Towards a One-Stop-Shop for Analysis, Transformation and Visualization of Software

Paul Klint[1,2], Bert Lisser[1], and Atze van der Ploeg[1]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] INRIA Lille Nord Europe, France

Abstract. Over the last two years we have been developing the meta-programming language RASCAL that aims at providing a concise and effective language for performing meta-programming tasks such as the analysis and transformation of existing source code and models, and the implementation of domain-specific languages.
However, meta-programming tasks also require seamlessly integrated visualization facilities. We are therefore now aiming at a "One-Stop-Shop" for analysis, transformation and visualization. In this paper we give a status report on this ongoing effort and sketch the requirements for an interactive visualization framework and describe the solutions we came up with. In brief, we propose a coordinate-free, compositional, visualization framework, with fully automatic placement, scaling and alignment. It also provides user interaction. The current framework can handle various kinds of charts, trees, and graphs and can be easily extended to more advanced layouts. This work can be seen as a study in domain engineering that will eventually enable us to create a domain-specific language for software visualization. We conclude with examples that emphasize the integration of analysis and visualization.

1 Introduction

Over the last two years we have been developing the meta-programming language RASCAL[1] [6] that aims at providing a concise and effective language for performing meta-programming tasks such as the analysis and transformation of existing source code and models, and the implementation of domain-specific languages. RASCAL is completely written in Java, integrates with Eclipse and gives access to programmable IDE features using IMP, The IDE Meta-Tooling Platform[2].

Given the large amounts of facts that emerge when analyzing software, meta-programming tasks also require seamlessly integrated visualization facilities We are therefore now aiming at a "One-Stop-Shop" for analysis, transformation and visualization. In this paper we give a status report on this ongoing effort and sketch the requirements for an interactive visualization framework and describe the solutions we came up with. In brief, we propose a coordinate-free, compositional, visualization framework, with fully automatic placement, scaling and alignment. It also provides user interaction.

[1] http://www.rascalmpl.org/
[2] http://www.eclipse.org/imp/

Software visualization is a relatively young and broad domain [3] and there are several ongoing efforts to create software vizualization frameworks, for instance, Rigi [14], Bauhaus,[3] and the Mondrian [10] visualization component of Moose[4] to mention just a few. In addition there is much excellent work in creating and exploring very specific visualizations ranging from version and process histories to application overviews, module interconnections and class structures, see, for instance, [3] or the proceedings of SoftVis.[5]

Software visualization systems can be positioned in a spectrum of visualization approaches and tools that differ in the level of automation and specialization for a specific domain. At the high end of the spectrum are domain-specific visualization systems. For instance, in the business domain, a limited number of visualizations can cater for the majority of visualization needs. As a case in point, Excel[6] provides a dozen chart types including line charts, bar charts, area charts, pie charts, histograms, Gantt charts and radar charts. Although all these charts are customizable, it is complex—and requires explicit low-level programming—to add a completely new chart type to Excel. Other examples in this category are GnuPlot[7] that aims at graph plotting in the scientific domain and Many Eyes[8] that aims at charts and data visualization.

At the low end of the automation and specialization spectrum are graphics package like AWT[9], Java2D[10], SWT[11] and Processing[12] that provide low level graphics. Everything is possible but has to be implemented from scratch using low-level primitives. On a slightly higher automation level is a system like Protovis[13] [1] that uses a declarative, data-driven, approach for data visualization. There is no global state and visual attributes are combined based on inheritance. However, mapping of measures to graphical attributes has to be programmed explicitly and scaling them requires code changes.

The domain of visualizing software facts exhibits too much variability and is too diverse to be covered by a limited set of standard visualizations from the high end of the spectrum. Unfortunately, using low-level primitives to implement software visualizations from scratch is time-consuming, error-prone and requires manual integration with the fact extractor. What is needed is a software visualization framework that takes the middle ground.

As an analogy, consider the field of layouts in user-interfaces: there is no set of layouts that can cater for all layout needs but manually programming them involves tedious computations for alignments, sizing and resize behavior as well as manual integration of the user interface elements with the layout. This has been solved by

[3] http://www.bauhaus-stuttgart.de/bauhaus/index-english.html
[4] http://www.moosetechnology.org/news/moose-4-0
[5] http://www.softvis.org
[6] http://office.microsoft.com/en-us/excel-help/available-chart-types-HA001034607.aspx
[7] http://www.gnuplot.info/
[8] http://www-958.ibm.com/software/data/cognos/manyeyes/
[9] http://java.sun.com/products/jdk/awt/
[10] http://java.sun.com/products/java-media/2D/index.jsp
[11] http://www.eclipse.org/swt/
[12] http://processing.org/
[13] http://vis.stanford.edu/protovis/

automatic layout managers such as the ones in Tcl/TK [11]. We aim to do the same for software visualization: alleviate programmers of tedious tasks by providing just the right level of abstraction and automation. Thus we aim to develop a software visualization framework that:

- enables non-experts to easily create, combine, extend and reuse interactive software visualizations;
- integrates seamlessly with existing techniques for software analysis (parsing, pattern matching, tree traversal, constraint solving) and software transformation (rewriting, string templates).

