# Drawing structured diagrams with SDDL

Mathieu Bourgeois and Roger Villemaire

## Abstract

We present SDDL, a Structured Diagram Description Language aimed at producing graphical representations for discrete mathematics and computer science. SDDL allows combining graphical objects (circles, lines, arrows, ...) and LaTeX boxes to produce diagrams representing discrete structures such as graphs, trees, etc. with an easy-to-use domain specific language.

## 1 What is SDDL?

SDDL, *Structured Diagram Description Language*, is a high-level domain specific language tailored for diagrams that have an inherent structure. Examples of these might be trees, graphs and automata. Any diagram that is naturally structured can be described in SDDL. However, it was designed especially for data structures appearing in computer science and discrete mathematics.

The main objective of this language is to realize drawings in a natural and structured way. Mainly, one describes a drawing in SDDL in a way that is similar to drawing on a blackboard. For example, specific shapes (like circles, ellipses, texts, boxes) are drawn at positions that can be either absolute or relative to specific points on already placed shapes. Since we aim at mathematical drawings, text is handled through LaTeX.

Since exhibiting the structure of a diagram is essential to our purpose, SDDL uses an object-oriented hierarchy, completely written in Java, under our high-level language. Having shapes as objects makes structuring of diagrams easier and more intuitive since most shapes are explicitly represented by a class. Furthermore, extension by the end user is quite feasible since Java is a well-known language.

At the lower level, SDDL uses another graphical description language for LaTeX, namely Asymptote [1]. Our tool produces Asymptote code, which is finally converted to an encapsulated PostScript (eps) vector file.

## 2 Canvas and shapes

A diagram in SDDL is defined by a main Canvas. It is just like a standard painting canvas, in the sense that we can place (or paint) different things on it as much as we like. The things we can place inside a Canvas are Shapes. Thus, for creating a "Hello, world!" diagram, we would use the following:

```
put Text with [text = "Hello, world!"] in main;
```

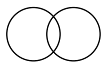
resulting in:

Hello, world!

In this example, Text is a Shape, and it possesses a property text, which is the LaTeX string used to render the text. Each Shape defines a certain number of such properties.

SDDL permits the definition of variables. It is a dynamically typed language, so variables don't have to be declared before being used. Variables make it possible to reuse the same Shape at multiple locations. For example,

```
a = Circle with [radius = 10.0];
put a at (-7.5, 0.0) in main;
put a at (7.5, 0.0) in main;
```

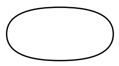


In this example, we specify the location where we want our Shape to be. If a location is not specified with the at clause, the Shape will be placed at the point of origin of the current Canvas, which is its center.

One important thing to note is that once a Shape has been put in a Canvas, it is immutable. It cannot be modified or removed. A Shape can be modified *after* it has been drawn on a Canvas, but this will have an effect only on later use of this object and will not modify the actual Canvas in any way; a Shape always appears in a Canvas as it was at the moment it was added.

When a Shape is created, every property specified by the user is set, in the given order. It is not necessary to specify all properties at once, nor to set all of them. Most of the properties have appropriate default values. However, some properties, if not set, will yield strange results. For example, a circle with no radius property set will have a default radius of 0. Usually, property order is irrelevant. For instance, giving for an ellipse the radius along the x-axis or the y-axis first will yield exactly the same result.

```
a = Ellipse;
a = a with [xradius = 20.0];
a = a with [yradius = 10.0];
put a in main;
```



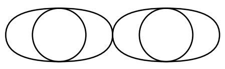
## 3 Structuring multiple canvases

Every diagram consists of a *main* Canvas. However, we can define other Canvas objects if we wish. An additional Canvas can be introduced using a typical variable assignment. However, when a put command is used, SDDL checks to see if the canvas variable is already defined. If it isn't, it automatically creates an empty `Canvas` for use.

