1. Overview

This document provides a complete overall picture of bridging between Grafset, Petri Net and PNML. It also 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 Grafset and in the opposite way.

So this document describes how bridges between Grafset, Petri Net and PNML have been built, using a model transformation language ATL. This construction is composed of five steps:

- Grafset Models conforming to its metamodel are injected from a textual definition of the grafset by means of a TCS (Textual Concrete Syntax) program (this part is out of the scope of the document).

- A transformation from Grafset in their Petri Net equivalent and inverse: the Grafset – Petri Net Bridge.

- A transformation from Petri Net generated with Grafset in their PNML equivalent and inverse: the Petri Net - PNML Bridge.

- A transformation from PNML generated with Petri Net in their XML equivalent and inverse: the PNML - XML Bridge.

- As a final step, the XML model is extracted to the textual XML representation using an ATL query.

The next sections will explain the different steps to realize these bridges between Grafset, Petri Net, PNML and XML. Section 2 presents all metamodels; Section 3 explains all bridges and their transformations.
2. Metamodels

2.1. Grafcet

2.1.1. Generalities about Grafcet

Grafcet is a mainly French-based representation support for discrete system. It is a mode of representation and analysis of an automatism, particularly adapted to sequential systems with evolution, i.e. decomposable in steps. The Grafcet’s name came from «graph» because this model had a graphic basis, and AFCET (Association française de cybernétique économique et technique) from the scientific association which supported it. The Grafcet represents graphically the operation of an automatism by: steps with associated action, transitions between steps, and directed connections between the steps and the transitions.

2.1.2. A simplified metamodel of Grafcet

Figure 2 - Grafcet metamodel
Description of this metamodel:

- “Grafcet”: the main or root element which represents a grafcet,
- It is composed of elements and connections which are abstract classes,
- Elements are “Step” or “Transition”,
- Connections are “StepToTransition” or “TransitionToStep”,
- Steps and transitions can have many incoming or outgoing connections.

2.2. Petri Net

2.2.1. Generalities about Petri Net

Petri nets are also known as a place/transition net or P/T net. Defined in 1962 by Carl Adam Petri, they extend state machines with a notion of concurrency. It is a graphical and mathematical representation of discrete distributed systems. Petri nets consist of places, transitions and directed arcs that connect them, so arcs run between places and transitions, not between places and places or transitions and transitions. There are two sorts of arcs connecting place to transition or transition to place.
2.2.2. A simplified metamodel of Petri Net

Description of the basic metamodel:

- "PetriNet": the main or root element which represent a Petri net,
- It is composed of elements and arcs which are abstract class,
- Elements are "Place" or "Transition",
- Arcs are "PlaceToTransition" or "TransitionToPlace",
- Places and transitions can have many incoming or outgoing arcs.

2.3. PNML

2.3.1. Generalities about PNML

The Petri Net Markup Language (PNML) is a proposal of an XML-based interchange format for Petri nets (see [1]). Originally, it was intended to serve as a file format for the Java version of the Petri Net Kernel. PNML is a concept for defining the overall structure of a Petri net file.
2.3.2. A simplified metamodel of PNML

Figure 4 - PNML metamodel
Description of the simplified metamodel:

- “PNMLDocument”: the main or root element which contains Petri nets.
- “NetElement” represents the Petri net; it is composed of “NetContent” which are “Arc”, “Place” and “Transition”.
- Arcs reference a source and a target (“Place” or “Transition”) but the two kinds of arcs are not differentiated in this model (PlaceToTransition and TransitionToPlace).
- Net elements and net contents can have a name which is a labelled element composed of labels.

2.4. XML

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 V.

```
Node
  +startLine[0..1] : Integer
  +startColumn[0..1] : Integer
  +endLine[0..1] : Integer
  +endColumn[0..1] : Integer
  +name[1] : String
  +value[1] : String

Element

Attribute

Text

Root
```

Figure 5 - XML metamodel
Description of the basic 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.
3. Bridges

3.1. The Grafcet – Petri Net Bridge

3.1.1. Grafcet to Petri Net Transformation

In order to realize the bridge, and as there is no markup language for Grafcet, we need a textual input file. So, the Grafcet model is imported to a textual representation by means of a TCS (Textual Concrete Syntax) program. This part is not documented in this document.

3.1.1.1. Description of the Transformation

This transformation takes a Grafcet model conforming to our Grafcet metamodel and maps all Grafcet’s features to Petri Net. In fact the two metamodels of Grafcet and Petri Net are very close. So this transformation is quite easy. The ATL code for this transformation consists of 5 rules and no helpers.

Rules:

- The PetriNet rule generates a PetriNet element from the input Grafcet element. The name of the generated PetriNet element is copied from the one of the input Grafcet. Its set of Elements corresponds to Elements generated by Place and Transition rules. And its set of Arcs corresponds to Connections generated by PlaceToTransition and TransitionToPlace rules.

- The Place rule generates a Place element from the input Step element. The name of the generated Place element is copied from the one of the input Step. Its set of incomingArcs corresponds to incomingConnections generated by TransitionToPlace rule. And its set of outgoingArc corresponds to outgoingConnections generated by PlaceToTransition rule.

- The Transition rule generates a Transition element from the input Transition element. The name of the generated Transition element is copied from the one of the input Transition. Its set of incomingArcs corresponds to incomingConnections generated by PlaceToTransition rule. And its set of outgoingArc corresponds to outgoingConnections generated by TransitionToPlace rule.

- The PlaceToTransition rule generates a PlaceToTransition element from the input StepToTransition element. The name of the generated PlaceToTransition element is copied from the one of the input StepToTransition. Its from and to references are also copied from the ones of the input StepToTransition.

