OMLAB®

MM-DSL:
An EBNF Specification

Version 1.0 – Draft –

Edited by Niksa Visic

University of Vienna, Faculty of Computer Science

August 2013

Remark:
*The content of this document has been exposed with the purpose to specify a **Modeling Method Domain-Specific Language**. It also enables future collaboration on the ongoing research and implementation.*

Contact:
Niksa Visic
niksa@dke.univie.ac.at

Table of Contents

Revision History

Editor	Date	Reason For Changes	Version
Niksa Visic	21.05.2013	Railroad Diagrams added; minor grammar changes	0.2
Niksa Visic	08.07.2013	Partial Algorithm support; examples; new terminals	0.3
Niksa Visic	15.07.2013	Document structure changed; minor content updates; new examples	0.4
Niksa Visic	22.07.2013	Grammar change; document content and structure change	0.5
Niksa Visic	24.07.2013	Grammar update; code embedding is now more generic; examples modified	0.6
Niksa Visic	19.08.2013	Minor formatting fixes	1.0

1. Introduction

1.1 Purpose

The Modeling Method Domain-specific Language (MM-DSL) is being developed for the purpose of specifying the modeling method requirements and their direct translation into a fully functional modeling tool. The MM-DSL is a textual programming language; therefore the EBNF notation has been applied to formally specify its grammar.

1.2 Intended Audience and Reading Suggestions

The intended audience includes, but is not limited to, academics and experts working in the field of computer science, particularly in software engineering, computer language design, modeling and metamodeling. It is implied that the reader is familiar with the general modeling and metamodeling concepts, such as *class*, *relation*, *attribute*, *instance*, etc.

1.3 Scope

The document at hand specifies the formal MM-DSL grammar using EBNF notation. The visualization of the grammar is presented in the form of Railroad (Syntax) Diagrams. The MM-DSL is currently in the prototyping phase, which means that its grammar is susceptible to change.

1.4 Notational Conventions

The EBNF notation used in this document has been adopted from the W3C (World Wide Web Consortium) specifications and recommendations. For more details please see the XML Recommendation (http://www.w3.org/TR/xml/), section “6 Notation”, as well as XQuery Recommendation (http://www.w3.org/TR/xquery/), section “A.1.1 Notation”.

The Railroad Diagrams have been generated using the online Railroad Diagram Generator (http://railroad.my28msec.com/). The orange rectangles with soft edges represent terminals, and the pale yellow rectangles with hard edges represent nonterminals. All the productions start with a symbol represented with double arrows facing right and end with a symbol represented as arrows facing each other.

2. General Description

Every grammar production rule is represented as a Railroad Diagram with accompanying EBNF notation, as well as a short textual description. The grammar rule instantiation (the MM-DSL code) is illustrated in multiple examples. The examples also show what kind of modeling artifacts are generated in a modeling tool, when the MM-DSL code is compiled. Keep in mind that not all code snippets that illustrate the grammar usage are not a valid MM-DSL program by themselves.

2.1 Getting Started

The MM-DSL is a domain-specific programming language. It is design for describing modeling language and modeling method requirements with a purpose of compiling (translating) them into the modeling tools.

The language is structured in three specialized parts covering the important aspects of modeling languages (methods): *metamodel*, *graphical representation* and *algorithms*. Every of these three sections have its own subset of sentences on top of the general sentences available to every part of the language.

In the time when this document was written, the grammar for algorithms part was not finished, therefore it was not included in the current document.

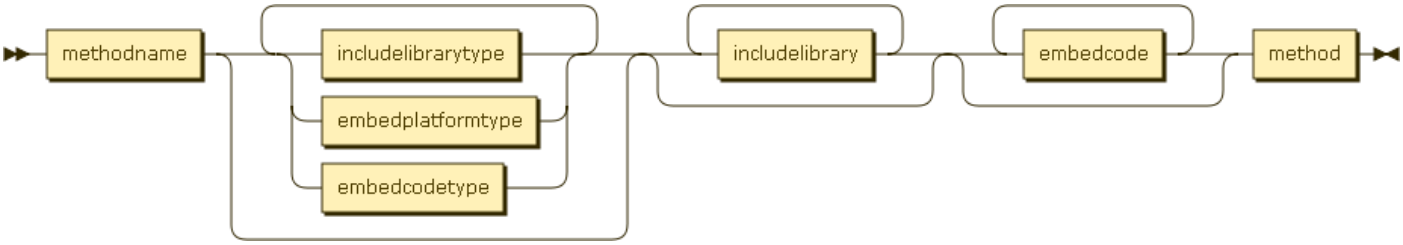
2.2 The MM-DSL Prototype

The prototype being developed at the University of Vienna, Faculty of Informatics, generates platform-specific code from the code written in MM-DSL. Generated code can be directly imported into a metamodeling platform, ergo creating a modeling tool. MM-DSL IDE (Integrated Development Environment) is developed upon the Eclipse platform (http://www.eclipse.org/), and is using ADOxx metamodeling platform (http://www.adoxx.org/) as an execution engine.

The language itself is platform independent. However, a translator which compiles the code written in MM-DSL to the ADOxx platform code is platform-dependent – it maps language concepts onto platform concepts (specifically, onto the platform meta-metamodel). This isn't an issue, because the MM-DSL IDE architecture allows the integration of multiple translators at the same time. A framework for the development of custom translators is included as well.

3. The MM-DSL Grammar

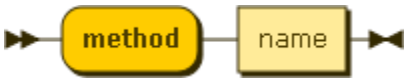
3.1 Root Element




```
root ::= methodname
       ( includelibrarytype | embedplatformtype | embedcodetype )*
       includelibrary*
       embedcode*
       method
```

Root is the nonterminal element defining the general structure of MM-DSL. Required nonterminal elements of every program are *methodname* and *method*.

3.2 Method Name



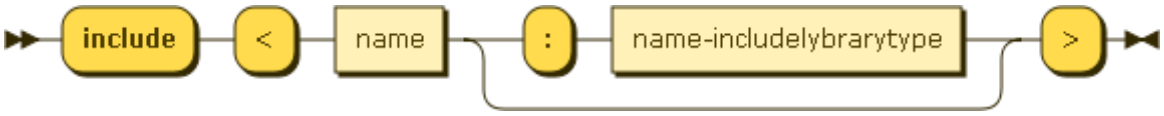

```
methodname ::= 'method' name
```

Every program in MM-DSL starts with the keyword *method*, followed by the method *name*, which is an identifier. Only one method is permitted in one program.

```
method MyMethod
```

Figure 1: Example for 3.2

3.3 Include Library




```
includelibrary ::= 'include'
                   '<' name ( ':' name-includelybrarytype )? '>'
```

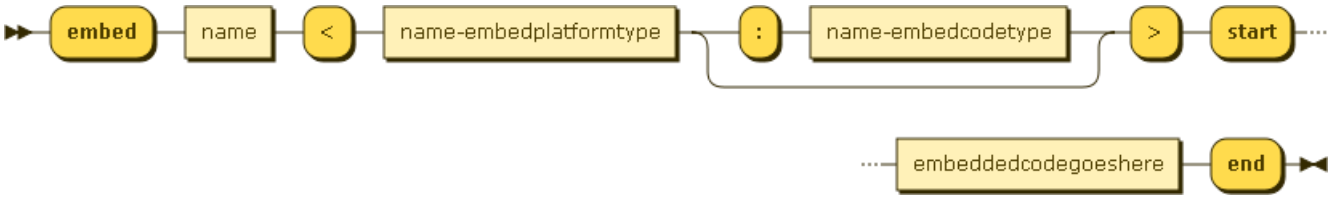
It is possible to include libraries containing various implementations. Libraries need to be included before the *method* keyword.

3.4 Include Library Type


```
includelibrarytype ::= 'def' 'IncludeLibraryType' name
```

This statement defines the name of the library used in the *includelibrary* statement.

3.5 Embed Code




```
embedcode::= 'embed' name '<' name-embedplatformtype
             ( ':' name-embedcodetype )? '>'
             'start' embeddedcodegoeshere 'end'
```

Embed code statement contains the name of the platform code is coming from, as well as the code type (one platform can have multiple formats and languages). Embedded code starts with the keyword *start* and ends with the keyword *end*. Embed code statement can be referenced by name from the *insert* statement.

