the current AlternativeTransition by the 'AlternativeTrans' helper.
615    -- The generated loop transition accepts the name of the input Transition as
616    -- name.
617    -- Its source corresponds to the 'pe_s' element generated for the input element
618    -- returned by the call of the 'getPreviousTransition()' helper.
619    -- Its target corresponds to the 'pe_s' element generated for the
620    -- AlternativeTransition element that contains the rule input PrimitiveTrans
621    -- element in one of its alternative pathes.
622    rule AltTrans2 {
623       from
624           tpe_pt : TextualPathExp!PrimitiveTrans (
625             thisModule.altTransitions2->includes(tpe_pt)
626             )
627       to
628          pe_t : PathExp!Transition (
629             name <- tpe_pt.name,
630             source <- thisModule.resolveTemp(
631                      tpe_pt.getPreviousTransition(),
632                      'pe_s'),
633             target <- thisModule.resolveTemp(
634                      TextualPathExp!AlternativeTrans.allInstances()
635                         ->select(a | a.altPaths
636                            ->collect(b | b.transitions)
637                            ->flatten()
638                            ->includes(tpe_pt)
639                         )->asSequence()
640                         ->first(),
641                      'pe_s')
642          )
643    }
```

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.

```
 1  module PathExp2PetriNet;
 2  create OUT : PetriNet from IN : PathExp;
 3
 4
 5  ------------------------------------------------------------------------------
 6  -- HELPERS ---------------------------------------------------------------------
 7  ------------------------------------------------------------------------------
 8
 9
10  -- This helper computes the Set containing all the Transitions of the input
11  -- PathExp model.
12  -- CONTEXT: thisModule
13  -- RETURN:  Set(PathExp!Transition)
14  helper def: allTransitions : Set(PathExp!Transition) =
15     PathExp!Transition.allInstances();
16
17
18  ------------------------------------------------------------------------------
19  -- RULES -----------------------------------------------------------------------
20  ------------------------------------------------------------------------------
21
22
23  -- Rule 'Main'
24  -- This rule generates a PetriNet element from the input PathExp element.
25  -- The name of the generated PetriNet is copied from the input PathExp element.
```

```
26    -- Its set of places and its set of transitions respectively correspond to the
27    -- elements generated for states and the transitions of the input PathExp.
28    -- Its set of arcs correspond to the 'pn_ia' and 'pn_oa' elements generated for
29    -- the input Transition elements.
30    rule Main {
31       from
32            pe : PathExp!PathExp
33       to
34          pn : PetriNet!PetriNet (
35             name <- pe.name,
36             places <- pe.states,
37             transitions <- pe.transitions,
38             arcs <- thisModule.allTransitions
39                      ->collect(e | thisModule.resolveTemp(e, 'pn_ia'))
40                      ->union(
41                         thisModule.allTransitions
42                         ->collect(e | thisModule.resolveTemp(e, 'pn_oa'))
43                      )
44          )
45    }
46
47
48    -- Rule 'State'
49    -- This rule generates a Place element from an input State element.
50    -- Generated Place accepts an empty string as name.
51    -- Its set of incoming arcs correspond to 'pn_oa' elements that are generated
52    -- for the incoming Transitions of the input State.
53    -- Its set of outgoing arcs correspond to 'pn_ia' elements that are generated
54    -- for the outgoing Transitions of the input State.
55    rule State {
56       from
57           pe_s : PathExp!State
58       to
59          pn_p : PetriNet!Place (
60             name <- '',
61             incoming <- pe_s.incoming
62                         ->collect(e | thisModule.resolveTemp(e, 'pn_oa')),
63             outgoing <- pe_s.outgoing
64                         ->collect(e | thisModule.resolveTemp(e, 'pn_ia'))
65          )
66    }
67
68
69    -- Rule 'Transition'
70    -- From an input PathExp Transition, this rule generates:
71    --  * a PetriNet Transition
72    --  * a PlaceToTransArc
73    --  * a TransToPlaceArc
74    -- The generated Transition accepts an empty string as name, the generated
75    -- 'pn_ia' PlaceToTransArc as incoming arc, and the generated 'pn_oa'
76    -- TransToPlaceArc as outgoing arc.
77    -- The generated PlaceToTransArc accepts the element generated for the source
78    -- of the input PathExpTransition as source, and the generated PetriNet
79    -- Transition as target.
80    -- The generated TransToPlaceArc accepts the generated PetriNet Transition as
81    -- source, and the element generated for the target of the input
82    -- PathExpTransition as target.
83    rule Transition {
84       from
85           pe_t : PathExp!Transition
86       to
87          pn_t : PetriNet!Transition (
88             name <- '',
89             incoming <- pn_ia,
90             outgoing <- pn_oa
91          ),
92
93          pn_ia : PetriNet!PlaceToTransArc (
```

```
 94            source <- pe_t.source,
 95            target <- pn_t,
 96            weight <- 1
 97         ),
 98
 99         pn_oa : PetriNet!TransToPlaceArc (
100            source <- pn_t,
101            target <- pe_t.target,
102            weight <- 1
103         )
104    }
```

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.