Our main objective is to liberate the creator of visualizations from many low-level chores, such as programming of explicit coordinates, mapping metrics related to software properties to sizes of shapes and figures, and to provide high-level features instead, like figure composition, fully automatic figure placement and symbolic links between visualizations elements. Since RASCAL already provides excellent facilities for software analysis and transformation, the main challenge is therefore to design a software visualization framework that integrates well with and provides full access to what RASCAL has to offer. Our contributions can be summarized as follows:

- A compositional, coordinate-free, visualization framework that provides primitives for drawing elementary shapes and composite figures.
- Mechanisms to associate numeric scales with arbitrary figures.
- The first attempt we are aware of to decompose charts into reusable primitives.
- The integration of this framework with the RASCAL language and infrastructure thus creating a true ”One-Stop-Shop” for software analysis, transformation and visualization.
- An analysis of the software visualization domain that can form the basis for a domain-specific language for software visualization.

The paper is organized as follows. In Section 2 we identify requirements, elaborate on the design principles we have adhered to and describe the actual design we came up with. In Section 3 we present some examples and in Section 4 we draw conclusions.

2 Requirements, Design and Architecture

We will now first summarize our requirements and global design (Section 2.1) and describe our global architecture (Section 2.2). In subsequent sections we will provide more details.

2.1 Requirements

In scientific visualization, the data of interest represent objects or processes in the physical world and thus mapping of these data to visual attributes (e.g., location, size, color, line width, line style, gradient, and transparency) is mostly dictated by physical properties or appearance. In software visualization, and in information visualization

in general, the data is more abstract and the mapping of the data to visual attributes is not prescribed and not easy to design. A key problem when designing a software visualization is to find good visual abstractions for software artifacts. Our software visualization framework should therefore make it easy to describe such mappings and to enable experiments to find the most appropriate ones. As a consequence the framework should provide reusable primitives for expressing such mappings. Typical use cases to be supported are the visualization of

- hierarchical and non-hierarchical software structures,
- (multi-dimensional) metrics associated with software components,
- functions that express software properties over time.

The visualization framework should not only promote creating new visualizations but should also include standard visual layouts (e.g., graphs, trees, tree maps) and primitives for common chart types (e.g., bar charts, scatter plots, Gantt charts). To achieve this goal, we reason that the software visualization framework should be *automatic and domain-specific*, *reusable*, *compositional* and *interactive*.

Automatic and Domain-specific. We aim for *as much as possible* automation and specialization for our software visualization framework. This implies eliminating low-level representation chores, such as layout and size computations, and introducing concepts that are specialized for the software visualization domain. There is no fixed border between general information visualization and software visualization and it is fruitful to exchange ideas and concepts between the two. We aim for very general solutions that have direct application in the software visualization domain.

Reusable. Creating visualizations is difficult and their reuse should therefore be enabled as much as possible. This implies that visualizations are treated as ordinary values and can, for instance, be used as arguments of a function or be the result computed by a function. The same applies to visual attributes. We want to be able to resize, parameterize and combine existing visualizations into new ones. A corollary is that arbitrary combinations and nesting of visualizations should be allowed and that the composition of visual attributes should be well-defined.

Compositional. Many graphics approaches are highly imperative. The color or line style of a global *pen* can be set and when a line is drawn these pen properties are used. As a consequence, drawing sequences that assume a different global state cannot be easily combined. We aim for declarative visualizations in which each visualization is self-contained, can be drawn independently, is easily composable and thus contributes to reusability. For some related work we refer to [5,4], HaskellCharts[14] and Degrafa[15]. Compositionality is a desirable goal but limits the solution space. The main challenge is to cover the whole spectrum of possible software visualizations with compositional primitives.

[14] `http://dockerz.net/twd/HaskellCharts`
[15] `http://www.degrafa.org/`

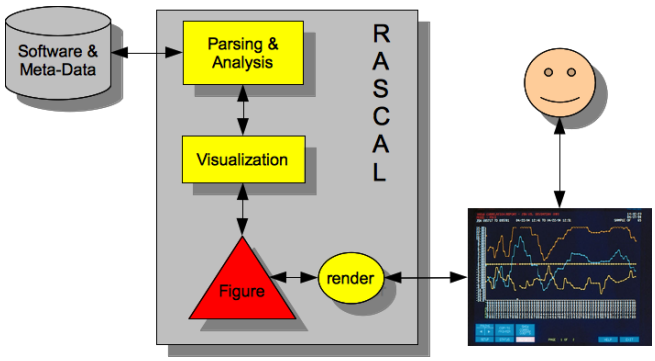


Fig. 1: Architecture of Figure visualization framework

Interactive. In the use cases we envisage, overwhelming amounts of data have to be understood and Schneiderman's *Overview First, Zoom and Filter, then Details-on-demand* mantra [12] applies. We need interaction mechanisms to enable the user to start with an overview visualization and to zoom in on details. This may include mixing visualization with further software analysis and requires tight integration with an Interactive Development Environment (IDE). For instance, given an initial overview of a software system, a user may interactively select a subsystem, fill in a dialog window to select the metrics to be computed, and inspect a new visualization that represents the computed metrics. We aim for a mix of automatic, out-of-the box, interactions and programmatically-defined ones. Since we will support various interaction elements (e.g., buttons, text fields) as well as layouts we achieve integration of pure visualization and user-interface construction.