One of the main reasons to use other `Canvas`es is to create an explicit structure in our diagram. Since a `Canvas` is a `Shape`, once a sub-`Canvas` has been created, it can be placed inside the `main` `Canvas`, or any other one for that matter. This will ensure that everything that is defined inside our sub-`Canvas` will be placed at the proper position in the final diagram. However, any `Canvas` that has been defined, but is not linked directly or indirectly with the *main* `Canvas`, will not be drawn in the final diagram.

As an example, let's say we want to make a diagram that consists of two identical "eyes". Instead of defining two identical objects, we will create one in a `Canvas` named *form* and put it at different positions inside our *main* `Canvas`. Our "eye" consists of a circle and an ellipse, both with the same center. All we need to do for this is place them at the default position of the `Canvas`. Finally, we can take this sub-`Canvas` and place it at the two positions required.

```
a = Circle with [radius=10.0];
b = Ellipse with [xradius=20.0, yradius=10.0];
put a in form;
put b in form;
put form at (-20.0, 0.0) in main;
put form at (20.0, 0.0) in main;
```



## 4 Paths

SDDL offers support for the description of paths. The way in which they are described is similar in syntax with MetaPost and Asymptote, though with some variations. A `Path` shape is described by linking points together with specific linking symbols. To draw a line between two points, the line symbol `--` can be used. To link some points using a curve, use the curved line symbol `~`, which will use Asymptote's positioning algorithms to create a nice-looking curve which passes through those points. For the moment, user control over the curve is limited to beginning and ending tangents, but could be expanded.

```
p1 = a--b--c;
p2 = a~b~c;
```

One of the other things you may want to do with a `Path` is to create an `Arrow` out of it. SDDL defines symbols for forward, backward and bidirectional arrows for both linear and curved lines. Right now, however, support is restricted to forward arrows.

```
p1 = a<-b--c->d;
p2 = a<~b~c~>d;
p3 = a<->b;
p4 = a<~>b;
```

Once a `Path` is defined, it can be used as any other `Shape`. However, one of the main things that you want to do is to put your `Path` somewhere. For this, you can write something like this:

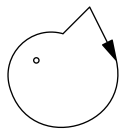
```
p = (0.0, 0.0) -- (10.0, 10.0);
put p in main;
```

This is the standard way to add `Shape`s to a `Canvas`. However, since `Path`s are usually used to link different `Shape`s together in the *main* `Canvas`, special handling is done in SDDL. Namely, a `Path` that is not explicitly assigned to a variable will be automatically placed in the *main* `Canvas`. Therefore the `Path` will be directly placed in *main* without any explicit `put` expression. Thus, linking has its own special syntax, which is more natural and convenient. As an example, this program

```
(0.0, 0.0) -- (10.0, 10.0);
```

yields exactly the same result as the one just above. Furthermore, points can be added and multiplied by a scalar (as vectors). Using a dot to locate the origin of the drawing, we can use the SDDL syntax to create this example:

```
a = (10.0, 10.0);
a--2.0*a->(30.0, 0.0);
a~(-1.0)*a~(30.0, 0.0);
```



Finally, as we will now see, `Shape`s also usually define specific points.

## 5 Drawing and linking shapes together

To draw a diagram, usually some shapes with a specific structure are first laid down. Once that is done, those objects are linked together with lines, curves and arrows. However, these links must be made between specific positions, usually derived from one or more specific `Shape`s. For example, we may want to link the northwestern point of a rectangle with the point on a circle at an angle of 45 degrees. These points could obviously be computed in advance. However, this becomes quite problematic when points are at peculiar angles or more complex relationships between points and `Shape`s are needed. Worst of all, if the position or any other property of a `Shape` is changed, all computations will have to be redone.

To ease object linking, SDDL provides so-called *reference points*. A reference point is a point that has no static value, unlike points defined by a pair of two real numbers. Instead, a reference point is defined by its relationship to a specific `Shape` appearing at a particular location. As a matter of fact, all reference points are defined along with the Shape because they are a natural part of it. This ensures that we always have nice-looking lines at exactly the positions we want them to be.

