Joint EUROGRAPHICS - IEEE TCVG Symposium on Visualization (2002)
D. Ebert, P. Brunet, I. Navazo (Editors)

An Open Toolkit for Prototyping Reverse Engineering Visualizations

Alexandru Telea[1], Alessandro Maccari[2], Claudio Riva[2]

[1] Department of Mathematics and Computer Science
Eindhoven University of Technology, The Netherlands
alext@win.tue.nl
[2] Software Technology Laboratory
Nokia Research Center, Helsinki, Finland
alessandro.maccari, claudio.riva@nokia.com

Abstract

Maintenance and evolution of complex software systems (such as mobile telephones) involves activities such as reverse engineering (RE) and software visualization. Although several RE tools exist, we found their architecture hard to adapt to the domain and problem 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. We compare our toolkit with other existing reverse engineering visual tools and outline the differences.

1. Introduction

Reverse engineering (RE) [15, 22, 20] is an essential part of maintenance and evolution of complex software systems, that involves several tasks such as program analysis [15], plan recognition [11], redocumentation [20], and architecture recovery [20, 22]. Overall, these tasks can be reduced to two generic operations: *construction* of a layered program representation by automatic and user-driven (interactive) operations, and the *visualization* of this representation in various manners, such as graphical navigable views at different detail levels [3, 17]. Several tools support (some of) the RE tasks by providing automatic and user-driven data extraction and visual data presentation [21, 9, 3]. In practice, attempts to reverse engineer large systems usually reach functional and/or structural limitations of such RE tools. Some tools focus on domain modelling and program analysis [15, 11] but provide little for the visual *examination* and/or *editing* of the extracted information. Other tools focus on visualization [3, 9, 21], but do not support program analysis or are hard to integrate with tools that perform this task. In comparison with scientific visualization (SciViz), where established generic system architectures and implementations thereof are now common [16], most RE tools are still narrowly specialized and hard to adapt e.g. to easily integrate program analysis with visualization facilities.

To address the above problems, we propose an open software toolkit for RE tools. Our first aim is easy prototyping of RE data exploration scenarios by combining and customizing existing software components and writing new ones. The presented toolkit accommodates a large range of RE data types, operations, and application scenarios. Secondly, this paper brings concrete data on current practice of constructing RE exploration and visualization applications in the software industry. Section 2 overviews how current software tools support reverse engineering in practice. Sections 3 and 4 present the architecture's core, respectively visualization back-end. Section 5 shows an end-user view of our tool. Section 6 illustrates the use of our RE tool for the analysis of concrete software data from the industry. Section 7 concludes the paper with future directions.

2. Reverse Engineering Overview

The RE tasks outlined in Sec. 1 can be refined into five generic operations a RE tool should support (see also Fig. 1 a, cf. [20, 22, 21]):

