Pragmatic Visualizations for Roassal: a Florilegium

Accepted to IWST 2013

Mathieu Dehouck[1] Usman Bhatti[13] Alexandre Bergel[2]
Stéphane Ducasse[1]

[1]RMoD, INRIA Lille Nord Europe, France
[2]Department of Computer Science (DCC), University of Chile, Santiago, Chile
[3]Synectique, Lille, France

ABSTRACT

Software analysis and in particular reverse engineering often involves a large amount of structured data. This data should be presented in a meaningful form so that it can be used to improve software artefacts. The software analysis community has produced numerous visual tools to help understand different software elements. However, most of the visualization techniques, when applied to software elements, produce results that are difficult to interpret and comprehend.

This paper presents five graph layouts that are both expressive for polymetric views and agnostic to the visualization engine. These layouts favor spatial space reduction while emphasizing on clarity. Our layouts have been implemented in the Roassal visualization engine and are available under the MIT License.

1. INTRODUCTION

Software analysis and reverse engineering large software systems are known to be difficult [DDN02]. Visualizing software eases analysis by using cognitive abilities to understand software and identify anomalies [**?**]. Visualizing software elements as a graph is a natural visual representation commonly employed:

- Graphs are relatively cheap and easy to visualize due to the amount of available dedicated libraries (*e.g.,* D3[1], Raphael[2]).
- Graphs are a structure effective to represent many different aspects of a software, including control flow and dependencies between structural elements.

Visualization techniques are known to be effective at analyzing package dependencies, correlating metric values, package connectivity and cycles, package evolution or the common usage of package classes (*e.g.,* [DLP05, LDDB09, vLKS+11]). A large body of existing work on software understanding is based on visualization approaches [HMM00, SvG05], in particular, on node-link visualizations [SM95, CIK03, KD03, HSSW06]. On one hand, some researchers explored matrix-based representation of graphs [HFM07, AvH04] or of software [MFM03] and its evolution [VTvW05]. On another hand, important progress has been made to support navigation over large graphs and to propose scalable and sophisticated node-link visualizations for visualizing the connectivity graph of software entities [GFC05, HSSW06, Hol09].

Roassal[3] [**?**] is a visualization engine for the Pharo language[4] [**?**]. The question here was not to invent a new way of representing information, but to find relevant existing layouts and to implement them in Roassal, alongside Roassal layout such as grid, circle, tree. The novelty of our approach is that even when the nodes do not have same size they are drawn correctly.

We have thus proposed five new graph layouts, each one focusing on a particular aspect: the *radial-tree* focuses on representing hierarchies, while with regular trees the root is repulsed to the top, radial-tree keeps the root at the center of the visualization. *Force-based* layout allows one to represent cyclic graphs such as dependency graphs. The *compact tree* family is just another implementation of trees using the same algorithm as radial-tree so that it saves space for large hierarchies. The *reversed radial tree* layout is another way of representing hierarchies where the position of an element does not depend on its depth but on its distance from the bottom of the graph. The *rectangle-packing* layout is an implementation of a rectangle packing algorithm to allow representing a lot of elements of different sizes in a reasonably restricted space.

To avoid confusion, we define terms used in this paper. A *layout* is an algorithm that determinate position of the graphical elements contained in a visualization following some particular constraints. A *node* or *vertex* is a basic element of a graph, typically in software analysis a package, a class or a method. An *edge* or a *link* represents a relation between two nodes, typically inheritance, composition or call. A *tree* is an acyclic directed graph, for example a simple object hierarchy. A *root* node is an entry point from which all the nodes are reachable by transitivity, typically the superclass of a class hierarchy.

In Section 2 we will introduce the problem and then in Section 3 describe the different layouts we have implemented, explaining for each the intention we had, the problems we encountered, the way we solved them and the limits of our solution.

[1] http://d3js.org
[2] http://raphaeljs.com

[3] http://objectprofile.com/#/pages/products/roassal/overview.html
[4] http://www.pharo-project.org

2. PROBLEM DESCRIPTION

In reverse engineering we deal with old and complex systems that are not understood easily. Moose provides powerful tools to analyse these pieces of software and we end up with a large amount of data and metrics. But it is hardly more understandable, thus we need a way of having a quick and smart overview of the relevant information.

The solution is to map the most important textual information to graphical features, and to organize them to be easily readable. The aim of the layouts is to organize this visual information. We may have to represent various kinds of data and we may want to focus on different features, it is therefore necessary to have several layouts which will organize data.

The main constraint is computing duration, hence the choice of a pragmatic answer. For example, we give to the rectangle packing layout a desired size for the resulting rectangle and do not really compute the optimal arrangement which would have minimized the surface because it is time consuming.

3. A FLORILEGIUM OF VISUALIZATIONS

In this section we present some algorithms we added to Roassal. In particular we present the general intention of the algorithm, the main challenges it poses and the solutions we chose.