However, a reference point's exact position in a Canvas will depend on the Shape's position. Worse, since the same Shape can appears at multiple locations in the same Canvas, we have to know which occurrence we are talking about!

Therefore, SDDL introduces a feature called a Drawing. A Drawing is an object that represents a Shape at a particular position, i.e. a specific occurrence of a Shape in a Canvas. Whenever a Shape is put inside a Canvas, a Drawing is returned and can be assigned to a variable. This gives a way to uniquely identify every occurrence of a Shape appearing in a Canvas. If the same Shape is placed twice, two different Drawings will be returned, each referring to a different occurrence of the same Shape.

```
a = Circle with [radius = 10.0];
d1 = put a at (-10.0, 0.0) in main;
d2 = put a at (10.0, 0.0) in main;
```

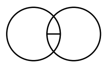
Once a drawing has been defined, a way to access its points is needed. SDDL defines a syntax for extracting points from drawings:

$$\textbf{coord of } \langle \textit{coord} \rangle \textbf{ ( } \langle \textit{args} \rangle \textbf{ ) in } \langle \textit{drawing} \rangle$$

Here one specifies the kind of point to use ($\langle \textit{coord} \rangle$) and any particular arguments required to obtain it. The available points are Shape specific, so only those kinds of points which make sense for the particular Shape can be used. For instance one can get a point at a particular angle on a Circle. This is also an example where an additional argument is needed. For instance, anglePoint(45.0) will give the point at 45 degrees.

Finally, to reference the drawing from which we take the point, its drawing Path (a dot separated path) from the main diagram must be given.

```
a = Circle with [radius = 10.0];
d1 = put a at (-7.5, 0.0) in main;
d2 = put a at (7.5, 0.0) in main;
(coord of anglePoint(0.0) in main.d1) --
   (coord of anglePoint(180.0) in main.d2);
```

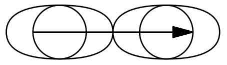


In this example, a line between two identical circles is drawn. The drawing Paths are simple here, since both drawings are made directly in *main*. Thus, only *main* followed by the drawing name is needed.

Simply giving a Drawing is not sufficient in order to make reference points non-ambiguous; complete drawing Paths are required, as we will now show. Let's go back to the two eyes we drew previously. The eye was defined by a circle and an ellipse, both put inside a sub-Canvas. Now, let's say we want to draw a line between the two circles. Each of those circles is defined in the sub-Canvas and they are, as a matter of fact, the same Drawing *a*. Adding a link between *a* and itself would add a link between the sub-Canvas' circle and itself (what could this mean?). Adding the sub-Canvas to the main Canvas twice, as we did before, would just duplicate this structure inside the main Canvas.

But since a Canvas is a Shape, we get a drawing when we put the sub-Canvas inside *main*. We can therefore identify each inner circle by listing the drawings required to access them, as shown in the following SDDL example. Thus, we can always link together Shapes that are deeply nested inside a sub-Canvas.

```
a = Circle with [radius=10.0];
b = Ellipse with [xradius=20.0, yradius=10.0];
c = put a in form;
put b in form;
f1 = put form at (-20.0, 0.0) in main;
f2 = put form at (20.0, 0.0) in main;
(coord of anglePoint(180.0) in main.f1.c) ->
  (coord of anglePoint(0.0) in main.f2.c);
```



## 6 Programming constructs and lists

SDDL also defines typical programming constructs. For instance lists are created and elements accessed in a syntax similar to most other dynamic languages. Since the content of a list is a general Java Object, you can create a list of objects that are not of the same type. Thus, an assignment like

```
l = [2.0, (0.0, 0.0), [], a--b];
```

is perfectly legal, even if not typically very useful! A more typical use of lists would be

```
list = [(0.0, 0.0), (10.0, 10.0)];
l = list[0];
l2 = list[1];
l--l2;
```

Lists are nice, but to use them properly, adequate programming constructs are needed, such as for and while loops. SDDL defines those, permitting us to draw arrays of Shapes or define points through a simple list.

