Complementing layout information with render information in SBML files

Ralph Gauges, Sven Sahle and Katja Wegner
EML Research
Schloss-Wolfsbrunnen Weg 33
D-69118 Heidelberg
Germany

February 1, 2006

1 Introduction

In 2003 we proposed an extension to the SBML file format that allowed programs to include layout and render information on graphical representation of reaction networks in SBML files. It soon became evident during the discussion on the SBML mailing list that a consensus for both layout and render information would not be reached fast, therefore we split the layout and the render part and concentrated on reaching consensus concerning the inclusion of layout information into SBML files. Now two years later, we consider the layout extension to be ready for general usage and as a matter of fact, it has been accepted as an official extension to be included into SBML Level 3. There are already several implementations for it and some programs already use it to exchange layout information on reaction networks. With the growing interest for graphical representations of reaction networks, we feel that it is now time to complement the layout extension with a render extension that builds on it and allows the user to define not only where the objects are to be located, but how they are to be rendered as well.

2 Design decisions

The first and as we think natural decision was to base the render extension on the existing layout extension. Secondly, we tried to make the render extension as flexible as possible in order to not impose any artificial limits on how programs can display their reaction networks. Since most programs do graphical representation of reaction networks in 2D, we decided to include only 2D render information in this first version of the render extension. If the need for 3D render information should arise, we will extend the render extension to include this information as well. We want to keep the render extension independant of the SBML model as well as the layout extension, therefore the render information will be stored as one or more seperate blocks. There can be one block of render information that applies to all layouts and blocks for each layout. In the beginning this render information will be stored in the annotation of the listOfLayouts or the annotation of a layout respectively.

The render information consists of a set of styles that are associated with objects from the layout either by a list of ids of layout objects or by roles of layout objects or their corresponding model elements. Consider as example, you can define a style that can be applied to all SpeciesReference objects that are products. The render information included in the annotation of the listOfLayouts element will only be able to define styles that associate render

information with roles of elements, it can not associate styles with individual objects from a layout. In order not to reinvent the wheel we loosely based this specification on SVG.

3 Defining render information

The render extension provides two locations where styles can be defined. First each layout can have its own set of render information located in the annotation of the layout itself. Second, a set of global render information located in the annotation of the listOfLayouts can be defined. It is important to note that each layout can have more then one style and that it is also possible to define more then one global style. Each style can also reference another style that complements it, this way the user can create styles that are based on other styles and only make slight modifications to the style they are based on. In contrast to local styles, the global styles can not reference individual layout elements by an id, it can only define role based or type based styles.

The top level element for the per layout render information is called `listOfRenderInformation` which can contain a list of one or more `renderInformation` elements of type LocalRenderInformation. The LocalRenderInformation datatype is based on the SBase datatype from SBML and has five attributes. The `id` attribute is of type SId just like the ids used in SBML. It is used to give the renderInformation element a unique id through which it can be referenced from other LocalRenderInformation objects. The optional attribute `name` is there to be able to give a LocalRenderInformation object a more user friendly name that can be displayed in programs. In cases where there is more then one LocalRenderInformation object per layout, this might make it easier for a user to find the render information he or she wants to associate with a given layout. The attributes `programName` and `programVersion` are optional and can be used to store information about the program that created the render information. Another optional attribute called `referenceRenderInformation` can be used to specify the id of another local (or global) render information object that complements the current render information object. So if a program can find no fitting render information in the current render information object, it can go on to the one referenced and see if it can find fitting information there. In order to avoid loops, only render information objects that have already been defined before may be referenced. So local render information objects may reference any global render information object as well as any local render information object that has already been defined.

In addition to those five attributes, the LocalRenderInformation object has three elements. The first element is called `listOfColorDefinitions` and is used to predefine a set of color to be referenced in styles. The second element is called `listOfGradientDefinitions` and it is used to predefine linear and radial gradients to be referenced in a style. How colors and gradients can be defined is explained in the section called **Colors and Gradients**. The third is called `listOfStyles` and it can hold one or more local style objects. Each local style object is located in an element called `style` and is of type LocalStyle.

