Proceedings of the International Workshop on Program Comprehension (IWPC-2002), Paris, 27-29 June 2002.

An Open Visualization Toolkit for Reverse Architecting

Alexandru Telea
Eindhoven University of Technology
Department of Mathematics and Computer Science,
The Netherlands
alext@win.tue.nl

Alessandro Maccari and Claudio Riva
Nokia Research Center
Software Architecture Group
P.O. Box 407, FIN-00045 NOKIA GROUP
{alessandro.maccari | claudio.riva}@nokia.com

Abstract

Maintenance and evolution of complex software systems (such as large telecom embedded devices) involve activities such as reverse engineering (RE) and software visualization. Although several RE tools exist, we found their architecture hard to adapt to the domain specific requirements posed by our current practice in Nokia. In this paper, we present an open architecture which allows easy prototyping of RE data exploration and visualization scenarios for a large range of domain models. We pay special attention to the visual and interactive requirements of the reverse engineering process. This article describes the basic architecture of our toolkit, compares it to the existing RE environments and present several visualizations taken from real cases.

Keywords
Reverse Engineering, Architecture Reconstruction, Software Visualization and RE tools.

1 INTRODUCTION

Reverse engineering (RE) is an essential part of the maintenance and evolution process of complex software systems. It may involve several tasks, such as program analysis, model capture and analysis, knowledge discovery and re-documentation. Reverse architecting is an extension of reverse engineering that also includes tasks such as abstraction of reverse engineering models and architecture recovery.

Reverse architecting (explained to some degree of detail in section 2) aims at extracting high level architectural models of the system. The models are typically extracted from the code and rendered at a higher level of abstraction in order to highlight the architectural structure [10] of the software system. The model thereby obtained must be rendered in a graphical form in order to give a user-visible representation of the architectural artifacts. This step is known as visualization, and can be performed in various manners, such as graphical navigable views, at different levels of detail.

Several tools support selected RE tasks by providing automatic and user-driven data extraction and visual data presentation [21], [7], [2]. In practice, attempts to reverse engineer large systems usually reach functional and/or structural limitations of such RE tools. A typical example of such limitations is the inability of most tools to display very large numbers of nodes and arcs.

Other tools focus on visualization, but usually offer little support for program analysis. They also do not integrate well with program analysis tools. In comparison with scientific visualization (SciViz), where established generic system architectures and implementations are now common, most RE tools are still too narrowly specialized, and are inadequate for reverse architecting (which implies easy integration between program analysis and visualization tools).

To address the above problems, we propose an open software toolkit for RE tools. The toolkit we present accommodates a large range of RE data types, operations, and application scenarios. Subsequently, we present examples of constructing RE exploration and visualization applications taken from industrial case studies. Section 2 summarizes the reverse architecting process and elaborates on how the existing software tools support it in practice. Section 3 introduces the architecture of the toolkit, while sections 4 and 5 present the core architecture of the tool (respectively, data model and back-end). Section 6 shows an end-user view of our tool. Section 7 illustrates the application of the tool to analyze the concrete data taken from industrial case studies. We conclude the paper with directions for future research in section 8.

2 REVERSE ENGINEERING OVERVIEW

In [13] and [12], we have proposed an architecture recovery method based on a four-step reverse engineering process: *definition of architectural concepts* (to select the architectural elements of the recovery), *extraction* (to capture a low level model of the system), *abstraction* (to inject domain knowledge in the model) and *visualization* (to present the architectural model with multiple views).

Visualization plays an important role for presenting the architectural model and to allow the user to navigate the model, to generate new views, and to interact during the abstraction phase. We can refine the RE tasks into the following five generic operations that a RE tool should support:

1. *Extract* the low-level model from the artefacts of the system (like the source code). This is achieved by external code analysers that typically offer APIs to access the internal symbol table (such as Source Navigator [11]).
2. *Refine* the low level model in order to emphasize the user sensitive information.
3. *Aggregate* the extracted artefacts into a hierarchical model according to the domain knowledge.
4. *Measure* the quality of the model with computed norms or calculate particular software metrics. If needed re-execute the aggregation differently
5. *Select* a sub-hierarchy of the model to focus the analysis on.
6. *Visualise* the data, e.g. by laying out the graph with a particular algorithm.

This process is applied iteratively and the tasks from 2 to 6 can take place in any order. Typically, the user tries to visualise the initial low-level model to get an overview of the graph. Then, he/she can apply some aggregations, measure its quality, re-visualise the graph. Once the high-level model has been generated the user needs to navigate it by creating multiple views of the same data set.

