1. ATL Transformation Example

1.1. Example: KM3 \rightarrow DOT

The KM3 to DOT example describes a transformation from a KM3 metamodel description into a class diagram drawn with dot. KM3 \cite{1} is a textual concrete syntax to describe metamodels. It has its advantages, yet having a graphical presentation of a metamodel can be sometimes enlightening. Dot is an automatic graph layout program from Graphviz \cite{2}. It can be used to create graphical files, such as PS, PNG... out of its layout.

1.2. Transformation overview

The aim of this transformation is to generate a rough visualization, in the form of a class diagram, of a KM3 model. A metamodel created with KM3 does not include any representation information, so dot, the Graphviz tool, is used to compute the layout and generate the output picture. To do this, the KM3 file has to be injected as a model, and then transformed into a dot model, which is then serialized into a dot file.

![Transformation diagram](image)

Figure 1. Transformation from end to end

In Table 1, a small example is given of a KM3 syntax and the corresponding dot syntax, in order to have the graphical result.

<table>
<thead>
<tr>
<th>KM3 textual syntax</th>
<th>Dot textual syntax</th>
<th>Graphical result</th>
</tr>
</thead>
</table>
| package MyPackage { class MyClass { attribute name: String; reference foos[*] : Foo; } class Foo { attribute bar: String; } } | digraph MyPackage { graph[rankDir=BT] node[shape=record] MyClass[label={MyClass | name: String | }] Foo[label={Foo | bar: String | }] MyClass -> Foo [headlabel='foos[*]'] } | MyClass
|                    |                    | MyPackage
|                    |                    | Foo
|                    |                    | bar: String
|                    |                    | foos[*]

Table 1. Syntaxes example

1.3. Metamodels

This transformation is based on the KM3 metamodel, of which an incomplete representation is given in Figure 2. Yet the only missing classes are those not necessary to understand the concepts used with this metamodel.
The transformation also relies on the DOT metamodel. It was defined for this transformation, and does not support every options that dot does. The classes that are most useful for the transformation are presented in Figure 3.

Figure 2. Simplified KM3 metamodel

Figure 3. Simplified DOT metamodel

In a nutshell, this metamodel is one for graph description in general, with some attributes that are dot-specific.

1.4. Rules Specification

Here are the main rules to transform a KM3 model into a DOT graph:
The **Metamodel** element is transformed into a **Graph** element, which will contain the different **Subgraphs**.

The **Package** elements are transformed into **Subgraphs** with a surrounding black rectangle.

Each **Datatype** is transformed into a **Record node** with the label « Datatype » followed by its name.

For each **Class**, a **Record node** is created, with the proper label, and every generalization **Arc**.

Each **Reference** is transformed into an **Arc**, with the proper **Arrow shapes** and **Labels** (roles and multiplicities).

### 1.5. ATL Code

The ATL code for the KM3 to DOT transformation consists in 18 helpers (9 of them being parameter values) and 7 rules.

#### 1.5.1. Helpers

There are nine helpers that are set at fixed values. They are parameters for this transformation. If the user wants to have a left to right layout of his diagram, for example, he can set this in the **RankingDirection()** helper.

The other nine helpers are mostly for label building. That is collecting the name of the class, its attributes and operations, or the roles and multiplicities of relations. The only one that differs is **relationsList()**. It is used to compute the list of relations, and then to match the proper number of relations. A relation may be bidirectional. So it is referred in both classes, and could be matched twice, and created twice. This helper is designed to avoid this, and have one bidirectional arrow rather than two monodirectional ones.

#### 1.5.2. Rules

These helpers are used in the 7 rules of this transformation.

The **Metamodel2Graph** rule generates a dot graph element. It uses many of the parameters helpers, such as **rankingDirection()**, **labelJustification()**, **labelLocation()** or **minimumNodeSeparation()**. Some options are fixed, as the **compound** attribute (to allow edges between subgraphs), because their modification does not affect the visual representation of metamodels. The generated graph is set with these attributes, and will contain one cluster **subgraph per package**.

The **Package** rule creates one cluster **subgraph per package**. Its name starts with 'cluster_' so that it will be represented by a black rectangle surrounding its included **subgraphs** and **nodes**. Indeed, there can be nested **packages** in the KM3 model, and dot supports nested **subgraphs**.

The **Datatype** rule treats the special case of **datatype** element. It generates a **record node** with the proper **label**, using **getDataTypeHead()**.