- The TransitionToPlace rule generates a TransitionToPlace element from the input TransitionToStep element. The name of the generated TransitionToPlace element is copied from the one of the input TransitionToStep. Its from and to references are also copied from the ones of the input TransitionToStep.
### 3.1.1.2. ATL Code

``` ATL
module Grafcet2PetriNet;
create OUT : PetriNet from IN : Grafcet;

-- The PetriNet rule generates a PetriNet element from the input Grafcet
element.
-- Name of the generated PetriNet element is copied from the one of the
input Grafcet.
-- Its set of Elements corresponds to Elements generated by Place and
Transition rules.
-- And its set of Arcs corresponds to Connections generated by
PlaceToTransition and TransitionToPlace rules.
rule PetriNet {
  from
    g : Grafcet!Grafcet
  to
    p : PetriNet!PetriNet
  {
    location <- g.location,
    name <- g.name,
    elements <- g.elements,
    arcs <- g.connections
  }
}

-- The Place rule generates a Place element from the input Step element.
-- Name of the generated Place element is copied from the one of the input
Step.
-- Its set of incomingArcs corresponds to incomingConnections generated by
TransitionToPlace rule.
-- And its set of outgoingArc corresponds to outgoingConnections generated
by PlaceToTransition rule.
rule Place {
  from
    g : Grafcet!Step
  to
    p : PetriNet!Place
  {
    location <- g.location,
    name <- g.name,
    net <- g.grafcet,
    incomingArc <- g.incomingConnections,
    outgoingArc <- g.outgoingConnections
  }
}

-- The Transition rule generates a Transition element from the input
Transition element.
-- Name of the generated Transition element is copied from the one of the
input Transition.
-- Its set of incomingArcs corresponds to incomingConnections generated by
PlaceToTransition rule.
```
-- And its set of outgoingArc corresponds to outgoingConnections generated
by TransitionToPlace rule.

rule Transition {
    from
    g : Grafcet!Transition
to
    p : PetriNet!Transition
    {
        location <- g.location,
        name <- g.name,
        net <- g.grafcet,
        incomingArc <- g.incomingConnections,
        outgoingArc <- g.outgoingConnections
    }
}

-- The PlaceToTransition rule generates a PlaceToTransition element from
the input StepToTransition element.
-- Name of the generated PlaceToTransition element is copied from the one
of the input StepToTransition.
-- Its from and to references are also copied from the ones of the input
StepToTransition.

rule PlaceToTransition {
    from
    g : Grafcet!StepToTransition
to
    p : PetriNet!PlaceToTransition
    {
        location <- g.location,
        name <- g.name,
        net <- g.grafcet,
        "from" <- g."from",
        "to" <- g."to"
    }
}

-- The TransitionToPlace rule generates a TransitionToPlace element from
the input TransitionToStep element.
-- Name of the generated TransitionToPlace element is copied from the one
of the input TransitionToStep.
-- Its from and to references are also copied from the ones of the input
TransitionToStep.

rule TransitionToPlace {
    from
    g : Grafcet!TransitionToStep
to
    p : PetriNet!TransitionToPlace
    {
        location <- g.location,
        name <- g.name,
        net <- g.grafcet,
        "from" <- g."from",
        "to" <- g."to"
    }
}
3.1.1.3. Configuration of the Transformation

As illustrated by the transformation configuration’s Figure 6 and Figure 7, there is one input metamodel (Grafcet) and one output (Petri Net). In Path Editor, place in “Grafcet” the path of the Grafcet metamodel; do the same for “PetriNet”. In “IN” place the path of an Ecore file (a model conforming to our Grafcet metamodel in Ecore format), and in “OUT” the path for the results (the generated file is an Ecore file conforming to the Petri Net metamodel).

Figure 6 - Grafcet to Petri Net configuration - part one
3.1.2. Petri Net to Grafcet Transformation

3.1.2.1. Description of the Transformation

As two metamodels of Grafcet and Petri Net are very close, this transformation is very similar to the previous one. The ATL code for the Petri Net to Grafcet transformation also consists of 5 rules and no helpers. All the rules are identical, only the input elements became output elements and in the reverse way.
Rules:

- The **GrafCet** rule generates a GrafCet element from the input Petri Net element. The name of the generated GrafCet element is copied from the one of the input Petri Net. Its set of Elements corresponds to Elements generated by Step and Transition rules. And its set of Connections corresponds to Arcs generated by StepToTransition and TransitionToStep rules.

- The **Step** rule generates a Step element from the input Place element. The name of the generated Step element is copied from the one of the input Place. Its set of incomingConnections corresponds to incomingArcs generated by TransitionToStep rule. And its set of outgoingConnections corresponds to outgoingArc generated by StepToTransition rule.

- The **Transition** rule generates a Transition element from the input Transition element. The name of the generated Transition element is copied from the one of the input Transition. Its set of incomingConnections corresponds to incomingArcs generated by StepToTransition rule. And its set of outgoingConnections corresponds to outgoingArc generated by TransitionToStep rule.

- The **StepToTransition** rule generates a StepToTransition element from the input PlaceToTransition element. The name of the generated StepToTransition element is copied from the one of the input PlaceToTransition. Its from and to references are also copied from the ones of the input PlaceToTransition.

- The **TransitionToStep** rule generates a TransitionToStep element from the input TransitionToPlace element. The name of the generated TransitionToStep element is copied from the one of the input TransitionToPlace. Its from and to references are also copied from the ones of the input TransitionToPlace.

### 3.1.2.2. ATL Code

```plaintext
module PetriNet2GrafCet;
create OUT : GrafCet from IN : PetriNet;

-- The GrafCet rule generates a GrafCet element from the input Petri Net element.
-- Name of the generated GrafCet element is copied from the one of the input Petri Net.
-- Its set of Elements corresponds to Elements generated by Step and Transition rules.
-- And its set of Connections corresponds to Arcs generated by StepToTransition and TransitionToStep rules.
rule GrafCet {
    from
        p : PetriNet!PetriNet
    to g : GrafCet!GrafCet
    {
        location <- p.location,
        name <- p.name,
        elements <- p.elements,
        connections <- p.arcs
    }
}
```
### ATL Transformation Example

Bridging Grafcet, Petri net, PNML and XML.