```
1   module PetriNet2XML;
2   create OUT : XML from IN : PetriNet;
3
4
5   ------------------------------------------------------------------------------
6   -- HELPERS ---------------------------------------------------------------------
7   ------------------------------------------------------------------------------
8
9
10  -- This helper computes a Sequence that contains all the Places of the input
11  -- PetriNet model.
12  -- CONTEXT: thisModule
13  -- RETURN:  Sequence(PetriNet!Place)
14  helper def: allPlaces : Sequence(PetriNet!Place) =
15     PetriNet!Place.allInstances()->asSequence();
16
17
18  -- This helper computes a Sequence that contains all the Transitions of the
19  -- input PetriNet model.
20  -- CONTEXT: thisModule
21  -- RETURN:  Sequence(PetriNet!Transition)
22  helper def: allTransitions : Sequence(PetriNet!Transition) =
23     PetriNet!Transition.allInstances()->asSequence();
24
25
26  -- This helper computes a Sequence that contains all the Arcs of the input
27  -- PetriNet model.
28  -- CONTEXT: thisModule
29  -- RETURN:  Sequence(PetriNet!Arc)
30  helper def: allArcs : Sequence(PetriNet!Arc) =
31     PetriNet!Arc.allInstances()->asSequence();
32
33
34  ------------------------------------------------------------------------------
35  -- RULES -----------------------------------------------------------------------
36  ------------------------------------------------------------------------------
37
38
39  -- Rule 'Main'
40  -- This rule generates the "pnml" root tag from the input PetriNet element.
41  -- This tag has an "xmlns" attribute and a "net" element as child element.
42  -- The "net" tag has an "id", a "type" and a "name" attributes, and the
43  -- following children elements:
44  --  * a "place" element for each Place of the input PetriNet model
45  --  * a "transition" element for each Transition of the input PetriNet model
46  --  * an "arc" element for each Arc of the input PetriNet model.
47  rule Main {
48     from
49          pn : PetriNet!PetriNet
50     to
51        root : XML!Root (
52           name <- 'pnml',
53           children <- Sequence{xmlns, net}
54        ),
55        xmlns: XML!Attribute (
56           name <- 'xmlns',
57           value <- 'http://www.example.org/pnpl'
58        ),
59
```

```
60          net : XML!Element (
61             name <- 'net',
62             children <- Sequence{
63                         id,
64                         type,
65                         name,
66                         thisModule.allPlaces,
67                         thisModule.allTransitions,
68                         thisModule.allArcs
69                      }
70          ),
71          id : XML!Attribute (
72             name <- 'id',
73             value <- 'n1'
74          ),
75          type : XML!Attribute (
76             name <- 'type',
77             value <- 'http://www.example.org/pnpl/PTNet'
78          ),
79
80          name : XML!Element (
81             name <- 'name',
82             children <- Sequence{text}
83          ),
84          text : XML!Element (
85             name <- 'text',
86             children <- Sequence{val}
87          ),
88          val : XML!Text (
89             value <- pn.name
90          )
91    }
92
93
94    -- Rule 'Place'
95    -- This rule generates a "place" tag from an input Place element.
96    -- This tag has an "id" attribute which value corresponds to the Place rank
97    -- within the allPlaces sequence.
98    -- The "place" tag also has a "name" child element, which has itself a "text"
99    -- child element that contains the name of the place (copied from the input
100   -- Place element).
101   rule Place {
102      from
103           pn_s : PetriNet!Place
104      to
105         xml_place : XML!Element (
106            name <- 'place',
107            children <- Sequence{id, name}
108         ),
109         id : XML!Attribute (
110            name <- 'id',
111            value <- thisModule.allPlaces->indexOf(pn_s).toString()
112         ),
113
114         name : XML!Element (
115            name <- 'name',
116            children <- Sequence{text}
117         ),
118         text : XML!Element (
119            name <- 'text',
120            children <- Sequence{val}
121         ),
122         val : XML!Text (
123            value <- pn_s.name
124         )
125   }
126
127
```