For each algorithm proposed here, we show the resultant layout with the Collection class hierarchy of Pharo: there are 131 classes in this hierarchy.

3.1 Radial Tree

Intention.

When dealing with inheritance it is natural to have large trees, and the problem with regular tree representation is that the root is repulsed to the top of the visualisation. Sometimes we want to avoid that, and to keep the root amidst the visualisation. This is the aim of the radial tree.

Difficulties.

There are several difficulties when drawing a radial tree.

- **Parent position node.** Firstly we had to choose if the position of a parent node would influence its children nodes position, or if the parent node position would be influenced by its children nodes position.
- **Supporting interaction.** Another constraint was that in Roassal we do not just want to represent data, but we also want to interact with them, so it was important to have an airy representation. This was the problem encountered with the old implementation, the representation was so compact that it was not possible to interact properly with the nodes.
- **Algorithm selection.** The last problem and maybe the most important one, is what kind of algorithm should we use to compute nodes position. If we choose to compute directly radial position for each node, as a circle has a finite perimeter then we take the risk of having to displace each node several times, that gives a complexity in $O(n^2)$, and we can do better for such layout.

Solutions.

We propose a solution inspired by the modified version of Reingold-Tilford algorithm [**?**]. We compute node position beginning at the leaves and then we ascend the tree to the root, displacing subtrees when they overlap. We do this in a Cartesian coordinate system, with some minor modifications to nodes position so that the radial tree looks nice at the end: typically the space between nodes depends on the layer they belong to. And then we transform our regular tree into a radial one wrapping the layers around the root (Figure 1).

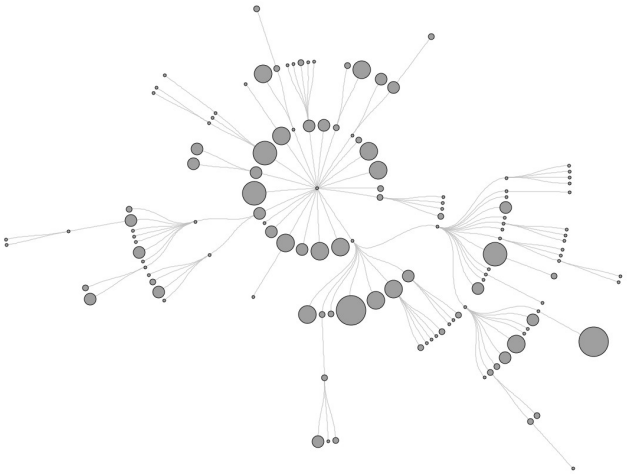

Figure 1: Radial-Tree Layout

Limits.

This layout is interesting for visualizing hierarchies, since we want to interact with the nodes, there must be enough space between them, and so when there are many nodes on a layer, then the tree has an enormous diameter and the root remains all alone in the middle of the visualization, far from its children. This is the main drawback of the radial-tree layout.

3.2 Force Based

Intention.

When dealing with methods invocations or module dependencies, trees are seldom encountered due to cyclic connections. And sometimes it does not make sense to give more importance to a node in particular, so a tree layout is not always appropriate. The force based layout considers nodes as repulsive charges and links as springs, then we have a representation which respect nodes connectivity.

Difficulties.

The main problem of force-based layout algorithms is their temporal complexity which is considered to be $O(n^3)$ for the most trivial implementations, as each iteration has a quadratic complexity (we compute force action for each pair of nodes) and we must iterate enough times (which is thought to be of the same order as the number of nodes) to reach a local minimum. And since the goal is to represent big graphs, it is necessary to have a less time consuming algorithm.

Solutions.

Our solution is inspired by D3 Javascript library implementation and the FADE algorithm [**?**]. Quadtrees reduce the number of cal-

culi at each step and thus give a *O(n log(n))* complexity. It is also possible to specify charge for a particular node, strength of a link, gravity center. Our force based layout is highly parametrisable, so that it is possible to focus on different aspects of a system (see Figure 2).

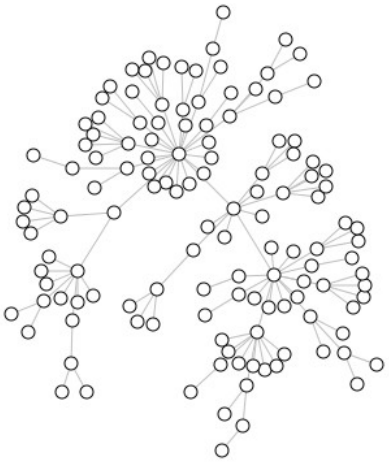

Figure 2: Force-Based Layout

Limits.

Here the limit is the running time. Even with a complexity in *O(n log(n))*, large graphs takes much longer, and then it may be difficult to use it in live.