3.6 Embed Platform Type


```
embedplatformtype ::= 'def' 'EmbedPlatformType' name
```

This statement defines the name of the platforms from which the embedded code comes from.

3.7 Embed Code Type


```
embedcodetype ::= 'def' 'EmbedCodeType' name
```

This statement defines the embedded code type. Both platform type and code type are later referenced by *embedcode* statement.

3.8 Insert Embed Code

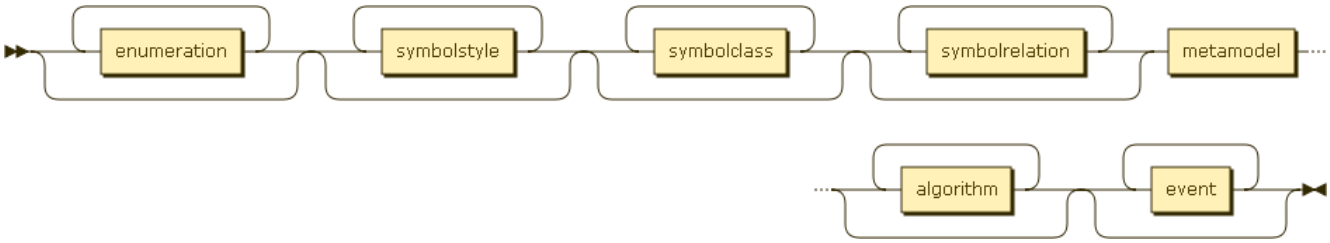

```
insertembedcode ::= 'insert' name-embedcode
```

This statement is used to insert embedded code defined in the embedded code blocks by the *embedcode* statement. To insert code on the place specified a reference to *embedcode* statement is used.

```
def EmbedPlatformType ADOxx
def EmbedCodeType GraphRep

embed ADODrawRoundRectangle <ADOxx:GraphRep>
start
embedded code goes here
end

…

classgraph RoundRect style IsParkedGraph.Black
{
	circle cx=0 cy=0 r=20
	insert ADODrawRoundRectangle
	circle cx=0 cy=0 r=30
}
```

Figure 2:Example for embedding code

3.9 Method

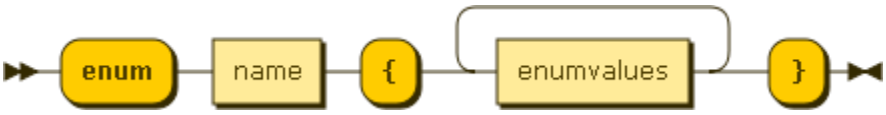



```
method ::= enumeration*
           symbolstyle*
           symbolclass*
           symbolrelation*
           metamodel
           algorithm*
           event*
```

The *method* statement is the starting point in creating your modeling language or modeling method. It contains all the necessary elements for definition of a modeling method. This statement is sequential in nature, which means that one needs to define enumerations, global symbol styles, and

class and relation symbol styles before they can be referenced in the metamodel part of the program. The metamodel needs to be defined before we can write algorithms that reference it. Algorithms need to be defined before they can be referenced by events.

3.10 Enumeration



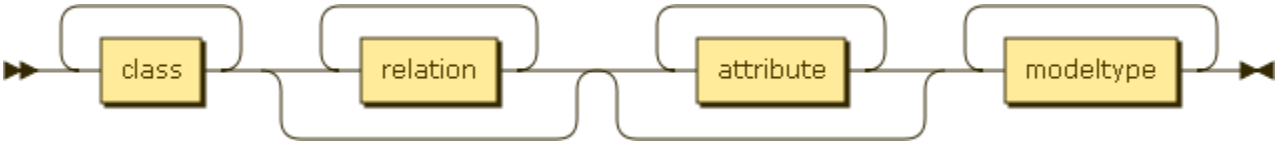

```
enumeration::= 'enum' name '{' enumvalues+ '}'
```

Enumeration allows us to create user specific sets that can be viewed as user defined data types.

```
enum MyEnum { "yes" "no" "maybe" }
```

Figure 3: Example for 3.10

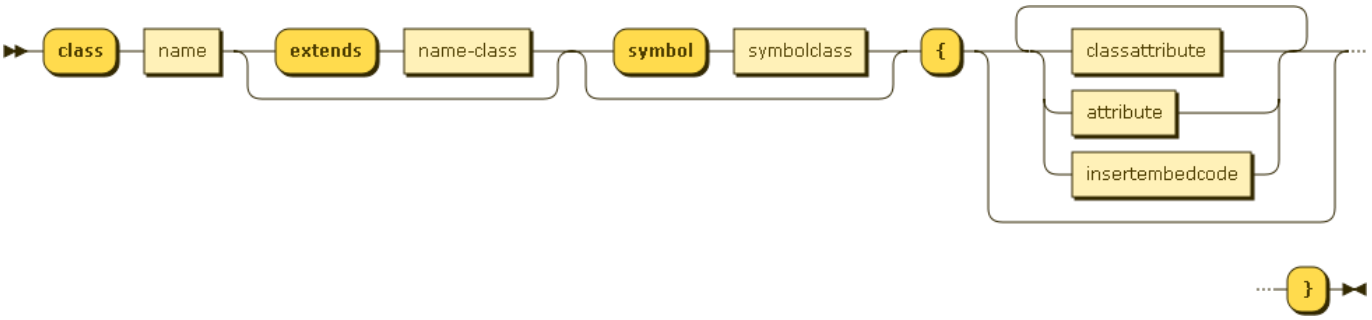
3.11 Metamodel




```
metamodel ::= class+
              relation*
              attribute*
              modeltype+
```

The *metamodel* nonterminal contains all the typical elements needed to describe a metamodel: class, relation, attribute, and a model type. Metamodel needs to have at least one class and one model type. Relations and attributes are optional.

3.12 Class



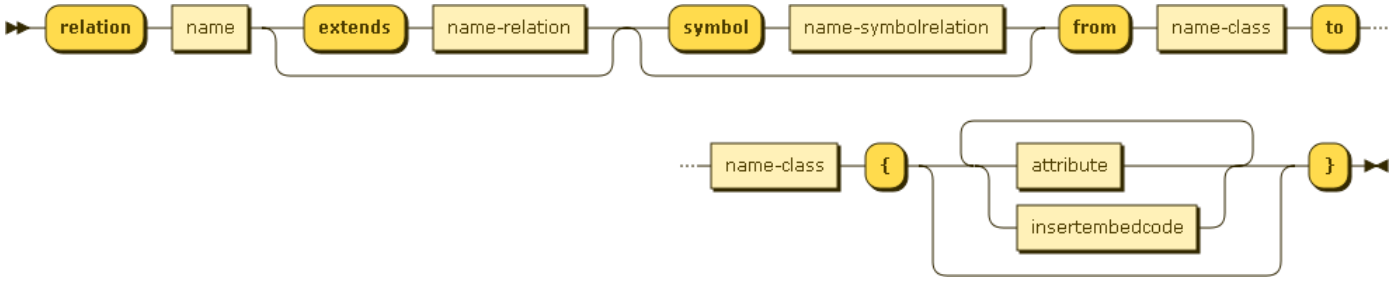

```
class ::= 'class' name ( 'extends' name-class )?
          ( 'symbol' symbolclass )? '{'
          ( classattribute | attribute | insertembedcode )* '}'
```

Every class sentence starts with the keyword *class* followed by an identifier. Class can extend another class, can be assigned a graphical representation (by using the keyword *symbol*) and can contain attributes.

```
class MyClass extends ClassA symbol CarGraph
{
	classattribute MyClassAttr : enum MyEnum
	attribute MyAttr1 : int
	attribute MyAttr2 : string
}
```

Figure 4: Example for defining a class

3.13 Relation



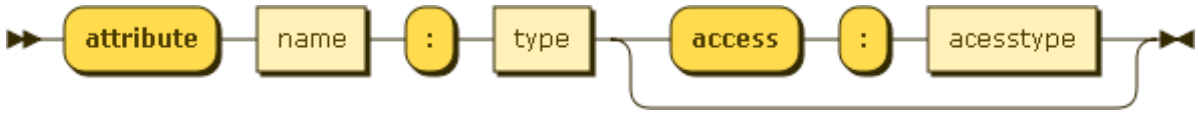

```
relation ::= 'relation' name ( 'extends' name-relation )?
             ( 'symbol' name-symbolrelation )?
             'from' name-class 'to' name-class
             '{' ( attribute | insertembedcode )* '}'
```