Contributor
Pierrick Guyard
pielepsy@gmail.com

Date 08/08/2005

---

#### Rule: Step

```xml
rule Step {
  from
    p : PetriNet!Place
to
    g : Grafcet!Step
  {
    location <- p.location,
    name <- p.name,
    grafcet <- p.net,
    isInitial <- false,
    isActive <- false,
    incomingConnections <- p.incomingArc,
    outgoingConnections <- p.outgoingArc
  }
}
```

---

#### Rule: Transition

```xml
rule Transition {
  from
    p : PetriNet!Transition
to
    g : Grafcet!Transition
  {
    location <- p.location,
    name <- p.name,
    grafcet <- p.net,
    incomingConnections <- p.incomingArc,
    outgoingConnections <- p.outgoingArc
  }
}
```

---

#### Rule: StepToTransition

```xml
rule StepToTransition {
  from
    p : PetriNet!PlaceToTransition
to
    g : Grafcet!StepToTransition
  {
    location <- p.location,
    name <- p.name,
    grafcet <- p.net,
    incomingConnections <- p.incomingArc,
    outgoingConnections <- p.outgoingArc
  }
}
```
from p : PetriNet!PlaceToTransition
to g : Grafcet!StepToTransition {
    location <- p.location,
    name <- p.name,
    grafcet <- p.net,
    "from" <- p."from",
    "to" <- p."to"
}

-- The TransitionToStep rule generates a TransitionToStep element from the
input TransitionToPlace element.
-- Name of the generated TransitionToStep element is copied from the one of
the input TransitionToPlace.
-- Its from and to references are also copied from the ones of the input
TransitionToPlace.
rule TransitionToStep {
    from p : PetriNet!TransitionToPlace
to g : Grafcet!TransitionToStep {
    location <- p.location,
    name <- p.name,
    grafcet <- p.net,
    "from" <- p."from",
    "to" <- p."to"
}
}
3.1.2.3. Configuration of the Transformation

As illustrated by the transformation configuration’s Figure 8 and Figure 9, there is one input metamodel (Petri Net) and one output (Grafcet). In Path Editor, place in “PetriNet” the path of the Petri net metamodel; do the same for “Grafcet”. In “IN” place the path of an Ecore file (a model conforming to our Petri net metamodel in Ecore format), and in “OUT” the path for the results (the generated file is an Ecore file conforming to the Grafcet metamodel).

![Figure 8 - Petri Net to Grafcet configuration - part one](image)
3.2. The Petri Net - PNML Bridge

3.2.1. Petri Net to PNML Transformation

3.2.1.1. Description of the Transformation

The ATL code for the Grafcet to Petri Net transformation consists of 4 rules and no helpers. In fact the two metamodels of Grafcet and Petri Net are quite close. So this transformation is quite easy.

Rules:

- The **PNMLDocument** rule generates a PNMLDocument and the NetElement which corresponds to the input PetriNet element. The name of the generated NetElement is copied from the one of the input PetriNet, by creating a PNML Name composed of a PNML Label which value is initialized by the PetriNet name. Its set of Contents corresponds to the union of the PetriNet Elements and Arcs.

- The **Place** rule generates a Place corresponds to the input PetriNet Place element. The name of the generated Place is copied from the one of the input Place, by creating a PNML Name composed of a PNML Label which value is initialized by the PetriNet Place name.

- The **Transition** rule generates a Transition corresponds to the input PetriNet Transition element. The name of the generated Transition is copied from the one of the input Transition, by creating a PNML Name composed of a PNML Label which value is initialized by the PetriNet Transition name.
• The **Arc** rule generates a Arc corresponds to the input PetriNet Arc element (TransitionToPlace and PlaceToTransition). The name of the generated Arc is copied from the one of the input Arc, by creating a PNML Name composed of a PNML Label which value is initialized by the PetriNet Arc name. Its source and target references are also copied from the input Arc and correspond respectively to from and to references.

### 3.2.1.2. **ATL Code**

```atl
module PetriNet2PNML;
create OUT : PNML from IN : PetriNet;

-- The PNMLDocument rule generates a PNMLDocument and the NetElement which corresponds to the input PetriNet element.
-- Name of the generated NetElement is copied from the one of the input PetriNet, by creating a PNML Name composed of a PNML Label which value is initialized by the PetriNet name.
-- Its set of Contents corresponds to the union of the PetriNet Elements and Arcs.
rule PNMLDocument {
  from
e : PetriNet!PetriNet
to
  n : PNML!PNMLDocument
  {
    location <- e.location,
    xmlns <- uri,
    nets <- net
  },
  uri : PNML!URI
  {
    value <- 'http://www.informatik.hu-berlin.de/top/pnml/ptNetb'
  },
  net : PNML!NetElement
  {
    name <- name,
    location <- e.location,
    id <- e.location,
    type <- type_uri,
    contents <- e.elements.union(e.arcs)
  },
  name : PNML!Name
  {
    labels <- label
  },
  label : PNML!Label
  {
    text <- e.name
  },
  type_uri : PNML!URI
  {
    value <- 'http://www.informatik.hu-berlin.de/top/pntd/ptNetb'
  }
}
```
-- The Place rule generates a Place corresponds to the input PetriNet Place element.
-- Name of the generated Place is copied from the one of the input Place, by creating a PNML Name composed of a PNML Label which value is initialized by the PetriNet Place name.
rule Place {
    from e : PetriNet!Place
to n : PNML!Place
    {
        name <- e.name,
id <- e.name,
        location <- e.location,
    },
    name : PNML!Name
    {
        labels <- label
    },
    label : PNML!Label
    {
        text <- e.name
    }
}

-- The Transition rule generates a Transition corresponds to the input PetriNet Transition element.
-- Name of the generated Transition is copied from the one of the input Transition, by creating a PNML Name composed of a PNML Label which value is initialized by the PetriNet Transition name.
rule Transition {
    from e : PetriNet!Transition
to n : PNML!Transition
    {
        name <- e.name,
id <- e.name,
        location <- e.location,
    },
    name : PNML!Name
    {
        labels <- label
    },
    label : PNML!Label
    {
        text <- e.name
    }
}