2.2 Architecture

The technical architecture of our visualization framework is shown in Figure 1. The given *Software & Meta-Data* is first parsed and then relevant analyses are performed (*Parsing & Analysis*). Parsing and analysis can be completely defined and arbitrary languages and data formats can therefore be parsed and analyzed. The analysis results are then turned into a figure (*Visualization*) and the result is an instance of the `Figure` data type, an ordinary RASCAL datatype that is used to represent our visualizations. Note, for later reference, that another data type, `FProperty`, is used to represent all possible visual properties of Figures. Figures are interpreted by a `render` function that transforms them in an actual on-screen display with which the user can interact. There is *two-way communication* between visualization and user: the visualization functions create a figure that is shown to the user, but this figure may contain call backs (RASCAL functions) that can be activated by user actions like pointing, hovering, selecting or scrolling.

Operator	Description
`text`	A text string
`outline`	Rectangular summary (with highlighted lines) of a source code file
`box`	A rectangle (may contain nested figures)
`ellipse`	An ellipse (may contain nested figures)

Table 1: Primitives and Sample Containers

Operator	Description
`id`	Name of subfigure (used for cross references)
`grow`	Horizontal (`hgrow`) and vertical (`vgrow`) size relative to children
`shrink`	Horizontal (`hshrink`) and vertical (`vshrink`) size relative to parent
`resizable`	Resizable in horizontal (`hresizable`) and vertical (`vresizable`) direction
`align`	Horizontal (`halign`) and vertical (`valign`) alignment
`lineWidth`	Width of lines
`lineColor`	Color of lines
`fillColor`	Fill color for closed figures

Table 2: Sample Properties

2.3 Figures and Properties

As already mentioned, visualizations are ordinary values and we use the datatypes `Figure` and `FProperty` to represent them. All primitives and properties will be represented as constructor functions for the datatypes `Figure` and `FProperty` .[16] In order to be able to give meaningful examples, we give some samples of both.[17]

Figure Primitives The primitive figures and some containers are listed in Table 1. The primitives `text` and `outline` are atomic in the sense that they cannot contain subfigures: `text` defines, unsurprisingly, text strings and `outline` is a rectangle with a list of highlighted lines inside that can be used to summarize findings on a specific source code file. The primitives `box` and `ellipse` are actually non-atomic containers that may contain a subfigure (e.g., box in a box).

Figure Properties Some figure properties are listed in Table 2, and they can define size, spacing, and color. A figure may have a name which is defined by the `id`-property (e.g., `id("A")`) and is used to express dependencies between figures and between properties.

Several properties are related to size, alignment, and space assignment when figures are being composed. We discuss these size-related properties together in later sections. Other properties define visual aspects such as color, line style and font, or specific properties of shapes. Properties for associating interactive behavior with a specific figure are given later in Table 5.

[16] In Section 4, we speculate on using the syntax definition facilities of RASCAL and giving a textual syntax to them, thus creating a true visualization DSL.

[17] Our complete framework provides dozens of primitives and properties. In this and following tables we only list items relevant for the current paper.

Operator	Description
`hcat`	Horizontal composition, grid with one row
`vcat`	Vertical compostion, grid with on collumn
`hvcat`	Horizontal and vertical compostion (resembling placing words in a text paragraph)
`overlay`	Stacked composed, i.e., figures are overlayed in the z-dimension.
`grid`	Place figures in a grid
`pack`	Place figures as close together as possible (bin packing)
`graph`	Place figures and edges in graph layout
`tree`	Place figures and edges in tree layout
`treemap`	Place figures in a treemap layout

Table 3: Composition Operators

Example The expression `box(fillColor("red"), lineColor("green"))` will create a red rectangle with a green border. When rendered, this rectangle will occupy all available space in the current window.

Property Inheritance Since figures and their subfigures usually have the same settings for most of their properties it is cumbersome to set all the properties of every figure individually. It should therefore be a possible to inherit properties from a parent figure, but a model where all properties are inherited is cumbersome as well: if a property should only apply to the current figure and not to its children then the programmer has to explicitly reset that property for all the children. Therefore we have opted for a model does not introduce this chore, but does allow inheritance:

- All properties are initialized with a standard (default) value.
- A property only applies to the figure in which it is set.
- A property can redefine the standard value of a property: that new standard value is then used in all its direct and indirect children (unless some child explicitly redefines that property itself). For example, `lineColor` defines the line color for the current figure, `std(lineColor)` defines it for all direct and indirect children of the current figure.

This model is similar in goal but simpler and more uniform than the inheritance model in cascading stylesheets in HTML, since in our model the inheritance behavior is the same for all properties, while in CSS this may differ per property as explained by Lie [8].

