# The diagram Package

Andreas Kupries ActiveState Software Inc. 409 Granville Vancouver, BC CA
andreask@ActiveState.com

## ABSTRACT

This is a small introduction to the diagram Package, a Tcl [6] based diagramming language and package inspired by Brian Kernighan's pic [1] processor. After a brief history and examples of its use we look into the internals and possibly interesting directions for the future.

## 1. OVERVIEW

diagram is a package for the easy construction of diagrams (sic), i.e. 2D vector graphics, on a Tk [7] canvas or other API compatible object. As such it is not a replacement for canvas, but a layer on top which makes it easier to use by abstracting away the minutiae of handling coordinates to position and size the drawn elements. The user can concentrate on the content of the diagram instead.

This is similar to Brian Kernighan's pic [1] language for troff [2], which is the spiritual ancestor of this package.

The remainder of this paper is structured as follows: In the next chapter an anecdotal overview of the history of this package is provided, followed by a chapter showing examples of the language with their results. After that, in chapter 4, a general overview of its design, implementation, and features is given. Lastly chapter 5 discusses possible applications and future directions for the package.

## 2. HISTORY

diagram's genesis is a long one, although with nothing much happening for long stretches of time.

The original idea was to have a package or tool with which to specify diagrams in writing, instead of having to click through an endless series of dialog boxes for object properties and the like. I had already played with Brian Kernighan's pic [1] in the past and wanted something like that. No dealing with absolute coordinates and the pain of minutely positioning objects to get the right alignment.

Nothing came of that for a long time, except that Arjen Markus got infected by the bug and wrote a small proof of concept. He put it up on the wiki first [3], and later put the code into tklib [8]. Then after the hassle I had with making the diagrams for last year's paper [14], struggling with the applications dia and InkScape I finally had enough motivation to put something together which went beyond a proof of concept.

First I based my work on Arjen's code, but abandoned it because the interface did not feel very smooth or Tclish. Then I tried to to work something out using the Tcllib Word Interpreter package [12], which forced a forthish style. That code became pretty much unusable due to escalating complexity of the internals while trying to fit in vector and point arithmetic. At last I went back to the basic roots of Arjen's code, i.e. using Tcl as the interpreter, and commands as the API. The main trick which got added in this iteration was the use of namespace unknown handlers to provide various pseudo-commands, which made the vector arithmetic and other things easy to write without requiring an infinite family of commands.

This third iteration became the code described here, also replacing the original proof of concept in tklib [8].

## 3. EXAMPLES

A few examples of what can be done with diagram, from the very trivial to the complex.

The basic shapes can be seen in figure 1.

Figure 2 demonstrates drawing of splines.

Figure 3 shows how most positioning can be done relative to previous elements, without using any coordinates at all.

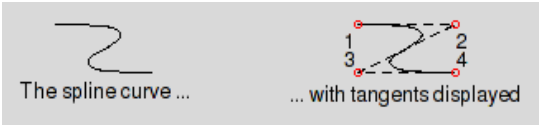
Tw more examples of such relative positioning are seen in the figures 4 and 5 which place things on the line between two points and at the intersection of 2 lines.

At last a larger example in figure 6, without the diagram description.

## 4. DESIGN & IMPLEMENTATION

The inner structure of diagram, i.e. the set of internal packages and their dependencies can be seen in figure 7. The main division is between a core managing the general drawing state and a package implementing the basic shapes seen in figure 1. The drawing state handled by the core is further split into the processing of element attributes, a database of named directions, i.e. angles, a database of the drawn elements, and the state of the automatic element layout. A supporting package provides commands to construct points and vectors and perform vector arithmetic on them. This is partially a repackaging of math::geometry, mainly adding type-tags differentiating absolute and relative coordinates.


*Tcl* '2010 Oakbrook Terrace/Chicago, IL, USA
.



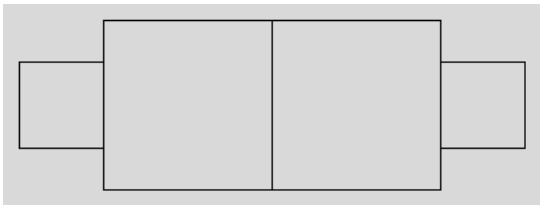

```
# -*- tcl -*- tcl.tk//DSL diagram//EN//1.0
proc showcorners {e corners} {
    foreach {c anchor text} $corners {
        circle radius 2 at [$e $c] \
            color red text $text \
            anchor $anchor
    }
}

block {
    down
    block {
        spline right \
            then down [1 cm] left \
            then right
    }
    text {The spline curve ...}
}
move
block {
    down
    block {
        line dashed                    right \
            then down [1 cm] left then right
        spline from [last line start] right \
            then down [1 cm] left then right
        showcorners [last line] {
            1 ne 1  2 nw 2  3 se 3  4 sw 4
        }
    }
    text {... with tangents displayed}
}
```

**Figure 2: Drawing of splines**

At the language level the public drawing commands are defined in an internal namespace for the lightweight encapsulation of the system's overall configuration. This is modeled after `pic` [1], using variables to hold the default values for various lengths and other attribute values. The core mostly provides the commands to declare such variables, the associated attributes, etc. and only declares a very small set of the language itself. The large majority of the drawing language is then provided by the package for the basic shapes.

