

1. ATL Transformation Example

1.1. Example: KM3 → DOT

The KM3 to DOT example describes a transformation from a KM3 metamodel description into a class diagram drawn with dot. KM3 [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 [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.

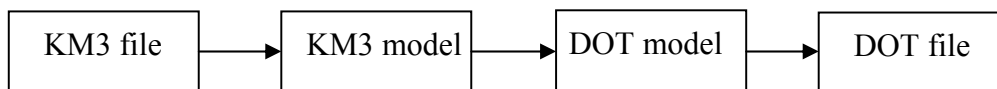


Figure 1. Transformation from end to end

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

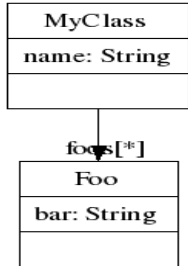
<pre> package MyPackage { class MyClass { attribute name: String; reference foos[*] : Foo; } class Foo { attribute bar: String; } } </pre>	<pre> digraph MyPackage { graph[rankDir=BT] node[shape=record] MyClass[label={MyClass name: String }] Foo[label={Foo bar: String }] MyClass -> Foo [headlabel='foos[*]']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

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.

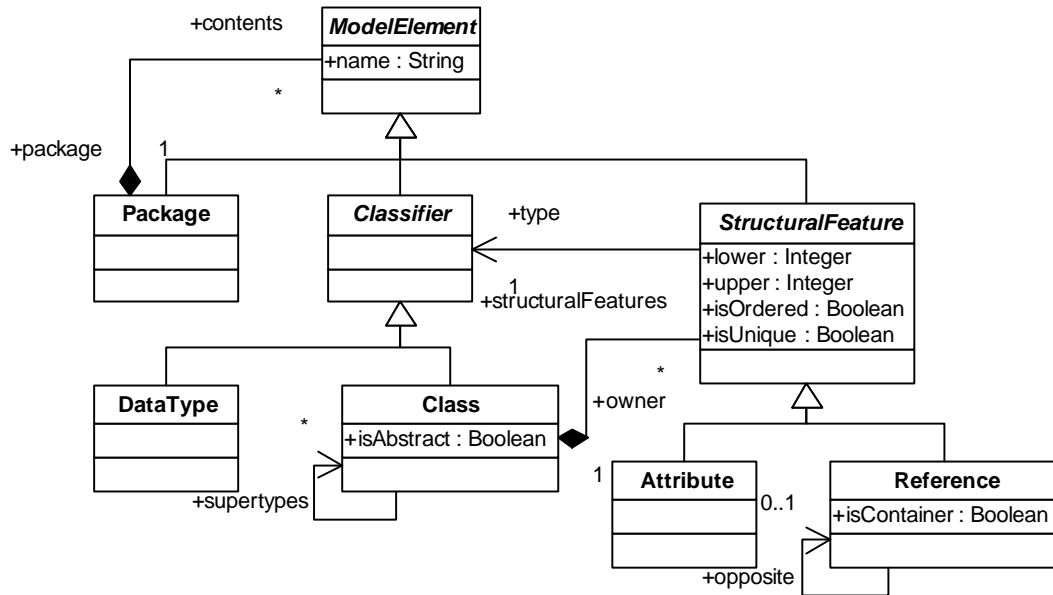


Figure 2. Simplified KM3 metamodel

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.