2.4 Figure Composition and Layout

In many approaches to graphics and visualization *coordinates* and *explicit sizes* are the primary mechanisms to express the layout of a visual entity. For instance, a rectangle is specified by its upper-left corner, width and height, or alternatively, by its upper-left and lower-right corner. Although this is common practice, there are disadvantages to this approach [2]:

- Explicit coordinates and sizes lead both to tedious computations to achieve a simple layout and to manual programming of resize behavior. This conflicts with our goals of automation and re-use.

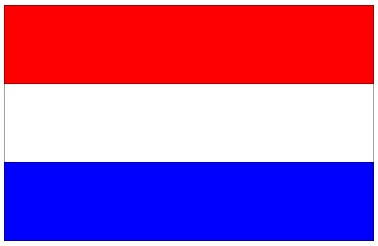


Fig. 2: Dutch Flag

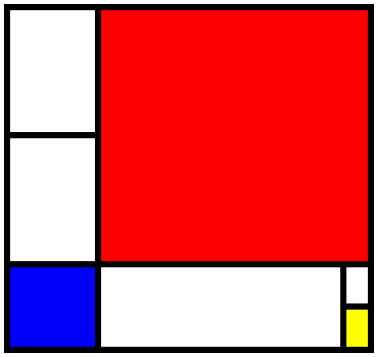
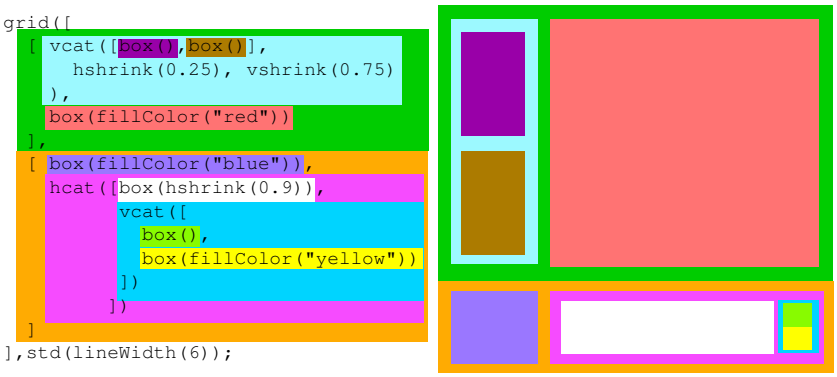


Fig. 3: Our version of *Composition II in Red, Blue, and Yellow* by Piet Mondriaan, 1930

– Explicit coordinates and sizes do not show the spatial relationships between the elements in a layout, this makes re-use and interaction more difficult.

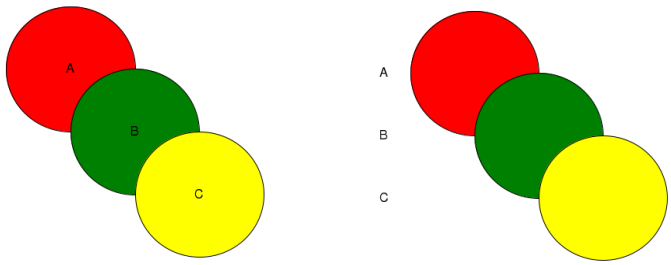
Therefore, we have chosen to avoid explicit coordinates and provide layouts based on figure composition. The position of a figure is described by nesting figures, for example an ellipse inside a box is written as `box(ellipse())` or by using the composition operators listed in figure Table 3. The most fundamental operators provide horizontal, vertical and overlayed (stacked) composition of figures. As an example, a resizable Dutch flag (Figure 2) can be created as follows: `vcat([ box(fillColor(c)) | c <- ["red","white","blue"] ])`.



```
grid([
  [ vcat([box(),box()],
      hshrink(0.25), vshrink(0.75)
    ),
    box(fillColor("red"))
  ],
  [ box(fillColor("blue")),
    hcat([box(hshrink(0.9)),
          vcat([
            box(),
            box(fillColor("yellow"))
          ])
         ])
  ]
],std(lineWidth(6));
```

Fig. 4: Code creating Mondriaan painting, the colors relate each piece of code to the corresponding area in the layout

The size of a figure should depend on its context. In this way we can use the same figure in different contexts, even though these contexts have different sizing requirements.



(a) An overlay (b) Using a screen

Fig. 5: The use of overlay and screen.

One way of expressing the size of a figure in terms of its context is through the `hshrink` and `vshrink` properties, which declare how much a figure shrinks relative to the horizontal or vertical size of its parent. The property `shrink` is a shorthand for setting both `hshrink` and `vshrink`.

As an example, consider our version of the painting *Composition II in Red, Blue, and Yellow* by Piet Mondriaan in Figure 3. Using our layout and sizing mechanisms this painting can be concisely described by the code in Figure 4. The sizes of the boxes are described in terms of their parents. If no sizing properties are given, the figure is given the size that is available.