First, a LocalStyle has an attribute called `id` that uniquely identifies it. It also has an optional roleList attribute which list all the roles the style applies to and it can have a `typeList` attribute which lists all the element types the style applies to. The valid types for the `typeList` attribute are a combination of one or more of the following values separated by whitespaces:

- COMPARTMENTGLYPH,
- SPECIESGLYPH,
- REACTIONGLYPH,
- SPECIESREFERENCEGLYPH,
- TEXTGLYPH,
- GRAPHICALOBJECT, and
- ANY.

The ANY keyword specifies that this styles applies to any type of glyph and would be equivalent to listing all the other keywords. Concerning the valid keywords for the `roleList` attribute we had thought about taking those from some kind of controlled vocabulary. Preferably, this would be some kind of ontology like SBO. The specifics of this will have to be discussed with other interested parties. LocalStyle objects can have one more optional attribute which is called `idList`. This is simply a list of ids of objects from the layout the style applies to.

The only subelement of a style is a `g` element which specifies how the element(s) covered by the idList, roleList and typeList are to be rendered. The details of this element are described in the section about grouping below.

ListOfLocalRenderInformation inherits from **SBase**		
renderInformation	:	LocalRenderInformation[1..∗]

LocalRenderInformation inherits from **SBase**		
id	:	SId
name	:	string {use=”optional”}
programName	:	string {use=”optional”}
programVersion	:	string {use=”optional”}
referenceRenderInformation	:	string {use=”optional”}
listOfColorDefinitions	:	ListOfColorDefinitions {use=”optional”}
listOfGradientDefinitions	:	ListOfGradientDefinitions {use=”optional”}
listOfStyles	:	ListOfLocalStyles {use=”optional”}

ListOfLocalStyles inherits from **SBase**		
style	:	LocalStyle[1..∗]

LocalStyle inherits from **Style**		
idList	:	string[1..∗] {use=”optional”}

Style inherits from **SBase**		
id	:	SId
roleList	:	string[1..∗] {use=”optional”}
typeList	:	string[1..∗] {use=”optional”}
g	:	Group

example:

```
<listOfLayouts xmlns="http://projects.eml.org/bcb/sbml/level2"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <layout id="Layout_1">
    <annotation>
      <listOfRenderInformation
           xmlns="http://projects.eml.org/bcb/sbml/render/level2">
        <renderInformation id="FancyRenderer_Default"
                           name="default style"
                          programName="FancyRenderer"
                          programVersion="0.1.1">
          <listOfColorDefinitions>
            <colorDefinition ... />
                        .
      .
      .
          </listOfColorDefinitions>
          <listOfGradientDefinitions>
```

```
            <linearGradient ... >
                   .
                   .
                   .
            </linearGradient>
            <radialGradient ... >
                   .
                   .
                   .
            </radialGradient>
                 .
                 .
                 .
          </listOfGradientDefinitions>
          <listOfStyles>
            <style id="CompartmentGlyphStyle" typeList="COMPARTMENTGLYPH">
              <g ...>
                .
                .
                .
              </g>
            </style>
               .
               .
               .
          </listOfStyles>
        </renderInformation>
      </listOfRenderInformation>
    </annotation>
         .
         .
         .
  </layout>
</listOfLayouts>
```

Global render information is specified very similar to local render information there are only some slight differences that one has to be aware of. Global render information is stored in an element called `listOfGlobalRenderInformation` which contains one ore more `renderInformation` elements of type GlobalRenderInformation. The attribute and elements of GlobalRenderInformation objects and LocalRenderInformation objects are the same. The only difference here is the fact that GlobalRenderInformation objects in their `referenceRenderInformation` attribute may only reference ids of other GlobalRenderInforma-

tion objects that have already been defined. The `listOfStyles` element of the GlobalRenderInformation object contains one or more `style` elements but this time these are of type GlobalStyle. The GlobalStyle datatype is also very similar to the LocalStyle datatype, the only difference being that the GlobalStyle does not have an `idList` attribute since referencing individual ids from a layout does not make sense for a global render information object. Save for those few differences, global and local render information is specified in the same way.