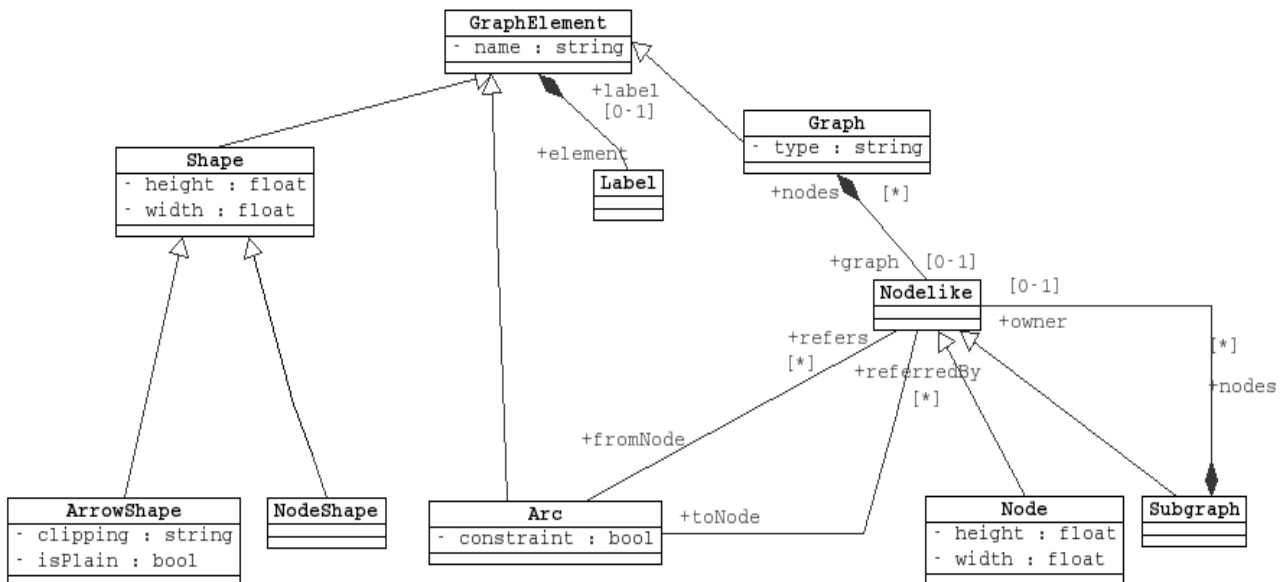


Figure 3. Simplified DOT metamodel

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.

```
1  module KM32DOT;
2  create OUT: DOT from IN: KM3;
3
4  -- Parameters
5
6  -- DiagramType
7  -- Specifies what kind of diagram shall be rendered: a Generalization one,
8  -- or one based on composition links, or a standard one.
9  -- Possible values:      'None' | 'Generalization' | 'Composition'
10 helper def: DiagramType(): String = 'None';
11
12 -- Mode
13 -- Specifies what association type shall be used so as to compute the
14 -- layout of the diagram
15 -- Possible values:      'All' | 'Generalization' | 'Composition' | 'Reference'
16 helper def: Mode(): String = 'All';
17
18 -- Invisible
19 -- Specifies whether the associations not used in the computation of the
20 -- layout have yet to be rendered. For example, if Invisible = true and
21 -- Mode = Generalization, the diagram will be rendered as if only
22 -- generalization links did matter. If Invisible = false, then only
23 -- generalization links are drawn, but the layout is the same as if
24 -- Possible values:      true | false
25 helper def: Invisible(): Boolean = false;
26
27 -- MinimumArcLength
28 -- Specifies the minimum length of an association, in inches.
29 -- Possible values:      any positive integer value
30 helper def: MinimumArcLength(): Integer = 2;
31
32 -- MinimumNodeSeparation
33 -- Specifies the minimum distance between two nodes, in inches.
34 -- Possible values:      any positive real value
35 helper def: MinimumNodeSeparation(): Real = 0.75;
36
37 -- RankingDirection
38 -- Specifies the direction in which the diagram should be rendered. Most
39 -- class diagrams are rendered with the value 'BT'
40 -- Possible values:      'BT' | 'TB' | 'LR' | 'RL' (Bottom to Top,
41 -- Top to Bottom, Left to Right, Right to Left)
42 helper def: RankingDirection(): String = 'BT';
43
44 helper def: LabelJustification(): String = 'l';
45
46 helper def: LabelLocation(): String = 't';
47
48 -- DataTypeHeader
49 -- The name of a datatype should begin with <<DataType>>
50 helper def: DataTypeHeader(): String = '&#171;DataType&#187;\n';
51
52 -- End Parameters
53
54 -- HELPERS
55 -- DiagramMode
56 -- Returns whether the DiagramMode parameter is the one tried or not
```

```
57     -- IN:     mode: String
58     -- OUT:    Boolean
59     helper def: DiagramMode(mode: String): Boolean =
60         mode = thisModule.Mode();
61
62     -- getDataTypeHead
63     -- Returns the name of the datatype, with its header
64     -- IN:     N/A
65     -- OUT:    String
66     helper context KM3!DataType def: getDataTypeHead(): String =
67         thisModule.DataTypeHeader() + self.name;
68
69     -- getLabel
70     -- Returns the correct SimpleLabel content for a KM3 Class:
71     -- Name | Attributes | Operations
72     -- IN:     N/A
73     -- OUT:    String
74     helper context KM3!Class def: getLabel(): String =
75         '{' + self.getName() + '|'
76         + self.getAttributes() + '|'
77         + self.getOperations() + '}';
78
79     -- getName
80     -- Returns the name of the class. If the class is abstract, the name
81     -- is put between slashes
82     -- IN:     N/A
83     -- OUT:    String
84     helper context KM3!Class def: getName(): String =
85         if self.isAbstract then
86             '/' + self.name + '/'
87         else
88             self.name
89         endif;
90
91     -- getAttributes
92     -- Returns the list of attributes of the class, with one attribute per line
93     -- and the correct multiplicities, using the getMultiplicity helper.
94     -- IN:     N/A
95     -- OUT:    String
96     helper context KM3!Class def: getAttributes(): String =
97         let attributes : Sequence(KM3!Attribute) = self.structuralFeatures->
98             select( e |
99                 e.oclIsKindOf(KM3!Attribute)) in
100         if attributes->notEmpty() then
101             attributes->iterate( e; acc: String = '' |
102                 acc + if acc = '' then '' else '\\\n' endif +
103                 e.name + e.getMultiplicity() + ' : ' + e.type.name
104             )
105         else
106             ''
107         endif;
108
109     -- getOperations
110     -- Returns the list of operation of the class, with one operation per line,
111     -- their parameters and return type.