Every *relation* sentence starts with a keyword relation followed by an identifier. Relation can extend another relation, however, only the graphical representation and attributes are transferred to the child class. Relation has from and to relationships followed by a class identifier. It can contain attributes.

```
relation MyRelation extends Flow symbol RelatedGraph from ClassA to MyClass
{
	attribute MyRelAttr1 : int
	attribute MyRelAttr2 : int
}
```

Figure 5: Example for defining a relation

3.14 Attribute



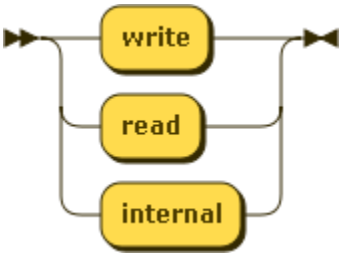
```
attribute ::= 'attribute' name ':' type
              ( 'access' ':' accesstype )?
```

Every attribute sentence starts with a keyword *attribute* and has an identifier and a type.

```
attribute MyAttr1 : int access:internal
attribute MyAttr2 : string access:read
attribute MyAttr3 : string access:write
```

Figure 6: Example for defining attributes and their access types

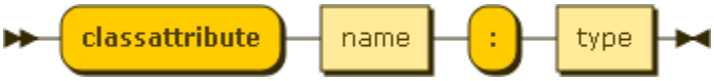
3.15 Attribute Access Type




```
accesstype ::= 'write'
             | 'read'
             | 'internal'
```

The *acesstype* sentance defines the access type to the attribute values. Write access allows all privileges, read allows read-only, and internal doesn't allow direct access to the attribute values.

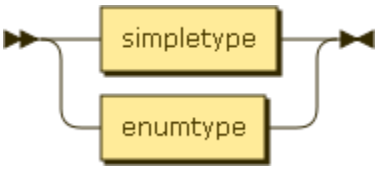
3.16 Class Attribute




```
classattribute ::= 'classattribute' name ':' type
```

Every class attribute sentence starts with a keyword *classattribute* and has an identifier and a type.

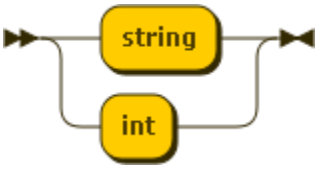
3.17 Type




```
simpletype ::= 'string'
             | 'int'
```

MM-DSL differentiates between simple types and enumerations, which are user defined collections of simple types, mostly strings or integers.

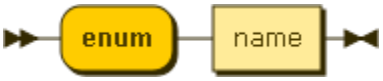
3.18 Simple Type




```
simpletype ::= 'string'
             | 'int'
```

Currently, MM-DSL supports only strings and integers. Support for other types is under development.

3.19 Enumeration Type



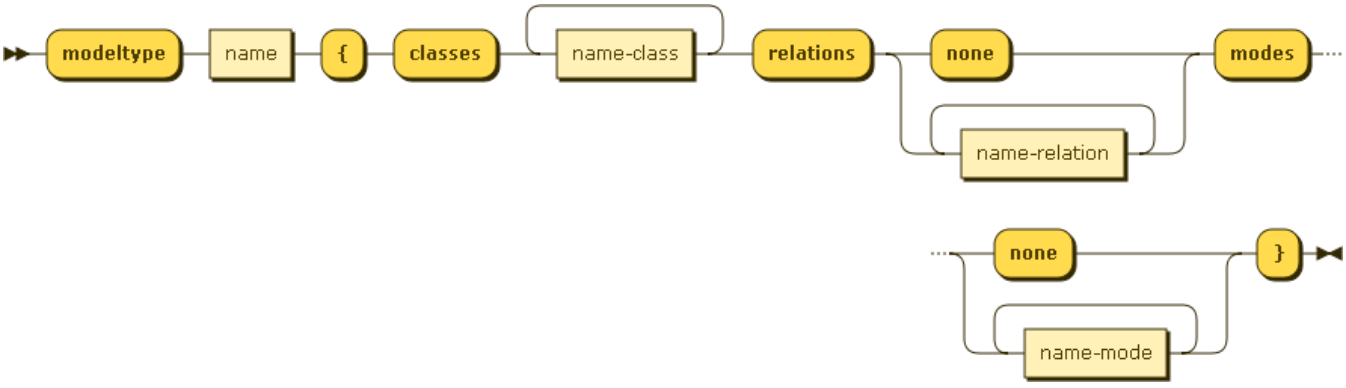

```
enumtype ::= 'enum' name
```

Every enumeration definition starts with a keyword *enum* followed by and identifier.

```
enum MyEnum { "yes" "no" "maybe" }
attribute MyAttr4 : enum MyEnum access:read
```

Figure 7: Example for defining and using enumerations

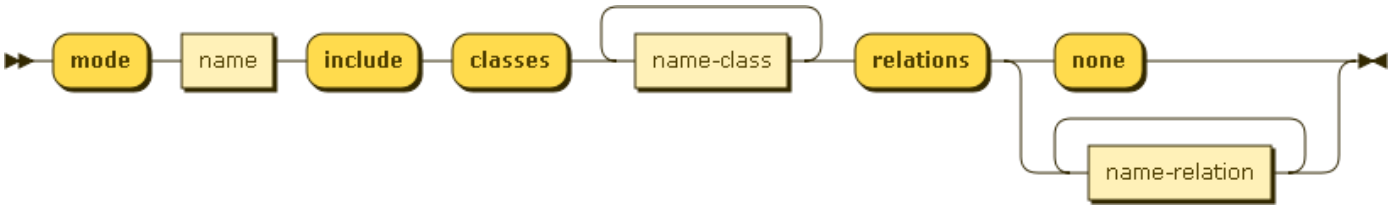
3.20 Model Type




```
modeltype ::= 'modeltype' name
              '{' 'classes' name-class+
              'relations' ( 'none' | name-relation+ )
              'modes' ( 'none' | name-mode+ ) '}'
```

A model type sentence starts with a *modeltype* keyword followed by an identifier. A model type must contain at least a single class. It can also contain relations and modes or views on the model type.

3.21 Mode



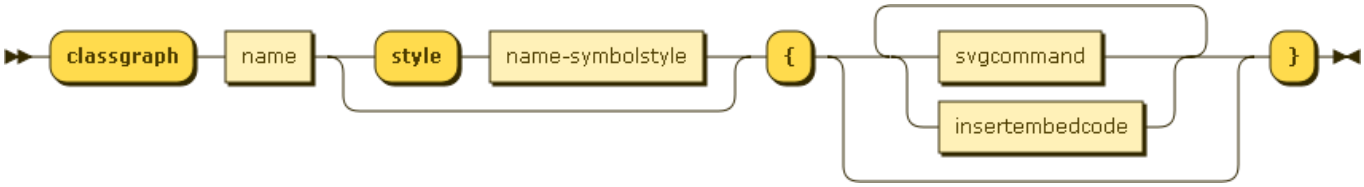

```
mode ::= 'mode' name 'include'
         'classes' name-class+
         'relations' ( 'none' | name-relation+ )
```

The *mode* sentence is always a part of the model type sentence. It starts with the keyword *mode* followed by an identifier and collection of class and relation identifiers following behind the keyword *include*.

```
modeltype NiceModel
{
      classes ClassA ClassB
      relations none
      modes
      mode ModeA include classes ClassA relations none
      mode ModeB include classes ClassB relations none
}
```