3.3 Compact Tree

Intention.

There were already tree layouts in Roassal, but they make large graphs since they only keep track of the biggest abscissa where a node has been set. Thus our goal here was to have a less space consuming algorithm, which permits us to draw condensed tree when there is not much space.

Difficulties.

- **Vacant position.** A trap in this kind of algorithm is that we need to know for each layer the abscissa where we can set nodes, this can be done multiple ways, but the trivial way consists of checking all the previously set nodes, and then you have a complexity in $O(n^2)$, which is a loss of time in this case since trees can be drawn with a smaller complexity.
- **Node shifting.** Then as this algorithm is recursive, when setting a child node we do not know where the parent node will be set, and then when setting the parent node, sometimes it occurs that we have to move children nodes, and here the trivial solution has also a complexity in $O(n^2)$.

Solutions.

Here we also use a Reingold-Tilford like algorithm with some improvements such as pointers for left-most and right-most children of a node. This is done so that we do not look at all the previously set nodes when we need to know where we can put a node. When placing a new node, we just skim the contour of the graph (the right-most and the left-most nodes of each layer) and it is less time consuming. In the same way, when we have to move children nodes to correspond to their parent node position, instead of moving them each time, the parent keeps a "modification" value, that spreads to the children when they are drawn, once again it saves time (see Figure 3).

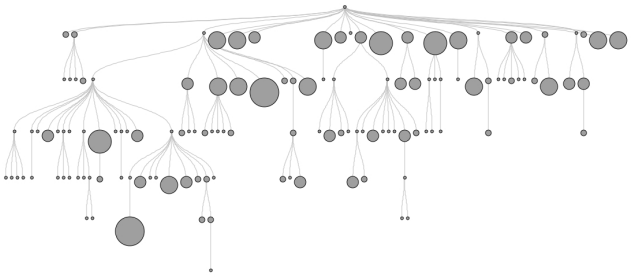

Figure 3: Vertical Compact Tree Layout

Limits.

Our solution is pragmatic thereby computed tree is not the narrowest since even when children order is not important the tree is drawn as if the children were ordered. But then it would be necessary to go through the hierarchy several times to sort the nodes in order to have the narrowest tree, and it would have a high complexity.

3.4 Reversed Radial Tree

Intention.

The reversed radial tree layout is another tree layout for hierarchy representation, but when most of the trees focus on the distance between the nodes and the root, the reversed radial tree layout focuses on the position of nodes compared to the whole tree, thus leaves are on the border regardless of their distance from the root.

Difficulties.

There are no real difficulties for the reversed radial tree layout. It is just important to avoid useless route in the graph.

Solutions.

We skim the tree from the leaves to the root, recording for each node the maximum distance to the leaves in the subtree induced by the node. And then as we have the list of leaves, we compute nodes position from the leaves to the root (see Figure 4).

Limits.

Here we have the same kind of problems as with the radial tree. As leaves are all on the border of the visualization, with many leaves, the diameter of the visualization may be large and the visualization may be almost empty, we will have lots of nodes on the border (the leaves) and very few nodes in the circle, with long edges between them.

3.5 Rectangle Packing

Intention.

Sometimes we want to represent a lot of elements of different

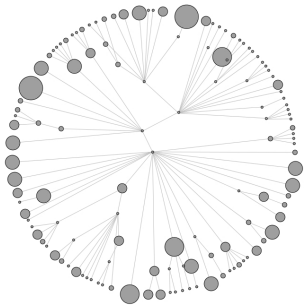

Figure 4: Reversed radial tree Layout

sizes and a grid layout is not always a good choice as it does not use the visual space efficiently. The goal of the rectangle packing layout is to show many elements of various sizes in the available restricted visual space.

Difficulties.

The problem of rectangle packing is NP-hard, that means that we cannot find a solution in polynomial time but we cannot afford excessive computation time.

Solutions.

Here our solution is very pragmatic: instead of looking for the arrangement that will minimize the surface occupied by the elements, we provide the layout a ratio (2/3 by default), which corresponds to the width divided by the height of the rectangle we want to fill with our elements (Figure 5). Then the layout starts placing the elements and resizes the containing rectangle until it has succeed in placing every element.

Limits.

There are two limits:

- **Running time.** Even without looking for the best arrangement, it is time consuming. Thus it is difficult to apply it on a large number of elements.

- **Biggest elements.** We are dependent of the biggest elements (typically the longest and the widest). Sometimes we have little elements and a few big ones, then if we ask for a shape oriented in the other direction as the big ones, we will not have it. In our example, we provided the ratio 1/1 since we wanted to arrange elements in a square, but as there is a very long and thin one, we do not have a square at all, but a thin rectangle.

  Here we may raise the question of node resizing which is a touchy one, since we may break the sense originally provided by node size. And then, how do we resize nodes? Do we resize all the nodes, or only the biggest ones?