-- The Arc rule generates a Arc corresponds to the input PetriNet Arc element (TransitionToPlace and PlaceToTransition).
-- Name of the generated Arc is copied from the one of the input Arc, by
creating a PNML Name composed of a PNML Label which value is initialized by
the PetriNet Arc name.
-- Its source and target references are also copied from the input Arc and
correspond respectively to the from and to references.

rule Arc {
  from
    e : PetriNet!Arc
to
    n : PNML!Arc
  {
    name <- name,
    location <- e.location,
    id <- e.name,
    source <- e."from",
    target <- e."to"
  },
  name : PNML!Name
  {
    labels <- label
  },
  label : PNML!Label
  {
    text <- e.name
  }
}
3.2.1.3. Configuration of the Transformation

As illustrated by the transformation configuration’s Figure 10 and Figure 11, there is one input metamodel (Petri Net) and one output (PNML). In Path Editor, place in “PetriNet” the path of the Petri net metamodel; do the same for “PNML”. In “IN” place the path of an Ecore file (a model conforming to our Petri net metamodel in Ecore format), and in “OUT” the path for the results (the generated file is an Ecore file conforming to the PNML metamodel).

![Figure 10 - Petri Net to PNML configuration - part one](image-url)
3.2.2. PNML to Petri Net Transformation

3.2.2.1. Description of the Transformation

The ATL code for the Grafcet to Petri Net transformation consists of 5 rules and no helpers. In fact the two metamodels of Grafcet and Petri Net are quite close. So this transformation is quite easy.

Rules:

- The PetriNet rule generates a PetriNet which corresponds to the input NetElement included in the PNMLDocument. The name of the generated PetriNet is copied from the one of the input NetElement, by recovering the value of the PNML Name of the NetElement. Its set of Elements is the corresponding set named “elementsSet” calculated in the using clause. And its set of Arcs is the corresponding set named “arcsSet” calculated in the using clause.

- The Place rule generates a Place which corresponds to the input Place. The name of the generated Place is copied from the one of the input Place, by recovering the value of the PNML Name of the PNML Place.

- The Transition rule generates a Transition which corresponds to the input Transition. The name of the generated Transition is copied from the one of the input Transition, by recovering the value of the PNML Name of the PNML Transition.

- The PlaceToTransition rule generates a PlaceToTransition which corresponds to the input Arc which has a Place for source and a Transition for Target. The name of the generated PlaceToTransition is copied from the one of the input Arc, by recovering the value of the
PNML Label included in the PNML Name of the PNML Arc. Its from and to references are also copied from the input Arc and correspond respectively to the source and target references.

- The TransitionToPlace rule generates a TransitionToPlace which corresponds to the input Arc which has a Transition for source and a Place for Target. The name of the generated TransitionToPlace is copied from the one of the input Arc, by recovering the value of the PNML Label included in the PNML Name of the PNML Arc. Its from and to references are also copied from the input Arc and correspond respectively to the source and target references.

### 3.2.2.2. ATL Code

``` ATL
module PNML2PetriNet;
create OUT : PetriNet from IN : PNML;

-- The PetriNet rule generates a PetriNet which corresponds to the input NetElement included in the PNMLDocument.
-- Name of the generated PetriNet is copied from the one of the input NetElement, by recovering the value of the PNML Label included in the PNML Name of the NetElement.
-- Its set of Elements is the corresponding set named “elementsSet” calculated in the using clause.
-- And its set of Arcs is the corresponding set named “arcsSet” calculated in the using clause.
rule PetriNet {
  from
  n : PNML!PNMLDocument
  using{
    elementsSet : Set(PetriNet!Element) =
      PNML!NetContentElement.allInstances();
    arcsSet : Set(PetriNet!Arc) =
      PNML!Arc.allInstances();
  }
  to
  p : PetriNet!PetriNet
  {
    location <- n.location,
    name <- n.nets.first().name.labels.first().text,
    elements <- elementsSet,
    arcs <- arcsSet
  }
}
```

-- The Place rule generates a Place which corresponds to the input Place.
-- Name of the generated Place is copied from the one of the input Place, by recovering the value of the PNML Label included in the PNML Name of the PNML Place.
```
rule Place {
  from
  n : PNML!Place
  to
  p : PetriNet!Place
  {
    location <- n.location,
  }
```
name <- n.name.labels.first().text,
net <- n.net.document
}

-- The Transition rule generates a Transition which corresponds to the
input Transition.
-- Name of the generated Transition is copied from the one of the input
Transition, by recovering the value of the PNML Label included in the PNML
Name of the PNML Transition.

rule Transition {
    from
    n : PNML!Transition
to
    p : PetriNet!Transition
{
    location <- n.location,
    name <- n.name.labels.first().text,
    net <- n.net.document
}

-- The PlaceToTransition rule generates a PlaceToTransition which
corresponds to the input Arc which has a Place for source and a Transition
for Target.
-- Name of the generated PlaceToTransition is copied from the one of the
input Arc, by recovering the value of the PNML Label included in the PNML
Name of the PNML Arc.
-- Its from and to references are also copied from the input Arc and
correspond respectively to the source and target references.

rule PlaceToTransition {
    from
    n : PNML!Arc
    { -- arc source must be a place and arc target a transition
        n.source.oclIsKindOf(PNML!Place) and
        n.target.oclIsKindOf(PNML!Transition)
    }
to
    p : PetriNet!PlaceToTransition
    {
        location <- n.location,
        name <- n.name.labels.first().text,
        net <- n.net.document,
        "from" <- n.source,
        "to" <- n.target
    }

-- The TransitionToPlace rule generates a TransitionToPlace which
corresponds to the input Arc which has a Transition for source and a Place
for Target.
-- Name of the generated TransitionToPlace is copied from the one of the
input Arc, by recovering the value of the PNML Label included in the PNML
Name of the PNML Arc.
rule TransitionToPlace {
  from n : PNML!Arc
    ( -- arc source must be a transition and arc target a place
       n.source.oclIsKindOf(PNML!Transition) and
       n.target.oclIsKindOf(PNML!Place)
     )
  to p : PetriNet!TransitionToPlace
    {
      location <- n.location,
      name <- n.name.labels.first().text,
      net <- n.net.document,
      "from" <- n.source,
      "to" <- n.target
    }
}
3.2.2.3. Configuration of the Transformation

As illustrated by the transformation configuration’s Figure 12 and Figure 13, there is one input metamodel (PNML) and one output (PetriNet). In Path Editor, place in “PetriNet” the path of the Petri net metamodel; do the same for “PNML”. In “IN” place the path of an Ecore file (a model conforming to our PNML metamodel in Ecore format), and in “OUT” the path for the results (the generated file is an Ecore file conforming to the Petri net metamodel).