ListOfGlobalRenderInformation inherits from **SBase**		
renderInformation	:	GlobalRenderInformation[1..∗]

GlobalRenderInformation inherits from **SBase**		
id	:	SId
name	:	string {use=”optional”}
programName	:	string {use=”optional”}
programVersion	:	string {use=”optional”}
referenceRenderInformation	:	string {use=”optional”}
listOfColorDefinitions	:	ListOfColorDefinitions {use=”optional”}
listOfGradientDefinitions	:	ListOfGradientDefinitions {use=”optional”}
listOfStyles	:	ListOfGlobalStyles {use=”optional”}

example:

```
<listOfLayouts xmlns="http://projects.eml.org/bcb/sbml/level2"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <annotation>
    <listOfGlobalRenderInformation
          xmlns="http://projects.eml.org/bcb/sbml/render/level2">
      <renderInformation id="FancyRenderer_GlobalDefault"
                         name="default global style"
                         programName="FancyRenderer"
                         programVersion="0.1.1">
        <listOfColorDefinitions>
        .
        .
        .
        </listOfColorDefinitions>
        <listOfGradientDefinitions>
               .
```

```
            .
            .
        </listOfGradientDefinitions>
        <listOfStyles>
            .
            .
            .
        </listOfStyles>
      </renderInformation>
    </listOfGlobalRenderInformation>
  </annotation>
</listOfLayouts>
```

3.1 Defining styles

3.2 Specifying positions and sizes

Positions and sizes for render elements can be specified either as absolute values where the default unit is pt (1/72 inch) like in the layout extension or alternatively as relative values in % where the % symbol has to be added to the value. All values are relative to the bounding box of the corresponding element in the layout. This bounding box basically specifies a viewbox for the render elements to be drawn on.

3.3 Colors and Gradients

Although, it is possible to specify the color for a graphical primitive directly, colors and especially gradients can be specified in a so called `listOfColorDefinitions` and `listOfGradient-Definitions` element which is a subelement of the RenderInformation data type. The `listOfColorDefinitions` element holds one or more elements called `colorDefinition` of type ColorDefinition. The ColorDefinition data type is derived from SBase and has two additional attributes. One `id` attribute which uniquely identifies the ColorDefinition object within a RenderInformation object and an attribute called `value` which holds a color value.

Color values are specified as a 6 to 8 digit hex string which defines the RGBA value of the color. If only the first six digits for the RGB value are given, the alpha value is assumed to be 0xFF which means that the color is totally opaque. SVG only has RGB color definitions and the A value is handled by the stroke-opacity and fill-opacity attribute. We feel that this is limiting and that RGBA color values are the standard way of handling transparency values in most drawing tools.

ColorDefinition inherits from **SBase**		
id	:	SId
value	:	string

example:

```
<listOfColorDefinitions>
  <colorDefinition id="darkred" value="#200000" />
          .
          .
          .
</listOfColorDefinitions>
```

All graphical primitives in the render extension have a stroke attribute that is used to specify the color of the stroke that is used to draw the curve or the outline of ellipses, rectangles or polygons. This `stroke` attribute can either hold a color value or it can hold the id of a predefined ColorDefinition object.