In most cases, the operations of *aggregation, selection, filtering, lay out* and *visualization* are the most important from the task of examining the extracted model. In our previous work [13] and [12], we have based our visualization on the Rigi environment [21] and extended it to support the visualization of Message Sequence Charts (MSCs) for dynamic analysis. Rigi satisfies most of our needs but it is not flexible enough in terms of visualization. We have found Rigi to be useful for creating a hierarchical model by aggregating nodes, navigating the hierarchical graphs, filtering according to user selection or entity types and creating multiple synchronised views. We have also made use of the domain paradigm that allows us to define the types of nodes/edges, the simple RSF input and the Tcl scripting that allows to customise the environment. However, Rigi lacks modern and robust visualization techniques: the selection of sub graphs is not well supported, we can specify only one containment relationships (while the users often need to aggregate and visualise the graph according to different part-of relationships), the visualization of nodes/edges cannot be customised (e.g. in the case of UML-like visualizations [14]), the layout capabilities are not scalable. In general, the visualization aspects are rather poor in Rigi and this is the biggest limitation we have experienced.

3 TOOLKIT ARCHITECTURE

We propose an architecture based on two concepts: datasets and operations. As in a classical SciViz dataflow, operations can read and write parts of datasets. The desired functionality is achieved by sequencing several operations in a specific order starting from a particular dataset. In our toolkit, we use a centralised data model where there is a single dataset *DS* across the whole system. This dataset is read and written by all the operations that we need to support. The operations can be initiated by the user or the system, and can be applied in any desired order.

The architecture of the toolkit consists of two major components: the *toolkit core* and the *user interface*.

The toolkit core contains the data set *DS* and the implementation of the operations for the creating the graphical visualizations. It is implemented as a C++ class library for performance and it is based on the Open Inventor toolkit [20] for the graphical visualizations. The user interface and the scripting layer are implemented in Tcl/Tk for flexibility. All the major functionality offered by the toolkit core is exported to the user with a Tcl API as a set of operations. This approach allows us to optimize the visualization engine for performance and to quickly prototype different type of visualization and interactions with the Tcl/Tk GUI.

Below we describe the data model, the operations and the user interface of the toolkit

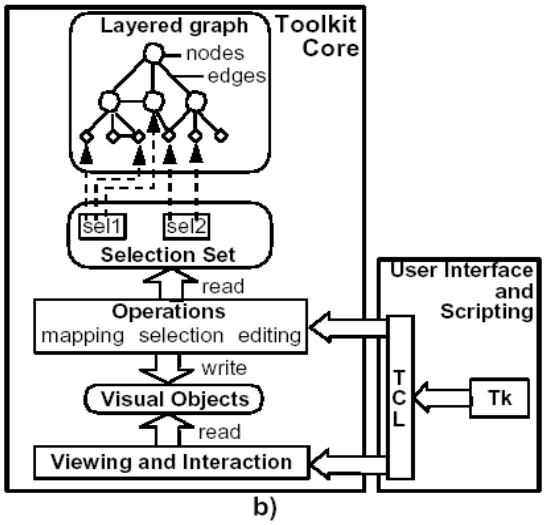

Figure 1. The toolkit architecture.

4 DATA MODEL

The basic data model is a *typed attributed graph* (similarly to those described in [15], [1], [17]) and contains three basic elements: structure, attributes and selections.

The structure is the set of nodes and edges of the graph. The nodes represent software artefacts extracted from the program analysis tools. The edges model both relational and containment information. The containment information is typically added during architectural recovery through a phase of plane assignment. We do not assume any distinction between this information and other relational information.

Nodes and edges may have key-value pair attributes. We implement the keys as string literals and the values as primitive types (integer, floating-point, pointer, or string). Each node and edge has a set of attributes with distinct keys, managed in a hash-table-like fashion. Attributes automatically change type if written with a value of another type. Several *attribute planes* can coexist in the graph. An attribute plane is defined implicitly as all attributes of a given set of nodes/edges for a given key. Our attribute model differs from the one used by most SciVis [16] and RE applications [2][21][7] which choose a fixed set of attributes of fixed types for all nodes/edges. Our choice is more flexible, since a) certain attributes may not be defined for all nodes, and b) attribute planes are frequently added and removed in a typical RE session.