```
128    -- Rule 'Transition'
129    -- This rule generates a "transition" tag from an input Transition element.
130    -- This tag has an "id" attribute which value corresponds to (the size of the
131    -- allPlaces sequence + the Transition rank within the allTransitions
132    -- sequence).
133    rule Transition {
134       from
135           pn_t : PetriNet!Transition
136       to
137          xml_trans : XML!Element (
138             name <- 'transition',
139             children <- Sequence{trans_id}
140          ),
141          trans_id : XML!Attribute (
142             name <- 'id',
143             value <- (thisModule.allPlaces->size() +
144                     thisModule.allTransitions->indexOf(pn_t)).toString()
145          )
146    }
147
148
149    -- Rule 'PlaceToTransArc'
150    -- This rule generates an "arc" tag from an input PlaceToTransArc element.
151    -- This tag has an "id", a "source" and a "target" attributes.
152    -- Value of the "id" attribute corresponds to (the size of the allPlaces
153    -- sequence + the size of the allTransitions sequence + the Arc rank within
154    -- the allArcs sequence).
155    -- Value of the "source" attribute corresponds to the source Place rank
156    -- within the allPlaces sequence.
157    -- Value of the "target" attribute corresponds to (the size of the allPlaces
158    -- sequence + the target Transition rank within the allTransitions sequence).
159    rule PlaceToTransArc {
160       from
161          pn_a : PetriNet!PlaceToTransArc
162       to
163          xml_arc : XML!Element (
164             name <- 'arc',
165             children <- Sequence{id, source, target}
166          ),
167          id : XML!Attribute (
168             name <- 'id',
169             value <- (thisModule.allPlaces->size() +
170                     thisModule.allTransitions->size() +
171                     thisModule.allArcs->indexOf(pn_a)).toString()
172          ),
173          source : XML!Attribute (
174             name <- 'source',
175             value <- thisModule.allPlaces
176                      ->indexOf(pn_a.source).toString()
177          ),
178          target : XML!Attribute (
179             name <- 'target',
180             value <- (thisModule.allPlaces->size() +
181                     thisModule.allTransitions
182                      ->indexOf(pn_a.target)).toString()
183          )
184    }
185
186
187    -- Rule 'TransToPlaceArc'
188    -- This rule generates an "arc" tag from an input TransToPlaceArc element.
189    -- This tag has an "id", a "source" and a "target" attributes.
190    -- Value of the "id" attribute corresponds to (the size of the allPlaces
191    -- sequence + the size of the allTransitions sequence + the Arc rank within
192    -- the allArcs sequence).
193    -- Value of the "source" attribute corresponds to (the size of the allPlaces
194    -- sequence + the source Transition rank within the allTransitions sequence).
195    -- Value of the "target" attribute corresponds to the target Place rank
```

```
196    -- within the allPlaces sequence.
197    rule TransToPlaceArc {
198       from
199          pn_a : PetriNet!TransToPlaceArc
200       to
201          xml_arc : XML!Element (
202             name <- 'arc',
203             children <- Sequence{id, source, target}
204          ),
205          id : XML!Attribute (
206             name <- 'id',
207             value <- (thisModule.allPlaces->size() +
208                     thisModule.allTransitions->size() +
209                     thisModule.allArcs->indexOf(pn_a)).toString()
210          ),
211          source : XML!Attribute (
212             name <- 'source',
213             value <- (thisModule.allPlaces->size() +
214                     thisModule.allTransitions
215                      ->indexOf(pn_a.source)).toString()
216          ),
217          target : XML!Attribute (
218             name <- 'target',
219             value <- thisModule.allPlaces
220                      ->indexOf(pn_a.target).toString()
221          )
222    }
```

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.