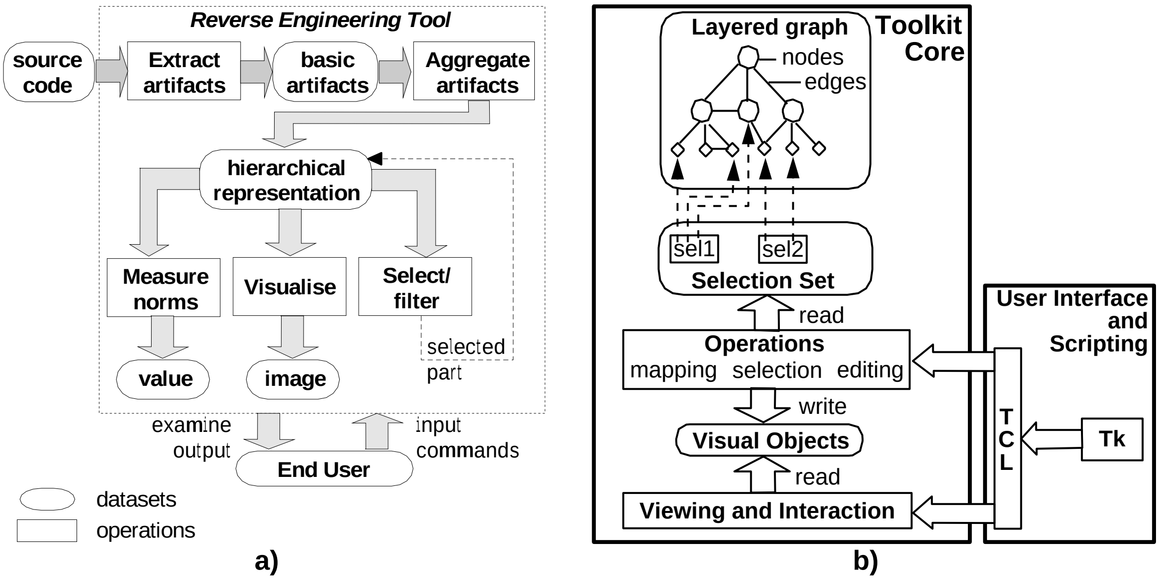


Figure 1: *Reverse engineering pipeline (a). Toolkit architecture overview (b)*

1. *extract* the low-level artifacts from the source code
2. *aggregate* the extracted artifacts into a hierarchical model
3. *measure* model's quality using computed norms. If needed, reexecute the aggregation differently.
4. *select* a sub-hierarchy to examine, if the whole is too large, complex, or unclear to display.
5. *visualize* the data, e.g. by producing a graph layout, followed by a drawing, of the selected data [6].

Steps 2 to 5 can occur in any order - one may e.g. first visualize the whole model produced by Step 1, then apply some user- or system-driven aggregation (Step 2), measure the result's quality (Step 3), select a feature to look at (Step 4), and then repeat from Step 2. Step 5 may provide different visualizations besides graph drawing. However, in most cases we are aware of, RE users desire to focus on the specific relations between (groups of) software components, so graph visualization is their first choice.

3. Toolkit Architecture

Our toolkit is built as a layered system (Fig. 1 b). The toolkit core is implemented as a C++ class library for performance. The user interface and scripting layer is implemented in Tcl/Tk for flexibility. We describe next the data and operation model used by the toolkit core.

3.1. Data Model

Our data model contains four elements: structure, attributes, and selections, as follows.

3.1.1. Structure

We model the basic RE data by a *hierarchical (layered) attributed graph*, similarly to [14, 2, 17]. The graph's nodes model software artifacts output from program analysis. The graph's layers model node aggregations (clusterings) done during plan assignment throughout architecture recovery. Several aggregations (nodes with several parents) model alternative system structurings common during RE sessions. The edges model both relational and containment information.

3.1.2. Attributes

Both nodes and edges may have key-value pair attributes. We implement keys as string literals and 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 to 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 [3, 21, 9] 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.

3.1.3. Selections

Selections, defined as sets of nodes and edges, allow executing toolkit operations on a *specific subset* of the whole graph. To make the toolkit flexible, we decouple the operations' definitions from the selections on which they are executed, similarly to the dataset-algorithm decoupling in SciViz. Selections are named, similarly to the attributes. All selections are kept 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 [10]. 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 [10] too, this is one of the main differences between SciViz and graph-based toolkits which leads to different architectures for the two.

3.2. Operation Model

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 distinguish three operation types, based on their read/write data access, as follows (see also Fig. 1 b). *Selection operations* create selection objects (Sec. 3.3). *Graph editing operations* modify the graph data (Sec. 3.4). *Mapping operations* map the graph data to visual objects (Sec. 4). The above data-operation interface allows the system to automatically update all components that depend on the modified data after an operation's execution. For example, the selections are automatically updated after a structure editing operation which deletes selected nodes or edges. Similarly, the data viewers (Sec. 4.0.7) are updated when the selections they monitor change. Although largely similar to the SciViz dataflow mechanism [16], we do not *explicitly* construct an operation pipeline. After attempting to do this, we have found that, in contrast to SciViz applications, RE operations are seldom executed in the same order in typical RE sessions, so building an explicit pipeline burdens more than helps users.