Selections, defined as sets of nodes and edges, allow us to execute the toolkit operations on a *specific subset* of the whole graph. To make the toolkit flexible, we decouple the definition of the operations from the selections on which they are executed, similarly to the dataset-algorithm decoupling in SciViz. Selections are named and stored in variables accessible to the user as key-value pairs selection-set, similarly to attributes. Overall, our graph and selection data model is quite similar to the one used by the GVF toolkit [8]. Our graphs are *structurally* equivalent to the node-and-cell dataset model in SciViz frameworks, whereas our selections do not have a direct structural equivalent. Selections are *functionally* equivalent to SciViz datasets, since they are the operations' inputs and outputs. As pointed out by [8], this is one of the main differences between SciViz and graph-based toolkits which leads to different architectures for the two.

5 THE OPERATIONS

The operations implement a particular functionality of the toolkit and modify the unique dataset *DS*. The operations have three types of inputs and outputs: *selections* that specify on which nodes and edges to operate, *attribute keys* that specify on which attribute plane(s) of the selection to work, and operation-specific *parameters* such as thresholds or factors. We can categories the operations according to their read/write data access: *selection operations*, *graph editing operations* and *mapping operations*.

The toolkit architecture (based on the dataset-operation paradigm) allows the system to automatically update all components that depend on the modified data after the execution of any operation. For example, the selections are automatically updated after a structure editing operation which deletes selected nodes or edges. Similarly, the data viewers are updated when the selections that they monitor change. Although this is largely similar to the SciViz dataflow mechanism [16], we do not *explicitly* construct an operation pipeline. The reason is that, in contrast to SciViz applications, reverse engineering operations are seldom executed in the same order in different RE sessions. The typical RE task can be summarised as follows:

- select a subset of nodes/edges of interest.
- apply some editing operations on the selection.
- map the selection to a particular visual representation.

Below, we describe the three categories of operations that we support.

Selection operations

Selection operations create a selection of objects by grouping nodes and edges. This is an important task in the toolkit since all the operations accept as input a particular selection. We implemented several basic selection operations, as follows. *Level selections* (called 'horizontal slices' in the RE literature [21]) gather all the nodes and association edges on a certain aggregation level in the layered graph, and are useful for visualizing the software at a given level of detail. *Tree selections* (called 'vertical slices' in [21]) gather all nodes and containment edges reachable from a user defined selection of nodes, and are useful e.g. for visualizing subsystem structures. *Conditional selections* (called 'filters' in most RE tools) gather all elements in an input selection that obey some attribute based condition, e.g. in queries like 'show all nodes where the *cost* attribute is higher than some threshold'.

Editing operations

Graph editing operations modify the graph dataset *DS*. There are two categories of editing operations: *structure editing* and *attribute editing*.

The structure editing operations construct and modify the graph itself. We have basic operations for adding/removing nodes and edges, and operations for reading several graph format such as RSF [21], GraphEd [5], DOT [9] and GXL [6]. Aggregation operations also

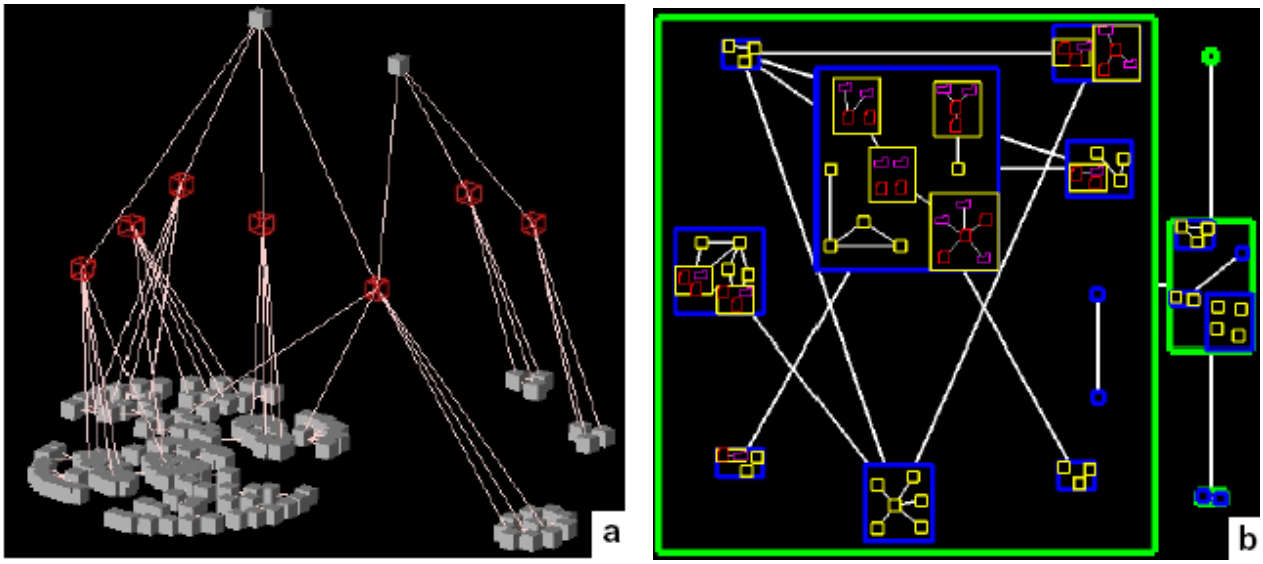

Figure 2. Stacked Layout (a) and nested layout (b).