Figure 8: Example for 3.26, 3.27

3.22 Symbol Class



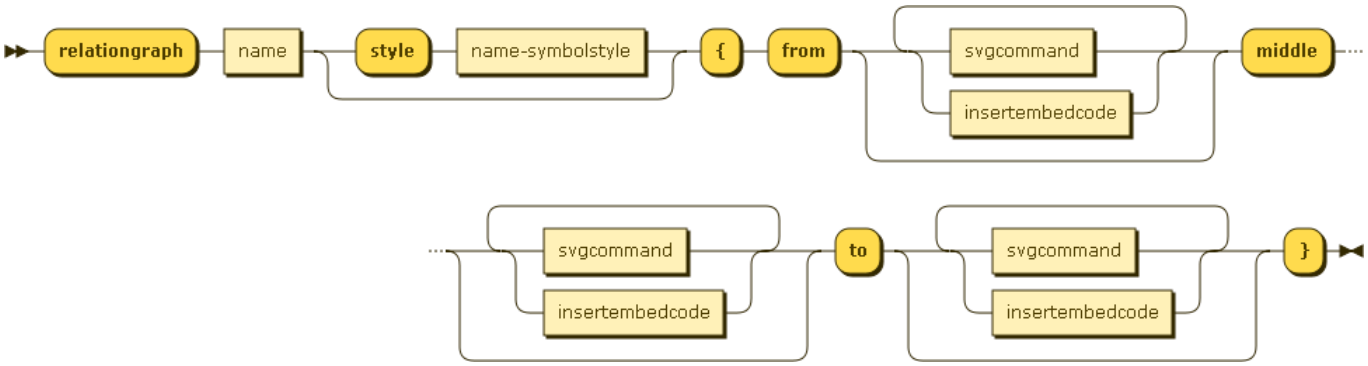

```
symbolclass ::= 'classgraph' name
                ( 'style' name-symbolstyle )? '{'
                ( svgcommand | insertembedcode )* '}'
```

The *symbolclass* sentence always begins with the keyword *classgraph* followed by an identifier. It can contain a global style and a list of simplified SVG commands. Graphical representation defined by this sentence can only be assigned to a class.

```
classgraph MotorGraph style RedBlue {
       rectangle x=-30 y=-30 w = 60 h = 60
       circle cx = 0 cy = 0 r = 30
       ellipse cx = 0 cy = 0 rx = 10 ry = 30
}
```

Figure 9: Example for defining a class symbol

3.23 Symbol Relation



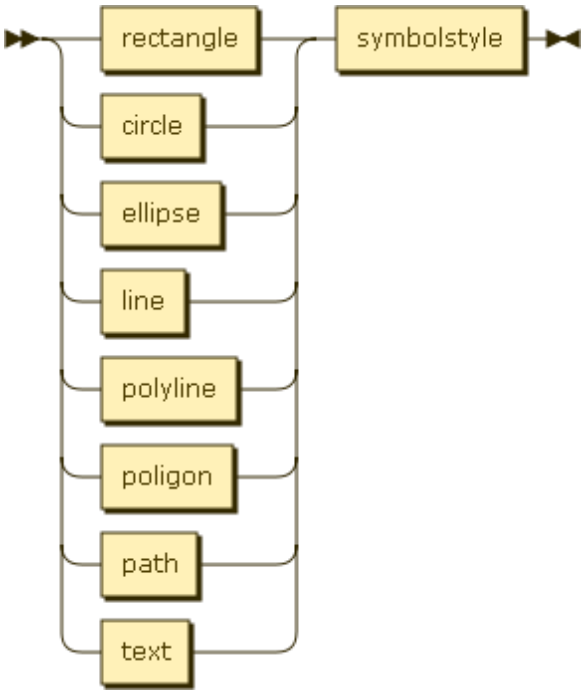

```
symbolrelation ::= 'relationgraph' name
                   ( 'style' name-symbolstyle )? '{'
                   'from' ( svgcommand | insertembedcode )*
                   'middle' ( svgcommand | insertembedcode )*
                   'to' ( svgcommand | insertembedcode )* '}'
```

The *symbolrelation* sentence always begins with the keyword *relationgraph* followed by an identifier. It can contain a global style. It has three sections where one can input a list of simplified SVG commands. The sections starts with the keywords: *from*, *line*, and *to*.

```
relationgraph RelatedGraph {
       from circle cx=0 cy=0 r=2
       middle text "relates to" x=0 y=0
       to polygon points = 0,3 6,0 0,-3
}

…

relation Relates symbol RelatedGraph from Car to Motor {
       attribute description : string access:write
       attribute importance : string access:internal
}
```

Figure 10: Example for defining a relation symbol and referencing it in a relation

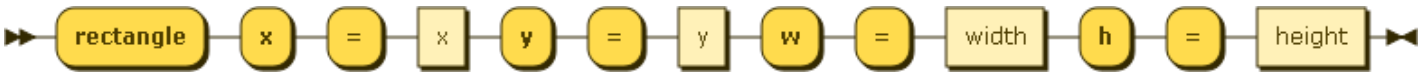
3.24 SVG Command




```
svgcommand ::= ( rectangle
               | circle
               | ellipse
               | line
               | polyline
               | poligon
               | path
               | text )
              symbolstyle
```

SVG command sentence is used to define graphical objects associated with a graphical symbol used either by a class or a relationship. The following simplified SVG graphical objects can be defined: *rectangle*, *circle*, *ellipse*, *line*, *polyline*, *polygon*, *path*, and *text*.

3.25 Rectangle



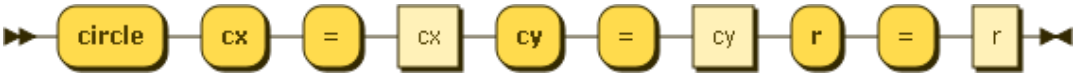

```
rectangle ::= 'rectangle'
              'x' '=' x
              'y' '=' y
              'w' '=' width
              'h' '=' height
```

The *rectangle* sentence starts with a keyword *rectangle* followed by parameters for x, y, width and height. It ends with a *symbolstyle* sub-sentence.

```
rectangle x=-10 y=-10 w=20 h=20 style Black { fill:black stroke:black stroke-width:1}
```

Figure 11: Example for defining a rectangle with a style explicitly specified

3.26 Circle



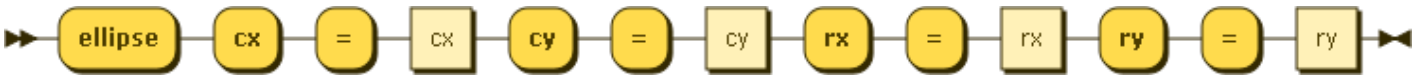

```
Circle ::= 'circle'
           'cx' '=' cx
           'cy' '=' cy
           'r' '=' r
```

The *circle* sentence starts with a keyword *circle* followed by parameters *cx*, *cy*, and *r*. It ends with a *symbolstyle* sub-sentence.

```
circle cx=0 cy=0 r=2 style Black { fill:black stroke:black stroke-width:1 }
```

Figure 12: Example for defining a circle with an explicitly specified symbol style

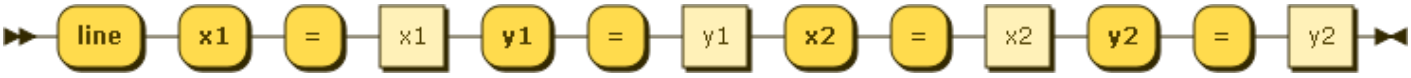
3.27 Ellipse




```
ellipse  ::= 'ellipse'
             'cx' '=' cx
             'cy' '=' cy
             'rx' '=' rx
             'ry' '=' ry
```

The *ellipse* sentence starts with a keyword *ellipse* followed by parameters *cx*, *cy*, *rx*, *ry*. It ends with a *symbolstyle* sub-sentence.

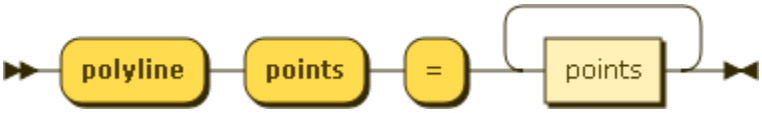
3.28 Line




```
line ::= 'line'
         'x1' '=' x1
         'y1' '=' y1
         'x2' '=' x2
         'y2' '=' y2
```

The *line* sentence starts with a keyword *line* followed by parameters *x1*, *y1*, *x2*, *y2*. It ends with a *symbolstyle* sub-sentence.

