Including Render Information to SBML Layouts
Version 0.2

Ralph Gauges, Ursula Rost, Sven Sahle and Katja Wegner
European Media Laboratory
Schloss-Wolfsbrunnen Weg 33
69118 Heidelberg
Germany

October 28, 2003

Introduction

This document is meant to complement the SBML Layout extension document presented earlier (http://projects.villa-bosch.de/bcb/sbml/) with render information for the layout objects. The ideas that went into this render extension are mostly the same as described in the layout extension. We wanted this extension to be as flexible as possible. In order for the user to make maximal use of defined render objects across several layouts, we choose to separate the render information completely from the layout information. Given the early stage of this draft, all name tags should be considered preliminary. Any comments and suggestions on those are always welcome.

Namespace

For the render extensions we use the same namespace as for the layout information since we assume that this document will be merged with the layout extension once it has matured enough. It should be specified as follows: xmlns:sl2="http://projects.eml.org/bcb/sbml/level2/".
A SBML file header that would utilize the extension could have the following form:

```
<?xml version="1.0" encoding="UTF-8"?>
   <sbml xmlns:sbml="http://www.sbml.org/sbml/level2" level="2"
         version="1"
         xmlns:sl2="http://projects.eml.org/bcb/sbml/level2"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://projects.eml.org/bcb/sbml/level2
         http://projects.eml.org/bcb/sbml/level2/layout2.xsd">
```

Meta information

Most of the render classes below are derived from a class called SBase which was taken from the SBML Level 2 schema specification (http://www.sbml.org/sbml/level2/version1/). This enables programs to store meta information with the render objects. Most objects do have a unique id of type SId as well.

XML Schema representation:

```
<xsd:complexType name="SBase" abstract="true">
 <xsd:sequence>
  <xsd:element name="notes" minOccurs="0">
   <xsd:complexType>
```

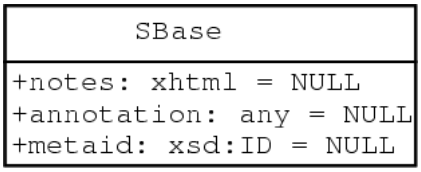

Figure 1: SBase class

```
    <xsd:sequence>
     <xsd:any namespace="http://www.w3.org/1999/xhtml"
              processContents="skip"
              maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
  <xsd:element name="annotation" minOccurs="0">
   <xsd:complexType>
    <xsd:sequence>
     <xsd:any processContents="skip" maxOccurs="unbounded"/>
    </xsd:sequence>
   </xsd:complexType>
  </xsd:element>
 </xsd:sequence>
 <xsd:attribute name="metaid" type="xsd:ID" use="optional"/>
</xsd:complexType>

<xsd:simpleType name="SId">
 <xsd:restriction base="xsd:string">
  <xsd:pattern value="(_|[a-z]|[A-Z])(_|[a-z]|[A-Z]|[0-9])*"/>
 </xsd:restriction>
</xsd:simpleType>
```

Overall structure

As stated above, the render information has been completely separated from the layout information. It will have a separate tag called **r**ender. There is just one render object per file which contains the rendering information as a listOfLayouts which can contain one or more layouts. The render tag also holds two optional attributes called background and foreground. They are both of type Color and can be used to specify a default background and a default foreground color for all render objects. (For the specification of the Color type see the next section.) Each layout object holds several lists: A listOfFilltypes for a list of possible filltypes, a listOfLinetypes holds all the

different line types, a listOfColors will hold a number of predefined color names which are easier to remember than RGBA values, a listOfShapes for all the complex and primitive shapes that will eventually be rendered and a listOfRenderGroups. All shapes are derived from a base class Shape. Shapes defined so far are curves, text labels, bitmaps and several forms of primitive shapes like ellipses, rectangle and triangle. The listOfRenderGroups hold groups of render objects that have been defined in the listOfShapes; this lets the user compose complex render objects from simple shapes. Those render groups can than be referenced by the layout part together with a transformation object that is to be used on the render object.