The `listOfGradientDefinitions` element holds one or more `linearGradient` or `radialGradient` subelements of type LinearGradient or RadialGradient respectively. A `linearGradient` element has eight attributes. The `id` attribute uniquely identifies it within a RenderInformation object. The attributes `x1`, `y1`, `z1`, `x2`, `y2` and `z2` are all optional and define a vector on which the gradient stops are mapped. If not specified, `x1`, `y1` and `z1` default to 0% and `x2`,`y2` and `z2` default to 100%. The last attribute called `spreadMethod` is also optional and specifies the method that is used to continue the gradient pattern if the

vector points do not span the whole bounding box of the object the gradient is applied to. The attribute can have three values called **pad**, **reflect** or **repeat**. A value of **pad**, which is also the default, means that the gradient color at the endpoint of the vector defines how the gradient is continued beyond that point. A value of **reflect** means the gradient continues from end to start and then from start to end again and so on. A value of **repeat** simply means that the gradient pattern is repeated from start to end over and over again.

To specify the mentioned gradient stops the linear gradient element can hold one or more subelements called `stop` which are of type GradientStop. The GradientStop datatype has two attributes. The first attribute, called `offset`, represents the distance from the starting point of the vector defined by `x1`, `y1`, `z1`, `x2`, `y2` and `z2`. The value is given as a positive percentage value (usually somewhere between 0% and 100%). The other attribute is called `stop-stroke` and defines the color for the given gradient stop. The attributes value can either be given as a hexadecimal color value or as the id of a ColorDefinition object from the `listOfColorDefinitions` (see above).

In case the two points that define the gradient vector are identical, the area will be painted with a single color taken from the last gradient stop element.

example:

```
<listOfGradientDefinitions>
  <linearGradient x1="30%" y1="50%" x2="70%" y2="50%">
    <stop offset="0%" stop-color="#0000A0" />
    <stop offset="100%" stop-color="darkred" />
  </lineaGradient>
          .
          .
          .
</listOfGradientDefinitions>
```

The RadialGradient data type has 9 attributes. Just like the LinearGradient it has an `id` that uniquely identifies it within

LinearGradient inherits from **SBase**		
id	:	SId
x1	:	string {use="optional" default="0%"}
y1	:	string {use="optional" default="0%"}
z1	:	string {use="optional" default="0%"}
x2	:	string {use="optional" default="100%"}
y2	:	string {use="optional" default="100%"}
z2	:	string {use="optional" default="100%"}
spreadMethod	:	string {use="optional" default="pad"}
stop	:	GradientStop[1..*]

GradientStop inherits from **SBase**		
offset	:	string
stop-color	:	string

a RenderInformation object and it also has the same `spreadMethod` attribute with the same possible values. The attributes `cx`, `cy` and `cz` define the center of the radial gradient. The attributes are optional and can either be given in absolute or relative coordinates. If ommited, these three attribute values default to 50%. The `r` attribute defines the radius of the gradient and it can also be specified in either absolute or relative coordinates. Specifying negative values for `r` is considered an error just like it is in SVG. The attributes `fx`, `fy` and `fz` specify the focal point of the gradient. The gradient will be drawn such that the 0% stop is mapped to (fx,fy,fz). The attributes `fx`, `fy` and `fz` are optional. If one is omitted it is considered to coincide with the value of `cx`, `cy` and `cz` respectively.

RadialGradient inherits from **SBase**		
id	:	SId
cx	:	string {use="optional" default="50%"}
cy	:	string {use="optional" default="50%"}
cz	:	string {use="optional" default="50%"}
r	:	string {use="optional" default="50%"}
fx	:	string {use="optional"}
fy	:	string {use="optional"}
fz	:	string {use="optional"}
spreadMethod	:	string {use="optional" default="pad"}
stop	:	GradientStop[1..*]

example:

```
<listOfGradientDefinitions>
  <radialGradient cx="50%" cy="50%" r="20" spreadMethod="repeat">
    <stop offset="10%" stop-color="#000040" />
    <stop offset="90%" stop-color="#0000C0" />
  </radialGradient>
       .
       .
       .
</listOfGradientDefinitions>
```

3.4 Graphical primitives

The graphical primitives polygons, rectangles and ellipses are based on the corresponding elements from SVG. For lines, arcs and general path primitives, we basicall reuse the curve element from the layout extension. There is however one difference to the Curve datatype from the layout extension. Whereas Point objects in the layout extension could only contain absolute values for their coordinates, Point objects in the render extension can contain relative coordinate values. Since polygons are very similar to general path primitives, we also make use of the Curve datatype to specify polygons in the render extension.