112     -- IN:     N/A
113     -- OUT:    String
114     helper context KM3!Class def: getOperations(): String =
115         let operations : Sequence(KM3!Operation) = self.operations in
116         if operations->notEmpty() then
117             operations->iterate( e; acc: String = '' |
118                 acc + e.name + e.getParameters() +
```

```
119         if e.type.oclIsUndefined() then
120             ''
121         else
122             ' : ' + e.type.name
123         endif + '\\n')
124     else
125         ''
126     endif;
127
128     -- getMultiplicity
129     -- Returns the multiplicity of the element
130     -- IN:    N/A
131     -- OUT:   String
132     helper context KM3!TypedElement def: getMultiplicity(): String =
133     if self.lower = 0 then
134         if self.upper = 0-1 then
135             '['*]'
136         else
137             '[' + self.lower.toString() + '-' + self.upper.toString() + ']'
138         endif
139     else
140         if self.upper = 1 then
141             ''
142         else
143             if self.upper = 0-1 then
144                 '[' + self.lower.toString() + '-' + '*' ]'
145             else
146                 '[' + self.lower.toString() + self.upper.toString() + ']'
147             endif
148         endif
149     endif;
150
151     -- getParameters
152     -- Returns the parameters of the current operation, with their types,
153     -- and separated with commas.
154     -- IN:    N/A
155     -- OUT:   String
156     helper context KM3!Operation def: getParameters(): String =
157     let parameters : Sequence(KM3!Parameters) = self.parameters in
158     '(' + parameters->iterate( e; acc: String = '' |
159     acc +
160     if e.name = parameters->last().name then
161         e.name + ' : ' + e.type.name
162     else
163         e.name + ' : ' + e.type.name + ','
164     endif)
165     + ')';
166
167     -- relationsList
168     -- This helper is used so as to match a reference only once. Indeed, in
169     -- KM3, if the relation is bidirectionnal, it is referenced in both its
170     -- edge classes.
171     -- It puts the container class the second part of the returned tuple.
172     -- IN:    N/A
173     -- OUT:   Sequence(Tuple (reference, opposite reference))
174     helper def: relationsList: Sequence(
175     TupleType(ref: KM3!Reference, opposite : KM3!Reference)) =
176     let references: Sequence(KM3!Reference) = KM3!Reference.allInstances()->
177     reject( e |
178     e.opposite.oclIsUndefined()) in
179     references->iterate( e;
180     acc: Sequence(TupleType(ref: KM3!Reference, opposite: KM3!Reference)) =
```

```
181     Sequence{} |
182         if acc->excludes(Tuple{ref = e, opposite = e.opposite}) then
183             if acc->excludes(Tuple{ref = e.opposite, opposite = e}) then
184                 if e.opposite.isContainer then
185                     acc->append(Tuple{ref = e, opposite = e.opposite})
186                 else
187                     acc->append(Tuple{ref = e.opposite, opposite = e})
188                 endif
189             else
190                 acc
191             endif
192         else
193             acc
194         endif);
195 -- END HELPERS
196
197 -- RULES
198 -- Metamodel2Graph
199 -- Transforms a KM3 Metamodel element into a DOT oriented graph element,
200 -- using many parameters defined at the beginning of this transformation.
201 -- The Graph elements contains then contents of the KM3 Metamodel element.
202 rule Metamodel2Graph {
203     from
204         m: KM3!Metamodel
205     to
206         out: DOT!Graph (
207             type <- 'digraph',
208             name <- 'KM3 Model in DOT',
209             rankDir <- thisModule.RankingDirection(),
210             labeljust <- thisModule.LabelJustification(),
211             labelloc <- thisModule.LabelLocation(),
212             compound <- true,
213             concentrate <- thisModule.DiagramMode('Generalization') and
214             not thisModule.Invisible(),
215             nodeSeparation <- thisModule.MinimumNodeSeparation(),
216             nodes <- m.contents
217         )
218 }
219
220 -- Package
221 -- Transforms a Package into a SubGraph that will be rendered within a
222 -- black box (because its name begins with 'cluster_' and its color is set
223 -- at black). It may contain nodes or subgraphs, dot supports nested