3.29 Polyline



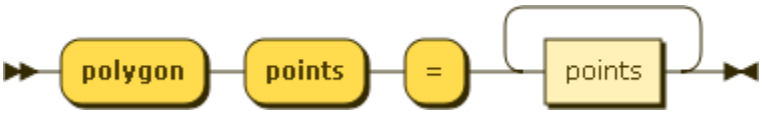

```
polyline ::= 'polyline'
             'points' '=' points+
```

The *polyline* sentence starts with a keyword *polyline* followed by a set of points. It ends with a *symbolstyle* sub-sentence.

```
polyline points=0,0 4,4 1,1 23,4
```

Figure 13: Example for defining a polyline

3.30 Polygon



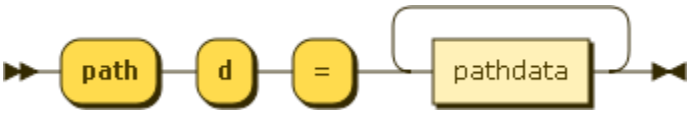

```
polygon  ::= 'polygon'
             'points' '=' points+
```

The *polygon* sentence starts with a keyword *polygon* followed by a set of points. It ends with a *symbolstyle* sub-sentence.

```
polygon points = 0,3 6,0 0,-3 style Red { fill:red stroke:red stroke-width:1 }
```

Figure 14: Example for defining a polygon with explicitly defined symbol style

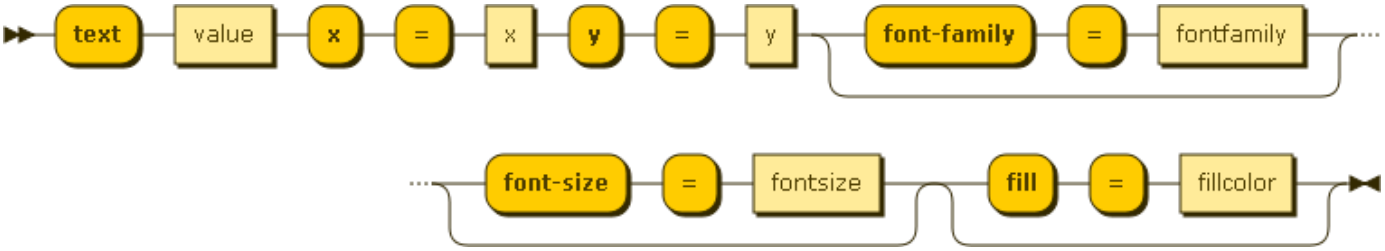
3.31 Path




```
path ::= 'path'
         'd' '=' pathdata+
```

The *path* sentence starts with a keyword *path* followed by *pathdata* parameters. It ends with a *symbolstyle* sub-sentence.

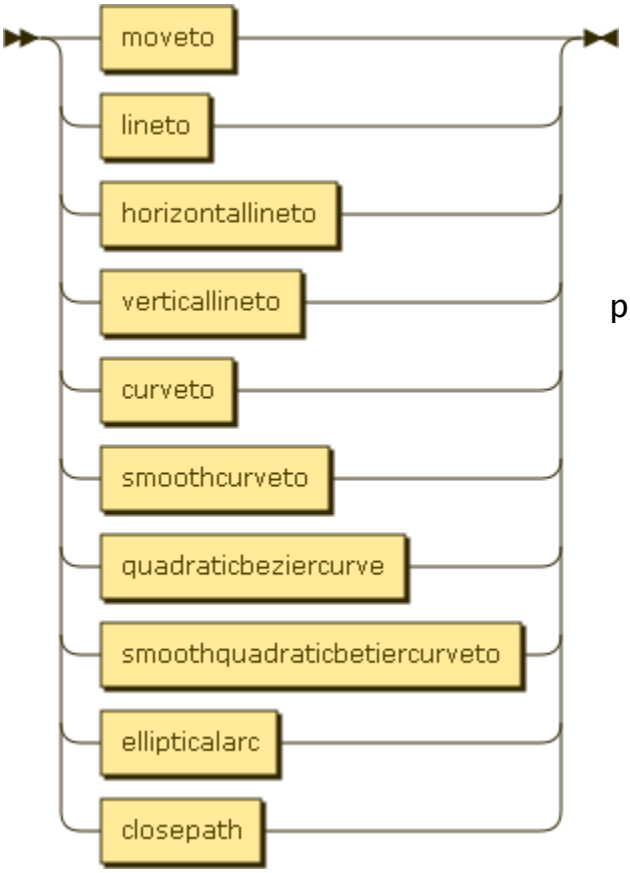
3.32 Text




```
text ::= 'text' value
         'x' '=' x
         'y' '=' y
         ( 'font-family' '=' fontfamily )?
         ( 'font-size' '=' fontsize )?
         ( 'fill' '=' fillcolor )?
```

The *text* sentence starts with a keyword *text* followed by the string value and position parameters *x*, and *y*. Additionally it contains style parameters: *font-family*, *font-size*, and *fill*.

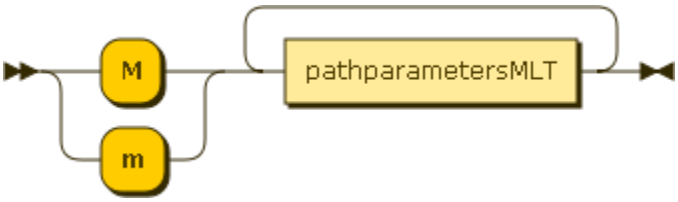
3.33 Path Data




```
pathdata ::= moveto
           | lineto
           | horizontallineto
           | verticallineto
           | curveto
           | smoothcurveto
           | quadraticbeziercurve
           | smoothquadraticbetiercurveto
           | ellipticalarc
           | closepath
```

The *pathdata* sentence contains all the parameters usable when defining the path graphical object (using the path sentence).

3.34 Move To



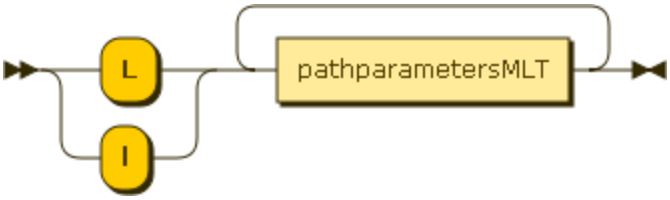

```
moveto ::= ( 'M' | 'm' ) pathparametersMLT+
```

The *moveto* sentence starts with *M* or *m* followed by a parameter list.

```
path d=M 2,3 3,5 -2,3 style Blue { fill:blue stroke:aliceblue stroke-width:2}
```

Figure 15: Example for defining a path with "move to" parameters

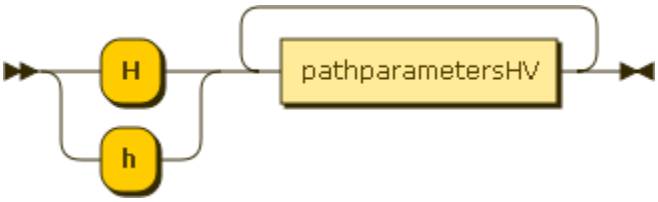
3.35 Line To




```
lineto ::= ( 'L' | 'l' ) pathparametersMLT+
```

The *lineto* sentence starts with *L* or *l* followed by a parameter list.