The **ClassWithSupertypesGeneralizationsDrawn2Node** rule applies to KM3 **classes** for which there are generalization links to be created. This depends on the **DiagramMode()** and **DiagramType()** parameters, and on the existence of supertypes for the **class**. This rule creates the corresponding **arcs** with the correct arrow shape. To generate the correct **label** it calls the **getLabel()** helper.

The **ClassWithSupertypesGeneralizationsNotDrawn2Node** rule applies when the previous one does not. It generates the correct **record node** but without the generalization links.

The **Reference2OneWayArc** rule generates a one way (with the arrow shape) **arc** with the correct role and multiplicity as **label**.
The Reference2Arc rule creates all the other association arcs, with their multiplicities, roles, and possible composition diamond shapes.

```plaintext
module KM32DOT;
create OUT: DOT from IN: KM3;

-- Parameters

-- DiagramType
-- Specifies what kind of diagram shall be rendered: a Generalization one,
-- or one based on composition links, or a standard one.
-- Possible values: 'None' | 'Generalization' | 'Composition'
help def: DiagramType(): String = 'None';

-- Mode
-- Specifies what association type shall be used so as to compute the
-- layout of the diagram
-- Possible values: 'All' | 'Generalization' | 'Composition' | 'Reference'
help def: Mode(): String = 'All';

-- Invisible
-- Specifies whether the associations not used in the computation of the
-- layout have yet to be rendered. For example, if Invisible = true and
-- Mode = Generalization, the diagram will be rendered as if only
-- generalization links did matter. If Invisible = false, then only
-- generalization links are drawn, but the layout is the same as if
-- Possible values: true | false
help def: Invisible(): Boolean = false;

-- MinimumArcLength
-- Specifies the minimum length of an association, in inches.
-- Possible values: any positive integer value
help def: MinimumArcLength(): Integer = 2;

-- MinimumNodeSeparation
-- Specifies the minimum distance between two nodes, in inches.
-- Possible values: any positive real value
help def: MinimumNodeSeparation(): Real = 0.75;

-- RankingDirection
-- Specifies the direction in which the diagram should be rendered. Most
-- class diagrams are rendered with the value 'BT'
-- Possible values: 'BT' | 'TB' | 'LR' | 'RL' (Bottom to Top,
-- Top to Bottom, Left to Right, Right to Left)
help def: RankingDirection(): String = 'BT';

help def: LabelJustification(): String = 'l';
help def: LabelLocation(): String = 't';

-- DataTypeHeader
-- The name of a datatype should begin with <<DataType>>
help def: DataTypeHeader(): String = '\#171;\#171; DataTypes#187;\n';
```

-- End Parameters
-- HELPERS
-- DiagramMode
-- Returns whether the DiagramMode parameter is the one tried or not
--- IN: mode: String
--- OUT: Boolean
\textbf{helper def: }DiagramMode(mode: String): Boolean =
mode = thisModule.Mode();

-- getDataTypeHead
-- Returns the name of the datatype, with its header
-- IN: N/A
-- OUT: String
\textbf{helper context KM3!DataType \textbf{def: }getDataTypeHead(): String =
thisModule.DataTypeHeader() + self.name;

-- getLabel
-- Returns the correct SimpleLabel content for a KM3 Class:
-- Name | Attributes | Operations
-- IN: N/A
-- OUT: String
\textbf{helper context KM3!Class \textbf{def: }getLabel(): String =
'{
' + self.getName() + ' |
+ self.getAttribute() + ' |
+ self.getOperations() + ' |
'};

-- getName
-- Returns the name of the class. If the class is abstract, the name
-- is put between slashes
-- IN: N/A
-- OUT: String
\textbf{helper context KM3!Class \textbf{def: }getName(): String =
if self.isAbstract then
  '/ ' + self.name + '/ '
else
  self.name
endif;

-- getAttributes
-- Returns the list of attributes of the class, with one attribute per line
-- and the correct multiplicities, using the getMultiplicity helper.
-- IN: N/A
-- OUT: String
\textbf{helper context KM3!Class \textbf{def: }getAttributes(): String =
let attributes : Sequence(KM3!Attribute) = self.structuralFeatures->
select( e | e.oclIsKindOf(KM3!Attribute)) \textbf{in}
if attributes->notEmpty() then
attributes->iterate( e; acc: String = '' |
acc + if acc = '' then '' else ' \n' endif +
e.name + e.getMultiplicity() + ' : ' + e.type.name
}
else
,
endif;