All graphical primitives have some attributes in common that specify some drawing properties. As mentioned in the **Colors**

and Gradients section, each graphical primitive has a stroke attribute that defines the color curves and outlines of geometric shapes are drawn. In addition to that, the `stroke-width` attribute specifies the width of the stroke and the `stroke-dasharray` is a list of numbers that specifies the lengths of dashes and gaps that are used to draw the line. In addition to those attributes, ellipses, polygons and rectangles have an attribute called `fill` that specifies the fill style of those elements. The fill style can either be a hexadecimal color value or the id of a ColorDefinition object or a GradientDefinition object.

Another attribute that all graphical primitives have in common is the `transform` attribute. This attribute can be used to specify a six element matrix which corresponds to a affine transformation matrix. The details are described in the section called "Transformations" below.

GraphicalPrimitive1D inherits from **SBase**		
stroke	:	string {use="optional"}
stroke-width	:	string {use="optional"}
stroke-dasharray	:	double[1..*] {use="optional"}
transform	:	string {use="optional"}

GraphicalPrimitive2D inherits from **GraphicalPrimitive1D**		
fill	:	string {use="optional"}

3.4.1 Curves

Simple lines and complex curves are represented by the curve element introduced in the layout extension. A curve has a listOfCurveSegments which can hold an arbitrary number of line segments and cubic bezier elements in any order. With this, any path (no matter how complex it is) can be represented. As mentioned above, Point objects used to specify the individual curve segments can contain relative values for their coordinates

as well as absolute values. The coordinate values are always with respect to the bounding box of the layout object the render information applies to.

Curve inherits from **GraphicalPrimitive1D**		
listOfCurveSegments	:	ListOfCurveSegments

ListOfCurveSegments inherits from **SBase**		
curveSegment	:	LineSegment[1..∗]

Point inherits from **SBase**		
id	:	SId {use=”optional”}
x	:	double
y	:	double
z	:	double {use=”optional” default=”0.0”}

LineSegment inherits from **SBase**		
start	:	Point
end	:	Point

CubicBezier inherits from **LineSegment**		
basePoint1	:	Point {use=”optional”}
basePoint2	:	Point {use=”optional”}

example:

```
<g ...>
 <curve stroke-width="2.0" stroke="#000000" >
  <listOfCurveSegments>
    <curveSegment xsi:type="LineSegment">
      <start x="0%" y="50%" />
      <end x="100%" y="50%" />
    </curveSegment>
  </listOfCurveSegments>
 </curve>
   .
   .
   .
</g>
```

3.4.2 Polygons

A polygon object is made up of a `polygon` element which contains a curve that defines the edge of the polygon. The major difference to the curve object is that the last point of the curve is connected to the first, so the polygon is always closed and that the polygon can therefore have a fill style that determines how the inside of the polygon is to be rendered. The fill style is applied according to the rules defined in the SVG specification.