This wellknown way of defining sizes is, for instance, also used in HTML tables. Another, novel, way of defining sizes in term of their context is through the `hgrow` and `vgrow` properties: which declares how much a figure grows relative to the horizontal or vertical size of its children. For example if we want a box containing the text "Rascal" that is twice as wide and three times as high as the enclosed text we could define this by `box(text("Rascal"),hgrow(2.0),vgrow(3.0))`.

To specify where in the available space a figure is positioned the `halign` and `valign` properties are used. Here 0.0 means completely on the left side or top side and 1.0 means completely on the right side or bottom side. Shorthands such as `left()` are available. Consider the following example (Figure 5a), where the `overlay` composition is used which stacks figures:

```
overlay([
  ellipse(text("A"), left(),top(),
              fillColor("red"),shrink(0.6)),
  ellipse(text("B"), center(),
              fillColor("green"),shrink(0.6)),
  ellipse(text("C"), right(), bottom(),
              fillColor("yellow"),shrink(0.6))
])
```

Now suppose we want to display the same image as our last example, but with the labels on the left as displayed in Figure 5b. We could achieve this effect by positioning the labels in a separate overlay and horizontally composing these two overlays. However

this means that the user must manually specify the alignment of the labels. To raise the level of automation in such cases we introduce the notion of a *screen*. A screen is a horizontal or vertical line on which figures can be projected. Thus to obtain the picture displayed in Figure 5b we place a screen on the left of the overlay and then project a label from each ellipse on this screen in the following way:

```
leftScreen("s",
  overlay([
    ellipse(project(text("A"),"s"), left(),top(),
                fillColor("red"), shrink(0.6)),
    ellipse(project(text("B"),"s"), center(),
                fillColor("green"), shrink(0.6)),
    ellipse(project(text("C"),"s"), right(), bottom(),
                fillColor("yellow"), shrink(0.6))
  ])
);
```

There are also composition for laying out arbitrary figures according to their relation in a graph or a tree since this common in software visualization. We argue that with these special purpose layout operators and the general purpose composition operators described above the user can describe most layouts needed in software visualization in a concise, declarative and reusable way.

2.5 Scales of Measurement

Representing software facts requires mapping measurements to scales. Mapping of software measures to visual properties was pioneered in polymetric views as described in [7]. We want to provide a similar mechanism but make the mapping from measurement to graphic representation explicitly, so that it can later be scaled or manipulated interactively. Traditionally, the following scales of measurement are distinguished [13]:

- A *nominal* scale consist of unordered data points and only equality of data points is defined. This kind of data can, for instance, be represented by text labels or color codes.
- An *ordinal* scale consists of ordered data points; this implies that a comparison between data points is possible but that differences between values are not meaningful.
- An *interval* scale consists of ordered values, with a constant scale, but no natural zero. The typical example are temperatures and dates.
- A *ratio* scale consists of ordered, constant scale, values with a natural zero. Examples are height and age.

We distinguish the following aspects of a scale (of which the last three are inspired by [9]):

- Its classification in the above four categories.
- The figures that are to be mapped to the scale. For example, the bars in a bar chart.

```
vcat([
  hcat( [ box(text(n),fillColor(convert(t,"accessType")))
        | <n,t> <- [<"equals","public">,
                    <"intersects","protected">,
                    <"toString","public">,
                    <"getPassword","private">,
                    <"union","protected">]
         ]),
  colorPallette("Access types of methods",
                "accesType",
                hshrink(0.5))]
)
```

Fig. 6: Example using `colorPallette`

- The specific property whose value is to be mapped to the scale. For example, the height of a bar in a bar chart or the coordinates of a point in a function or scatter plot.
- The figure that explains the mapping. For example, an axis or a color legend.

We introduce a figure type for each kind of scale of measurement and annotate values of figure properties with the name given to the desired scale figure to establish the mapping. All scales are figures, and can be placed in the visualization just like ordinary figures. They are visualized as an explanation of the mapping, i.e., axis or color legend. Our ambition is to support all four scales of measurement as built-in primitives, but currently we only support the nominal and the ratio scale.

Nominal scales Assume that a list of Java method names and their access modifiers are given. We want to visualize each method as a named box with a color that represents its access modifier. The access modifiers can be represented by a nominal scale in which each access modifier is mapped to a different color.

Figure 6 shows how this can be achieved. First we create horizontally concatenated boxes with the method name as text and a fillColor that is determined by the value of each access modifier converted to a nominal scale with name `accessType`. Below these boxes, we then place the nominal scale `colorPallette` with a title and — as expected — the name (`accessType`); it will convert the nominal values that were added in the box declarations to a color on the pallette. The result is shown in Figure 7.

Interval Scales: Axes Figure 8 gives the anatomy of a single bar in a typical bar chart: the bar has a height (a numerical value) that should be mapped to units on the vertical axis, and the bar should also be mapped to a nominal label on the horizontal axis.

To map numeric values, we introduce the notion of an *axis*. An axis takes care of the proper interval and scale of values, and of major and minor tick marks. It has a name and contains a figure with subfigures whose dimensions are mapped using the `convert` operator. Axes exist in different flavors depending on their placement relative to the list of figures.