3.3. Selection Operations

Selection operations add nodes and edges of selection objects. We implemented several such 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 nodes in an input selection, 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, and are useful in queries such as 'show all nodes where the *cost* attribute is higher than some threshold'.

3.4. Graph Editing Operations

Graph editing operations edit the graph structure or the node/edge attributes, as follows.

3.4.1. Structure Editing

Structure editing operations construct and modify the graph. Such operations include the standard node and edge addition and removal, as well as reading several graph formats such as file formats such as RSF [21], GraphEd [7], and DOT [12], and GXL [10]. Aggregation operations usually take the nodes in an input selection and produce a unique parent node. The input selection can be either programmatically constructed or can be the output of user interaction (Sec. 4.0.7). Currently we are working on more complex aggregation methods, such as automatic topology-based graph simplification.

3.4.2. Attribute Editing

These operations create, modify, and delete attributes from the nodes' and edges' attribute-sets (Sec. 3.1.2). Besides the selection input, attribute operations have also one or several attribute-plane names as inputs. These names refer to attribute-planes that the operation reads and/or writes, as follows.

3.4.3. Metrics

We treat RE metrics as attribute editing operations. Examples of RE metrics are computing the number of provisions, requirements, and internalizations for some selected nodes [21, 15]. Metrics may produce new attribute-planes, as the above metrics do, or single values, e.g. the cyclomatic number or size for a subgraph. Decoupling the metric's selection input from the selection operation allows by default applying any metric on any subgraph (which is not the case in other RE tools [21, 9]). Moreover, explicitly specifying the input and output attribute-plane names allows easy run-time prototyping of various combinations of metrics, similarly to the way one works with function or matrix objects in systems such as Matlab of Mathematica [23]. Finally, the above decoupling allows the implementations of metrics, attributes, and selections to evolve independently from each other in the toolkit.

3.4.4. Graph Layouts

In contrast to other systems [21, 9, 10], we treat graph layouts simply as attribute editing operations and thus decouple them *completely* from mapping and visualization. This has several benefits. First, we can lay out different subgraphs separately, e.g. using spring embedders [12, 4] for call graphs and tree layouts [18, 12] 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 [6]. We have implemented