modify the graph and typically take the nodes of a selection and connect them to a parent node by containment arcs. The input selection can be either a user selection or constructed as a sequence of Tcl commands.

The attribute editing operations create, modify and delete the attributes from the nodes' and edges' attribute-sets. The inputs of the attribute operations are the input selection and one or several attribute-plane names where the operation reads and/or writes. There are two important attribute operations that we describe below: *calculation of RE metrics* and *graph layouts*.

The calculation of RE metrics is treated as an attribute editing operation that produces new attribute-planes. Examples of RE metrics are the number of provisions, requirements, internalisations of the aggregated nodes or the cyclomatic number of a subgraph. Decoupling the metric calculation from the selection operation allow us to apply any metric on any subgraph (which is not the case of other RE tools [21], [7]).

We treat graph layouts simply as attribute editing operations and thus decouple them completely from mapping and visualization. This has several benefits, in contrast to other approaches [21], [7], [8]. First, we can lay out different subgraphs separately, e.g. using spring embedders [9], [3] for call graphs and tree layouts [9], [18] for containment hierarchies. Second, we can precompute several layouts e.g. to quickly switch between them. Finally, we can cascade different layouts on the same position attributes, e.g. to apply a fish-eye distortion or refine an existing layout [4]. We have implemented several custom layouts by cascading simple ones, as follows. *Stacked layouts* (in Figure 2a) lay out a selection spanning several layers of a graph by applying a given 2D layout (e.g. spring embedder) per layer and then stacking the layers in 3D. Stacked layouts visualize effectively both containment (vertical) and association (horizontal) relations of a software system. *Nested layouts* (in Figure 2b) lay out a similar selection as above, by recursively laying out the contents of every node separately and then laying out the bounding boxes of the containing nodes. Nested layouts produce images similar to UML class diagrams and are very helpful in RE applications. Users can easily combine any 2D layouts as the building bricks for the stacked and nested layouts For example, in Figure 2a we use a tree layout, whereas in Figure 2b we use a spring embedder as basic layout.

Concretely, we use the AT&T's DOT package [9] for tree

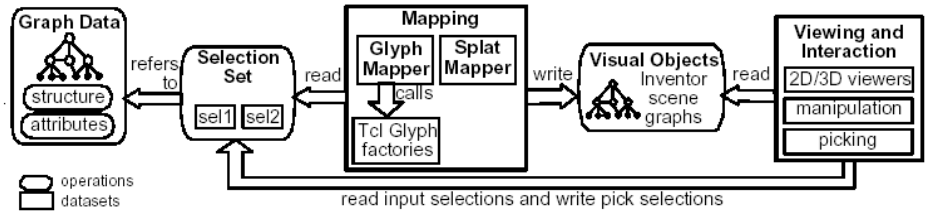

Figure 3. The components of the mapping and visualization operations.

layouts and AT&T's NEATO, GraphEd, and GEM [5][3] for spring embedding. We have conducted several tests on graphs up to 2000 nodes on which DOT was faster, more robust, and produced visually better results than the layouts of RE tools such as [21][7]. In about 70% of our tests, GEM produced better layouts quicker than NEATO, especially for graphs over 1000 nodes, but was more sensitive to the parameter choice. Adding new layouts to the toolkit is reasonably simple. Adding DOT, NEATO, or GEM (whose implementations exceed 50000 C lines) were wrapped by less than 100 C++ lines each, whereas our custom layouts have each under 200 C++ lines.

Mapping and visualization operations

Mapping and visualization operations map the graph to visual objects. They allow the user to visualise and interact with the graph data. In the toolkit, there are four components responsible to implement these operations: *mappers*, *viewers*, *glyph factories* and *glyphs* as shown in Figure 3. The visualization operations are implemented using the Open Inventor C++ toolkit [20].