3.36 Horizontal Line To



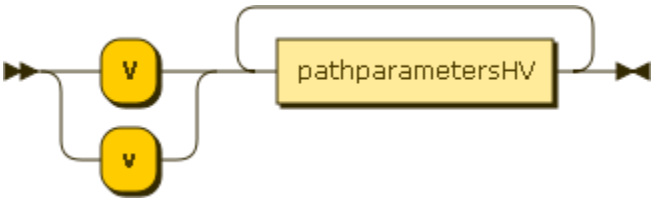

```
horizontallineto ::= ( 'H' | 'h' ) pathparametersHV+
```

The *horizontallineto* sentence starts with *H* or *h* followed by a parameter list.

```
path d=h 2 42 -11
```

Figure 16: Example of a path with "horizontal line to" paremeters

3.37 Vertical Line To



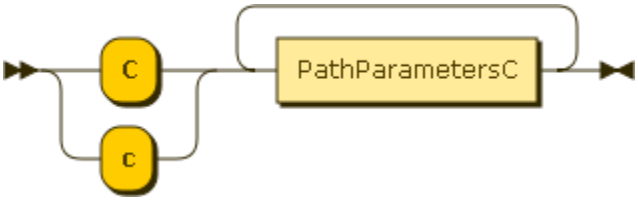

```
verticallineto ::= ( 'V' | 'v' ) pathparametersHV+
```

The *verticallineto* sentence starts with *V* or *v* followed by a parameter list.

```
path d=V 3 23 12 -32 style Blue { fill:blue stroke:aliceblue stroke-width:2}
```

Figure 17: Example of a path with "vertical line to" parameters and explicitly defined style

3.38 Curve To



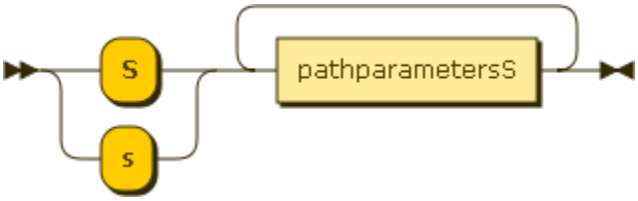

```
curveto ::= ( 'C' | 'c' ) PathParametersC+
```

The *curveto* sentence starts with *C* or *c* followed by a parameter list.

```
path d=c 1 2 3 4 5 6
```

Figure 18: Example of a path with "curve to" parameters

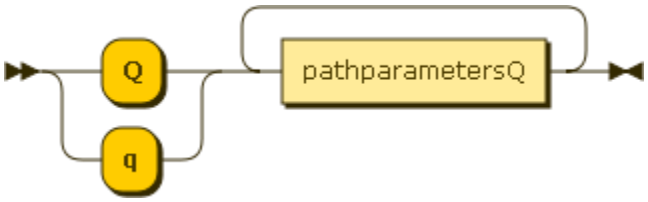
3.39 Smooth Curve To




```
smoothcurveto ::= ( 'S' | 's' ) pathparametersS+
```

The *smoothcurveto* sentence starts with *S* or *s* followed by a parameter list.

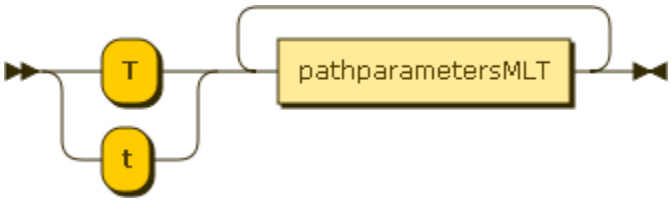
3.40 Quadratic Bezier Curve




```
quadraticbeziercurve ::= ( 'Q' | 'q' ) pathparametersQ+
```

The *quadraticbeziercurve* sentence starts with Q or q followed by a parameter list.

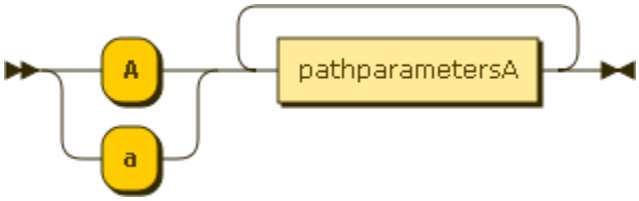
3.41 Smooth Quadratic Bezier Curve To




```
smoothquadraticbeziercurveto ::= ( 'T' | 't' ) pathparametersMLT+
```

The *smoothquadraticbeziercurveto* sentence starts with T or t followed by a parameter list.

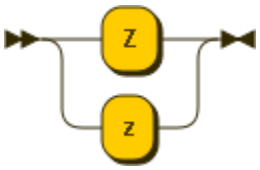
3.42 Elliptical Arc




```
ellipticalarc ::= ( 'A' | 'a' ) pathparametersA+
```

The *ellipticarc* sentence starts with A or a followed by a parameter list.

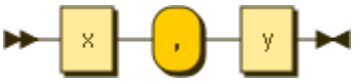
3.43 Close Path




```
closepath ::= 'Z' | 'z'
```

The *closepath* sentence starts indicates the end of a path. It contains only *Z* or *z*.

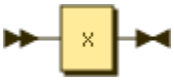
3.44 Points




```
points ::= x ',' y
```

Points is a sentence containing comma-separated parameters *x* and *y*.

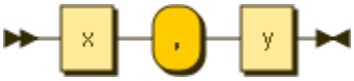
3.45 Path Parameters HV




```
pathparametersHV ::= x
```

The pathparametersHV contains a parameter *x*.

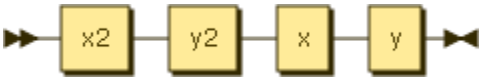
3.46 Path Parameters MLT




```
pathparametersMLT ::= x ',' y
```

The *pathparametersMLT* contains comma-separated parameters *x* and *y*. Because of future considerations, pathdata sentences use this nonterminal instead of *points*.

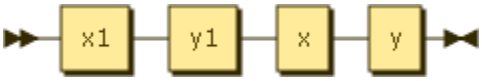
3.47 Path Parameters S




```
pathparametersS ::= x2 y2 x y
```

The *pathparametersS* contains parameters *x2*, *y2*, *x*, and *y* separated by space.

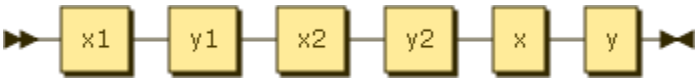
3.48 Path Parameters Q




```
pathparametersQ ::= x1 y1 x y
```

The *pathparametersQ* contains parameters *x1*, *y1*, *x*, and *y* separated by space. Because of future considerations, a different parameter list is used for *quadraticbeziercurve* than for *smoothcurveto*.

3.49 Path Parameters C




```
pathparametersC ::= x1 y1 x2 y2 x y
```

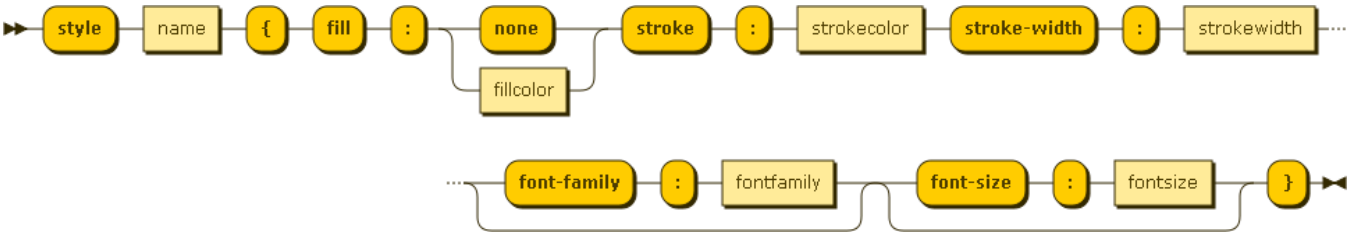
The *pathparametersC* contains parameters x1, y1, x2, y2, x, and y separated by space.

3.50 Path Parameters A


```
pathparametersA ::= rx ',' ry xaxisrot largearcflag sweepflag x y
```

The *pathparametersA* contains parameters *rx*, *ry*, *xaxisrot*, *largearcflag*, *sweepflag*, *x* and *y*.

3.51 Symbol Style



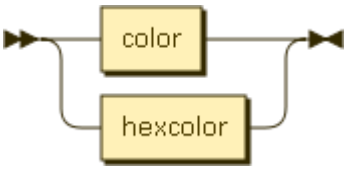

```
symbolstyle ::= 'style' name
                '{' 'fill' ':' ( 'none' | fillcolor )
                'stroke' ':' strokecolor
                'stroke-width' ':' strokewidth
                ( 'font-family' ':' fontfamily )?
                ( 'font-size' ':' fontsize )? '}'
```

The *symbolstyle* sentence defines the graphical style used for representing graphical symbols associated with classes and relations. It can be applied globally on the whole symbol, or individually on the graphical objects from which the symbol is formed. It begins with the keyword *style* followed by the following graphical options: *fill*, *stroke*, *stroke-width*, *font-family*, and *font-size*.

```
style VioletBlack { fill:violet stroke:black stroke-width:2 }
style RedBlue { fill:red stroke:blue stroke-width:4 }
style Custom {fill:#123456 stroke:#ffaaee stroke-width:2 font-family:Georgia font-size:12}
```

Figure 19: Example for 3.57

3.52 Fill Color




```
fillcolor ::= colors
            | hexcolor
```

The *fillcolor* sentence defines the color format typically used in the Symbolstyle sentence. The color format can be either *color*, or *hexcolor*.