```
1    module XML2PetriNet;
2    create OUT : PetriNet from IN : XML;
3
```

```
4
5   ------------------------------------------------------------------------
6   -- HELPERS ---------------------------------------------------------------
7   ------------------------------------------------------------------------
8
9
10  -- This helper computes the Set containing all the XML!Element of the input
11  -- XML model that are named 'place'.
12  -- CONTEXT: thisModule
13  -- RETURN:  Set(XML!Element)
14  helper def: allPlaces : Set(XML!Element) =
15     XML!Element.allInstances()
16        ->select(e | e.name = 'place');
17
18
19  -- This helper computes the Set containing all the XML!Element of the input
20  -- XML model that are named 'transition'.
21  -- CONTEXT: thisModule
22  -- RETURN:  Set(XML!Element)
23  helper def: allTransitions : Set(XML!Element) =
24     XML!Element.allInstances()
25        ->select(e | e.name = 'transition');
26
27
28  -- This helper computes the Set containing all the XML!Element of the input
29  -- XML model that are named 'arc'.
30  -- CONTEXT: thisModule
31  -- RETURN:  Set(XML!Element)
32  helper def: allArcs : Set(XML!Element) =
33     XML!Element.allInstances()
34        ->select(e | e.name = 'arc');
35
36
37  -- This helper computes the name value of an input XML!Element.
38  -- For this purpose, it first selects among its elements children the one
39  -- named 'name'. It then selects, among children of this new element, the one
40  -- named 'text'. It then selects the XML!Text child of this last element and
41  -- returns its value.
42  -- CONTEXT: XML!Element
43  -- RETURN:  String
44  helper context XML!Element def : getName() : String =
45     self.children
46        ->select(c | c.oclIsTypeOf(XML!Element) and c.name = 'name')
47        ->first().children
48        ->select(c | c.oclIsTypeOf(XML!Element) and c.name = 'text')
49        ->first().children
50        ->first().value;
51
52
53  -- This helper calculates the value of a given attribute (identified by the
54  -- name provided as a parameter) of the contextual XML!Element.
55  -- To this end, it selects among its attribute children the one which has the
56  -- name provided in parameter, and returns its value.
57  -- CONTEXT: XML!Element
58  -- IN:      String
59  -- RETURN:  String
60  helper context XML!Element def : getAttributeValue(name : String) : String =
61     self.children
62        ->select(c | c.oclIsTypeOf(XML!Attribute) and c.name = name)
63        ->first().value;
64
65
66  -- This helper calculates the value of the 'id' attribute of the contextual
67  -- XML!Element. For this purpose, it calls the 'getAttributeValue' with 'id'
68  -- as parameter.
69  -- CONTEXT: XML!Element
70  -- RETURN:  String
71  helper context XML!Element def : getId() : String =
```

```
72      self.getAttributeValue('id');
73
74
75    -- This helper calculates the value of the 'target' attribute of the contextual
76    -- XML!Element. For this purpose, it calls the 'getAttributeValue' with
77    -- 'target' as parameter.
78    -- CONTEXT: XML!Element
79    -- RETURN:  String
80    helper context XML!Element def : getTarget() : String =
81       self.getAttributeValue('target');
82
83
84    -- This helper calculates the value of the 'source' attribute of the contextual
85    -- XML!Element. For this purpose, it calls the 'getAttributeValue' with
86    -- 'source' as parameter.
87    -- CONTEXT: XML!Element
88    -- RETURN:  String
89    helper context XML!Element def : getSource() : String =
90       self.getAttributeValue('source');
91
92
93    ------------------------------------------------------------------------------
94    -- RULES -----------------------------------------------------------------
95    ------------------------------------------------------------------------------
96
97
98    -- Rule 'Main'
99    -- This rule generates a PetriNet element from the XML!Element called 'net'.
100   -- Name of the generated PetriNet is computed by the 'getName' helper.
101   -- Its places, transitions and arcs respectively correspond to the elements
102   -- generated for the XML!Elements named 'place', 'transition', and 'arc'.
103   rule Main {
104      from
105           xml_net : XML!Element (
106            xml_net.name = 'net'
107           )
108      to
109         pn : PetriNet!PetriNet (
110            name <- xml_net.getName(),
111            places <- thisModule.allPlaces,
112            transitions <- thisModule.allTransitions,
113            arcs <- thisModule.allArcs
114         )
115   }
116
117
118   -- Rule 'State'
119   -- This rule generates a Place element for each XML!Element called 'place'.
120   -- Name of the generated Place is computed by the 'getName' helper.
121   -- Its incoming arcs correspond to the elements generated for the XML!Element
122   -- named 'arc' whose target is the input 'place' XML!Element.
123   -- Its outgoing arcs correspond to the elements generated for the XML!Element
124   -- named 'arc' whose source is the input 'place' XML!Element.
125   rule Place {
126      from
127           xml_place :  XML!Element (
128            xml_place.name = 'place'
129           )
130      to
131         pn_p : PetriNet!Place (
132            name <- xml_place.getName(),
133            incoming <- thisModule.allArcs
134                        ->select(a | a.getTarget() = xml_place.getId()),
135            outgoing <- thisModule.allArcs
136                        ->select(a | a.getSource() = xml_place.getId())
137         )
138   }
139
```