The ability to use loops significantly simplifies many diagrams in computer science. In this example, we first define a list of points. Afterward, using a for loop, we link those points into a linear Path.

```
a = [(10.0, 10.0), (20.0, 10.0),
     (20.0, 20.0), (10.0, 20.0)];
for i from 0 to 2 do
```

```
  a[i]--a[i+1];
end
```

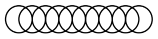
Alternatively, we could get the same result using the following while loop.

```
i = 0;
while i != 3 do
  a[i]--a[i+1];
  i = i + 1;
end
```

## 7 Functions

SDDL also permits the creation of functions inside the code. Those must be written before the diagram description. A SDDL function is defined with the keyword `fun`. The list of arguments does not need to be typed, only named. The return type (if it exists) does not need to be specified either. To call the function in the resulting code, a typical C-like call is used.

```
fun dropMany(s, i, v, n, c){
  for j from 1 to n do
    put s at i + j*v in c;
  end
}
circle = Circle with [radius = 5.0];
dropMany(circle, (0.0, 0.0),
         (5.0, 0.0), 10.0, main);
```



In this case, we define a function that drops many instances of a `Shape` *s* at different positions on a `Canvas` *c*. The positions of the `Shape` are determined by an initial point *i* and a translation vector *v*. Finally, the number of `Shape`s placed is also given as a parameter *n*. Each `Shape` is then placed at the initial point translated by the translation vector a certain number of times. This permits us to create many circles along one line.

One important point is the parameter for the `Canvas` in which we place the `Shape`s. However, we pass *main* as our parameter. The reason for this is that SDDL does not possess global variables. A function can only access its parameters and variables that have been defined inside the function. Thus, trying to place something in *main* from inside a function will yield an error message saying the variable *main* is unknown.

## 8 Primitives

SDDL, as mentioned before, is written in Java. One of the nice features of Java is its libraries. Many classes have been written for any number of different tasks and the number of functions and algorithms implemented is astonishing. To rewrite libraries that are already defined in Java or to link them one by one in SDDL seemed pointless. It is much nicer to directly use those functions, since they're already there.

Since Java is a reflexive language, in the sense that the program knows about itself and can modify itself at will, functions can be dynamically accessed at runtime. We already use this feature to simplify the definition of the SDDL interpreter: the interpreter doesn't have to know every possible type of `Shape` in order to work. Using these same mechanisms permits loading classes at runtime and gives the user the power to call from inside SDDL any Java function.

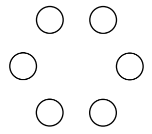
This is done through so-called primitives, which are defined using the keyword `primitive` followed by the name of the function, as it will appear in SDDL. Following that, two strings are given: one for the package in which the function is defined and one for the name of the function as it was defined in Java. Any primitive declaration must be made prior to any function definition.

Let's look at an example. One of the main things a diagram description language would require is some basic mathematics and geometry libraries. Java possesses a nice `java.lang.math` package which we would like to use, especially the `sin` and `cos` functions. For this, we will need to define two primitives.

```
primitive static sin "java.lang.Math" "sin";
primitive static cos "java.lang.Math" "cos";
```

Once they are defined, they can be called just like any other SDDL functions. In this case, we will use those functions to place circles around an invisible circle.

```
for i from 1 to 6 do
  put Circle with [radius = 5.0] at
            (20.0 * cos(i * 3.14 / 3.0),
             20.0 * sin(i * 3.14 / 3.0))
            in main;
end
```