-- getOperations
-- Returns the list of operation of the class, with one operation per line,
-- their parameters and return type.
-- IN: N/A
-- OUT: String
\textbf{helper context KM3!Class \textbf{def: }getOperations(): String =
let operations : Sequence(KM3!Operation) = self.operations \textbf{in}
if operations->notEmpty() then
operations->iterate( e; acc: String = '' |
acc + e.name + e.getParameters() +

if e.type.oclIsUndefined() then
    "
else
    ' : ' + e.type.name
endif + '\n')
else
    '',
endif;

-- getMultiplicity
-- Returns the multiplicity of the element
-- IN:  N/A
-- OUT:  String
helper context KM3!TypedElement def: getMultiplicity(): String =
    if self.lower = 0 then
        '[[*]]'
    else
        '[ ' + self.lower.toString() + '=' + self.upper.toString() + ']'
    endif
else
    if self.upper = 1 then
        '
    else
        if self.upper = 0-1 then
            '[[ ' + self.lower.toString() + '=' + ']]'
        else
            '[ ' + self.lower.toString() + self.upper.toString() + ']'
        endif
    endif
endif;

-- getParameters
-- Returns the parameters of the current operation, with their types,
-- and separated with commas.
-- IN:  N/A
-- OUT:  String
helper context KM3!Operation def: getParameters(): String =
    let parameters : Sequence(KM3!Parameters) = self.parameters
    in
        '(' + parameters->iterate( e; acc: String = '' | acc +
            if e.name = parameters->last().name then
                e.name + ' : ' + e.type.name
            else
                e.name + ' : ' + e.type.name + ','
            endif
        + ')');

-- relationsList
-- This helper is used so as to match a reference only once. Indeed, in
-- KM3, if the relation is bidirectionnal, it is referenced in both its
-- edge classes.
-- It puts the container class the second part of the returned tuple.
-- IN:  N/A
-- OUT:  Sequence(Tuple (reference, opposite reference))
helper def: relationsList: Sequence(TupleType(ref: KM3!Reference, opposite : KM3!Reference)) =
    let references : Sequence(KM3!Reference) = KM3!Reference.allInstances() ->
        reject( e | e.opposite.oclIsUndefined() ) in
        references->iterate( e; acc: Sequence(TupleType(ref: KM3!Reference, opposite: KM3!Reference)) =

Sequence() | 1
if acc->excludes(Tuple(ref = e, opposite = e.opposite)) then 2
  if acc->excludes(Tuple(ref = e.opposite, opposite = e)) then 3
    if e.opposite.isContainer then 4
      acc->append(Tuple(ref = e, opposite = e.opposite)) 5
    else 6
      acc->append(Tuple(ref = e.opposite, opposite = e)) 7
    endif 8
  else 9
    acc 10
  endif 11
else 12
  acc 13
endif 14
-- END HELPERS 15
-- RULES 16
-- Metamodel2Graph 17
-- Transforms a KM3 Metamodel element into a DOT oriented graph element, 18
-- using many parameters defined at the beginning of this transformation. 19
-- The Graph elements contains then contents of the KM3 Metamodel element. 20
rule Metamodel2Graph { 21
  from m: KM3!Metamodel 22
to out: DOT!Graph ( 23
type <- 'digraph', 24
name <- 'KM3 Model in DOT', 25
rankDir <- thisModule.RankingDirection(), 26
labeljust <- thisModule.LabelJustification(), 27
labelloc <- thisModule.LabelLocation(), 28
compound <- true, 29
concentrate <- thisModule.DiagramMode('Generalization') and 30
not thisModule.Invisible(), 31
nodeSeparation <- thisModule.MinimumNodeSeparation(), 32
nodes <- m.contents 33
) 34
}
-- Package 35
-- Transforms a Package into a SubGraph that will be rendered within a 36
-- black box (because its name begins with 'cluster_' and its color is set 37
-- at black). It may contain nodes or subgraphs, dot supports nested 38
-- subgraphs, and KM3 supports nested packages. 39
rule Package { 40
  from p: KM3!Package 41
to out: DOT!SubGraph ( 42
name <- 'cluster_' + p.name, 43
label <- SubGraphLabel, 44
color <- 'black', 45
labelloc <- thisModule.LabelLocation(), 46
nodes <- p.contents 47
), 48
SubGraphLabel: DOT!SimpleLabel ( 49
content <- p.name 50
)
)
-- Datatypes 51
-- Transforms a Datatype into a Record Node using the datatype header
rule Datatypes {
  from d: KM3!DataType
  to out: DOT!Node {
    name <- d.name,
    shape <- NodeShape,
  },
  NodeShape: DOT!RecordNodeShape {
    name <- 'record',
    label <- NodeLabel,
  },
  NodeLabel: DOT!SimpleLabel {
    content <- '{' + d.getDataTypeHead() + '|' + '}'
  }
}