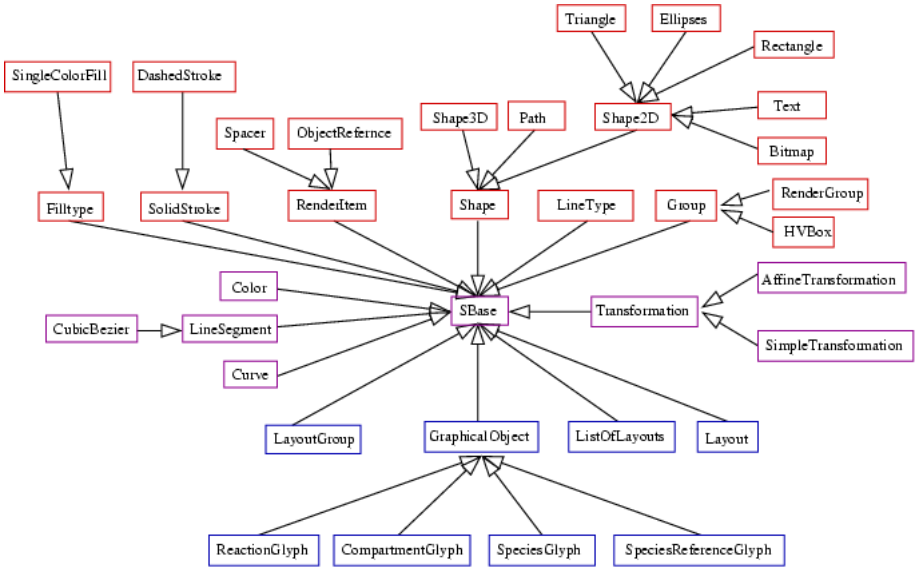

Figure 2: Inheritance tree for the layout and render classes

XML Schema representation:

```
<xsd:complexType name="ListOfLinetypes">
 <xsd:complexContent>
  <xsd:extension base="sl2:SBase">
   <xsd:sequence>
    <xsd:element name="linetype" type="sl2:Linetype"
                 maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ListOfFilltypes">
 <xsd:complexContent>
```

```
  <xsd:extension base="sl2:SBase">
   <xsd:sequence>
    <xsd:element name="filltype" type="sl2:Filltype" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ListOfShapes">
 <xsd:complexContent>
  <xsd:extension base="sl2:SBase">
   <xsd:sequence>
    <xsd:element name="shape" type="sl2:Shape" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ListOfRenderGroups">
 <xsd:complexContent>
  <xsd:extension base="sl2:SBase">
   <xsd:sequence>
    <xsd:element name="group" type="sl2:RenderGroup" maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Render">
  <xsd:sequence>
    <xsd:element name="listOfColors" type="sl2:ListOfColors" minOccurs="0"/>
    <xsd:element name="listOfLinetypes" type="sl2:ListOfLinetypes" minOccurs="0"/>
    <xsd:element name="listOfShapes" type="sl2:ListOfShapes" minOccurs="0"/>
    <xsd:element name="listOfRenderGroups" type="sl2:ListOfRenderGroups" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="background" type="sl2:Color" use="optional">
   <xsd:attribute name="foreground" type="sl2:Color" use="optional">
</xsd:complexType>
```

Colors

A Color object has 5 attributes. The id attribute is used to give a name to the color object so it can be referenced from other objects. The red, green and blue attributes specify the value for the red, green and blue channel respectively. They are integers values in the range of 0 to 255. The alpha attribute is also an integer value in the range of 0 to 255 and specifies the transparency of the color. 0 meaning fully transparent and 255 meaning opaque. The default value for the alpha attribute is 255 (fully opaque).

XML Schema representation:

```
<xsd:simpleType name="ColorChannel">
 <xsd:restriction base="xsd:integer">
  <xsd:minInclusive value="0"/>
  <xsd:maxInclusive value="255"/>
```

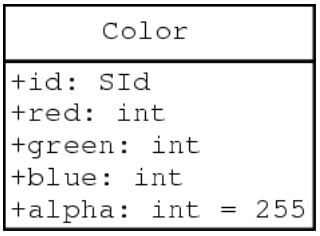

Figure 3: Color class

```
 </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="Color">
 <xsd:complexContent>
  <xsd:extension base="sl2:SBase">
   <xsd:attribute name="id" type="sl2:SId"/>
   <xsd:attribute name="red" type="sl2:ColorChannel"/>
   <xsd:attribute name="green" type="sl2:ColorChannel"/>
   <xsd:attribute name="blue" type="sl2:ColorChannel"/>
   <xsd:attribute name="alpha" type="sl2:ColorChannel" use="optional" default="255"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Linetypes

The listOfLinetypes holds zero or more Linetype objects. A linetype object has an attribute for the thickness of the line and one for the stroke. So far we have defined two types of strokes. A SolidStroke which is a normal solid line and a DashedStroke which is a line made up of little line segments of variable length interrupted by spaces of variable length. The length of the dashes and spaces is specified by the sequence attribute which consists of a list of double values. The values alternately correspond to the length of a dash followed by the length of a space, followed by the length of the next dash and so on. If the line that is drawn with such a dashed stroke is longer than the sequence given, the pattern is repeated until the full length of the line is reached. The sequence must consists of at least two values since having only one value would correspond to a solid line and would therefor not make much sense.

The shapes are normally just drawn as an outline. In order to be able to fill a shape we have defined one filltype called SingleColorFill. The only attributes are an id to reference the filltype from other objects and the color with which the shape should be filled. Other filltypes that might be useful

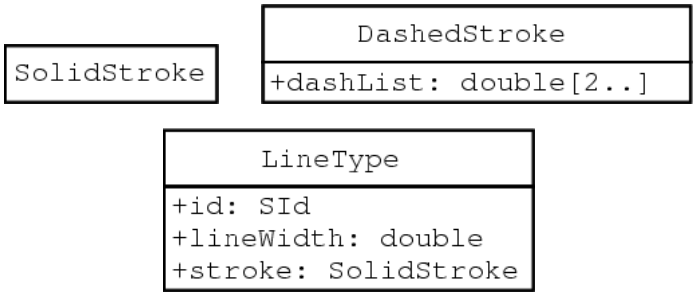

Figure 4: Strokes and Linetypes

are a gradient and a pattern filltype, but those will only be defined if there is an actual need for them.

Figure 5: Filltypes

XML Schema representation:

```
<xsd:complexType name="StrokeType" abstract="true">
 <xsd:complexContent>
  <xsd:extension base="sl2:SBase">
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="SolidStroke" abstract="true">
 <xsd:complexContent>
  <xsd:extension base="sl2:StrokeType">
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="DashedStroke" abstract="true">
 <xsd:complexContent>
  <xsd:extension base="sl2:StrokeType">
    <xsd:attribute name="sequence">
      <xsd:simpleType>
        <xsd:list itemType="xsd:double" minLength="2" maxLength="unbounded"/>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Linetype">
 <xsd:complexContent>
  <xsd:extension base="sl2:SBase">
```

```
   <xsd:sequence>
     <xsd:element name="stroke" type="sl2:StrokeType"/>
   </xsd:sequence>
   <xsd:attribute name="id" type="sl2:SId"/>
   <xsd:attribute name="lineWidth" type="xsd:double"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Filltype" abstract="true">
 <xsd:complexContent>
  <xsd:extension base="sl2:SBase">
   <xsd:attribute name="id" type="sl2:SId"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="SingleColorFill">
 <xsd:complexContent>
  <xsd:extension base="sl2:Filltype">
   <xsd:attribute name="color" type="sl2:Color"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Shapes

This will probably be the most difficult part of the render extension and we hope to get much feedback as to what shapes others need for their applications. As stated above all shapes will be derived from a base type called Shape which extends SBase and has a unique id of type SId. Further on we derive two more classes namely Shape2D and Shape3D which will form the basis for all 2D and 3D shapes to come. Since probably very few people are considering 3D layout for their applications, we will concentrate an 2D specific things first. The 2D shapes that we have come up with so far are curves, text labels, bitmaps, ellipses, rectangles and triangles. Of those, the curves are somewhat different since they are not 2D but apply to 3D as well, therefore we derived the Path object directly from Shape. Some issues that we will have to think about sooner or later are:

- Should text and bitmap objects be scaled at all?
- How should 2D layout programs treat 3D objects?
- How should 2D objects be treated in an 3D environment?

XML Schema representation:

```
<xsd:complexType name="Shape" abstract="true">
  <xsd:complexContent>
```

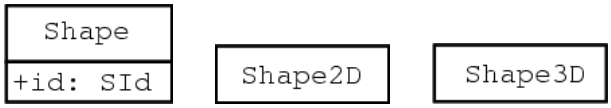

Figure 6: Abstract shape baseclasses

```
    <xsd:extension base="sl2:SBase">
      <xsd:attribute name="id" type="sl2:SId"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Shape2D" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="sl2:Shape">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Shape3D" abstract="true">
  <xsd:complexContent>
    <xsd:extension base="sl2:Shape">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Ellipses

The ellipses is a subclass of Shape2D. Additional attributes for the ellipses are linecolor, linetype and filltype as well as the x and y radius given in the xr and yr attribute. The attributes for linecolor, linetype and filltype are of type Color, Linetype and Filltype respectively and all three attributes are optional. If no linecolor is given, the default foreground color should be used. The default linetype is a SolidStroke with width 1. If filltype is omitted, the shape is not to be filled.

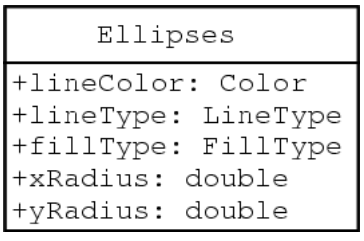

Figure 7: Ellipses class

```
<xsd:complexType name="Ellipses">
```

```
  <xsd:complexContent>
    <xsd:extension base="sl2:Shape2D">
        <xsd:sequence>
            <xsd:element name="linecolor" type="sl2:Color" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="linetype" type="sl2:Linetype" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="filltype" type="sl2:Filltype" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="xr" type="xsd:double"/>
        <xsd:attribute name="yr" type="xsd:double"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Triangle

The triangle shape is defined by three points in 2D space which are specified by the x1,y1, x2,y2 and x3,y3 attributes. The three points are given relative to the position of the corresponding glyph object from the layout. The additional attributes for linecolor, linetype and filltype are treated the same as for the ellipses (see above).