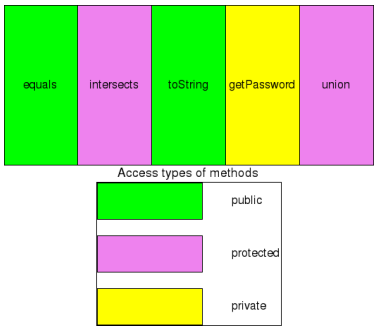


Fig. 7: A nominal scale using a color pallette

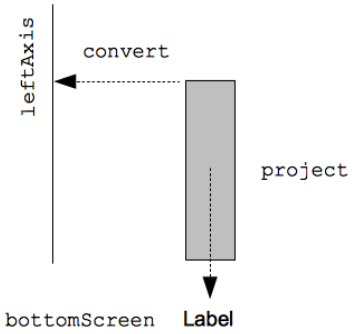


Fig. 8: Anatomy of a bar in a bar chart

```
Figure hBarChart(map[str,num] vals){
        return bottomScreen("categories",leftAxis("y",
                        hcat([box(height(convert(vals[k],"y")),
                                  project(text(k),"categories"),
                                  fillColor("blue"))
                              | k <- vals], hgrow(1.2))
                ));
}
```

Fig. 9: `hBarChart`: a simple bar chart

For instance,

```
leftAxis("y", [ box(height(convert(10, "y"))),
                box(height(convert(15, "y")))
              ])
```

creates a vertical axis with two boxes to the right of it.

Axis and screen form the building blocks for many common chart types and we illustrate in Figure 9 how to apply them to create simple bar charts in the following function `hBarChart` that takes a map from strings to numbers and returns a bar chart that visualizes this information. See Section 3.1 for an application of this function.

2.6 Figure Interaction

A plotting package like GnuPlot, takes a description of a desired plot and delivers a static rendering of it. Given our interaction requirements (Section 2.1) we add properties and operators for interaction as listed in Tables 5, and respectively, 6. The properties allow associating interactive behavior with a specific figure, such as showing a second figure when the mouse is over the first one or handling a mouse click on a figure. The interaction operators represent separate user-interface elements like buttons, checkboxes and text fields, to which RASCAL *closures* (functions and their local context) can be

Operator	Description
`leftAxis`	A vertical axis to the left of a figure, similar: `rightAxis`, `bottomAxis`, `topAxis`
`convert`	Convert a size to a datapoint on an axis
`bottomScreen`	Horizontal projection screen at the bottom of a figure, similar: `topScreen`, `leftScreen`, `rightScreen`
`project`	Project a figure to a screen

Table 4: Axes and Screens

Operator	Description
`mouseOver`	Add a figure when mouse is over current subfigure
`onClick`	Handle mouse clicks
`onMouseOver`	Handle the mouse entering the current subfigure
`onMouseOff`	Handle the mouse leaving the current subfigure

Table 5: Interaction properties

attached as call backs. Although RASCAL has value-based semantics, it does allow assignment to variables and this can be used to represent the state of the user-interface inside the RASCAL program.

Examples In Figure 10 we illustrate how to define a function that returns a figure property, in this case, a `mouseOver` property that will display a yellow box with text when the mouse hovers over the figure which has this property. The box will be 1.2 times larger than the text and it is not resizable.

In Figure 11 the creation of a textfield is illustrated. Note how the `textfield` property has a closure parameter

```
void(str s){TERM = s;}
```

that acts as call back function. It assigns to the environment variable `TERM` and in this way global state can be maintained across calls to call back functions.

Operator	Description
`computeFigure`	Compute a new (sub)figure triggered by interaction
`button`	Button with call back
`textField`	Text entry field with call back
`combo`	Combo box with call back
`choice`	List of choices with call back
`checkbox`	A check box with call back

Table 6: Interaction Figures

```
public FProperty popup(str S){
   return mouseOver(box(text(S),
                    fillColor("lightyellow"),
                    grow(1.2),
                    resizable(false)));
}
```

Fig. 10: Defining a popup

```
    str TERM = ""; // Initial search term
    searchField =
         hcat([ text("Enter search term:"),
                textfield("<TERM>", void(str s){TERM = s;})
              ]);
```