```
140
141    -- Rule 'Transition'
142    -- This rule generates a Transition element for each XML!Element called
143    -- 'transition'.
144    -- Generated Place accepts an empty string as name.
145    -- Its incoming arcs correspond to the elements generated for the XML!Element
146    -- named 'arc' whose target is the input 'transition' XML!Element.
147    -- Its outgoing arcs correspond to the elements generated for the XML!Element
148    -- named 'arc' whose source is the input 'transition' XML!Element.
149    rule Transition {
150       from
151            xml_trans :  XML!Element (
152             xml_trans.name = 'transition'
153            )
154       to
155          pn_t : PetriNet!Transition (
156             name <- '',
157             incoming <- thisModule.allArcs
158                         ->select(a | a.getTarget() = xml_trans.getId()),
159             outgoing <- thisModule.allArcs
160                         ->select(a | a.getSource() = xml_trans.getId())
161          )
162    }
163
164
165    -- Rule 'PlaceToTransArc'
166    -- This rule generates a PlaceToTransArc element for each XML!Element called
167    -- 'arc' whose source is a 'place' XML!Element.
168    -- The source of the generated PlaceToTransArc corresponds to the element
169    -- generated for the 'place' XML!Element whose 'id' is equal to the source of
170    -- the input 'arc' XML!Element.
171    -- The target of the generated PlaceToTransArc corresponds to the element
172    -- generated for the 'transition' XML!Element whose 'id' is equal to the target
173    -- of the input 'arc' XML!Element.
174    rule PlaceToTransArc {
175       from
176            xml_arc :  XML!Element (
177             if xml_arc.name = 'arc' then
178                thisModule.allPlaces
179                   ->collect(p | p.getId())
180                   ->includes(xml_arc.getSource())
181             else
182                false
183             endif
184            )
185       to
186          pn_a : PetriNet!PlaceToTransArc (
187             weight <- 1,
188             source <- thisModule.allPlaces
189                      ->select(b | b.getId() = xml_arc.getSource())
190                      ->first(),
191             target <- thisModule.allTransitions
192                      ->select(b | b.getId() = xml_arc.getTarget())
193                      ->first()
194          )
195    }
196
197
198    -- Rule 'TransToPlaceArc'
199    -- This rule generates a TransToPlaceArc element for each XML!Element called
200    -- 'arc' whose source is an 'transition' XML!Element.
201    -- The source of the generated TransToPlaceArc corresponds to the element
202    -- generated for the 'transition' XML!Element whose 'id' is equal to the source
203    -- of the input 'arc' XML!Element.
204    -- The target of the generated TransToPlaceArc corresponds to the element
205    -- generated for the 'place' XML!Element whose 'id' is equal to the target of
206    -- the input 'arc' XML!Element.
207    rule TransToPlaceArc {
```

```
208       from
209            xml_arc :  XML!Element (
210             if xml_arc.name = 'arc' then
211                thisModule.allTransitions
212                   ->collect(p | p.getId())
213                   ->includes(xml_arc.getSource())
214             else
215                false
216             endif
217            )
218       to
219          pn_a : PetriNet!TransToPlaceArc (
220             weight <- 1,
221             source <- thisModule.allTransitions
222                      ->select(b | b.getId() = xml_arc.getSource())
223                      ->first(),
224             target <- thisModule.allPlaces
225                      ->select(b | b.getId() = xml_arc.getTarget())
226                      ->first()
227          )
228    }
```

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.

```
 1  module PetriNet2PathExp;
 2  create OUT : PathExp from IN : PetriNet;
 3
 4
 5  ------------------------------------------------------------------------------
 6  -- RULES -----------------------------------------------------------------------
 7  ------------------------------------------------------------------------------
 8
 9  -- Rule 'Main'
10  -- This rule generates a PathExp from the input PetriNet element.
11  -- Name of the generated PathExp is copied from the PetriNet one.
12  -- Its set of states and transitions respectively correspond to the elements
13  -- that are generated for the input Places and Transitions.
14  rule Main {
15     from
16          pn : PetriNet!PetriNet
17     to
18        pe : PathExp!PathExp (
19           name <- pn.name,
20           states <- pn.places,
21           transitions <- pn.transitions
22        )
23  }
24
25
26  -- Rule 'Place'
27  -- This rule generates State for each input Place element.
28  -- The set of incoming transitions of the generated Place corresponds to the
29  -- elements generated for Transitions that are source of the incoming
30  -- PetriNet!Arc.
31  -- The set of outgoing transitions of the generated Place corresponds to the
32  -- elements generated for Transitions that are tagret of the outgoing
33  -- PetriNet!Arc.
34  rule Place {
35     from
36          pn_p : PetriNet!Place
37     to
38        pe_s : PathExp!State (
```

```
39              incoming <- pn_p.incoming
40                          ->collect(e | e.source)
41                          ->flatten(),
42              outgoing <- pn_p.outgoing
43                          ->collect(e | e.target)
44                          ->flatten()
45          )
46      }
47
48
49      -- Rule 'Transition'
50      -- This rule generates a PathExp!Transition for each PetriNet!Transition.
51      -- Source of the generated Transition corresponds to the element generated for
52      -- the Place that is the source of the incoming PetriNet!Arc.
53      -- Target of the generated Transition corresponds to the element generated for
54      -- the Place that is the target of the outgoing PetriNet!Arc.
55      rule Transition {
56         from
57              pn_t : PetriNet!Transition
58         to
59            pe_t : PathExp!Transition (
60               name <- '',
61               source <- pn_t.incoming
62                         ->collect(e | e.source)
63                         ->flatten()
64                         ->first(),
65               target <- pn_t.outgoing
66                         ->collect(e | e.target)
67                         ->flatten()
68                         ->first()
69          )
70      }
```

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**.