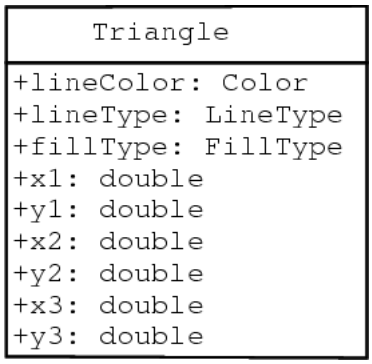

Figure 8: Triangle class

```
<xsd:complexType name="Triangle">
  <xsd:complexContent>
    <xsd:extension base="sl2:Shape2D">
        <xsd:sequence>
            <xsd:element name="linecolor" type="sl2:Color" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="linetype" type="sl2:Linetype" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="filltype" type="sl2:Filltype" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="x1" type="xsd:double"/>
        <xsd:attribute name="y1" type="xsd:double"/>
        <xsd:attribute name="x2" type="xsd:double"/>
        <xsd:attribute name="y2" type="xsd:double"/>
        <xsd:attribute name="x3" type="xsd:double"/>
        <xsd:attribute name="y3" type="xsd:double"/>
    </xsd:extension>
```

```
  </xsd:complexContent>
</xsd:complexType>
```

Rectangle

The rectangle objects has two attributes called width and height which specify the size of the rectangle. The linecolor, linetype and filltype attributes are also treated the same way as for the ellipses (see above). We added another two attributes for people that would like to have rounded edges on their rectangles. The rx and ry attributes specify the radius of the rectangles edges. A low value would correspond to only slightly rounded edges, whereas two high values would convert the rectangle into an ellipses. The rx and ry attribute are of type double are optional and if not specified they default to 0.0 (normal edges). As in SVG, if only one is given, they are assumed to be equal. An additional constraint is that rx may not exceed half the width of the rectangle and ry likewise may not exceed half the height of the rectangle. Even if those values are given, they can easily be ignored if the program does not support rounded edges.

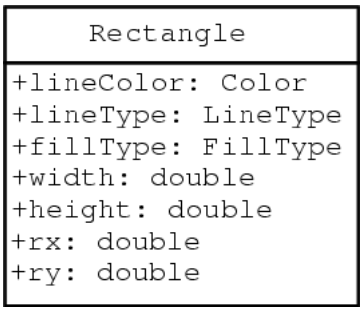

Figure 9: Rectangle class

```
<xsd:complexType name="Rectangle">
  <xsd:complexContent>
    <xsd:extension base="sl2:Shape2D">
        <xsd:sequence>
            <xsd:element name="linecolor" type="sl2:Color" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="linetype" type="sl2:Linetype" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="filltype" type="sl2:Filltype" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="width" type="xsd:double"/>
        <xsd:attribute name="height" type="xsd:double"/>
        <xsd:attribute name="rx" type="xsd:double" use="optional" default="0.0"/>
        <xsd:attribute name="ry" type="xsd:double" use="optional" default="0.0"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Text

The text object has an optional linecolor attribute of type Color. If omitted, the default foreground color is to be used. Additionally the text object has an attribute called family which is either serif, sansserif or monospaced. the size attribute specifies the size of the font in points. The weight attribute is optional and is either normal (default) or bold. The same goes for the style attribute, it is optional as well and either normal (default) or italic. The actual text belonging to the text object is given as UTF-8 encoded Unicode string between the opening and the closing text tag. This corresponds much to the way text is handled in SVG. Unfortunately even in SVG, there is no way (at least I could not find one) to make the size of an object dependent on the size of another object. That is, you can not specify that a certain shape should be surrounded by some other object, like a rectangle. So in order to have some text surrounded by a rectangular box, the size of the box would have to be fixed. In some cases, this could well mean that the text will not fit into the box on some other system because of the font properties being slightly different. Unfortunately we think that this is something we will have to live with, but any suggestions as to how we could do something like that are always welcome. There also is an ImplicitText object which is derived from text. The difference between a text label and an implicit text label is that the actual text belonging to the object is not given with the object, but is derived from the id given by the reference attribute. If the object connected with the id has a name attribute, this is to be used as the actual text, otherwise, the id itself is the textual representation. This scheme is very simple and may not be sufficient for all situations. A somewhat more elaborate way of handling implicit text would be to specify the text by an object id together with an attribute name. For example to specify a textual representation for the initial amount of some species, one would give the id of the species and the keyword *initalAmount*.