Figure 12 - PNML to Petri Net configuration - part one
3.3. The PNML - XML Bridge

3.3.1. PNML to XML Transformation: Extractor

3.3.1.1. Description of the Transformation

The ATL code for the Grafcet to Petri Net transformation consists of 4 rules and 1 helper.

Helper:

- The `getRoot` helper is a constant helper. It seeks the root element of PNML model: the PNML document. This helper allows to link elements and their parents, thanks to a “resolveTemp” instruction.

Rules:

- The `Root` rule generates the XML Root element as well as a collection of attributes and elements and Text node from the input PNMLDocument element. The generated Root element is a “pnml” tag that has an “xmlns” Attribute and a “net” Element as children. The value of the “xmlns” attribute is copied from the PNMLDocument. The “net” Element has an “id” and a “type” Attribute, a “name” sub-Element. The “id” attribute and the “type” attribute are also copied from the input element. Finally, the “name” Element contains a “text” Element, which itself contains a Text node whose value corresponds to the name of the input PNMLDocument element.

- The `Place` rule generates three XML Elements, one XML Attribute and one XML Text for each PNML Place input element. The first generated Element, “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 one of the PNML Place. 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 **Transition** rule generates three XML Elements, one XML Attribute and one XML Text for each PNML Transition input element. The first generated Element, “transition”, is a “transition” tag which accepts an “id” Attribute as well as a child “name” Element. The value of the “id” attribute corresponds to the one of the PNML Transition. 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 Transition.

- The **Arc** rule generates three XML Elements, three XML Attributes and one XML Text for each PNML Arc input element. The generated Element is an “arc” tag that has three Attribute children: “id”, “source” and “target”, as well as a child “name” Element. The value of the “id” attribute corresponds to the one of the PNML Arc. Values of the “source” and “target” attributes respectively correspond to the id of the source and the id of the target of the input Arc. 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 Transition.

### 3.3.1.2. ATL Code

```atl
module PNML2XML;
create OUT : XML from IN : PNML;

-- The getRoot helper, is a constant helper. It seeks the root element of PNML model : the PNML document.
-- This helper allows to link elements and their parents, thanks to a “resolveTemp” instruction and to the helper.
-- CONTEXT: n/a
-- RETURN: PNML!PNMLDocument
helper def: getRoot() : PNML!PNMLDocument =
  PNML!PNMLDocument.allInstances()->asSequence()->first();

-- The Root rule generates the XML Root element as well as a collection of attributes and elements from the input PNMLDocument 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 copied from the PNMLDocument. The “net” Element has an “id” and a “type” Attribute, a “name” sub-Element. The “id” attribute and the “type” attribute are also copied from the input element.
-- Finally, the “name” Element contains a “text” Element, which itself contains a Text node whose value corresponds to the name of the input PNMLDocument element.
rule Root {
  from
    n : PNML!PNMLDocument
to
  e : XML!Root
  
```
name <- 'pnml',
-- value = name of the net contained by this document
value <- n.nets.first().name.labels.first().text,
children <- Sequence (document_name, document_xmlns, document_net)
},
document_name : XML!Element
{
  name <- 'name',
  parent <- n,
  children <- document_text
},
document_text : XML!Element
{
  name <- 'text',
  parent <- document_name,
  children <- document_xml_text
},
document_xml_text : XML!Text
{
  value <- n.nets.first().name.labels.first().text,
  parent <- document_text
},
document_xmlns : XML!Attribute
{
  name <- 'xmlns',
  value <- n.xmlns.value,
  parent <- n
},
document_net : XML!Element
{
  name <- 'net',
  value <- n.nets.first().name.labels.first().text,
  parent <- n,
  children <- Sequence (net_name, net_id, net_type)
},
net_name : XML!Element
{
  name <- 'name',
  parent <- document_net,
  children <- net_text
},
net_text : XML!Element
{
  name <- 'text',
  parent <- net_name,
  children <- net_xml_text
},
net_xml_text : XML!Text
{
  value <- n.nets.first().name.labels.first().text,
  parent <- net_text
},
net_id : XML!Attribute
{
name <- 'id',
value <- n.nets.first().id,
parent <- document_net
},
net_type : XML!Attribute
{
    name <- 'type',
value <- n.nets.first().type.value,
parent <- document_net
}
}

-- The Place rule generates three XML Elements, one XML Attribute and one XML Text for each PNML Place input element.
-- The first generated Element, “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 one of the PNML Place.
-- 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.
rule Place {
    from
    n : PNML!NetContentElement
    { n.oclIsKindOf(PNML!Place) }
    to
    place : XML!Element
    {
        name <- 'place',
        value <- n.name.labels.first().text,
        parent <- thisModule.resolveTemp(thisModule.getRoot(),
            'document_net'),
        children <- Sequence{place_id, place_name}
    },
    place_id : XML!Attribute
    {
        name <- 'id',
        value <- n.id,
        parent <- n
    },
    place_name : XML!Element
    {
        name <- 'name',
        parent <- n,
        children <- place_text
    },
    place_text : XML!Element
    {
        name <- 'text',
        parent <- place_name,
        children <- place_xml_text
    },
    place_xml_text : XML!Text
value <- n.name.labels.first().text,
        parent <- place_text
    }
}

-- The Transition rule generates three XML Elements, one XML Attribute and
one XML Text for each PNML Transition input element.