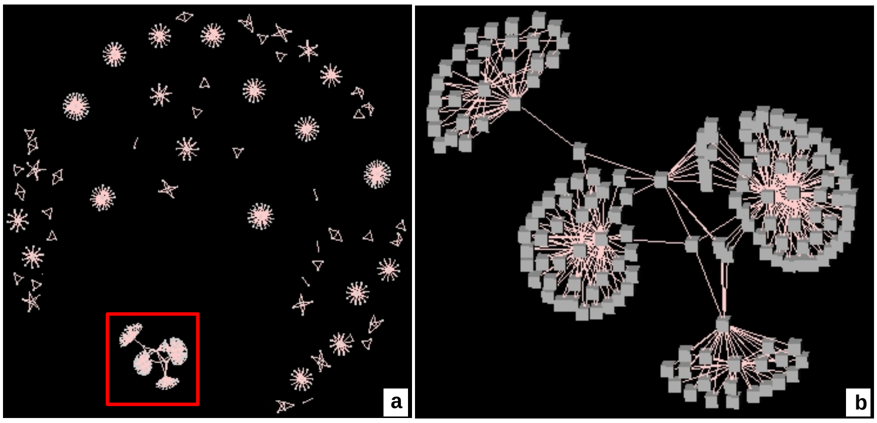


Figure 2: *Software visualization overview (a) and detail (b)*

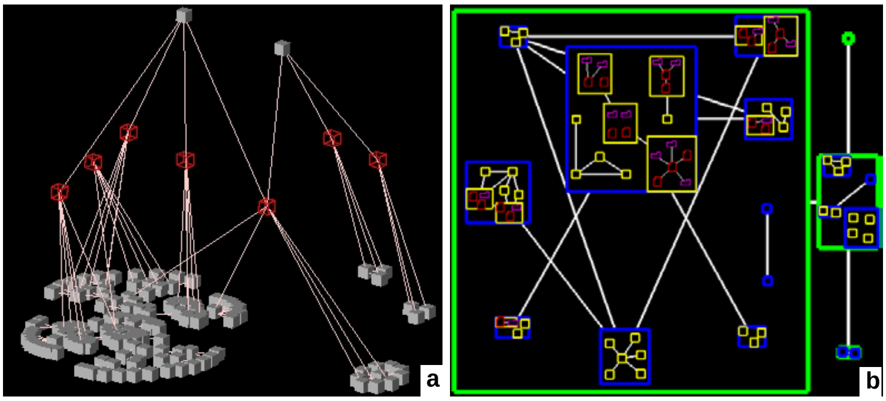


Figure 3: *Custom layouts: stacked layout (a) and nested layout (b)*

several custom layouts by cascading simpler ones, as follows. *Stacked layouts* (Fig. 3 a) 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 in a software system. *Nested layouts* (Fig. 3 b) 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 package UML 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 In the example in Fig. 3 a we use a tree layout, whereas in Fig. 3 b we use a spring embedder as basic layout. Concretely, we use the AT&T's DOT package [12] for tree layouts and AT&T's NEATO, GraphEd, and GEM [7, 4] for spring embedding. We have conducted some hundreds of 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, 9]. 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.

4. Data Mapping and Visualization

Mapping and visualization operations enable users to see and interact with the graph data. These operations have four sub-components: mappers, viewers, glyph factories, and glyphs (Fig. 4). These operations are implemented using the Open Inventor C++ toolkit [19], which offers sophisticated mechanisms for object direct manipulation, picking, and rendering, and are described next.

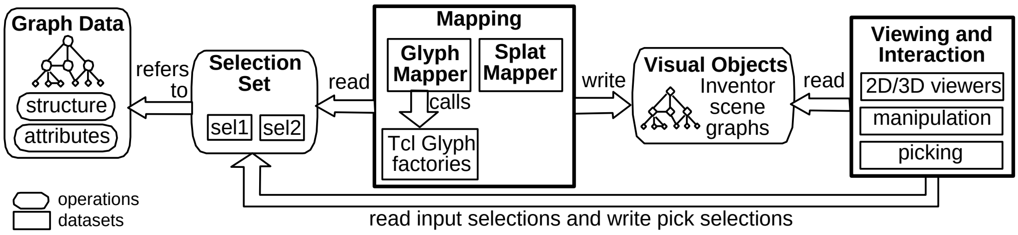


Figure 4: *Software components of the mapping operation*

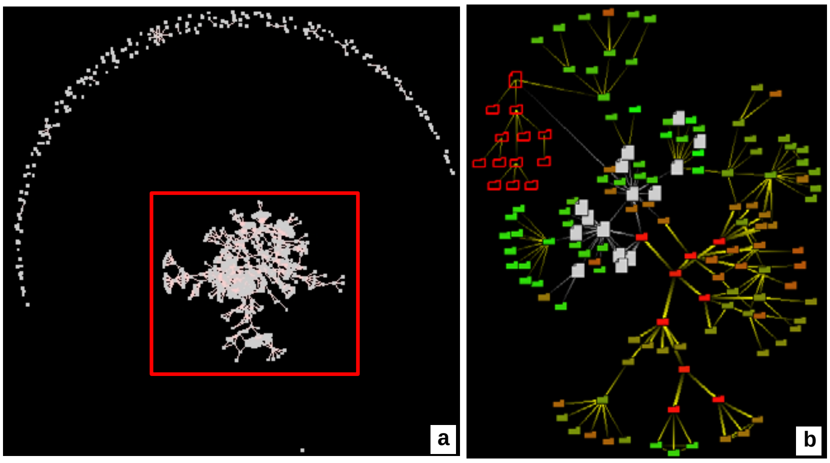


Figure 5: *Visualization of program analysis tool (a) and clustered core detail using glyphs(b)*

4.0.5. The Mappers

The central visualization component is the mapper, which maps selections to Inventor scene graphs. We have implemented several mappers, as follows. 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 an attribute plane of the input nodes and/or edges. This attribute-plane is constructed before mapping by a layout operation (Sec. 3.4). In contrast, the *splat mapper* produces a splat field from the selected subgraph, by the method described in [13]. The splat field can be viewed as a color or elevation plot (see Figs. 7,8,9,10, and 11).