```
1   module PathExp2TextualPathExp;
2   create OUT : TextualPathExp from IN : PathExp;
3
4
5   --------------------------------------------------------------------------
6   -- HELPERS ---------------------------------------------------------------
7   --------------------------------------------------------------------------
8
9   -- This helper computes the root State of the input PathExp model.
10  -- To this end, it selects among all State instances the one that has no
11  -- incoming transition.
12  -- CONTEXT: thisModule
13  -- RETURN:  PathExp!State
14  helper def: rootState : PathExp!State =
15     PathExp!State.allInstances()
16        ->select(s | s.incoming->isEmpty())
17        ->asSequence()
18        ->first();
19
20
21  -- This helper computes a boolean value stating whether a loop transition is
22  -- defined for the contextual State.
23  -- For this purpose, the helper checks if there exists an incoming transition
24  -- whose source is the contextual State.
25  -- CONTEXT: PathExp!State
26  -- RETURN:  Boolean
27  helper context PathExp!State def: existLoop() : Boolean =
28     self.incoming
29        ->select(e | e.source = self)
30        ->notEmpty();
31
32
33  -- This helper returns the loop Transition defined for the contextual State.
34  -- To this end, it returns the first incoming transition that has the
35  -- contextual State as source.
36  -- PRECOND: a loop transition must be defined for the contextual State.
37  -- CONTEXT: PathExp!State
38  -- RETURN:  PathExp!Transition
39  helper context PathExp!State def: getLoop() : PathExp!Transition =
```

```
40      self.incoming
41         ->select(e | e.source = self)
42         ->asSequence()
43         ->first();
44
45
46   -- This helper computes the set of non-loop incoming transitions of the
47   -- contextual State.
48   -- To this end, it selects among incoming transitions the ones that do not
49   -- target the contextual State.
50   -- CONTEXT: PathExp!State
51   -- RETURN:  Sequence(PathExp!Transition)
52   helper context PathExp!State def: getInT() : Sequence(PathExp!Transition) =
53      self.incoming
54         ->select(e | e.source <> self)
55         ->asSequence();
56
57
58   -- This helper computes the set of non-loop outgoing transitions of the
59   -- contextual State.
60   -- To this end, it selects among outgoing transitions the ones that do not
61   -- target the contextual State.
62   -- CONTEXT: PathExp!State
63   -- RETURN:  Sequence(PathExp!Transition)
64   helper context PathExp!State def: getOutT() : Sequence(PathExp!Transition) =
65      self.outgoing
66         ->select(e | e.target <> self)
67         ->asSequence();
68
69
70   -- This helper computes the set of States whose transitions lead to the
71   -- contextual State (except the contextual State if it is reachable from itself
72   -- by means of a loop transitions).
73   -- For this purpose, the helper simply collects the source of the transitions
74   -- returned by the call of the 'getInT' helper onto the contextual State.
75   -- CONTEXT: PathExp!State
76   -- RETURN:  Sequence(PathExp!State)
77   helper context PathExp!State def: getPrevStates() : Sequence(PathExp!State) =
78      self.getInT()->collect(e | e.source);
79
80
81   -- This helper computes the set of States that can be reached by means of
82   -- outgoing transitions of the contextual State (except the contextual State
83   -- if it is reachable from itself through a loop transitions).
84   -- For this purpose, the helper simply collects the target of the transitions
85   -- returned by the call of the 'getOutT' helper onto the contextual State.
86   -- CONTEXT: PathExp!State
87   -- RETURN:  Sequence(PathExp!State)
88   helper context PathExp!State def: getNextStates() : Sequence(PathExp!State) =
89      self.getOutT()->collect(e | e.target);
90
91
92   -- This helper computes the sequence of both Path!State and Path!Transition
93   -- input elements that correspond to the transitions of the Path initiated by
94   -- the contextual State.
95   -- The helper accepts a Boolean parameter that encodes the fact that what
96   -- cooresponds to a nested alternative transition has just been parsed. The
97   -- helper is initially called with false as parameter.
98   --  * A contextual State with several non-loop incoming transitions along with
99   --    a false nested parameter, identifies the end of the current Path. The
100  --    helper therefore returns an empty sequence.
101  --  * If the contextual State has several non-loop outgoing transitions with a
102  --    true along with a nested parameter, this identifies the beginning of a
103  --    new nested alternative transition within the current Path. The helper
104  --    then returns a sequence made of 1) the loop transition of the contextual
105  --    State, if it is defined, 2) the contextual State itself, and 3) the
106  --    sequence returned by a recursive call of 'getTransitionsFromState' on the
107  --    state that closes the new alternative transition (it is computed by the
```