3.53 Color


```
color ::= standardHTMLcolors
```

The *color* sentence contains all the standard HTML color names.

3.54 Hex Color


```
hexcolor ::= standardHEXcolor
```

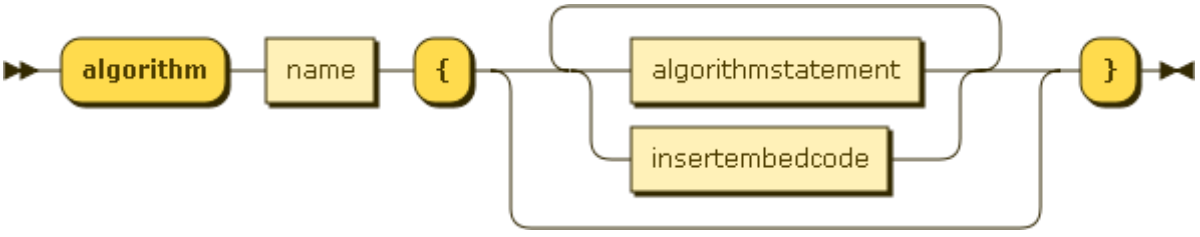
The *hexcolor* sentence defines the standard hexadecimal representation of a color (e.g., #123456)

3.55 Font Family


```
hexcolor ::= standardHEXcolor
```

Defining fonts is allowed either by using well known Windows fonts (enumeration list) or as a string.

3.56 Algorithm



```
algorithm ::= 'algorithm' name '{'
              ( algorithmstatement | insertembedcode )* '}'
```

The *algorithm* sentence starts with the *algorithm* keyword followed by an identifier and a list of statements. At this point statements that are a part of the algorithm definition are under development. Therefore, they are not included in this version of MM-DSL grammar. However, an experiment with statements from a language called Xbase (see the following link for more info: https://github.com/eclipse/xtext/blob/master/plugins/org.eclipse.xtext.xbase/src/org/eclipse/xtext/xbase/Xbase.xtext) is undergoing.

```
algorithm MyAlgorithm
{
	// algorithm statements go here
	ui.infobox "Hello" "Hello World!" ok
	// create and discard a model from model type MyModelType
	model.create MyModel MyModelType
	model.discard MyModel
}
```

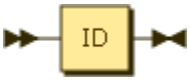
Figure 20: Example for 3.62

3.57 Event


```
event ::= 'event' name '.' 'execute' '.' name-algorithm
```

This statement triggers the referenced algorithm when an event in the modeling tool occurs.

3.58 Name




```
name ::= ID
```

The *name* is always instantiated as an identifier. Identifiers are unique in every MM-DSL program.

4. The MM-DSL Approach to Modeling Method Engineering

The MM-DSL is a great tool for fast prototyping of modeling languages and modeling methods, either by starting from the original idea, or by integrating multiple methods (languages) into a hybrid one. It is a language dedicated to the domain experts and it doesn't require a vast previous programming language experience. On the contrary, it is very easy to learn and use. It also doesn't require that the user has an in depth knowledge of the metamodeling platform utilized to execute the MM-DSL code.

The development of a modeling method with MM-DSL starts with defining an integrated metamodel: *classes*, *relations*, and their correlations. Every *class* and every *relation* can contain *attributes* of various data types (e.g., *int*, *string*), as well as user defined types (currently only definable through *enumeration* attribute type). The integrated metamodel can be later divided into multiple smaller metamodels. We continue by specifying the graphical representation, which can be shared between multiple classes (relations). The graphical representation is a collection of graphical objects (lines and polygons) and it has a specific *style* associated with it. Styles define the general properties of the graphical objects, like line and fill color, line thickness, and text font. Finally, algorithms that work on the previously specified concepts can be implemented. Algorithms interact with previously mentioned concepts and modify their instances (e.g., class and relation instances, attribute values) for the purpose of the qualitative and quantitative model analysis and simulation.

4.1 The Smallest Modeling Method

The smallest possible modeling method that can be translated into a fully functional modeling tool looks like this in MM-DSL code:

```
// the smallest method possible
method SmallestInstantiable

class Abc {}

modeltype ModelTypeA
{
        classes Abc
        relations none
        modes none
}
```

It contains only one class and one model type. To be able to instantiate this code into a graphical modeling tool, MM-DSL assigns predefined graphical representations to every class and every relation that doesn't possess it.

4.2 Integrated Metamodel

All the defined classes and relations initially belong to the same metamodel, called an integrated metamodel. We can assign classes and relations into multiple smaller metamodels called *model types*. We have already seen a simple definition of a model type in the previous MM-DSL code. At least one *model type* needs to be defined to be able to compile the code. However, that doesn't mean that all the classes and relations have to be assigned to it. The following code is valid:

```
// method with enumeration, attributes
method Minimalistic

enum YesNo { "Yes" "No" "Maybe"}
```

```
class ClassA
{
        attribute intAttr : int
        attribute strAttr : string
        attribute enuAttr : enum YesNo
}
class ClassB extends ClassA
{
        attribute intAttr2 : int
}

modeltype NiceModel
{
        classes ClassB
        relations none
        modes none
}
```

4.3 Inheritance, Attribute Types and Enumerations

If we take a closer look, we can see a couple of new language concepts: attributes and enumerations, as well as the concept of inheritance. Currently, only types allowed are *int*, *string*, and user defined *enums*. Enumerations always take the first given value as the default value. In the previous example, default *enum* value would be *"Yes"*. We use inheritance similarly as in almost every object-oriented language (e.g., Java, C#). In our example, *ClassB* inherits attributes of *ClassA*.

4.4 Classes and Relations

In the next example we introduce relations and their association between classes:

```
// introducing relations
method TestMethod_No3

enum YesNo { "Yes" "No" "Maybe" }

class ClassA
{
        attribute intAttrA : int
        attribute strAttrA : string
        attribute enuAttrA : enum YesNo
}
class ClassB extends ClassA
{
        attribute intAttrB : int
}

relation Flow from ClassA to ClassB
{
        attribute intAttrFlow : int
        attribute enumAttrFlow : enum YesNo
}

modeltype NiceModel
{
        classes ClassA ClassB
        relations Flow
        modes none
}
```

Relations, comparing to classes, have some additional properties: *from* and *to*. These properties indicate which class is an outgoing and which class is an ingoing class for the current relation. In our example relation *Flow* is going from *ClassA* to *ClassB*.

4.5 Graphical Representation

The MM-DSL has the possibility not only to define and structure a metamodel of a modeling language, but also the means to define a graphical representation and associate it with different elements of a metamodel. Let us take a look at the following example:

```
// graphical representation of classes and relations
// using symbols an styles
method GraphicalSyntaxPreview

enum Colors { "red" "white" "blue" "black" "yellow" }

style VioletBlack { fill:violet stroke:black stroke-width:2 }
style RedBlue { fill:red stroke:blue stroke-width:4 }

classgraph CarGraph style VioletBlack {
        rectangle x=-30 y=-30 w=60 h=60 style Blue { fill:blue stroke:black stroke-width:2 }
        circle cx=0 cy=0 r=30 style Yellow { fill:yellow stroke:black stroke-width:2 }
        ellipse cx=0 cy=0 rx=30 ry=15
}

classgraph MotorGraph style RedBlue {
        rectangle x=-30 y=-30 w = 60 h = 60
        circle cx = 0 cy = 0 r = 30
        ellipse cx = 0 cy = 0 rx = 10 ry = 30
}

relationgraph RelatedGraph {
        from
        circle cx=0 cy=0 r=2 style Black { fill:black stroke:black stroke-width:1 }
        middle
        text "relates to" x=0 y=0
        to
        polygon points = 0,3 6,0 0,-3 style Red { fill:red stroke:red stroke-width:1 }
}

class Vehicle {
        classattribute alternativeName : string
        attribute velocity:int access:write
        attribute color:enum Colors access:write
        attribute price:int access:read
        attribute year:string
}

class Car extends Vehicle symbol CarGraph {
        attribute numberOfDoors:int access:read
        attribute braksOnAllTiers:enum YesNo access:write
}

class Motor extends Vehicle symbol MotorGraph {
        attribute biggerFrontTire:enum YesNo access:read
        attribute brand:string access:read
}

relation Relates symbol RelatedGraph from Car to Motor {
        attribute description : string access:write
        attribute importance : string access:internal
}

modeltype CarRelationship {
        classes Car Motor
```

```
        relations Relates
        modes none
}
```