```
<xsd:SimpleType name="FontFamily">
 <restriction base="string">
   <xsd:enumeration value="serif"/>
   <xsd:enumeration value="sansseerif"/>
   <xsd:enumeration value="monospaced"/>
 </restriction>
</xsd:SimpleType

<xsd:SimpleType name="FontWeight">
 <restriction base="string">
   <xsd:enumeration value="normal"/>
   <xsd:enumeration value="bold"/>
 </restriction>
</xsd:SimpleType

<xsd:SimpleType name="FontStyle">
```

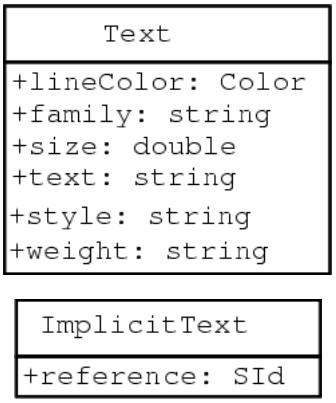

Figure 10: Text label classes

```
 <restriction base="string">
   <xsd:enumeration value="normal"/>
   <xsd:enumeration value="italic"/>
 </restriction>
</xsd:SimpleType

<xsd:complexType name="Text">
  <xsd:complexContent>
    <xsd:extension base="sl2:Shape2D">
        <xsd:sequence>
            <xsd:element name="linecolor" type="sl2:Color" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
        <xsd:attribute name="family" type="sl2:FontFamily"/>
        <xsd:attribute name="size" type="xsd:double"/>
        <xsd:attribute name="weight" type="sl2:FontWeight" use="optional" default="normal"/>
        <xsd:attribute name="style"  type="sl2:FontStyle" use="optional" default="normal"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ImplicitText">
  <xsd:complexContent>
    <xsd:extension base="sl2:Text">
        <xsd:attribute name="reference" type="sl2:SId"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Path

The Path object consists of a curve, which itself consists of a list of line segments. In addition, the user can specify a linecolor and a linetype which are both optional. If not given, the default foreground color will be used as the line color and a SolidStroke with width 1 will be used as the line type. The **curve** tag contains a **listOfCurveSegments** which as the name

already suggests contains an arbitrary number of curve segments. For now we provide the definitions for two types of curve segments (**LineSegment** and **CubicBezier**) but leave it open if this should in future be restricted to only one type or even generalized to more different line types. All segment types, which is just CubicBezier so far, are derived from the LineSegment type. The type of the curve segment has to be specified with a **xsi:type** attribute in the **curveSegment** tag.