## 9 A concrete example

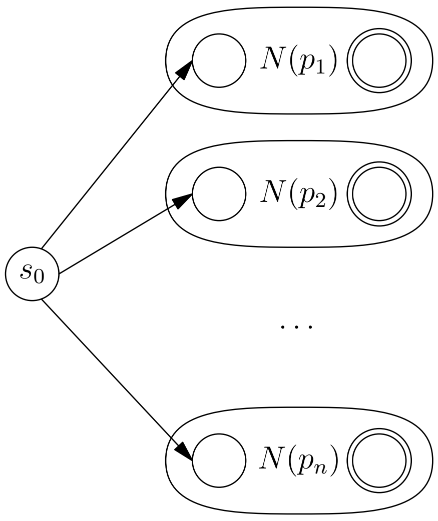
Now that we have all our tools, let's use SDDL to describe a simple diagram of an automaton. This is the complete code, along with the resulting image.

```
circle1 = Circle with [radius = 10.0];
circle2 = Circle with [radius = 12.0];
ellipse = Ellipse with [xradius = 50.0,
                        yradius = 20.0];
circle = put circle1 at (-30.0, 0.0) in form;
put circle1 at (30.0, 0.0) in form;
put circle2 at (30.0, 0.0) in form;
put ellipse in form;

d1 = put form at (100.0, 100.0) in main;
d2 = put form at (100.0, 50.0) in main;
d3 = put form at (100.0, -50.0) in main;
put Text with [text="$N(p_1)$"]
         at (100.0, 100.0) in main;
put Text with [text="$N(p_2)$"]
         at (100.0, 50.0) in main;
put Text with [text="\dots"]
         at (100.0,0.0) in main;
put Text with [text="$N(p_n)$"]
         at (100.0, -50.0) in main;

put Text with [text = "$s_0$"]
         at (0.0, 20.0) in main;
d4 = put circle1 at (0.0, 20.0) in main;

(coord of anglePoint(70.0) in main.d4)->
(coord of anglePoint(180.0) in main.d1.circle);
(coord of anglePoint(0.0) in main.d4)->
(coord of anglePoint(180.0) in main.d2.circle);
(coord of anglePoint(-70.0) in main.d4)->
(coord of anglePoint(180.0) in main.d3.circle);
```



## 10 Modifying the hierarchy by adding shapes

Using what has been described above, most diagrams can be created. However, many simplifications can be made and some domains may find it relevant to add classes specific to their trade. SDDL eases the modification of the Java hierarchy underneath the language by using reflexivity. Any user who respects some basic guidelines will be able to have its class work automatically in SDDL without any specific linking required.

To do this, every class must possess a certain number of properties. Those properties are defined through setters and getters. For instance, for the radius of a circle, these are called `setRadius` and `getRadius`. Any property that follows this rule will be accessible.

Furthermore, an empty constructor must also be defined that sets, as much as possible, default values for all properties of the defined `Shape`. There are also some small functions to be defined, like the `draw` and `obtainPath` functions. Any reference points must also be defined with a function to obtain them.

As an example, let's say we would like to add a class to represent a cloud. A cloud will have a number of "spikes" and a size. The class definition (without function definition) that we would require would be the following:

```
public class Cloud extends Shape{
  double size;
  int numberOfSpikes;
  public Cloud(){...}
  public void setSize(double size){...}
  public double getSize(){...}
  public void setNumberOfSpikes(int n){...}
  public int getNumberOfSpikes(){...}
  public void draw(AbstractPoint a,
                   PrintWriter w){...}
  public Path obtainPath(){...}
}
```

## 11 Future additions and availability

SDDL is available now at `http://www.info2.uqam.ca/~villemaire_r/Recherche/SDDL/`. It is functional, though not by any means complete. Many additional `Shape`s could be added (in particular tree and graph classes) and options could be added to existing classes. Development of the application continues for the time being and a more thorough version will be made available.

## References

[1] John C. Bowman and Andy Hammerlindl. Asymptote: A vector graphics language. *TUGboat: The Communications of the TEX Users Group*, 29:288–294, 2008.

◇ Mathieu Bourgeois
Université du Québec à Montréal
Montréal, Canada
`bourgeois dot mathieu dot 2 (at) courrier dot uqam dot ca`

◇ Roger Villemaire
Université du Québec à Montréal
Montréal, Canada
`villemaire dot roger (at) uqam dot ca`
`http://intra.info.uqam.ca/personnels/Members/villemaire_r`