```
108    --    'findNextState' helper), with the nested parameter set to true.
109    --  * If the contextual State has a single non-loop outgoing transition, the
110    --    helper returns a sequence made of 1) the loop transition of the
111    --    contextual State, if it is defined, 2) the outgoing transition of the
112    --    contextual State, and 3) the sequence returned by a recursive call of
113    --    'getTransitionsFromState' onto the the next state of the contextual
114    --    State, with the nested parameter set to false.
115    --  * Finally, a contextual State with no outgoing Transitions indicates the
116    --    end of the input PathExp and (also) of the current Path. The  helper
117    --    therefore returns an empty sequence.
118    --
119    -- NOTE: the result type of the helper is currently encoded as a sequence of
120    -- strings since 1) the oclAny type is not implemented yet 2) and no type
121    -- verifications are performed by the current atl version.
122    --
123    -- CONTEXT: PathExp!State
124    -- IN:      Boolean
125    -- RETURN:  Sequence(oclAny)
126    helper context PathExp!State
127       def: getTransitionsFromState(nested : Boolean) : Sequence(String) =
128       let nextStates : Sequence(PathExp!State) = self.getNextStates()
129       in let prevStates : Sequence(PathExp!State) = self.getPrevStates()
130       in let loop : Sequence(PathExp!Transition) =
131          if self.existLoop() then
132             self.getLoop()
133          else
134             Sequence{}
135          endif
136       in
137
138       if prevStates->size() > 1 and not nested then
139          Sequence{}
140       else
141          if nextStates->size() > 1 then
142             let state : PathExp!State = nextStates->first().findNextState(0)
143             in Sequence{
144                   loop,
145                   self,
146                   state.getTransitionsFromState(true)
147                }->flatten()
148          else
149             if nextStates->size() = 1 then
150                Sequence{
151                   loop,
152                   self.getOutT()->first(),
153                   nextStates->first().getTransitionsFromState(false)
154                }->flatten()
155             else
156                Sequence{}
157             endif
158          endif
159       endif;
160
161
162    -- This helper computes the sequence of both Path!State and Path!Transition
163    -- input elements that correspond to the transitions of the Path initiated by
164    -- the contextual Transition.
165    -- The returned sequence is composed of the contextual transition followed by
166    -- the result of the call of the 'getTransitionsFromState' helper onto the
167    -- target of this contextual transition.
168    --
169    -- NOTE: the result type of the helper is currently encoded as a sequence of
170    -- strings since 1) the oclAny type is not implemented yet 2) and no type
171    -- verifications are performed by the current atl version.
172    --
173    -- CONTEXT: PathExp!State
174    -- RETURN:  Sequence(oclAny)
175    helper context PathExp!Transition
```