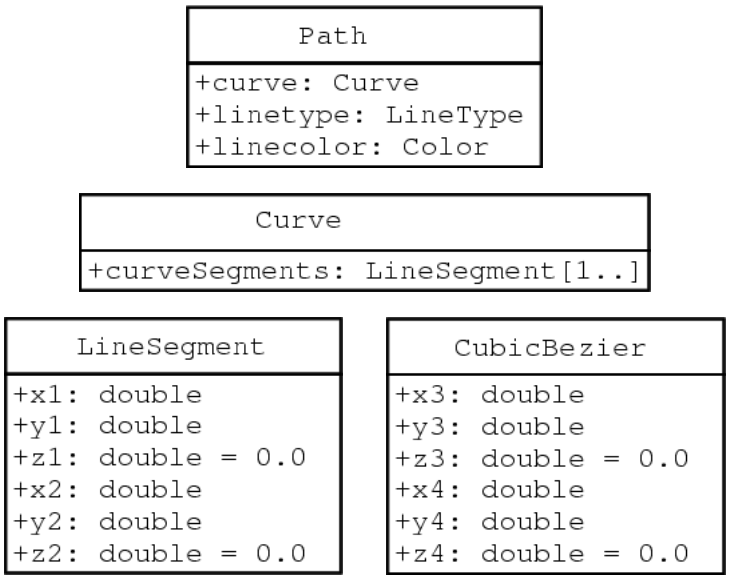

Figure 11: Curve classes

```
<xsd:complexType name="LineSegment">
  <xsd:complexContent>
    <xsd:extension base="sl2:SBase">
      <xsd:attribute name="x1" type="xsd:double"/>
      <xsd:attribute name="y1" type="xsd:double"/>
      <xsd:attribute name="z1" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="x2" type="xsd:double"/>
      <xsd:attribute name="y2" type="xsd:double"/>
      <xsd:attribute name="z2" type="xsd:double" use="optional" default="0.0"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="CubicBezier">
  <xsd:complexContent>
    <xsd:extension base="sl2:LineSegment">
      <xsd:attribute name="x3" type="xsd:double"/>
      <xsd:attribute name="y3" type="xsd:double"/>
      <xsd:attribute name="z3" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="x4" type="xsd:double"/>
      <xsd:attribute name="y4" type="xsd:double"/>
```

```
      <xsd:attribute name="z4" type="xsd:double" use="optional" default="0.0"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="ListOfCurveSegments">
  <xsd:complexContent>
    <xsd:extension base="sl2:SBase">
      <xsd:sequence>
        <xsd:element name="curveSegment" type="sl2:LineSegment"
                     minOccurs="1" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Curve">
  <xsd:complexContent>
    <xsd:extension base="sl2:SBase">
      <xsd:sequence>
        <xsd:element name="listOfCurceSegments" type="sl2:ListOfCurveSegments"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Path">
  <xsd:complexContent>
    <xsd:extension base="sl2:Shape">
        <xsd:sequence>
            <xsd:element name="Curve" type="sl2:ListOfCurves" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="linetype" type="sl2:LineType" minOccurs="0" maxOccurs="1"/>
            <xsd:element name="linecolor" type="sl2:Color" minOccurs="0" maxOccurs="1"/>
        </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Bitmap

For the time being, the bitmap objects only attribute is the name of a bitmap file.

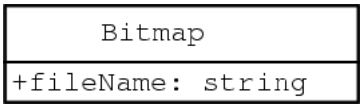

Figure 12: Bitmap class

```
<xsd:complexType name="Bitmap">
  <xsd:complexContent>
    <xsd:extension base="sl2:Shape2D">
        <xsd:attribute name="filename" type="xsd:string"/>
    </xsd:extension>
```

```
  </xsd:complexContent>
</xsd:complexType>
```

Render Groups

A render group is made up of one or more ObjectReference objects. The ObjectReference object itself consists of a reference to either a shape object defined in the listOfShapes or a reference to another Group object which can be either another RenderGroup or a HVBox (see below). Users should be careful not to create loops here by having objects reference themselves directly or indirectly. Additionally the ObjectReference contains a transformation that is to be used on the object. There are two types of transformations so far; one is called SimpleTransformation and allows for translation, rotation and scaling of the object. A more elaborate transformation is the AffineTransformation which lets the user specify a 4x3 transformation matrix. (see http://mathworld.wolfram.com/AffineTransformation.html) In essence, the affine transformation gives you some additional transformations that you can not do with the SimpleTransformation, but for 99% of the users, the SimpleTransformation should suffice. All attributes for both transformations are optional and default to the identity transformation.

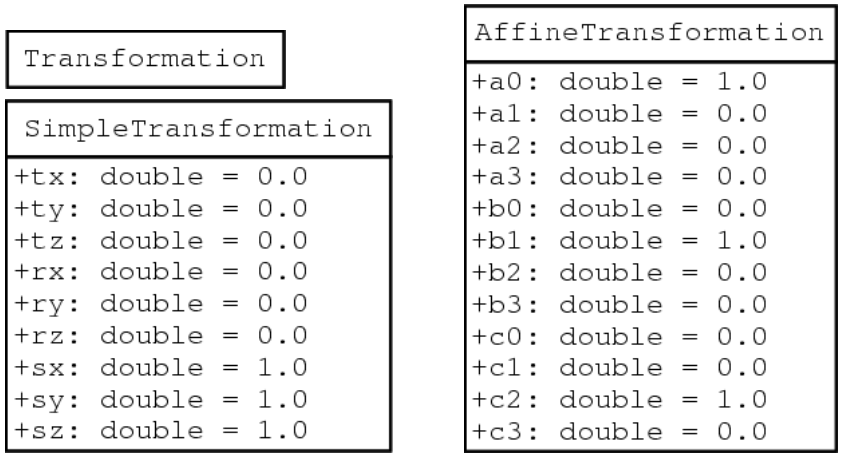

Figure 13: Transformation classes

XML Schema representation:

```
<xsd:complexType>
  <xsd:complexContent>
    <xsd:extension base="sl2:SBase">
    </xsd:extension>
  </xsd:complexContent>
```

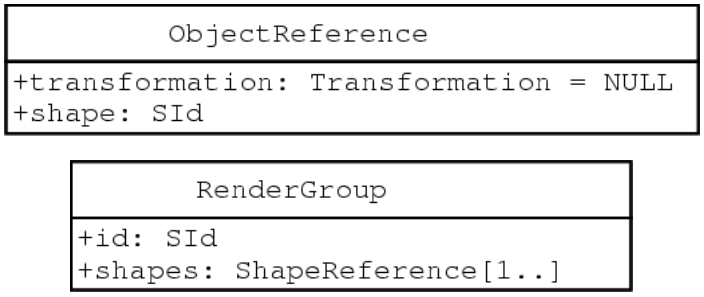

Figure 14: Absolute layout classes