Another reason for using a namespace was the ability to implement the special forms of the language (see table 1) via a namespace-specific unknown handler, allowing them for the language without polluting the global namespace and possibly affecting other parts of the application the package is used in.

Encapsulation in a full-blown safe interpreter was considered for the implementation, but ruled out as too restrictive and possibly a premature optimization. As the diagram application package shows, encapsulation in a safe interpreter can be added after the fact, where needed. So this is not something required for the implementation. With the current setup of keeping everything in a namespace a diagram specification is able to access the global and other namespaces should this is needed and.or allowed for a particular application.



```
# -*- tcl -*- tcl.tk//DSL diagram//EN//1.0

box
box width [4 cm] height [4 cm]
box same
box
```

**Figure 3: Relative positioning**

## 5. FUTURE DIRECTIONS

To conclude this document some thoughts on the future of `diagram`, i.e in which directions we can go with it.

1. The package has a variety of possibilities for extension, allowing the implementation of higher DSLs for specific types of diagrams, like mathematical formulas, chemical structures, or visual record descriptions (data types, file formats, ...).

   (a) Diagrams can define their own commands for custom aggregation of shapes or to create new shapes.

   (b) The package's extension APIs enable the declaration of new attributes, elements (shapes), named angles, and commands, the latter either like procedures, or as aliases to to command prefixes. They also allow the registration of command prefixes to call when encountering unknown attributes, enabling special forms for attribute specification.

2. The `dia application` written to work on top of the package[1] currently uses the packages `canvas::snap` [9] and `Tkimg` [5] to produce raster images from descriptions, and `canvas::mvg` [10] to generate `ImageMagick` [16]'s MVG vector format.

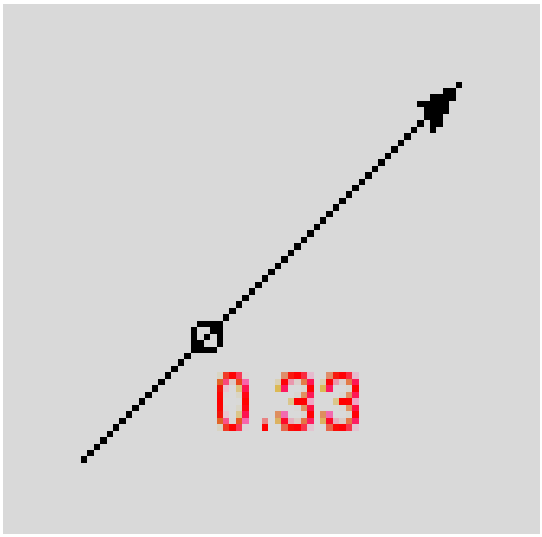
   Obviously, any extension of `Tkimg` [5] will automatically extend the number of formats supported by diagram.

   There is also a package, hidden in the `Coccinella` [15] chat application, to dump the contents of a canvas widget to the SVG format. This has not been integrated yet.

3. Both of the converters to vector formats work by using canvas' introspection abilites to determine the items to convert, and their attributes. This has the disadvantage of requiring a canvas widget, therefore Tk, therefore X11.

   So, another direction would be the creation of object classes which are API compatible to canvas, but whose machinery does not require X11. Karl Lehenbauer's `tcl.gd` [17] comes to mind, for example.

[1]Do not confuse it with the X11 based drawing application I mentioned in chapter 2




```
# -*- tcl -*- tcl.tk//DSL diagram//EN//1.0

arrow up right
circle radius 2 \
    at [0.33 between \
            [last arrow start] \
            [last arrow end]] \
    text 0.33 anchor nw textcolor red
```

**Figure 4: Interpolated positioning**

4. As `diagram is` mainly a package, it can be integrated whereever simple 2D vector graphics are required. One example I hope to pursue is integraqtion within the `doctools` [13], to give writers of manpages access to graphics without having them to leave the environment.