-- ClassWithSupertypesGeneralizationsDrawn2Node
-- Transforms a class into a node, and creates the generalization arcs
-- foreach superclass
rule ClassWithSupertypesGeneralizationsDrawn2Node {
  from c: KM3!Class {
    not(c.supertypes->oclIsUndefined()) and
    (thisModule.Invisible() or
     (thisModule.DiagramMode('Generalization') or
      thisModule.DiagramMode('All')))
  } to out: DOT!Node {
    name <- c.name,
    shape <- nodeShape,
    refers <- Sequence {c.structuralFeatures->select( e |
      e.oclIsKindOf(KM3!Reference))}->append(supertypeClasses)
  },
  nodeShape: DOT!RecordNodeShape {
    name <- 'record',
    label <- NodeLabel,
  },
  NodeLabel : DOT!SimpleLabel {
    content <- c.getLabel()
  },
  supertypeClasses: distinct DOT!DirectedArc
  foreach(sup in c.supertypes) {
    constraint <- (thisModule.DiagramType() = 'Generalization' or
                   thisModule.DiagramType() = 'None'),
    style <- if thisModule.DiagramMode('Generalization') or
              thisModule.DiagramMode('All') then 'none' else 'invis' endif,
    fromNode <- c,
    toNode <- super,
    group <- super.name,
    minlen <- thisModule.MinimumArcLength(),
    arrowHead <- arrowHeadShape
  },
  arrowHeadShape: distinct DOT!ArrowShape
  foreach(sup in c.supertypes) {
    name <- 'normal',
    isPlain <- true
  }
}

-- ClassWithSupertypesGeneralizationsNotDrawn2Node
-- Transforms a class into a node, and does not create the generalization
-- arcs either because it has no superclass, or because the parameters set
-- for the transformation imply not drawing any generalization arc

rule ClassWithSupertypesGeneralizationsNotDrawn2Node {
  from
    c: KM3!Class {
      c.supertypes->oclIsUndefined()
      or
      (not(c.supertypes->oclIsUndefined())
      and
      (thisModule.Invisible() or
      not(thisModule.DiagramMode('Generalization')
      or
      thisModule.DiagramMode('All'))))
  }
  to
    out: DOT!Node {
      name <- c.name,
      shape <- nodeShape,
      refers <- Sequence {c.structuralFeatures->select( e |
        e.oclIsKindOf(KM3!Reference))
      },
      nodeShape: DOT!RecordNodeShape {
        name <- 'record',
        label <- NodeLabel
      },
      NodeLabel : DOT!SimpleLabel {
        content <- c.getLabel()
      }
    }
}

-- Reference2OneWayArc
-- Transforms a one way reference into a unidirectional arc, with the
-- proper arrowhead and arrowtail (there may be one way compositions for
-- instance), and with its role and multiplicity

rule Reference2OneWayArc {
  from
    r: KM3!Reference {
      r.opposite.oclIsUndefined()
      and
      (thisModule.Invisible() or
      (if r.isContainer then
        thisModule.DiagramMode('Composition')
      else
        thisModule.DiagramMode('Reference')
      endif or
      thisModule.DiagramMode('All')))}
  to
    out: DOT!DirectedArc {
      fromNode <- r.owner,
      toNode <- r.type,
      arrowHead <- arrowHeadShape,
      arrowTail <- arrowTailShape,
      group <- r.type.name,
      style <- if thisModule.DiagramMode('All')
      or
      thisModule.DiagramMode('Reference')
      then 'none'
      else
      if thisModule.DiagramMode('Composition')
      then 'none'
      else
        'invis'
      endif endif,
      minlen <- thisModule.MinimumArcLength(),
      headlabel <- ArcHeadLabel,
      constraint <- (r.isContainer
      and (thisModule.DiagramType() = 'Composition' or
      thisModule.DiagramType() = 'None'))
    },
    ArcHeadLabel : DOT!SimpleLabel {
    }
}
content <- r.name + r.getMultiplicity() + if r.isOrdered then
    '({ordered})
else
    '',
endif,
arrowHeadShape: DOT!ArrowShape {
    name <- 'vee',
    isPlain <- false,
    clipping <- 'none'
},
arrowTailShape: DOT!ArrowShape {
    name <- if r.isContainer then 'diamond' else 'none' endif,
    isPlain <- false,
    clipping <- 'none'
}
}