-- The first generated Element, “transition”, is a “transition” tag which
accepts an “id” Attribute as well as a child “name” Element. The value of
the “id” attribute corresponds to the one of the PNML Transition.

-- 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 Transition.

rule Transition {
    from
        n : PNML!NetContentElement
    to
        transition : XML!Element
            name <- 'transition',
            value <- n.name.labels.first().text,
            parent <- thisModule.resolveTemp(thisModule.getRoot(),
                'document_net'),
            children <- Sequence{transition_id, transition_name}
    },
    transition_id : XML!Attribute
        name <- 'id',
        value <- n.id,
        parent <- n
    },
    transition_name : XML!Element
        name <- 'name',
        parent <- n,
        children <- transition_text
    },
    transition_text : XML!Element
        name <- 'text',
        parent <- transition_name,
        children <- transition_xml_text
    },
    transition_xml_text : XML!Text
        value <- n.name.labels.first().text,
        parent <- transition_text
    }
}
The Arc rule generates three XML Elements, three XML Attributes and one XML Text for each PNML Arc input element. The generated Element is an "arc" tag that has three Attribute children: "id", "source" and "target", as well as a child "name" Element. The value of the "id" attribute corresponds to the one of the PNML Arc. Values of the "source" and "target" attributes respectively correspond to the id of the source and the id of the target of the input Arc. 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 Transition.

```
rule Arc {
    from
    n : PNML!Arc
to
    arc : XML!Element {
        name <- 'arc',
        value <- n.name.labels.first().text,
        parent <- thisModule.resolveTemp(thisModule.getRoot(),
        'document_net'),
        children <- Sequence {arc_name, arc_id, source, target}
    },
    arc_id : XML!Attribute {
        name <- 'id',
        value <- n.id,
        parent <- n
    },
    arc_name : XML!Element {
        name <- 'name',
        parent <- n,
        children <- arc_text
    },
    arc_text : XML!Element {
        name <- 'text',
        parent <- arc_name,
        children <- arc_xml_text
    },
    arc_xml_text : XML!Text {
        value <- n.name.labels.first().text,
        parent <- arc_text
    },
    -- source and target attribute are initialised by the id of the element pointed
    source : XML!Attribute {
        name <- 'source',
        value <- n.source.id,
        parent <- n
    }
}
```
247 },
248 target : XML!Attribute
249 {
250     name <- 'target',
251     value <- n.target.id,
252     parent <- n
253 }
254 }
3.3.1.3. Configuration of the Transformation

As illustrated by the transformation configuration’s Figure 14 and Figure 15, there is one input metamodel (PNML) and one output (XML). In Path Editor, place in “PNML” the path of the PNML metamodel; do the same for “XML”. In “IN” place the path of an Ecore file (a model conforming to our PNML metamodel in Ecore format), and in “OUT” the path for the results (the generated file is an Ecore file conforming to the XML metamodel).

Figure 14 - PNML to XML configuration - part one
3.3.2. XML to PNML Transformation: Injector

3.3.2.1. Description of the Transformation

The ATL code for the Grafcet to Petri Net transformation consists of 5 rules and 3 helpers.

Helpers:

- The first helper `getAttribute`, returns the value of an attribute (identified by its name, passed as a parameter) of the contextual XML Element. For this purpose, it 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 `getLink` helper collects all instances of xml element and search the one whose id matches the id passed in parameter. The helper returns the first xml element of the collection.
Rules:

- **The PNMLDocument** rule generates a PNMLDocument from the input XML Root Element.

- **The Net** rule generates a NetElement from each “net” XML Element input element. The name of the generated NetElement is computed by calling the `getName` helper. Its set of Places, Transitions and Arcs are initialized by the other rules. The link to its parent, the PNMLDocument, is also created.

- **The Place** rule generates a PNML Place for each “place” XML Element. The name of the generated Place is computed by a call to the `getName` helper. Its id is copied from the one of the input XML Element. The link to its parent, the NetElement, is also created.

- **The Transition** rule generates a PNML Transition for each “transition” XML Element. The name of the generated Transition is computed by a call to the `getName` helper. Its id is copied from the one of the input XML Element. The link to its parent, the NetElement, is also created.

- **The Arc** rule generates a PNML Arc for each “arc” XML Element. The name of the generated Arc is computed by a call to the `getName` helper. Its id is copied from the one of the input XML Element. Its source (obtained by means of the `getLink` helper) corresponds to the XML Element which id is contained in the child attribute named “source”. Idem for the target. The link to its parent, the NetElement, is also created.

### 3.3.2.2. ATL Code

```plaintext
module XML2PNML;
create OUT : PNML from IN : XML;

-- The getAttrVal 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.
-- CONTEXT: XML!Element
-- RETURN: String
helper context XML!Element def: getAttrVal(name : String) : String =
  let a : Sequence(XML!Attribute) = self.children->select(c |
  c.oclIsTypeOf(XML!Attribute) and c.name = name) in
  if a.isEmpty() then
    ''
  else
    a.first().value
  endif;

-- 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.
-- CONTEXT: XML!Element
-- RETURN: String
helper context XML!Element def : getName() : String =
```
self.children->select(c | c.oclIsTypeOf(XML!Element) and c.name = 'name')->first().children
    ->select(d | d.oclIsTypeOf(XML!Element) and d.name = 'text')-
>first().children
    ->select(e | e.oclIsKindOf(XML!Text))->first().value;

-- The getLink helper, collects all instances of xml element and search the
one whose id matches the id passed in parameter.
-- The helper returns the first xml element of the collection.
-- CONTEXT: n/a
-- RETURN: XML!Element

helper def: getLink(id : String) : XML!Element =
    XML!Element.allInstances()->select(z | z.getAttrVal('id') = id)->first();