Fig. 11: Defining a text entry field

3 Examples

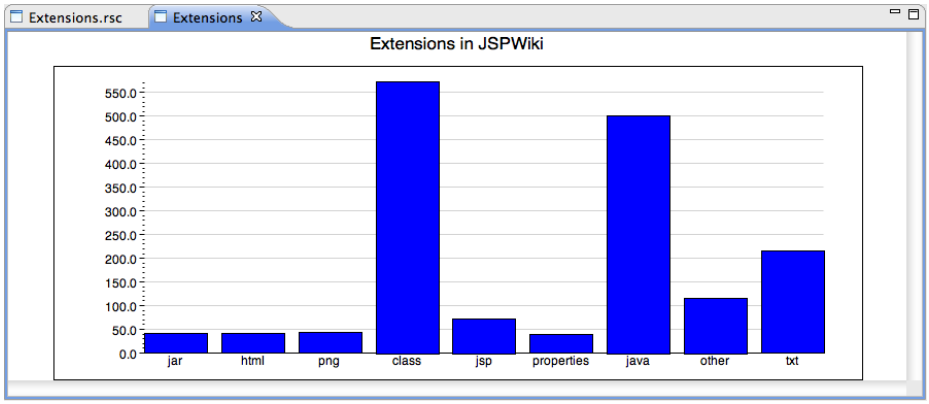


Fig. 12: Bar chart with frequencies of file name extensions in JSPWiki

3.1 Bar Chart of File Name Extensions

The first task we want to solve is to extract all files from an Eclipse project, count the file name extensions (i.e., different file types) and draw a bar chart of the result. The code is shown in Figure 13 and re-uses the `hBarChart` function defined earlier in Figure 9. The auxiliary function `getExtensions` use deep pattern matching (`/`) to search for all files in the project and to count their frequencies. It returns a map from extensions to

frequencies.[18] In the statement `m[l.extension]?0 += 1;` the current value associated with the value of `l.extension` in table `m` is incremented. The binary *undefined operator* `?` caters for the case that that key value is not yet in the table and uses 0 instead. Application to a sample project as in `drawBarChart(|project://JSPWiki|)`, and rendering the result gives the bar chart shown in Figure 12.

```
public Figure drawBarChart(loc project) {
  e = getExtensions(getProject(project));
  return vcat([ text("Extensions in <project.host>",
                                fontSize(17)),
                box(hBarChart(e),grow(1.1))]);
}

// Extract all file extensions
public map[str, num] getExtensions(Resource r) {
  m = ();
  for (/file(loc l) := r)
      m[l.extension]?0 += 1;
  return m;
}
```

Fig. 13: Visualizing file name extensions

3.2 Search and Browse Files

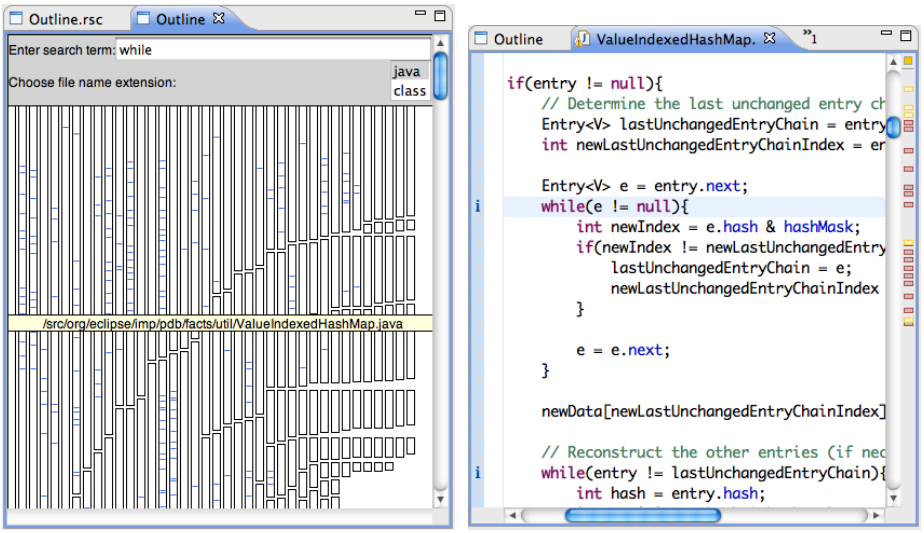
The second final task we want to solve deals with search and file exploration. It integrates analysis, visualization and user interface construction and consists of the following steps:

- Present the user with a search field to enter a search term.
- Present the user with an option to search Java files or class files.
- Present the search results in the form of an outline per file, with colored lines representing hits.
- When hovering over an outline, the corresponding file name is shown as a popup.
- When clicking on an outline, a new editor is opened for the corresponding file, with the occurrences of the search term highlighted.

Figure 14a shows an example of visualizing the results of a query for the term ”while”, and Figure 14b shows an editor that appears when clicking on one of the file outlines in Figure 14a.

The function `mkOutline` (Figure 15) takes the location of a source file, a word to search for, a message to attach to each line, and a scale factor to compute the relative position of hits in the file. It returns an outline figure, with line information associated with it. This outline has two properties that implement interaction:

[18] Extensions with a frequency below 2% are collected in the category ”other”; this is not shown in the code.