-- Reference2Arc
-- Transforms a bidirectional reference into a bidirectional arc, with its
-- roles, multiplicities and arrowshapes.
rule Reference2Arc {
    from: r: KM3!Reference {
        if not(r.opposite.oclIsUndefined()) then
            thisModule.relationsList->includes(Tuple{ref = r, opposite = r.opposite}) and
            (thisModule.Invisible() or
             (if r.isContainer or r.opposite.isContainer then
                thisModule.DiagramMode('Composition')
             else
                thisModule.DiagramMode('Reference')
             endif
             or thisModule.DiagramMode('All')))}
        else
            false
        endif
    }
    to: DOT!DirectedArc {
        fromNode <- r.owner,
        toNode <- r.type,
        group <- r.type.name,
        minlen <- thisModule.MinimumArcLength(),
        arrowHead <- arrowHeadShape,
        arrowTail <- arrowTailShape,
        taillabel <- ArcTailLabel,
        style <- if thisModule.DiagramMode('All') or
            thisModule.DiagramMode('Reference') then 'none' else
            if thisModule.DiagramMode('Composition')
                and (r.isContainer or r.opposite.isContainer) then 'none' else
                'invis' endif endif,
        constraint <- ((r.isContainer or r.opposite.isContainer) and 
            (thisModule.DiagramType() = 'Composition' or
             thisModule.DiagramType() = 'None')),
        headlabel <- ArcHeadLabel
    }
}
ArcHeadLabel : DOT!SimpleLabel {
    content <- r.name + r.getMultiplicity() + if r.isOrdered then
        '{ordered}'
    else
        ''
    }
}
endif

ArcTailLabel : DOT!SimpleLabel {
  content <- r.opposite.name + r.opposite.getMultiplicity() +
  if r.opposite.isOrdered then
    '{ordered}'
  else ''
  endif

}),
arrowHeadShape: DOT!ArrowShape {
  name <- if r.opposite.isContainer then 'diamond' else 'none' endif,
  isPlain <- false,
  clipping <- 'none'
},
arrowTailShape: DOT!ArrowShape {
  name <- if r.isContainer then 'diamond' else 'none' endif,
  isPlain <- false,
  clipping <- 'none'
}

-- END RULE
I. DOT metamodel in KM3 format

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

package DOT {

    -- Labels
    abstract class Label {
        reference element : GraphElement oppositeOf label;
    }

    class SimpleLabel extends Label {
        attribute content : String;
    }

    class ComplexLabel extends Label {
        reference compartments[1-*] ordered container : Compartment oppositeOf complexLabel;
    }

    abstract class Compartment {
        reference complexLabel : ComplexLabel oppositeOf compartments;
        reference compartments[0-1] ordered container : Compartment;
        reference anchor[0-1] : Anchor oppositeOf source;
    }

    class VerticalCompartment extends Compartment {
    }

    class HorizontalCompartment extends Compartment {
    }

    class SimpleCompartment extends Compartment {
        attribute content : String;
    }

    class Anchor {
        attribute name : String;
        reference source[0-1] : Compartment oppositeOf anchor;
    }

    -- End Labels

    -- GraphElements
    abstract class GraphElement {
        attribute name : String;
        reference label[0-1] container : Label oppositeOf element;
        attribute style[0-1] : String; -- invis | filled | rounded |
        diagonals | dashed | dotted | none
        attribute color[0-1] : String;
    }
}
class Graph extends GraphElement {
    attribute type : String; -- digraph | graph
    attribute rankDir[0-1] : String;
    attribute labeljust[0-1] : String;
    attribute labelloc[0-1] : String;
    attribute concentrate[0-1] : Boolean;
    reference nodes[*] ordered container : Nodelike oppositeOf graph;
    attribute boundingBox[0-1] : String;
    attribute compound[0-1] : Boolean;
    reference layers[*] container : Layer oppositeOf graph;
    attribute nodeSeparation[0-1] : Double;
    attribute ordering[0-1] : String;
    attribute size[0-1] : String;
    attribute ratio[0-1] : String;
    attribute center[0-1] : Boolean;
}

class Layer extends GraphElement {
    reference nodes[*] : Nodelike oppositeOf layers;
    reference arcs[*] : Arc oppositeOf layers;
    reference graph : Graph oppositeOf layers;
    attribute layerSeparator[0-1] : String;
}