```
</xsd:complexType>

<xsd:complexType name="SimpleTransformation">
  <xsd:complexContent>
    <xsd:extension base="sl2:Transformation">
      <xsd:attribute name="tx" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="ty" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="tz" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="rx" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="ry" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="rz" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="sx" type="xsd:double" use="optional" default="1.0"/>
      <xsd:attribute name="sy" type="xsd:double" use="optional" default="1.0"/>
      <xsd:attribute name="sz" type="xsd:double" use="optional" default="1.0"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="AffineTransformation">
  <xsd:complexContent>
    <xsd:extension base="sl2:Transformation">
      <xsd:attribute name="a0" type="xsd:double" use="optional" default="1.0"/>
      <xsd:attribute name="a1" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="a2" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="a3" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="b0" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="b1" type="xsd:double" use="optional" default="1.0"/>
      <xsd:attribute name="b2" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="b3" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="c0" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="c1" type="xsd:double" use="optional" default="0.0"/>
      <xsd:attribute name="c2" type="xsd:double" use="optional" default="1.0"/>
      <xsd:attribute name="c3" type="xsd:double" use="optional" default="0.0"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="RenderItem">
  <xsd:complexContent>
    <xsd:extension base="sl2:SBase">
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

```
<xsd:complexType name="ObjectReference">
  <xsd:complexContent>
    <xsd:extension base="sl2:RenderItem">
    <xsd:sequence>
       <xsd:choice>
         <xsd:element name="transformation" type="sl2:Transformation"/>
         <xsd:element name="transformation" type="sl2:AffineTransformation"/>
       </xsd:choice>
    </xsd:sequence>
      <xsd:attribute name="object" type="sl2:SId"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:SimpleType name="RenderTypes">
 <restriction base="string">
   <xsd:enumeration value="species"/>
   <xsd:enumeration value="compartment"/>
   <xsd:enumeration value="reaction"/>
   <xsd:enumeration value="speciesreference"/>
 </restriction>
</xsd:SimpleType>

<xsd:complexType name="Group">
  <xsd:extension base="sl2:SBase">
    <xsd:attribute name="id" type="sl2:SId"/>
    <xsd:attribute name="defaultType" type="sl2:RenderTypes" use="optional"/>
  </xsd:extension>
</xsd:complexType>

<xsd:complexType name="RenderGroup">
 <xsd:complexContent>
  <xsd:extension base="sl2:Group">
   <xsd:sequence>
    <xsd:element name="item" type="sl2:ObjectReference" minOccurs="1"
                 maxOccurs="unbounded"/>
   </xsd:sequence>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>
```

Relative Object Layout

So far, objects can only be laid out with absolute positioning, but sometimes you might want to lay out objects relative to each other. For example you might want to place a text label below a bitmap representation of a species. Since the bitmap object does not specify the size of the bitmap, you can not do this with absolute positioning. For cases like that, we introduce two new simple objects, an HVBox and a Spacer object. The HVBox is a subclass of Group and has two additional attributes as well as a list of objects of type RenderItem. The first attribute is called direction and specifies the direction in which the components of the HVGroup are to be laid out. The direction can be horizontal or vertical (default). A HVBox with horizontal layout will place its components beside each other in one row, an HVBox

with direction vertical will place the components in a column one below the next. This layout concept is present in most GUI toolkits and should be easy to implement. The other attribute is called alignment and is a double value in the range of 0.0 to 1.0. This value specifies how the individual components are to be aligned in the direction perpendicular to the Boxes direction. E.g. if you have a HVBox with direction vertical and specify an alignment value of 0.5, the box will be as wide as the widest component and all components are centered in the horizontal direction. A value of 0.0 would mean they are right aligned and a value of 1.0 would have them left aligned. All components can still have a transformation, but the translational part of the transformation is ignored. The spacer object is just what the name suggests an object to create some space between components of an HVBox. E.g. if you want to have a 10pt space between all the components of the Box, you just add a spacer with a size value of 10 between each pair of objects.