The central visualization component is the mapper, which maps a selection to an Inventor scene graph. We have implemented two mappers: the *glyph* and the *splat* mapper. The glyph mapper creates a glyph for each node and edge in the input selection and positions these glyphs at the 2D or 3D coordinates provided by a particular attribute plane (previously created with a layout operation). The splat mapper produces a splat field [19] from the input selection and an assigned density function, as described in Section 7 .

The *glyph* is a 2D or 3D graphical object that visualises a node or an edge. The user can customise the glyphs with a Tcl script called the *glyph factory*. The script sets the glyph's graphical properties (color, shape, size, annotation, and so on) from the attributes of the input node or edge. When the mapper is executed, the tcl script

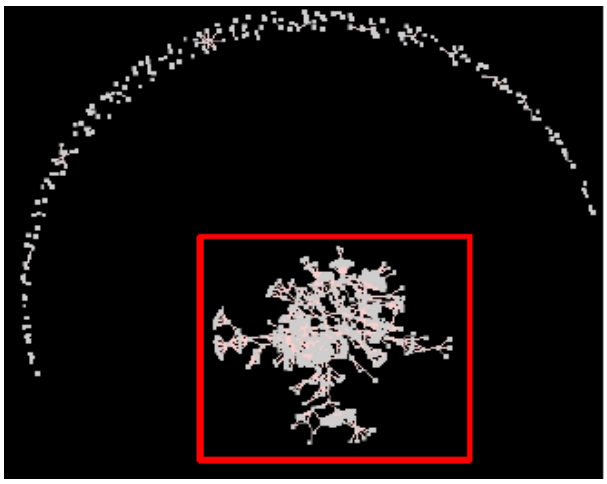

Figure 4. Visualization of the RE model.

is invoked to build the desired glyph as an Inventor node. Since the users may freely edit these scripts at run-time, it is very easy to customize the visualization at hand. Figure 5 shows a glyph-based visualization of the software of a program analysis system developed at Nokia. The left image shows the graph laid out with a DOT layout while the right image with the spring embedder. In this case, the two views are based on the same selection but use a different position attributes.

6 THE GUI

The core of the toolkit is exported to the user with a simple Tcl API. The user interface is implemented with the Tcl/Tk. The scripts and GUIs add custom functionality, such as examining and editing node attributes, selection objects, domain models, and viewers, loading and saving data, and so on. These integrated applications are functionally very similar to other RE tools such as Rigi [21] and VANISH [7]. The main difference is that our toolkit's core architecture is based on a few loosely coupled, orthogonal components: graph

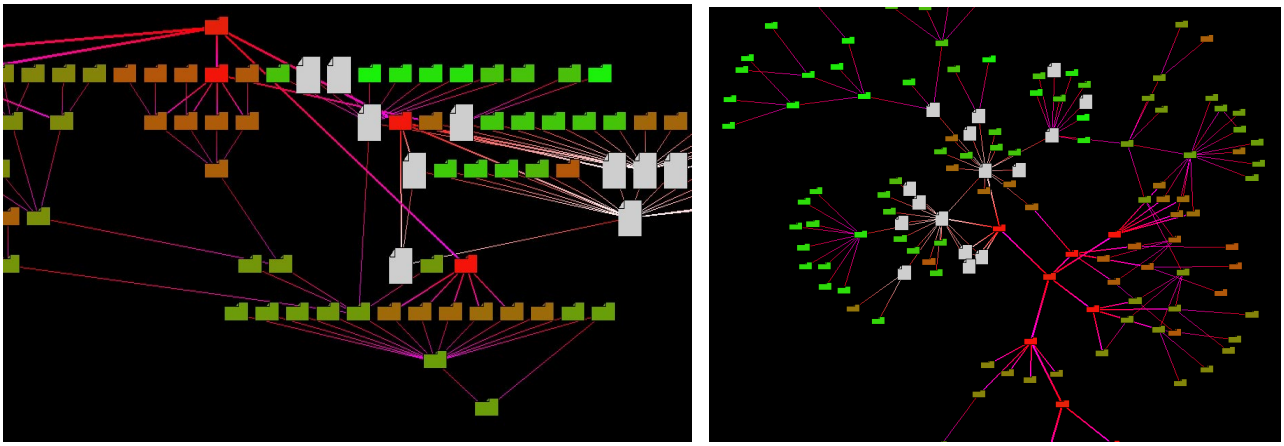

Figure 5. Visualization of a RE clustered graph with glyphs in DOT (a) and spring (b) layout.