(a) Search results for ”while”

(b) Clicking on outline opens editor

Fig. 14: Browsing search results

```
public Figure mkOutline(loc f, str word, str msg, int scale){
  lineInfo = [];
  lines =  readFileLines(f);
  for(i <- index(lines)){
    if(/<word>/ := lines[i])
      lineInfo += info(i + 1, msg);
  }
  return outline(lineInfo,
                 size(lines),
                 size(5, size(lines)/scale),
                 popup(f.path),
                 onClick(void () {edit(f, lineInfo);}));
}
```

Fig. 15: `mkOutline` create an outline with search results

```
public Figure find(loc project, str word, str suffix){
 if(word == "")
    return
     box(text("No results", center()), fillColor("silver"));
 P = getProject(project);
 outlines = [ mkOutline(l, word, "Found: <word>", 2)
            | /file(loc l) := P, l.extension == suffix
            ];
 return pack(outlines,  top(), left(), gap(3));
}
```

Fig. 16: `find`: top level function to create the visualization

```
public void searchGUI(){
    // Create text field for search term
    str TERM = ""; // Initial search term
    searchField =
         hcat([ text("Enter search term:"),
                textfield("<TERM>", void(str s){TERM = s;})
              ]);
    // Create choice box for file extensions
    str EXT = "java"; // Initial file name extension
    extField =
         hcat([ text("Choose file name extension:"),
                choice(["java", "class"], void(str s){EXT = s;})
              ]);
    // Combine both fields in a box
    pane = box(vcat([searchField, extField]),
               fillColor("lightgrey"),
               stdVresizable(false), vgrow(1.1));
    // Create computed figure for visualization of result
    result = computeFigure(
               Figure(){ return find(project, TERM, EXT);}
             );
    // Render the pane and search result
    render("Outline", vcat([pane, result],  left()));
}
```

Fig. 17: `searchGUI`: create the user-interface

– a `mouseOver` that displays the file name when the mouse hovers over this outline; this re-uses the function `popup` discussed earlier in Section 2.6
– an `onClick` properties that defines a parameterless function to be called when a mouse click occurs on this outline. This function calls, in turn, the library function `edit` that opens a source code editor for the given file.

The function `find` (Figure 16), searches through all files with the right file extension, applies `mkOutline` to each file and `pack`s the resulting outlines as densely as possible. Finally, the function `searchGUI` (Figure 17) creates the user-interface and uses `find` as utility.

4 Conclusions

We have presented a high-level overview of the RASCAL visualization framework that we are currently developing and we hope that this overview has convinced you that a declarative, coordinate-free, visualization approach is both feasible and highly applicable. Once the framework has stabilized, we envisage to explore how interaction facilities can be further extended and how animation can be added. We are also planning to systematically describe the most popular chart types with the primitives presented here as starting point. A series of case studies will help to assess the applicability of our framework and to further improve it.

This effort acts as a domain analysis for software visualization and we expect that the concepts we have identified can form the basis for a true DSL for software visualization. You will have observed that we have presented the framework as an abstract data type with prefix constructor functions and this leaves the advanced RASCAL facilities for syntax definition and parsing completely unused. Another next step is therefore to design a concise textual syntax for the visualization primitives presented here and to integrate them even further in the RASCAL language.

Acknowledgements

We thank our colleagues from the RASCAL team for the brainstorms, suggestions and support. Jurgen Vinju commented on drafts of this paper.

References

1. M. Bostock and J. Heer. Protovis: A Graphical Toolkit for Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1121–1128, Sept. 2009.
2. J. Coutaz. A layout abstraction for user-system interface. *SIGCHI Bull.*, 16:18–24, January 1985.
3. S. Diehl. *Software visualization: visualizing the structure, behaviour, and evolution of software*. Springer, July 2007.
4. C. Elliott. Functional images. In *The Fun of Programming*, "Cornerstones of Computing" series. Palgrave, Mar. 2003.
5. S. Finne and S. Peyton Jones. Pictures: A simple structured graphics model. *Glasgow Workshop on Functional Programming*, Jan. 1995.

6. P. Klint, T. van der Storm, and J. Vinju. EASY meta-programming with Rascal. In J. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, editors, *Generative and Transformational Techniques in Software Engineering III*, volume 6491 of *Lecture Notes in Computer Science*, pages 222–289. Springer Berlin / Heidelberg, 2011.
7. M. Lanza and S. Ducasse. Polymetric views - a lightweight visual approach to reverse engineering. *Software Engineering, IEEE Transactions on*, 29(9), Sept. 2003.
8. H. Lie. *Cascading Style Sheets*. PhD thesis, Faculty of Mathematics and Natural, Sciences University of Oslo, 2005.
9. W. Lucas and S. M. Shieber. A simple language for novel visualizations of information. In J. Filipe, B. Shishkov, M. Helfert, and L. A. Maciaszek, editors, *Software and Data Technologies*, volume 22 of *Communications in Computer and Information Science*, pages 33–45. Springer Berlin Heidelberg, 2009.
10. M. Meyer, T. Gîrba, and M. Lungu. Mondrian: an Agile Information Visualization Framework. In *Proceedings of the 2006 ACM symposium on Software visualization - SoftVis '06*, pages 135–144, New York, New York, USA, 2006. ACM Press.
11. J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, Reading Massachusetts, 1994.
12. B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343, 1996.
13. S. Stevens. On the Theory of Scales of Measurement. *Science, New Series*, 103(2684):677–680, June 1946.
14. M.-A. Storey and K. Wong. Rigi: A Visualization Environment for Reverse Engineering. In *Software Engineering, 1997., Proceedings of the 1997 (19th) International Conference on*, pages 606–607. ACM, 1997.