```
176       def: getTransitionsFromTrans() : Sequence(String) =
177       Sequence{self, self.target.getTransitionsFromState(false)}->flatten();
178
179
180    -- This helper aims to compute the State that closes the alternative transition
181    -- that is started at the contextual State of the initial call.
182    -- It accepts an Integer as parameter which indicates the number of opened
183    -- nested alternative transitions. It is initially called with n = 0.
184    -- In order to compute its closing State, the helper recursively parses the
185    -- Path:
186    --  * if the contextual State has more than one incoming transition and no
187    --    nested alternative trans. are opened (n=0), the helper returns the
188    --    contextualState.
189    --  * if the contextual State has more than one incoming transition and more
190    --    than one outgoing transition, the helper returns the value provided by
191    --    the recursive call of 'findNextState(n)' onto one of the next states of
192    --    the contextual state.
193    --  * if the contextual State has more than one incoming transition but a
194    --    single outgoing transition, the helper returns the value provided by the
195    --    recursive call of 'findNextState(n-1)' onto the next state of the
196    --    contextual state.
197    --  * if the contextual State has a single incoming transition and more than
198    --    one outgoing transition, the helper returns the value provided by the
199    --    'findNextState(n+1)' onto one of the next states of the contextual state.
200    --  * finally, if the contextual State has a single incoming transition and a
201    --    single outgoing transition, the helper returns the value provided by the
202    --    recursive call of 'findNextState(n+)' onto the next state of the
203    --    contextual state.
204    -- CONTEXT: PathExp!State
205    -- IN:      Integer
206    -- RETURN:  PathExp!State
207    helper context PathExp!State def: findNextState(n : Integer) : PathExp!State =
208       let prevStates : Sequence(PathExp!State) = self.getPrevStates() in
209       let nextStates : Sequence(PathExp!State) = self.getNextStates() in
210       if prevStates->size() > 1 and n = 0 then
211          self
212       else
213          if prevStates->size() > 1 then
214             if nextStates->size() > 1 then
215                nextStates->first().findNextState(n)
216             else
217                nextStates->first().findNextState(n-1)
218             endif
219          else
220             if nextStates->size() > 1 then
221                nextStates->first().findNextState(n+1)
222             else
223                nextStates->first().findNextState(n)
224             endif
225          endif
226       endif;
227
228
229    ----------------------------------------------------------------------------
230    -- RULES -------------------------------------------------------------------
231    ----------------------------------------------------------------------------
232
233    -- Rule 'Main'
234    -- This rule generates both a TextualPathExp and its main Path from the root
235    -- PathExp input element.
236    -- The generated TextualPathExp accepts the Path generated by the rule as path.
237    -- The sequence of transitions contained by the generated Path is returned by
238    -- the call of the 'getTransitionsFromState' helper onto the root State element
239    -- of the input model.
240    rule Main {
241       from
242            pe : PathExp!PathExp
243       to
```

```
244         tpe : TextualPathExp!TextualPathExp (
245            path <- p
246         ),
247
248         p : TextualPathExp!Path (
249            transitions <- thisModule.rootState.getTransitionsFromState(false)
250         )
251   }
252
253
254   -- Rule 'Loop'
255   -- This rule generates a multiple PrimitiveTrans from a loop Transition.
256   -- The generated PrimitiveTrans accepts an empty string as name. Its
257   -- 'isMultiple' attribute is set to 'true'.
258   rule Loop {
259      from
260          t : PathExp!Transition (
261            t.source = t.target
262          )
263      to
264         pt : TextualPathExp!PrimitiveTrans (
265            name <- '',
266            isMultiple <- true
267         )
268   }
269
270
271   -- Rule 'STransition'
272   -- This rule generates a simple PrimitiveTrans from a non-loop transition
273   -- which is the only outgoing transition of its source State.
274   -- The generated PrimitiveTrans accepts an empty string as name. Its
275   -- 'isMultiple' attribute is set to 'false'.
276   rule STransition {
277      from
278          t : PathExp!Transition (
279            t.source <> t.target and
280            t.source.getOutT()->size() = 1
281          )
282      to
283         pt : TextualPathExp!PrimitiveTrans (
284            name <- '',
285            isMultiple <- false
286         )
287   }
288
289
290   -- Rule 'MTransition'
291   -- This rule generates a simple PrimitiveTrans from a non-loop transition
292   -- which is NOT the only outgoing transition of its source State.
293   -- The generated PrimitiveTrans accepts an empty string as name. Its
294   -- 'isMultiple' attribute is set to 'false'.
295   rule MTransition {
296      from
297          t : PathExp!Transition (
298            t.source <> t.target and
299            t.source.getOutT()->size() > 1
300          )
301      to
302         pt : TextualPathExp!PrimitiveTrans (
303            name <- '',
304            isMultiple <- false
305         )
306   }
307
308
309   -- Rule 'State'
310   -- This rule generates both an AlternativeTransition and the different Paths
311   -- that compose that compose this alternative transition from an input State
```

```
312    -- that has multiple non-loop outgoing Transitions.
313    -- Paths of the generated AlternativeTransition are those that are generated
314    -- by the rule. Its 'isMultiple' attribute is set to 'false'.
315    -- A distinct Path is generated for each non-loop outgoing Transition of the
316    -- input State. The sequence of transitions that is assigned to a generated
317    -- Path is the corresponding (ie. at same rank) sequence of model elements in
318    -- the 'transitions2' sequence (calculated in the using clause).
319    rule State {
320       from
321            s : PathExp!State (
322             s.getOutT()->size() > 1
323            )
324       using {
325          transitions2 : Sequence(String) =
326             s.getOutT()->collect(e | e.getTransitionsFromTrans());
327       }
328       to
329          at : TextualPathExp!AlternativeTrans (
330             altPaths <- paths,
331             isMultiple <- false
332          ),
333
334          paths : distinct TextualPathExp!Path foreach(e in transitions2) (
335             transitions <- transitions2
336          )
337    }
```

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.