-- Nodelikes
abstract class Nodelike extends GraphElement {
    reference owner[0-1] : SubGraph oppositeOf nodes;
    reference refers[*] : Arc oppositeOf fromNode;
    reference referredBy[*] : Arc oppositeOf toNode;
    reference graph[0-1] : Graph oppositeOf nodes;
    reference layers[*] : Layer oppositeOf nodes;
}

class SubGraph extends Nodelike {
    reference nodes[*] ordered container : Nodelike oppositeOf owner;
    attribute labelloc[0-1] : String;
}

class Node extends Nodelike {
    attribute fixedSize[0-1] : Boolean;
    attribute fontname[0-1] : String;
    attribute fontsize[0-1] : Integer;
    attribute height[0-1] : Integer;
    attribute width[0-1] : Integer;
    reference shape[0-1] container : NodeShape;
}

-- End Nodelikes

-- Arcs
abstract class Arc extends GraphElement {
    reference fromNode : Nodelike oppositeOf refers;
    reference toNode : Nodelike oppositeOf referredBy;
    reference layers[*] : Layer oppositeOf arcs;
    attribute constraint[0-1] : Boolean;
    attribute group[0-1] : String;
    attribute minlen[0-1] : Integer;
attribute sameHead[0-1] : String;
attribute sameTail[0-1] : String;
reference lhead[0-1] : Nodelike;
reference ltail[0-1] : Nodelike;
attribute decorate[0-1] : Boolean;
}
-- if self.lhead.oclIsKindOf(DOT!SubGraph) or self.ltail.oclIsKindOf
-- (DOT!SubGraph) then self.getEnclosingGraph().compound
-- else false endif

class DirectedArc extends Arc {
  reference arrowHead[0-1] container : ArrowShape;
  reference headlabel[0-1] : Label;
  reference taillabel[0-1] : Label;
  reference arrowTail[0-1] container : ArrowShape;
  attribute tail_lp[0-1] : Double;
  attribute head_lp[0-1] : Double;
}
class UndirectedArc extends Arc {
}
-- End Arcs

-- Shapes
abstract class Shape extends GraphElement {
  attribute width : Integer;
  attribute height : Integer;
  attribute peripheries : Integer;
}
abstract class NodeShape extends Shape {
}

-- name may be : box | ellipse | circle | egg | triangle | plaintext |
one
-- diamond | trapezium | parallelogram | house | pentagon | hexagon |
septagon |
-- octagon | doublecircle | doubleoctagon | tripleoctagon | invtriangle
|-- invtrapezium | invhouse | rect | rectangle
-- These shapes have not been used, as for a class diagram they are not
needed.
class SimpleNodeShape extends NodeShape {
}
class PointNodeShape extends NodeShape {
}
abstract class ComplexNodeShape extends NodeShape {
-- name = polygon
class PolygonNodeShape extends ComplexNodeShape {
  attribute sides : Integer;
  attribute skew : Integer;
```plaintext
attribute distortion : Integer;
attribute isRegular : Boolean;
attribute orientation : Integer;
}

-- name may be : Mdiamond | Msquare | Mcircle
class MNodeShape extends ComplexNodeShape {
    reference toplabel[0-1] container : Label;
    reference bottomlabel[0-1] container : Label;
}

-- name may be : record | Mrecord
class RecordNodeShape extends ComplexNodeShape {
}

-- Arrow Shape :
    -- Name may be : box | crow | diamond | dot | inv | none | normal |
    -- tee | vee
    -- Clipping : left | right | none
    -- Clipping other than none has no sense with arrowShapes : dot |
    none
    -- isPlain = false has no sense with arrowShapes : crow | none |
    tee | vee
class ArrowShape extends Shape {
    attribute clipping : String;
    attribute isPlain : Boolean;
    attribute size : Integer;
}

-- End Shapes
-- End GraphElements
```
References