# APPENDIX

# A. REFERENCES


[1] Brian Kernighan, pic.
`http://www.troff.org/papers.html`

[2] troff
`http://www.troff.org/`

[3] Arjen Markus. Drawing diagrams.
`http://wiki.tcl.tk/13434`

[4] Wolf-Dieter Busch, canvas2mvg.
`http://wiki.tcl.tk/26859`

[5] Jan Nijtmans, tkImg.
`https://sourceforge.net/projects/tkimg/`

[6] Various, Tcl.
`https://tcl.sourceforge.net`

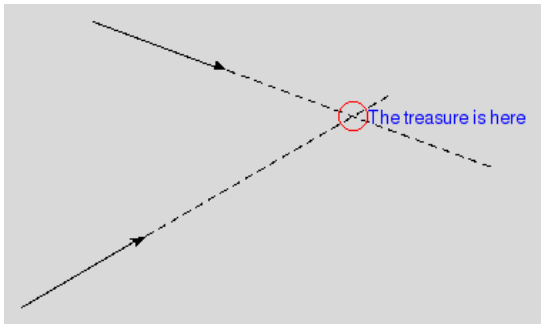
[7] Various, Tk.
`https://tcl.sourceforge.net`

[8] Various, Tklib.
`https://sourceforge.net/projects/tcllib/`

[9] Various, canvas::snap.
`https://sourceforge.net/projects/tcllib/`

[10] Various, canvas::mvg.
`https://sourceforge.net/projects/tcllib/`

[11] Various, Tcllib.
`https://sourceforge.net/projects/tcllib/`


```
# -*- tcl -*- tcl.tk//DSL diagram//EN//1.0

proc extend {s e} {
    line dashed from [$e end] \
        to [$s between [$e start] [$e end]]
}

proc dot {p anchor text} {
    circle radius 10 at $p color red
    text with w at [last circle e] \
        text $text anchor $anchor \
        textcolor blue
}

set A [arrow from [ 0    0] to [by 100 -20]]
extend 3 $A

set B [arrow from [-50 200] to [by 100  30]]
extend 3 $B

dot [intersect $A $B] w {The treasure is here}
```

**Figure 5: Computing intersections**


[12] Various, Word Interpreter.
`https://sourceforge.net/projects/tcllib/`

[13] Various, Documentation Tools.
`https://sourceforge.net/projects/tcllib/`

[14] Andreas Kupries, Reflected And Transformed Channels, Tcl 2009, Portland.
`http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2009.html`

[15] Various, Coccinella.
`http://wiki.tcl.tk/4005`

[16] Various, ImageMagick.
`http://wiki.tcl.tk/12849`,
`http://www.imagemagick.org/`

[17] Karl Lehenbauer, tcl.gd.
`http://code.google.com/p/flightaware-tcltools/`


```
# -*- tcl -*- tcl.tk//DSL diagram//EN//1.0

box "box" ; move
line "line" "" ; move
arrow "arrow" "" ; move
circle "circle" ; move
ellipse "ellipse" width [3 cm] ; move
group { arc "arc" } ; move
diamond "diamond" height [2 cm] ; move
drum "drum" ; move
text text "text"
```