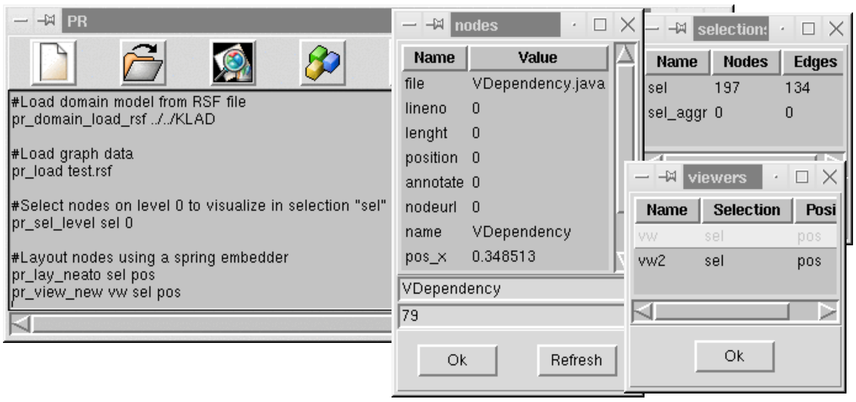
4.0.6. The Glyphs

A *glyph* is a 2D or 3D graphical object that visualizes a node or edge. The glyph mapper calls, for every node and edge it maps, a Tcl script, called a *glyph factory*, which builds the desired glyph as an Inventor node. The script sets the glyph's graphical properties (color, shape, size, annotation, and so on) from the attributes of the input node or edge. Since these scripts may be freely edited by users 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 all 1200 software artifacts (methods, classes, packages, and files) extracted from the code. The right image shows a simplified view of the system's core, after several graph-editing operations (Sec. 3.4) have been applied to cluster the extracted artifacts into higher-level units. Different glyphs have been used to show the different unit types, whereas the subsystem coupling strength [15] is visualized by edge glyphs of different thicknesses. The separation of the glyph placement, done in the layout phase, and the glyph construction, done in the mapping phase, is a simple but powerful way in specifying the visualization. New glyph factories can be developed without being concerned by the layout, whereas new layout tools can be added to operate on existing glyphs.

Although glyph-based visualizations are well known in SciViz, few software visualization systems support glyph mapping. One of the few such systems is VANISH [9]. However, although VANISH can build impressive glyph-based graph visualization, it provides very little freedom for graph manipulation, layouts, as discussed further in Sec. 5.

4.0.7. Viewing and Picking

Viewers are both output components (they display their input selection and provide 2D and 3D navigation) and input components (they edit a so-called *pick selection*). The pick



```
#Load domain model from RSF file
pr_domain_load_rsf ../../KLAD

#Load graph data
pr_load test.rsf

#Select nodes on level 0 to visualize in selection "sel"
pr_sel_level sel 0

#Layout nodes using a spring embedder
pr_lay_neato sel pos
pr_view_new vw sel pos
```

Name	Value
file	VDependency.java
lineno	0
lenght	0
position	0
annotate	0
nodeurl	0
name	VDependency
pos_x	0.348513

VDependency

79

Name	Nodes	Edges
sel	197	134
sel_aggr	0	0

Name	Selection	Posi
vw	sel	pos
vw2	sel	pos

Figure 6: *Tcl/Tk interface of the integrated reverse engineering application*

selection is a subset of the input selection that is displayed in a special color-highlighted manner, as shown by the tree in the upper-left part of Figs. 11 g and 5 b). The pick selection is edited interactively when users pick nodes and/or edges displayed by the viewers. The pick selection is automatically modified upon these pick events, whereupon all toolkit components are updated as described in Sec. 3.2. Moreover, users can attach Tcl scripts to pick events to customize the picking action, e.g. to interactively inspect, delete, aggregate, hide, lay out , or apply metrics on the pick selection. As for the Tcl glyph factories, such scripts usually have 10 to 20 Tcl lines and can be edited on the fly.