Polygon inherits from **GraphicalPrimitive2D**		
curve	:	Curve

example:

```
<g ...>
 <polygon fill="darkred" stroke="#000000" >
   <curve stroke-width="2.0">
     <listOfCurveSegments>
       <curveSegment xsi:type="LineSegment">
         <start x="0%" y="50%" />
         <end x="100%" y="50%" />
       </curveSegment>
     </listOfCurveSegments>
   </curve>
 </polygon>
      .
      .
      .
</g>
```

3.4.3 Rectangles

The rectangle definition was taken from SVG and allows the definition of rectangles with or without rounded edges. The rectangle has the attributes `x`, `y` and `z` to specify its position within the bounding box of the enclosing layout object and a `width` and `height` attribute that specifies the width and height of the rectangle, either in absolute values or as a percentage of the width and height of the enclosing bounding box. The

default value for the optional `z` attribute is 0.0. Additionally the rectangle has two optional attributes `rx` and `ry` that specify the radius of the corner curvature. If only `rx` or `ry` is specified, the other is presumed to have the same value. The default value for `rx` and `ry` is 0.0 which means that the edges are not rounded.

Rectangle inherits from **GraphicalPrimitive2D**		
x	:	string
y	:	string
z	:	string {use="optional" default="0.0"}
width	:	string
height	:	string
rx	:	string {use="optional" default="0.0"}
ry	:	string {use="optional" default="0.0"}

example:

```
<g ...>
 <rectangle x="0%" y="0%" width="100%" height="100%" rx="5%"
            fill="darkred" stroke="#000000" />
      .
      .
      .
</g>
```

3.4.4 Ellipses

The definition of an ellipse was also taken directly from SVG. The ellipse has the attributes `cx`, `cy` and `cz` to specify the center of the ellipse and `rx` and `ry` to specify the radius of the ellipse along the x-axis and the y-axis respectively. If only rx or ry is specified, the other is presumed to have the same value. Circles are a special case of an ellipse where `rx` and `ry` are equal. Again `cz` is optional and its default value is 0.0.

Ellipse inherits from **GraphicalPrimitive2D**		
cx	:	string
cy	:	string
cz	:	string {use="optional" default="0.0"}
rx	:	string
ry	:	string {use="optional"}

example:

```
<g ...>
 <ellipse cx="50%" cy="50%" rx="30%" fill="#00FF00" stroke="#000000" />
      .
      .
      .
</g>
```

3.4.5 Text elements

In order to draw text, we use the text element from SVG with slight modifications. Like the text element in SVG, our text element has the attributes `font-family` to specify which font to use and `font-size` to specify the size of the font. Both attributes are optional. If specified, `font-size` must be a positive absolute or relative size.

For reasons of simplicity, we limit the display of text to normal text, outlined or filled-outlined text are not supported. Also in order to simplify text display we think it would be best practice if programs would limit the choice of the font-family attribute to the generic families serif, sansserif and monospace. But since those only apply to western languages, it make sense to use other values for font-familie in certain cases.

The horizontal alignment of text element can be specified by the `text-anchor` attribute. Allowed values are start, middle and end. SVG does not seem to provide any means for the vertical alignment of text, therefore in order to make it easier for implementers we don't. On the other hand if people think that this is a necessary feature, we can either extend the allowed

values for the `text-anchor` attribute, or we can add a second attribute with the same allowed values to do vertical text alignment. Depending on the value of the `text-anchor` attribute, the `x`, `y` and `z` attributes of the text element either specify the left bottom corner, the middle bottom position or the right bottom corner of the text. As in rectangles and ellipses, the `z` attribute is optional and its default value is 0.0.

The `text` element has two more attributes. One is called `font-weight` and specifies wether a font is to be drawn bold. The only values allowed for `font-weight` are **bold** and **normal**. Likewise the `font-style` attribute determines wether a font is to be drawn italic or normal and consequently the only allowed values are **italic** and **normal**. Both attributes are optional.

Text inherits from **GraphicalPrimitive1D**		
x	:	string
y	:	string
z	:	string {use="optional" default="0.0"}
font-family	:	string {use="optional"}
font-size	:	string {use="optional"}
font-weight	:	string {use="optional"}
font-style	:	string {use="optional"}
text-anchor	:	string {use="optional"}

example:

```
<g ...>
 <text x="50%" y="50%" text-anchor="middle" stroke="#FF0000"
       font-family="serif" font-size="20.0" >This is a Text</text>
      .
      .
      .
</g>
```

3.4.6 Bitmaps

To include bitmaps into a graphical representation we use the image element from SVG. The image element in SVG can also be

used to include complete SVG vector images which we explicitely exclude in this version of the proposal since we think it would be too complicated. If the community feels that there is the need to include vector graphics, we could integrate the according SVG specificatin without changes.