**Figure 1: The basic shapes**

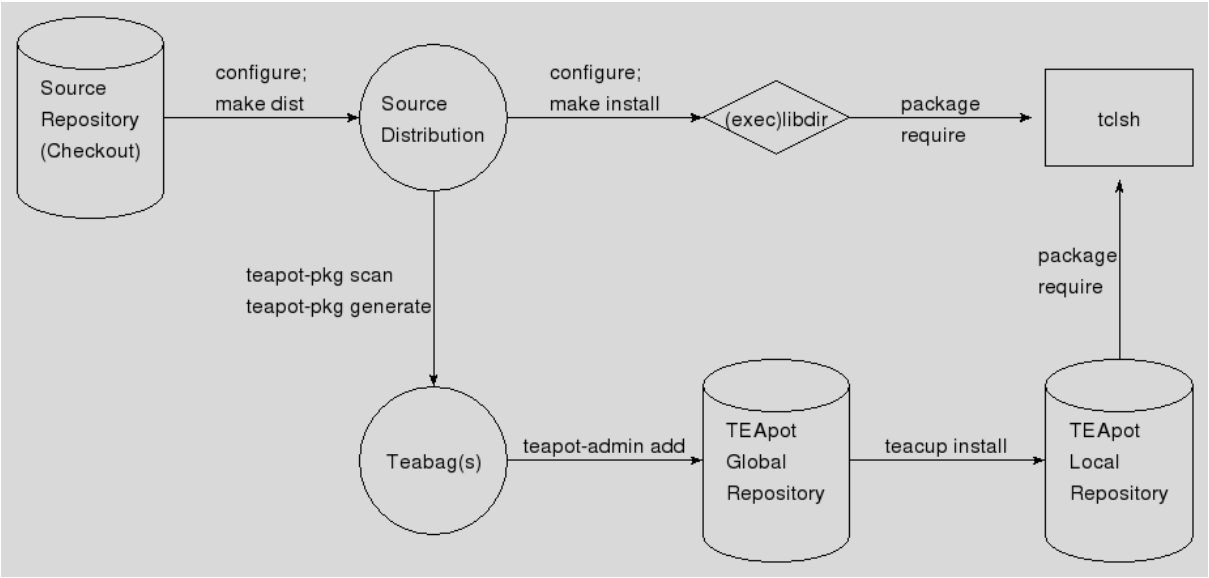



**Figure 6: Teapot Dataflow**

| Syntax | Meaning |
|---|---|
| *named-direction* | Tell the layout engine to follow this direction from now on as default for new elements. |
| *number* cm\|mm\|inch\|point | Return pixel equivalent for the length. This uses `tk scaling` as basis for the conversion. |
| *number number* | Construct an absolute location. |
| *number* between *point point* | Construct point by linearly interpolating between two others. |
| *point* by *distance direction* | Construct point by moving from the given point a certain distance in the specified direction. |
| *point* +\|- *point* | Vector arithmetic on two points. |
| *nth shape?corner?*<br>*nth* last *shape?corner?*<br>last *shape?corner?*<br>*nth* last *?corner?*<br>last *?corner?* | Access to named locations in previously drawn elements. |
| *element?corner...??*names *pattern?* | Access to named locations in elements. |

**Table 1: Special command forms**

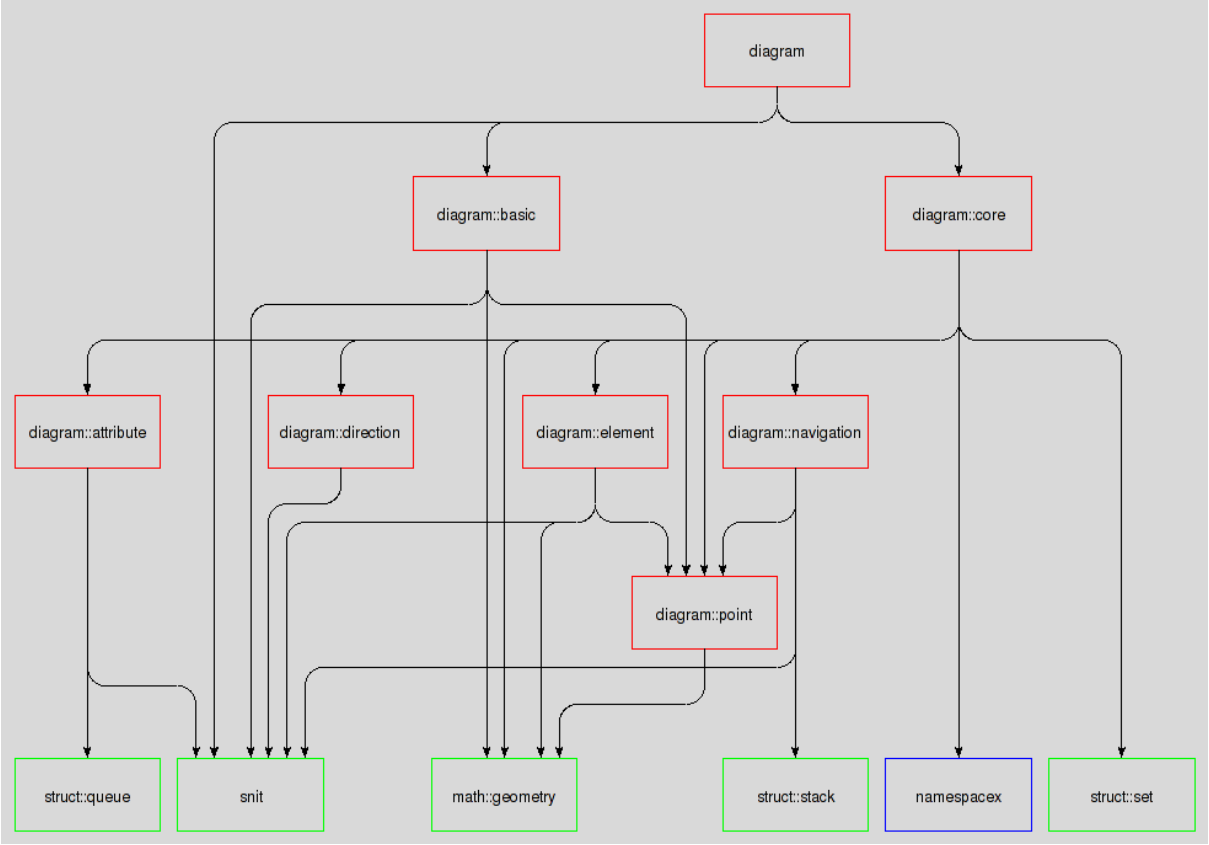



**Figure 7: diagram package and its dependencies**