5. User Interaction and Assessment

We have built several integrated applications based on the C++ toolkit core presented so far. These applications extend the Tcl interface to the core's C++ API with several Tk-based graphical user interfaces (GUIs) and Tcl scripts. 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 (Fig. 6). These integrated applications are functionally very similar to other RE tools such as Rigi [21], VANISH [9], or graph visualization tools such as Royere [10]. However, several differences are to be mentioned. The main difference is our toolkit's core architecture which is based on a few loosely coupled, orthogonal components: graph and selection data objects, operations, mappers, glyphs, and viewers. The data-operation loose coupling, via selections, makes it natural for developers to write small, independent operations - so far all our operations range from 20 to 150 C++ or Tcl lines. For example, the script shown in the background window of Fig. 6 that produces the graph visualization in Fig. 5 a, has 11 Tcl lines. In contrast, Rigi [21] uses a monolithic core architecture. Although somewhat adaptable via Tcl scripts, this architecture offers no subclassing or composition mechanisms *for the core itself*. It is not possible, for example, to change the graphic glyphs, the interactive selection policy, or to add a new mapper without recoding the core. Similarly, adding a new layout, selection operation, or metric involves a low level API to access nodes and edges, as Rigi has no notion of manipulating these as selections. VANISH [9] provides a way to build custom glyphs very similar to our glyph factories (Sec. 4.0.6). However, VANISH uses node and edge attributes based on compiled C++ classes which prove inflexible for our targeted RE scenarios (Sec. 3). Finally, we should mention the large class of library-level toolkits, such as GVF [10], GTL, or Graphlet [8]. These toolkits provide basic graph data manipulation and usually do not address visualization, interaction, and RE-specific operations. From these, our toolkit resembles GVF the most. However, we found GVF's Java-based API rather complex to understand and use, especially for non object-oriented expert end users, which led us to our choice for a light Tcl customization layer to a C++ core.

6. Applications

We have used the presented integrated GUI application for the exploration of reverse engineering data obtained from several software systems built at Nokia. First, we extract an attributed graph from the original Java, C, or C++ program source code. The graph's nodes are software entities such as functions, classes, files, and packages. The arcs map relationships such as 'uses', 'contains', 'calls', and 'implements'. Various code attributes such as names, number of code lines, version numbers, change dates, etc are stored as node/edge attributes. The graph is loaded in our RE tool after which the RE operation pipeline (Sec. 2), i.e. selection, aggregation, metric computation, layout, mapping, and viewing, is executed.

Figure 2 a shows a simplification displaying about 20% of a graph of 4000 nodes, layed out with a spring embedder. The about 850 clusters shown here correspond to different loosely-coupled subsystems in the original software. The