224 -- subgraphs, and KM3 supports nested packages.
225 rule Package {
226     from
227         p: KM3!Package
228     to
229         out: DOT!SubGraph (
230             name <- 'cluster_' + p.name,
231             label <- SubGraphLabel,
232             color <- 'black',
233             labelloc <- thisModule.LabelLocation(),
234             nodes <- p.contents
235         ),
236         SubGraphLabel: DOT!SimpleLabel (
237             content <- p.name
238         )
239 }
240
241 -- Datatypes
242 -- Transforms a Datatype into a Record Node using the datatype header
```

```
243 rule Datatypes {
244   from
245     d: KM3!DataType
246   to
247     out: DOT!Node (
248       name <- d.name,
249       shape <- NodeShape
250     ),
251     NodeShape: DOT!RecordNodeShape (
252       name <- 'record',
253       label <- NodeLabel
254     ),
255     NodeLabel: DOT!SimpleLabel (
256       content <- '{' + d.getDataTypeHead() + '|' + '}'
257     )
258 }
259
260 -- ClassWithSupertypesGeneralizationsDrawn2Node
261 -- Transforms a class into a node, and creates the generalization arcs
262 -- foreach superclass
263 rule ClassWithSupertypesGeneralizationsDrawn2Node {
264   from
265     c: KM3!Class (
266       not(c.supertypes->oclIsUndefined()) and
267       (thisModule.Invisible() or
268        (thisModule.DiagramMode('Generalization') or
269         thisModule.DiagramMode('All')))
270     )
271   to
272     out: DOT!Node (
273       name <- c.name,
274       shape <- nodeShape,
275       refers <- Sequence {c.structuralFeatures->select( e |
276         e.oclIsKindOf(KM3!Reference))}->append(supertypeClasses)
277     ),
278     nodeShape: DOT!RecordNodeShape (
279       name <- 'record',
280       label <- NodeLabel
281     ),
282     NodeLabel : DOT!SimpleLabel (
283       content <- c.getLabel()
284     ),
285     supertypeClasses: distinct DOT!DirectedArc
286     foreach(super in c.supertypes) (
287       constraint <- (thisModule.DiagramType() = 'Generalization' or
288        thisModule.DiagramType() = 'None'),
289       style <- if thisModule.DiagramMode('Generalization') or
290        thisModule.DiagramMode('All') then 'none' else 'invis' endif,
291       fromNode <- c,
292       toNode <- super,
293       group <- super.name,
294       minlen <- thisModule.MinimumArcLength(),
295       arrowHead <- arrowHeadShape
296     ),
297     arrowHeadShape: distinct DOT!ArrowShape
298     foreach(super in c.supertypes) (
299       name <- 'normal',
300       isPlain <- true
301     )
302 }
303
304 -- ClassWithSupertypesGeneralizationsNotDrawn2Node
```



```
305 -- Transforms a class into a node, and does not create the generalization
306 -- arcs either because it has no superclass, or because the parameters set
307 -- for the transformation imply not drawing any generalization arc
308 rule ClassWithSupertypesGeneralizationsNotDrawn2Node {
309   from
310     c: KM3!Class (
311       c.supertypes->oclIsUndefined() or
312       (not(c.supertypes->oclIsUndefined()) and
313        (thisModule.Invisible() or
314         not(thisModule.DiagramMode('Generalization') or
315          thisModule.DiagramMode('All'))))
316     )
317   to
318     out: DOT!Node (
319       name <- c.name,
320       shape <- nodeShape,
321       refers <- Sequence {c.structuralFeatures->select( e |
322         e.oclIsKindOf(KM3!Reference))}
323     ),
324     nodeShape: DOT!RecordNodeShape (
325       name <- 'record',
326       label <- NodeLabel
327     ),
328     NodeLabel : DOT!SimpleLabel (
329       content <- c.getLabel()
330     )
331 }
332
333 -- Reference2OneWayArc
334 -- Transforms a one way reference into a unidirectional arc, with the
335 -- proper arrowhead and arrowtail (there may be one way compositions for
336 -- instance), and with its role and multiplicity
337 rule Reference2OneWayArc {
338   from
339     r: KM3!Reference (
340       r.opposite.oclIsUndefined() and
341       (thisModule.Invisible() or
342        (if r.isContainer then
343          thisModule.DiagramMode('Composition')
344        else thisModule.DiagramMode('Reference')
345        endif or
346         thisModule.DiagramMode('All'))))
347   to
348     out: DOT!DirectedArc (
349       fromNode <- r.owner,
350       toNode <- r.type,
351       arrowHead <- arrowHeadShape,
352       arrowTail <- arrowTailShape,