The image element has six attributes. The `x`, `y` and `z` attributes specify the position of the image within the bounding box and the `width` and `height` attributes specify its width and height. The `z` attribute is optional and its default value is 0.0. The actual image data is not embedded in the render information, but the image element has an attribute called `xlink:href` that references an external JPEG or PNG file. If the referenced image is larger then the given width and height, it has to be scaled to the given dimensions.

Image inherits from **SBase**		
x	:	string
y	:	string
z	:	string {use=”optional” default=”0.0”}
width	:	string
height	:	string
href	:	string

example:

```
<g ...>
 <image x="10%" y="10%" width="80" height="100" href="Glucose.png" />
      .
      .
      .
</g>
```

3.5 Transformations

In order to be able to display text that is not aligned horizontally or vertically or to effectively compose groups of objects from primitives, transformation like rotation, translation and scaling are needed. SVG, among other options, allows the user to specify a 3x3 matrix transformation matrix. Since the last row of the matrix is always 0 0 1, the matrix is specified as a six vector. Therefore, in the render extension each group or graphical primitive can have a `transform` attribute just as in SVG. The allowed value for the attribute is the form: `a, b, c, d, e, f`, where a-f denote the values for the transformation matrix as stated above.

example:

```
<g ...>
  <text x="50%" y="50%" text-anchor="middle" stroke="#FF0000"
       font-family="serif" font-size="20.0"
       transform="1.0, 3.0, 2.5, 1.4, 4.0, 5.0">This is a Text</text>
     .
     .
     .
</g>
```

3.6 Grouping

Like in SVG, several graphical primitives can be grouped inside a `g` element to generate more complex render information. `stroke`, `stroke-width`, `stroke-dasharrays`, `transform`, `fill`, `font-family`, `font-size`, `font-weight`, `font-style` and `text-anchor` attributes can be applied to groups. If any of those attributes is specified for a Group object, it specifies the corresponding attribute for all graphical primitives and groups defined within this group. If a graphical primitive or a group redefines one or more of those attributes, the newly defined values take effect. If an object within the group does not redefine those values, those of the group apply. If an attribute is not

attribute	default value
stroke-width	0.0
stroke-dasharrays	*empty list*
transform	1.0, 0.0, 0.0, 0.0, 1.0, 0.0
fill	None
font-family	sans-serif
font-size	0
font-weight	normal
font-style	normal
text-anchor	start

Table 1: Attribute default values.

defined in any object of a style, its default values take effect. These default values are listed in 1.

It might seem a little unusual that the default values for stroke-width and text-size are set to 0. The reason for this is that a style that only contains an empty group is meant to define that the element the style applies to is not to be rendered. Since the render information for curves in SpeciesReferenceGlyphs and ReactionGlyphs as well as the render information for TextGlyphs is defined via attributes from the outermost group element of a style (see below), the group would explicitely have to define the stroke-width or the text-size to be 0 which would be inconsistent with the implied meaning of an empty group.

Each group has an id through which it can be referenced. This way groups can be nested by referencing one group within another. In order not to create loops, only groups that have been defined previously in the render information can be referenced. No forward declarartions as references are allowed. In addition to those attributes a Group object can contain 0 or more child elements that form the render information. These child elements can either be graphical primitives, e.g. rectan-

gles, ellipses, curves, polygons, text elements images or groups.