4. CONCLUSION

For large amounts of data, Roassal and similar visualization engines need to find a pertinent representation so that data are presented in a meaningful form and understood by the end users. In this paper, we have presented five graph layouts that are both expressive for polymetric views and agnostic to the visualization engine. The layouts favor spatial space reduction while emphasizing

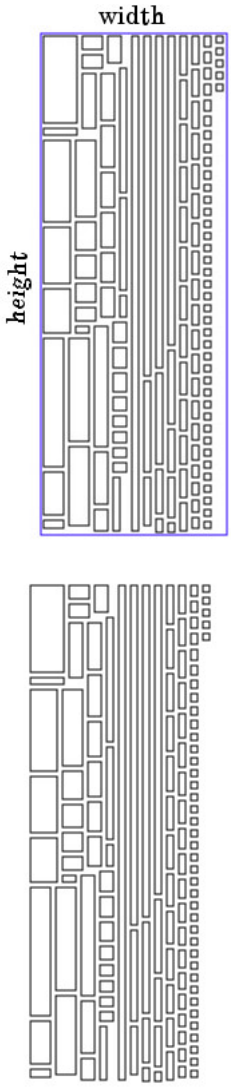

Figure 5: Rectangle Packing Layout

on clarity. Our solution is tractable and diverse, as the variety of layouts allows to analyze data in various forms. It should be noted that even with good layouts, if the amount of information is too big then it is hardly understandable, and the user has to himself select the most relevant information to be shown. We can make our layouts even more customisable, by for example proposing nodes staggering which can sometimes be a good way of saving even more space.

Acknowledgements.

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013', the Cutter ANR project, ANR-10-BLAN-0219 and the MEALS Marie Curie Actions program FP7-PEOPLE-2011- IRSES MEALS.

This work has been partially funded by Program U-INICIA 11/06 VID 2011, grant U -INICIA 11/06, University of Chile, and FONDECYT project 1120094. We thank the support from the Plomo INRIA associated Team.

5. REFERENCES

[AvH04] James Abello and Frank van Ham. Matrix zoom: A visual interface to semi-external graphs. In *10th IEEE Symposium on Information Visualization (InfoVis 2004), 10-12 October 2004, Austin, TX, USA*, pages 183–190. IEEE Computer Society, 2004.

[CIK03] Neville Churcher, Warwick Irwin, and Ron Kriz. Visualising class cohesion with virtual worlds. In *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*, pages 89–97, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[DDN02] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[DLP05] Stéphane Ducasse, Michele Lanza, and Laura Ponisio. Butterflies: A visual approach to characterize packages. In *Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 70–77. IEEE Computer Society, 2005.

[GFC05] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. On the readability of graphs using node-link and matrix-based representations: a controlled experiment and statistical analysis. *Information Visualization*, 4(2):114–135, 2005.

[HFM07] Nathalie Henry, Jean-Daniel Fekete, and Michael J. McGuffin. Nodetrix: a hybrid visualization of social networks. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1302–1309, 2007.

[HMM00] Ivan Herman, Guy Melançon, and M. Scott Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[Hol09] Danny Holten. *Visualization of Graphs and Trees for Software Analysis*. PhD thesis, Computer science department, 2009. ISBN 978-90-386-1882-1.

[HSSW06] Holt, Schürr, Sim, and Winter. Gxl: A graph-based standard exchange format for reengineering. *Science of Computer Programming*, 60(2):149–170, April 2006.

[KD03] Said Karouach and Bernard Dousset. Visualisation de relations par des graphes interactifs de grande taille. *Journal of ISDM (Information Sciences for Decision Making)*, 6(57):12, March 2003.

[LDDB09] Jannik Laval, Simon Denier, Stéphane Ducasse, and Alexandre Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.

[MFM03] Andrian Marcus, Louis Feng, and Jonathan I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.

[SM95] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP Views. In *Proceedings of ICSM '95 (International Conference on Software Maintenance)*, pages 275–284. IEEE Computer Society Press, 1995.

[SvG05] Margaret-Anne D. Storey, Davor Čubranić, and Daniel M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *SoftVis'05: Proceedings of the 2005 ACM symposium on software visualization*, pages 193–202. ACM Press, 2005.

[vLKS+11] Tatiana von Landesberger, Arjan Kuijper, Tobias Schreck, Jörn Kohlhammer, Jarke J. van Wijk, Jean-Daniel Fekete, and Dieter W. Fellner. Visual analysis of large graphs: State-of-the-art and future research challenges. *Comput. Graph. Forum*, 30(6):1719–1749, 2011.

[VTvW05] Lucian Voinea, Alex Telea, and Jarke J. van Wijk. Cvsscan: visualization of code evolution. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 47–56, New York, NY, USA, 2005. ACM.