353       group <- r.type.name,
354       style <- if thisModule.DiagramMode('All') or
355         thisModule.DiagramMode('Reference') then 'none' else
356         if thisModule.DiagramMode('Composition') and
357         r.isContainer then 'none' else
358         'invis' endif endif,
359       minlen <- thisModule.MinimumArcLength(),
360       headlabel <- ArcHeadLabel,
361       constraint <- (r.isContainer
362         and (thisModule.DiagramType() = 'Composition' or
363          thisModule.DiagramType() = 'None'))
364     ),
365     ArcHeadLabel : DOT!SimpleLabel (
366
```

```
367         content <- r.name + r.getMultiplicity() + if r.isOrdered then
368             '{ordered}'
369         else
370             ''
371         endif
372     ),
373     arrowHeadShape: DOT!ArrowShape (
374         name <- 'vee',
375         isPlain <- false,
376         clipping <- 'none'
377     ),
378     arrowTailShape: DOT!ArrowShape (
379         name <- if r.isContainer then 'diamond' else 'none' endif,
380         isPlain <- false,
381         clipping <- 'none'
382     )
383 }
384
385 -- Reference2Arc
386 -- Transforms a bidirectional reference into a bidirectional arc, with its
387 -- roles, multiplicities and arrowshapes.
388 rule Reference2Arc {
389     from
390         r: KM3!Reference (
391             if not(r.opposite.oclIsUndefined()) then
392                 thisModule.relationsList->includes(
393                     Tuple{ref = r, opposite = r.opposite}) and
394                     (thisModule.Invisible() or
395                     (if r.isContainer or r.opposite.isContainer then
396                         thisModule.DiagramMode('Composition')
397                     else
398                         thisModule.DiagramMode('Reference')
399                     endif
400                     or thisModule.DiagramMode('All')))
401             else
402                 false
403             endif
404         )
405     to
406         out: DOT!DirectedArc (
407             fromNode <- r.owner,
408             toNode <- r.type,
409             group <- r.type.name,
410             minlen <- thisModule.MinimumArcLength(),
411             arrowHead <- arrowHeadShape,
412             arrowTail <- arrowTailShape,
413             taillabel <- ArcTailLabel,
414             style <- if thisModule.DiagramMode('All') or
415                 thisModule.DiagramMode('Reference') then 'none' else
416                 if thisModule.DiagramMode('Composition')
417                     and (r.isContainer or r.opposite.isContainer) then 'none' else
418                     'invis' endif endif,
419             constraint <- ((r.isContainer or r.opposite.isContainer) and
420                 (thisModule.DiagramType() = 'Composition' or
421                 thisModule.DiagramType() = 'None')),
422             headlabel <- ArcHeadLabel
423         ),
424         ArcHeadLabel : DOT!SimpleLabel (
425             content <- r.name + r.getMultiplicity() + if r.isOrdered then
426                 '{ordered}'
427             else
428                 ''
```

```
429         endif
430     ),
431     ArcTailLabel : DOT!SimpleLabel (
432         content <- r.opposite.name + r.opposite.getMultiplicity() +
433         if r.opposite.isOrdered then
434             '{ordered}'
435         else
436             ''
437         endif
438     ),
439     arrowHeadShape: DOT!ArrowShape (
440         name <- if r.opposite.isContainer then 'diamond' else 'none' endif,
441         isPlain <- false,
442         clipping <- 'none'
443     ),
444     arrowTailShape: DOT!ArrowShape (
445         name <- if r.isContainer then 'diamond' else 'none' endif,
446         isPlain <- false,
447         clipping <- 'none'
448     )
449 }
450
451 -- 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 |
none
-- 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;
}
```



**ATL
TRANSFORMATION EXAMPLE**

Contributor
Jean Paliès

KM3 to DOT

Date 30/06/05

```
    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
}
```



**ATL
TRANSFORMATION EXAMPLE**

Contributor

Jean Paliès

KM3 to DOT

Date 30/06/05

References

- [1] Kernel MetaMetaModel, <http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/gmt-home/subprojects/ATL/doc/KernelMetaMetaModel%5Bv00.03%5D.pdf>
- [2] Graphviz Tools, <http://www.graphviz.org>