Group inherits from **GraphicalPrimitive1D**		
fill	:	string
font-family	:	string
font-size	:	string {use="optional" default="0.0"}
font-weight	:	string
font-style	:	string
text-anchor	:	string
children	:	(GraphicalPrimitive1D and/or Image)[0..*]

example:

```
<g stroke="#000000" font-family="serif" >
  <rectangle x="0%" y="0%" width="100%" height="100%"
             fill="blueLinearGradient"  />
  <text x="50%" y="50%" font-size="80%" text-anchor="middle"
        stroke="#FF0000" />
</g>
```

4 Style resolution

When a program tries to resolve which style applies to a certain object it should follow the rule that more specific style definitions take precedence over less specific ones and that if there are several styles with the same specificity, the first one encountered in the file is to be taken. In essence this means that a program first has to search the local render information for a style that references the id of the object, if none is found it searches for a style that mentions the role of the object if it has one (see next section). If it does not find one, it searches for a style for the type of the object. If a render information references another render information object via its `referenceRenderInformation` attribute, the program has to go through that one as well to see if a more specific render information is present there. If the chain of referenced RenderInformation objects has been searched and no style has been found that fits, it is up to the program how

the object is rendered. If a program explicitely wants to define render information that states that some objects are not to be rendered at all, it has to define a style that does nothing, i.e. has no render information but applies to the objects that are not to be rendered. If several type based styles are found that would fit, a style that applies to only one type takes precedence over a style that applies to several types.

5 Role resolution

This render extension explicitely provides means to write render information that renders layout objects based on certain roles those render objects or their corresponding model objects have. So far SBML models or layouts do not contain such role information or only for a limited number of objects if one would consider the role attribute of SpeciesReferenceGlyph objects to fall into this category. Although there is currently no means to specify these roles, there are already initiatives underway that try to complement SBML files with more biological information based on ontologies. One of these initiatives, the sboTerms, is about to be included into SBML Level 2 Version 2. This ontology or a similar one could provide this role information in the future. For now, we are going to assume that some means to specify roles has already been specified and we now need to specify how these roles are going to be resolved in case of overlapping or contradicting role information.

A specific style can reference one or more roles to which it applies. Roles of objects could be taken from the sboTerm of the corresponding object and we expect model objects as well as layout objects to use sboTerms to specify roles for individual objects. When a program tries to determine which style applies to a specific object it might have to determine the role of the object first. If the layout object itself has a role, this will be taken,

otherwise if the layout object is associated with an object in the model, the program should get the role from the associated object. If none of them has a role, no role based style can be applied to the object.

6 Style information for ReactionGlyphs and SpeciesReferenceGlyphs

When defining a style for a ReactionGlyph or SpeciesReferenceGlyph object, one has to distinguish wether the layout only specifies a bounding box for the object or if it specifies a curve. In the case of a bounding box, you want to define complete render information, whereas in the case of a curve, you only want to set certain attributes that determine certain aspects about how the curve is drawn, e.g. its color. To resolve this conflict, the style for such an object has to define render information for both cases. The render information for the case of a bounding box is specified just like render information for any other object within a Group. Render information for the case of a curve is defined by the appropriate attributes that are in effect in the outermost Group object itself. Those attributes include `stroke`, `stroke-width` and `stroke-dasharray`. If the group does not define one or more of these attributes, the default value is used (see section on Grouping).

7 Style information for TextGlyphs

Just as in the case of curves in ReactionGlyphs and SpeciesReferenceGlyphs, TextGlyphs already can be considered to be render information which is located in the layout because a TextGlyph already specifies the text to be rendered and it therefore does not need additional render information in the form of a

`text` element. On the other hand, it needs render information about the font properties the text is to be rendered with. Just as for the curve object for ReactionGlyphs and SpeciesReferenceGlyphs, this render information is taken from the font related attributes of the outermost group element of the style that is used to render a TextGlyph. Any additional information within the group is ignored. If the group does not specify any of the `font-family`, `font-size`, `font-weight`, `font-style` or `text-anchor` attributes, the default values are to be used.