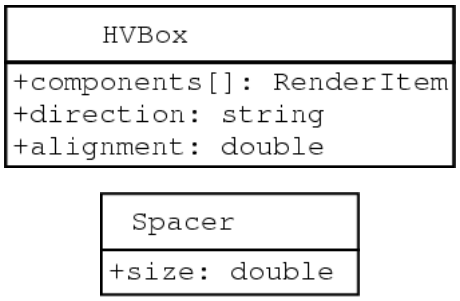

Figure 15: Relative layout classes

```
<xsd:simpleType name="HVBoxDirection">
 <restriction base="string">
   <xsd:enumeration value="horizontal"/>
   <xsd:enumeration value="vertical"/>
 </restriction>
</xsd:simpleType>

<xsd:simpleType name="AlignmentRange">
 <xsd:restriction base="xsd:double">
  <xsd:minInclusive value="0.0"/>
  <xsd:maxInclusive value="1.0"/>
 </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="HVBox">
 <xsd:complexContent>
  <xsd:extension base="sl2:Group">
   <xsd:sequence>
     <xsd:element name="item" type="sl2:RenderItem" minOccurs="1"
                  maxOccurs="unbounded"/>
   </xsd:sequence>
```

```
   <xsd:attribute name="direction" type="sl2:HVBoxDirection" use="optional"
                  default="vertical"/>
   <xsd:attribute name="alignment" type="sl2:AlignmentRange" use="optional"
                  default="0.0"/>
  </xsd:extension>
 </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="Spacer">
    <xsd:complexContent>
    <xsd:extension base="sl2:RenderItem">
        <xsd:attribute name="size" type="xsd:double"/>
    </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Connection to Layout

In the documentation to the layout part of this extension we described, that the layout contains compartmentGlyphs, reactionGlyphs and speciesReferenceGlyphs as well as additionalGraphicalObjects. In order to link the render information to the layout, all those objects have been extended by an attribute called renderGroup which would reference the id of the corresponding renderGroup. Additionally we added a transformation object to all the glyphs mentioned above as to allow a transformation to be used on the renderGroup. We put this transformation here rather than in the renderGroup itself because this would allow the user to use the same render group with several glyphs but at different sizes, rotations and positions etc.

```
<listOfLayouts>
    .
    .
    .
    <speciesGlyph id="ATP_Glyph" species="ATP" x="20.0" y="10.0" w="50" h="50"
                  renderGroup="specRefRender2">
    </speciesGlyph>
    <speciesGlyph id="ATP_Glyph" species="ATP" x="70.0" y="80.0" w="100" h="100"
                  renderGroup="specRefRender1">
        <transformation xsi:type="sl2:Transformation mx="10" my="10" sx="0.8" sy="0.8"/>
    </speciesGlyph>
    .
    .
    .
</listOfLayouts>
<render>
    <listOfColors>
        <color id="Color_Green" red="0" green="255" blue="0"/>
        <color id="Color_Black" red="0" green="0" blue="0"/>
    </listOfColors>
    <listOfLinetypes>
        <lineType id="SolidLine" width="1">
            <strokeType xsi:type="sl2:SolidStroke"/>
        </lineType>
    </listOfLinetypes>
    <listOfShapes>
        <shape xsi:type="sl2:Circle" id="StandCircle" lineType="SolidLine"
               linecolor="Color_Green" radius="1.0"/>
```

```
        <shape xsi:type="sl2:Text" id="label1" family="sansserif" size="12"
               linecolor="Color_Black">Glucose</shape>
        <shape xsi:type="sl2:Text" id="label2" family="sansserif" size="12"
               linecolor="Color_Black">ATP</shape>
    </listOfShapes>
    <listOfRenderGroups>
        <renderGroup id="specRefRender1">
            <shape shape="StandCircle">
                <transformation sx="100" sy="100"/>
            </shape>
            <shape shape="label1">
                <transformation mx="10" my="44"/>
            </shape>
        </renderGroup>
        <renderGroup id="specRefRender2">
            <shape shape="StandCircle">
                <transformation sx="50.0" sy="50.0"/>
            </shape>
            <shape shape="label1">
                <transformation mx="7.0" my="19.0"/>
            </shape>
        </renderGroup>
    </listOfRenderGroups>
</render>
```

O.K. As this is a made up example it probably has tons of mistakes, but we hope that it still shows the general principal that we think could be used to connect layout and render information. So what this example is supposed to show is a layout part with (among other things) two speciesGlyphs. Those have different sizes and different positions. In the render part, we define some colors and a line type. We don't use any fill types in this example. Next come the basic shapes that we will use to build up the render information. Each shape can in principal be used in more than one renderGroup as demonstrated with the circle. Last but not least we define two renderGroups each with a circle and a text label. Since the textLabels have different sizes, the circle has to be scaled to the right size with a transformation. Now we can reference these two renderGroups in our speciesGlyphs from the layout part. There is now a tag called renderGroup and an optional transformation to be used on the renderGroup. While the first renderGroup is rendered unchanged, the second one is scaled down a little bit and then moved as to be in the center of the bounding box of the speciesGlyph again.

Todo

- Make more examples
- Come up with some easy way to have a frame around an object
- Do the actual implementation on top of libsbml

- Correct inconsistencies between text and diagrams and all other bugs for that matter

Changes

Version 0.2

- render tag got two new attributes to define a default foreground an background color.
- Some 2D Shapes have been defined.
- Handling of text labels has been refined.
- Two new classes for relative positioning were introduced.
- Species-, Compartment-, Reaction- and SpeciesReferenceGlyphs can now have a default render object
- Some UML diagrams were added
- Rectangles can have rounded edges

Outlook

We are aware that this is just a first draft of what might one day become the render information part of the SBML layout extension as proposed by us. We think that this can again be used as a starting point and as a base for discussion with everyone from the sbml community who is interested in this extension. With the help of all those people it is our hope that both extension will be in a usable state before long.