-- The PNMLDocument rule generates a PNMLDocument from the input XML Root
Element.
rule PNMLDocument {
    from
        x : XML!Root
to
document : PNML!PNMLDocument	notation <- x.getAttrVal('xmlns'),
    uri : PNML!URI	notation.value <- x.getAttrVal('xmlns')
}

-- The Net rule generates a NetElement from each "net" XML Element input
element.
-- Name of the generated NetElement is computed by calling the getName
helper.
-- Its set of Places, Transitions and Arcs are initialized by the other
rules.
-- The link to its parent, the PNMLDocument, is also created.
rule Net {
    from
        x : XML!Element
to
    net_element : PNML!NetElement
        name <- named_element,
    type <- type_uri,
        -- pointer on the root element
document <- x.parent
    type_uri : PNML!URI
}
value <- x.getAttrVal('type')
},
named_element : PNML!Name
{
    labels <- label
},
label : PNML!Label
{
    text <- x.getName()
}
}

-- The Place rule generates a PNML Place for each “place” XML Element.
-- Name of the generated Place is computed by a call to the getName helper.
-- Its id is copied from the one of the input XML Element.
-- The link to its parent, the NetElement, is also created.

rule Place {
  from
  x : XML!Element
  {
    x.name = 'place'
  }
  to
  n : PNML!Place
  {
    name <- named_element,
    -- pointer on the net element
    net <- x.parent,
    id <- x.getAttrVal('id'),
    location <- ''
  },
  named_element : PNML!Name
  {
    labels <- label
  },
  label : PNML!Label
  {
    text <- x.getName()
  }
}

-- The Transition rule generates a PNML Transition for each “transition” XML Element.
-- Name of the generated Transition is computed by a call to the getName helper.
-- Its id is copied from the one of the input XML Element.
-- The link to its parent, the NetElement, is also created.

rule Transition {
  from
  x : XML!Element
  {
    x.name = 'transition'
  }
  to
n : PNML!Transition
{
  name <- named_element,
  -- pointer on the net element
  net <- x.parent,
  id <- x.getAttrVal('id')
},
named_element : PNML!Name
{
  labels <- label
},
label : PNML!Label
{
  text <- x.getName()
}
-- The Arc rule generates a PNML Arc for each "arc" XML Element.
-- Name of the generated Arc is computed by a call to the getName helper.
-- Its id is copied from the one of the input XML Element.
-- Its source (obtained by means of the getLink helper) corresponds to the
-- XML Element which id is contained in the child attribute named "source".
-- Idem for the target.
-- The link to its parent, the NetElement, is also created.
rule Arc {
  from
  x : XML!Element
  {
    x.name = 'arc'
  }
  to
  n : PNML!Arc
  {
    name <- named_element,
    id <- x.getAttrVal('id'),
    net <- x.parent,
    -- seek of the element pointed by the source id contained in the xml
    source <- thisModule.getLink(
      x.children->select(c | c.oclIsKindOf(XML!Attribute) and c.name = 'source')->&first().value)
    -- seek of the element pointed by the target id contained in the xml
    file
    target <- thisModule.getLink(
      x.children->select(c | c.oclIsKindOf(XML!Attribute) and c.name = 'target')->&first().value)
    },
    named_element : PNML!Name
    {
      labels <- label
    },
    ...
ATL
TRANSFORMATION EXAMPLE

Bridging Grafcet, Petri net, PNML and XML.

Contributor
Pierrick Guyard
pielepsy@gmail.com

Date 08/08/2005

191     label : PNML!Label
192     {
193         text <- x.getName()
194     }
195 }
196

Page 40/50
3.3.2.3. Configuration of the Transformation

As illustrated by the transformation configuration's Figure 16 and Figure 17, there is one input metamodel (XML) and one output (PNML). In Path Editor, place in “XML” the path of the XML metamodel; do the same for “PNML”. In “IN” place the path of an Ecore file (a model conforming to our XML metamodel in Ecore format), and in “OUT” the path for the results (the generated file is an Ecore file conforming to the PNML metamodel).

Figure 16 - XML to PNML configuration - part one
3.3.3. XML to PNML text (Extract XML)

3.3.3.1. Description of the Transformation

The ATL code, that allows generating a PNML valid and well-formed XML text file from an XML model, for this transformation consists in 4 helpers and 1 query.

The aim of this query is to extract each of the elements that compose the input XML model into an output XML file. Contrary to rules that are implemented to generate a model from another model, a query allows calculating output text files from an input model (see [3]). This is the reason why we need to use queries for this type of transformation: generating an XML file from an XML model.

The implemented query get the Root element of the XML model and call the “toString2()” helper on it. The content is generated by the “toString2()” helper called on the Root element of the XML model.

There are three “toString2()” helpers with different contexts. The XMLAttribute one simply returns the name and the value of an attribute in the correct string format. The XMLText one only returns the string value contained in a text node. The XMLElement one returns the valid and well-formed content of the output XML file by parsing recursively all the element of the input XML model. Note that it sometimes calls the XMLAttribute and XMLText “toString2()” helpers.

3.3.3.2. ATL Code

```plaintext
query XML2Text = XML!Root.allInstances()
->asSequence()
```
->first().toString2('').writeTo('C:\... Complete this path ...\Grafce-PetriNet\Models\XML2Text_example.xml');

helper context XML!Element def: toString2(indent : String) : String =
  let na : Sequence(XML!Node) =
    self.children->select(e | not e.oclIsKindOf(XML!Attribute))
  in
  let a : Sequence(XML!Node) =
    self.children->select(e | e.oclIsKindOf(XML!Attribute))
  in
  indent + '<' + self.name +
  a->iterate(e; acc : String = '' |
    acc + '' + e.toString2())
  ) +
  if na->size() > 0 then
    '>'
    + na->iterate(e; acc : String = '' |
      acc +
      if e.oclIsKindOf(XML!Text) then
        ''
      else
        '\r\n'
      endif
    + e.toString2(indent + ' ')
  ) +
  if na->first().oclIsKindOf(XML!Text) then
    '</' + self.name + '>'
  else
    '
' + indent + '</' + self.name + '>
  endif
else
  '/>'
endif;

helper context XML!Attribute def: toString2() : String =
  self.name + '='"" + self.value + '"';

helper context XML!Text def: toString2() : String =
  self.value;
3.3.3.3. Configuration of the Transformation

As illustrated by the transformation configuration's Figure 18 and Figure 19, there is one input metamodel (XML). In Path Editor, place in “XML” the path of the XML metamodel. In “IN” place the path of an Ecore file (a model conforming to our XML metamodel in Ecore format).