We have introduced a couple of new language concepts: *style*, *classgraph* and *relationgraph*. Style defines a general style of the graphical elements which have it assigned (e.g. line color and width of a stroke, fill color). Classgraph defines the graphical representation that can be assigned to any class. The same is true for the relationgraph, with an exception that it can only be associated with relations. Classgraph and relationgraph can contain multiple graphical elements (e.g. circles, rectangles, text, …). Style is applied globally to all the graphical elements inside classgraph (relationgraph), but it is overwritten if a graphical element has its own style associated with it.

4.6 Attribute Access Types

In the previous MM-DSL code, we can see there is one more language concept associated with attributes – *access*. Access influences the modification of an attribute value (it is important to understand that attributes can have values assigned only when the modeling tool is instantiated and used to construct models). *Read* access means that the value cannot be changed, but it can be seen in the modeling tool. *Write* access allows attribute value modification. An attribute with an *internal* access will not be shown in the modeling tool, and users cannot directly modify it.

As we can see from the previous examples, it is very easy and straight forward to code modeling methods in MM-DSL. When the language specification is complete, it will also be possible to write algorithms that can work on the defined metamodel, as well as on the graphical representation concepts.

5. Appendix

5.1 Simple Modeling Method Implementation in MM-DSL

This section illustrates how a modeling method described in MM-DSL code is instantiated using a concrete execution engine – the ADOxx platform.

5.1.1 MM-DSL Code:

```
// a simple modeling method example
method MySimpleCarParkMethod

enum VehicleType { "Automobil" "Motocycle" "Bicycle" }
enum TrueFalse { "True" "False" }

style BlueBlack { fill:blue stroke:black stroke-width:1 }
style RedBlack { fill:red stroke:black stroke-width:1 }

classgraph GarageGraph style BlueBlack {
        rectangle x=-30 y=-30 w=60 h=60 style Green {fill:greenyellow stroke:black stroke-width:2}
        circle cx=-30 cy=30 r=5
        circle cx=-20 cy=30 r=5
        circle cx=-10 cy=30 r=5
        circle cx=0 cy=30 r=5
        circle cx=10 cy=30 r=5
        circle cx=20 cy=30 r=5
        circle cx=30 cy=30 r=5
}

classgraph VehicleGraph style RedBlack {
        circle cx=0 cy=0 r=15
}

relationgraph IsParkedGraph {
        from
        circle cx=0 cy=0 r=2 style Black { fill:black stroke:black stroke-width:1 }
        rectangle x=-3 y=-3 w=6 h=6 style Black { fill:black stroke:black stroke-width:1}
        middle
        text "is parked in" x=0 y=0
        to
        polygon points = 0,3 6,0 0,-3 style Red { fill:red stroke:red stroke-width:1 }
}

class Vehicle symbol VehicleGraph {
        // access type of attributes is implicitly set write
        attribute Type : enum VehicleType
        attribute Description : string
        attribute Color : string
}

class Garage symbol GarageGraph {
        attribute Full : enum TrueFalse
        attribute Description : string
        attribute NumberOfVehicles : int access:read
}

relation IsParked symbol IsParkedGraph from Vehicle to Garage {}

modeltype ParkingGarage {
        classes Vehicle Garage
        relations IsParked
        modes none
}
```

5.1.2 Instantiated Modeling Tool:

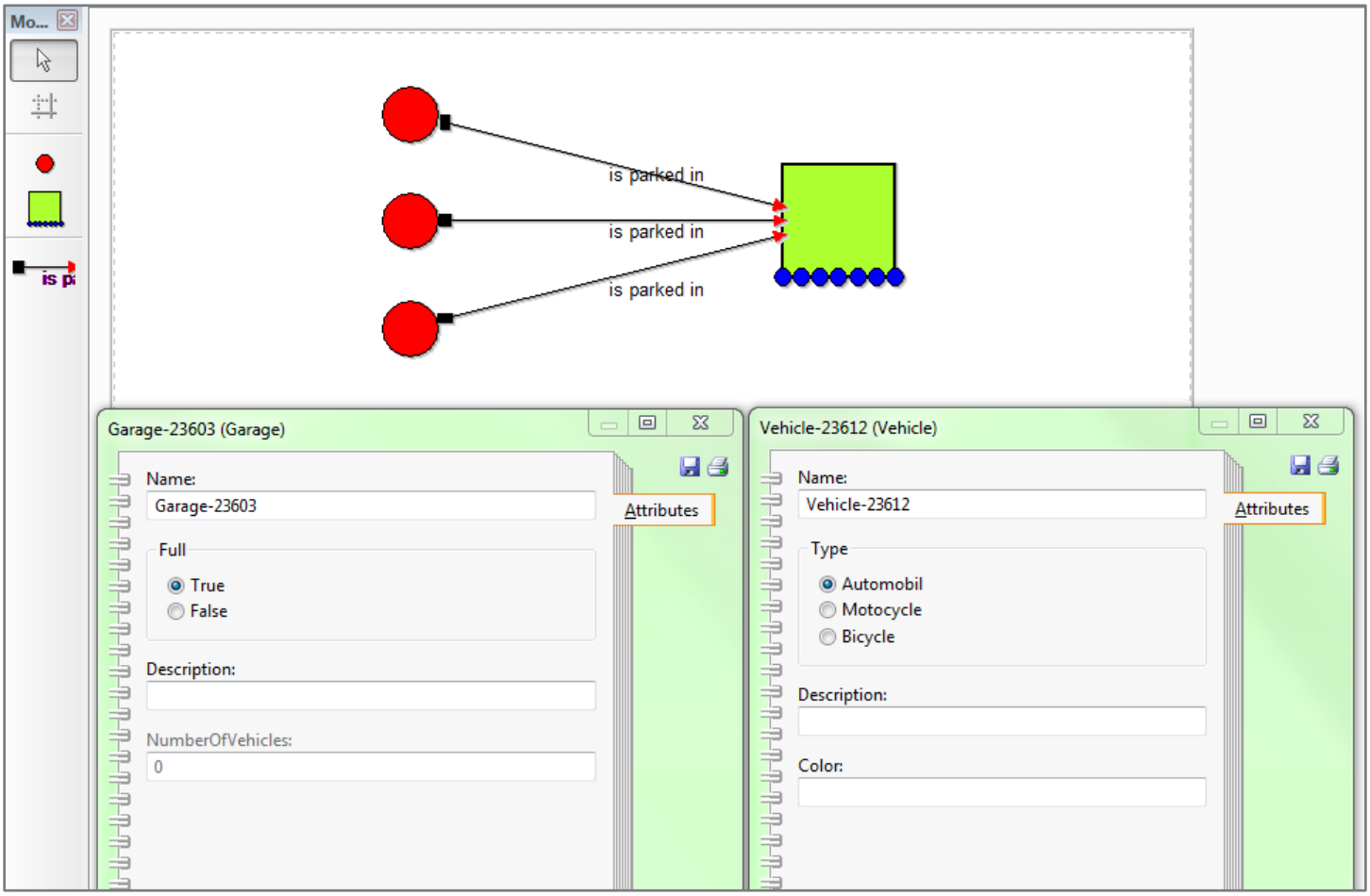


Figure 21: Modeling tool instantiated using the ADOxx platform

The screenshot above is not modified in any way. The modeling tools is completely generated out of the MM-DSL code presented in the previous section.

http://omilab.org

ISBN 978-3-902826-02-2