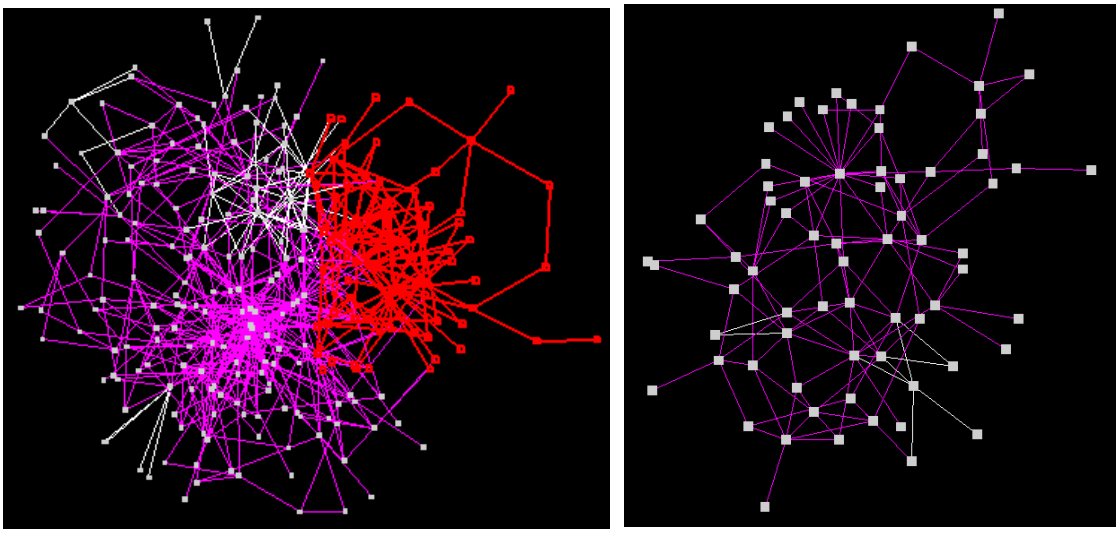

Figure 6. A selected sub graph (a) and its visualization in a separate view (b).

and selection data objects, operations, mappers, glyphs, and viewers. The data operation loose coupling and the selections make it natural for developers to write small, independent operations - so far all our operations range from 20 to 150 C++ or Tcl lines.

7 EXPERIENCES WITH THE TOOLKIT

We have exploited the prototype with several reverse engineering data extracted from Nokia software systems implemented in various languages, C, C++, Java and Tcl/Tk. The nodes of the graphs represent software entities like classes, functions, packages or architectural concepts like subsystems and components. The edges represent relationships between them, like function calls, containment, or architectural concepts, like messages.

We present some examples based on the glyph mapper and the splat mapper.

The Glyph Mapper

The Figure 4 show the visualization of the whole model extracted from the code. The graph (about 1200 nodes) has been laid out with a spring embedder. The Figure 5 shows two examples where we have selected a subset of the whole graph and applied a particular customization of the nodes. The nodes of the files and directories are represented as colored “document” and “folder” glyphs. We have applied the DOT layout on the Figure 4a and the spring embedder layout on the Figure 4b.

In Figure 6, we show the graph extracted from a Tcl/Tk system. The nodes represent procedures and the edges represent function calls. In the graph in Figure 6a, we have selected a subgraph. The subgraph (in the upper-

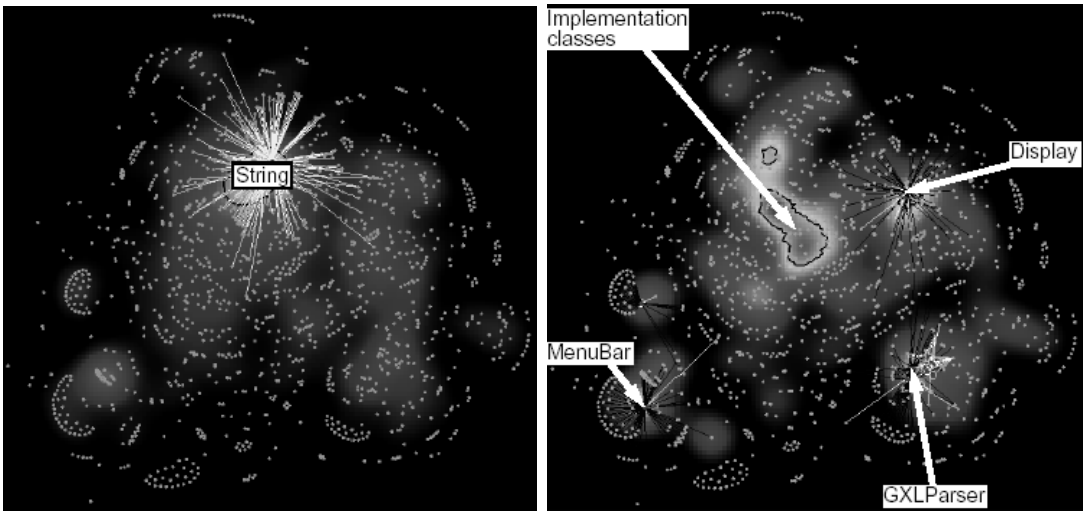

Figure 7. The splat field showing provisions (a) and requirements (b) of the graph.