The generated file is an Ecore file conforming to the XML metamodel. This file does not appear in the configuration, it is defined in the ATL code of the transformation. So in the XML to Text ATL file, ensure that the output file path is correct at the top of the file (Figure 20).

Figure 18 - XML to Text configuration - part one
Bridging Grafcet, Petri net, PNML and XML.

Figure 19 - XML to Text configuration - part two

Figure 20 - XML to Text ATL file
I. Grafcet metamodel in KM3 format

package Grafcet {
    abstract class LocatedElement {
        attribute location : String;
    }
    abstract class NamedElement extends LocatedElement {
        attribute name : String;
    }
    class Grafcet extends NamedElement {
        reference elements[*] container : Element oppositeOf grafcet;
        reference connections[*] container : Connection oppositeOf grafcet;
    }
    -- @begin elements
    abstract class Element extends NamedElement {
        reference grafcet : Grafcet oppositeOf elements;
    }
    class Step extends Element {
        attribute isInitial : Boolean;
        attribute isActive : Boolean;
        attribute action : String;
        reference incomingConnections[*] : TransitionToStep oppositeOf to;
        reference outgoingConnections[*] : StepToTransition oppositeOf from;
    }
    class Transition extends Element {
        attribute condition : String;
        reference incomingConnections[*] : StepToTransition oppositeOf to;
        reference outgoingConnections[*] : TransitionToStep oppositeOf from;
    }
    -- @end elements
    -- @begin connections
    abstract class Connection extends NamedElement {
        reference grafcet : Grafcet oppositeOf connections;
    }
    class StepToTransition extends Connection {
        reference from : Step oppositeOf outgoingConnections;
        reference to : Transition oppositeOf incomingConnections;
    }
    class TransitionToStep extends Connection {
        reference from : Transition oppositeOf outgoingConnections;
        reference to : Step oppositeOf incomingConnections;
    }
    -- @end connections
}
package PrimitiveTypes {
    datatype String;
    datatype Boolean;
}
II. Petri Net metamodel in KM3 format

package PetriNet {
    abstract class LocatedElement {
        attribute location : String;
    }
    abstract class NamedElement extends LocatedElement {
        attribute name : String;
    }
    -- @comment top element
    class PetriNet extends NamedElement {
        reference elements[*] container : Element oppositeOf net;
        reference arcs[*] container : Arc oppositeOf net;
    }
    -- @begin elements
    abstract class Element extends NamedElement {
        reference net : PetriNet oppositeOf elements;
    }
    class Place extends Element {
        reference incomingArc[*] : TransitionToPlace oppositeOf to;
        reference outgoingArc[*] : PlaceToTransition oppositeOf from;
    }
    class Transition extends Element {
        reference incomingArc[1-*] : PlaceToTransition oppositeOf to;
        reference outgoingArc[1-*] : TransitionToPlace oppositeOf from;
    }
    -- @end elements
    --@begin arcs
    abstract class Arc extends NamedElement {
        attribute weight : Integer;
        reference net : PetriNet oppositeOf arcs;
    }
    class PlaceToTransition extends Arc {
        reference from : Place oppositeOf outgoingArc;
        reference to : Transition oppositeOf incomingArc;
    }
    class TransitionToPlace extends Arc {
        reference from : Transition oppositeOf outgoingArc;
        reference to : Place oppositeOf incomingArc;
    }
    --@end arcs
}
package PrimitiveTypes {
    datatype String;
    datatype Integer;
}
III. PNML metamodel in KM3 format

```java
package PNML {
    abstract class LocatedElement {
        attribute location : String;
    }
    abstract class IdedElement extends LocatedElement {
        attribute id : String;
    } -- @begin declaration of types
    class URI extends LocatedElement {
        attribute value : String;
    } -- @end declaration of types
    -- @begin comment single top element (like in XML document)
    class PNMLDocument extends LocatedElement {
        reference xmlns container : URI;
        reference nets[1-+] container : NetElement oppositeOf document;
    } -- @comment a petri net element
    class NetElement extends IdedElement {
        -- @comment type reference the PNTD associated with the net
        reference type container : URI;
        reference document : PNMLDocument oppositeOf nets;
        reference contents[*] container : NetContent oppositeOf net;
        reference name[0-1] container : Name oppositeOf net;
    } -- @comment content of a petri net element
    abstract class NetContent extends LocatedElement {
        reference net : NetElement oppositeOf contents;
        reference name[0-1] container : Name oppositeOf netContent;
    } -- @comment element used for abstraction (Name, Inscription and InitialMarking)
    abstract class LabeledElement extends LocatedElement {
        reference labels[*] container : Label oppositeOf labeledElement;
    } class Label extends LocatedElement {
        attribute text : String;
        reference labeledElement : LabeledElement oppositeOf labels;
    } class Name extends LabeledElement {
        reference net[0-1] : NetElement oppositeOf name;
        reference netContent[0-1] : NetContent oppositeOf name;
    } -- @comment element used for abstraction (Place and Transition)
    abstract class NetContentElement extends NetContent,IdedElement {
    } class Arc extends NetContent,IdedElement {
        reference source : NetContentElement;
        reference target : NetContentElement;
    } class Place extends NetContentElement {
    } -- @comment a transition element
    class Transition extends NetContentElement {
    }
}
package PrimitiveTypes { datatype String; }
```
IV. XML metamodel in KM3 format

```java
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;
}
```
V. References