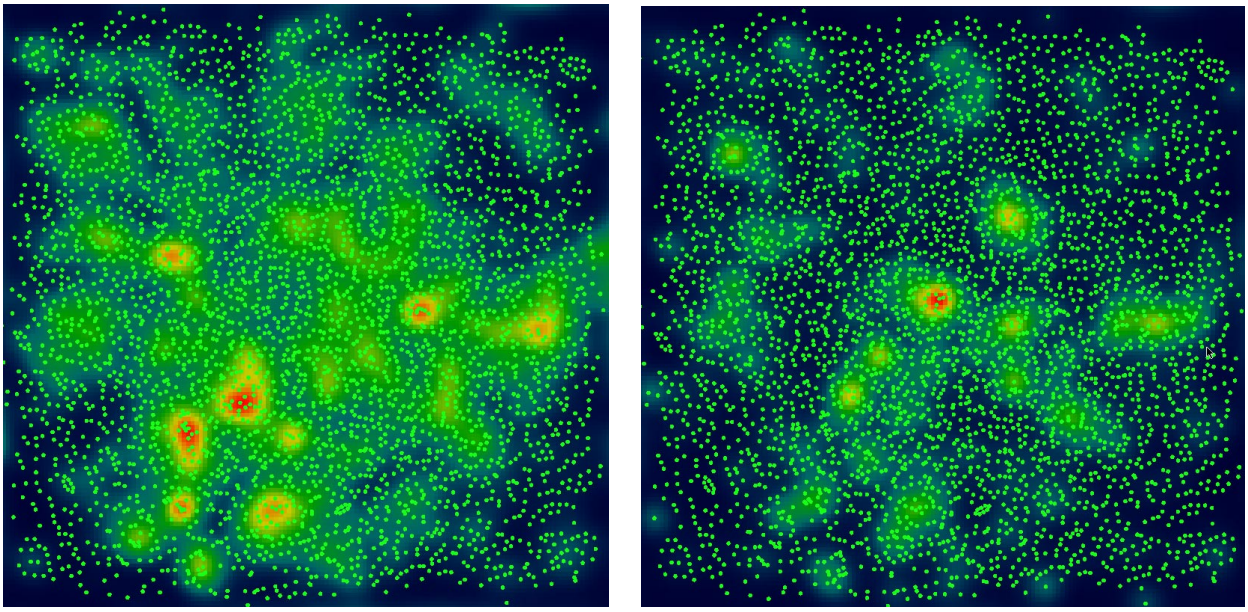

Figure 8. Splat visualization of the provisions (a) and requirements (b) of the functions in the Linux file system.

right corner of the picture) is highlighted. This selection is visualized with the spring embedder in the Figure 6b. The view is calculated according the selection. Any changes to the selection are updated also in the second view.

The Splat Mapper

We used the splat matter to create a splat field of a graph. We first layout the graph using a spring embedder and then we select a particular scalar density function. In Figure 7 we have applied the splat mapper on a graph of about 2000 nodes extracted from a Java system. The nodes represent classes and the edges represent the calls between the methods of the classes. In Figure 7a, the scalar density is the number of provisions of the classes (the number of external classes that are calling the methods of a class), in Figure 7b the density function is the number of requirements for the classes (the number of external classes that are called by a particular class). The white zones represent a high concentration of nodes with a high value of the density function. In Figure 7a, we can identify the String class whose methods are invoked by a large numbers of other classes. In Figure 7b, we can quickly identify classes that are using a high number of external classes (like the ones marked by the arrows). In Figure 8, we have created the splat visualization for the functions that implement the file system of a Linux release. The nodes represent the functions and the edges (not shown) represent function calls. The density function is the number of provisions (in Figure 8a) and number of requirements (in Figure 8b) of the functions. We can easily identify the areas where there is an high concentrations of highly coupled (the spring layout aggregates highly coupled nodes) functions with a large number of provisions (in the case of Figure 8a) and of requirements (in the case of Figure 8b).

8 CONCLUSIONS AND FUTURE WORK

The toolkit we present in this paper is a first step towards full support of the reverse architecting process. We have implemented the basic functionality and graphical user interface, and have experimented it on a number of industrial case studies. The outcome has been promising, and encourages further development of the tool, for instance in the support for a better usability of the visualization concepts, support for reverse engineering specific graph transformations, selection of pre-defined architectural views and in general to support a set of visualizations that are visually closer to the mental images that the architects have of a software architecture.

Further research on the subject could look at expanding the support for reverse architecting product family software, as well as rendering the visualization in some standard notation and exchange format, such as UML or XML. Also, the case studies we have worked on were of relatively small size (in industrial terms). Therefore, the tool should be tested on large and very large systems before its suitability is demonstrated. We also plan to implement a particular layout algorithm to support the visualization of message sequence charts that are a key requirement for the dynamic analysis.

9 REFERENCES

1. Card D. and Mackinlay J and Shneiderman B., Readings in Information Visualization, M. Kaufmann, 1999.

2. Eick S. and Wills G., Navigating large Networks with Hierarchies, in Readings in Inf. Vis.[1]
3. Frick A., Ludwig A. and Mehldau H., A fast adaptive layout algorithm for undirected graphs, Proc. Of Graph Drawing '94, Springer, 1995.
4. Herman, Melancon G. and Marshall M.S., Graph Visualization and Navigation in Information Visualization: a Survey, IEEE TVCG, 2000.
5. Himsolt M., GraphEd user manual, Technical report, Fakultat fur Informatik, Universitat Passau, 1992.
6. Holt R. Holt and Winter A., GXL: Representing Graph Schemas presentated at WCRE 2000 - 7th Working Conference on Reverse Engineering November, 23 - 25, 2000, Brisbane, Queensland, Australia
7. Kazman R. KAZMAN and Carriere J., Rapid Prototyping of Information Visualizations using VANISH, Proc. IEEE InfoVis '95, IEEE CS Press, 1995.
8. Marshall M. S., Herman I. HERMAN and Melancon G. MELANCON, An Object-Oriented Design for Graph Visualization, Software: Practice & Experience, 31, pp. 439–756, 2001.
9. North S. C. and Koutsofios E. KOUTSOFIOS, DOT and NEATO User's Guide, AT&T Bell Labs Reports, http://www.research.att.com, 1996.
10. Ran A., "ARES Conceptual Framework for Software Architecture" in M. Jazayeri, A. Ran, F. van der Linden (eds.), Software Architecture for Product Families Principles and Practice, Addison Wesley, 2000.
11. RedHat Source Navigator, http://sources.redhat.com/sourcenav/
12. Riva C. and J. V. Rodriguez, Combining Static and Dynamic Views for Architecture Reconstruction, Proceedings of the Sixth European Conference on Software Maintenance and Reengineering (CSMR 2002), IEEE Computer Society Press, 11-13 March 2002 in Budapest, Hungary, to appear.
13. Riva C., Reverse Architecting: an Industrial Experience Report, Proceedings of the 7th Working Conference on Reverse Engineering (WCRE2000), Brisbane, Australia, 23-25 November, 2000.
14. Riva C., Xu J. and Maccari A., Architecting and Reverse Architecting in UML, presented at the International Workshop on Describing Software Architecture with UML, International Conference on Software Engineering 2001 (ICSE), Toronto, May 2001.
15. Rohrich J., Graph Attribution with Multiple Attribute Grammars, ACM SIGPLAN 22 (11), pp.55-70, 1987.
16. Schroeder W., Martin K. and Lorensen, The Visualization Toolkit, 2nd edition, Prentice Hall, 1998
17. Stasko J., Domingue J. DOMINGUE and Brown M. H. Prince, Software Visualization – Programming Multimedia Experience, MIT Press, 1998.
18. Sugiyama K. SUGIYAMA, Tagawa S. TAGAWA and Toda M., Methods for Visual Understanding of Hierarchical Systems Structure, IEEE Trans. Systems, Man, and Cybernetics, Vol. 11, No. 2, pp.109-125, 1989.
19. van Liere R., Studies in Interactive Visualization, PhD thesis, CWI, Amsterdam, 2001.
20. Wernecke J., The Inventor Mentor: Programming Object-Oriented 3D Graphics, Addison-Wesley, 1993.
21. Wong K., Rigi User's Manual version 5.4.4, Dept. of Computer Science, Univ. of Victoria, Canada.