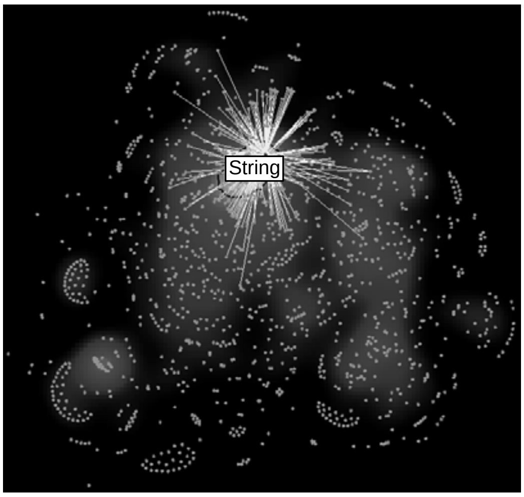


Figure 7: *Software graph splatting. Packages using String class*

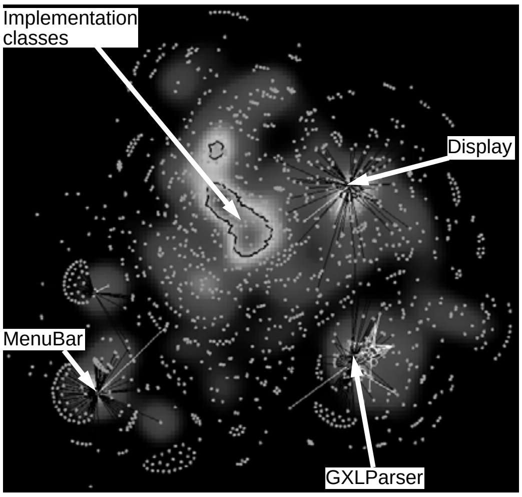


Figure 8: *Software graph splatting. Packages' requirements*

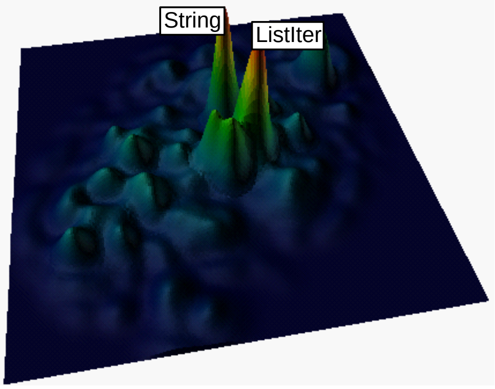


Figure 9: *Software splatting visualized with elevation plots*

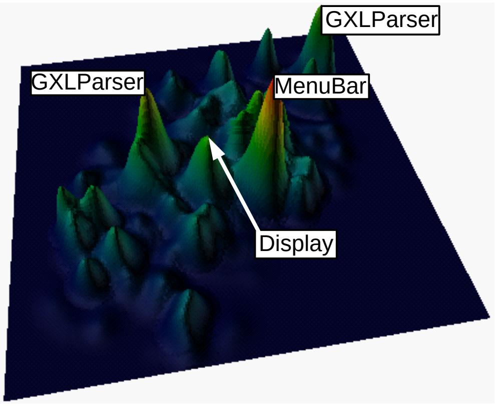


Figure 10: *Software splatting visualized with elevation plots*

largest subsystem, shown in the lower left part of Figure 2 a, was selected interactively by the user and then displayed separately in a second viewer (Figure 2 b). In this image, we can easily detect the 'bridge' (or interface) software components as being those nodes that connect the large, densely coupled subgraphs. In Fig. 5 a, 1200 entities from a differently architected software are visualized. The central dense graph is the system's core, whereas the elements scattered around it are user interface and I/O code. In Figs. 7 and 8, about 2000 entities extracted from a third system are visualized. As the extracted graph is strongly connected, visualizing it directly is not effective. Therefore, we visualize it using graph splatting on a spring embedder layout [13], as follows. In Fig. 7 (11 d in color), the scalar density function (splat field) shows the number of *provide relationships* (who is called by whom). The user has picked all packages using the `String` Java class. As expected, a large part of the code uses this class. Figure 8 (11 e in color) shows the number of *require relationships* (who calls what). The splatting maxima denote the key system components: the Display (visualization), GXL parser (data reader), and MenuBar (GUIs) package. The 'hot area' in the center of the image is populated by several implementation classes, such as strings, container classes, and so on, that refer to each other very frequently. Figures 9 and 10 (11 f in color) shows the same data as before, displayed with an elevation plot of the splat field. Even though this time we use different layout parameters, peaks denoting the same software components as before emerge.

The above scenarios, starting from the RSF data delivered by the code parser, took each under 10 minutes to build. This implied the execution, via the tool's GUI and its Tcl command-line, of less than 20 operations. To produce the zoomed-in image shown on the right, a Tcl procedure of less than 15 lines was written that takes the left viewer's highlight selection output (Sec. 4.0.7, applies a spring embedder layout, and maps it in a new viewer. To visualize node details, e.g. attribute names and values, one user wrote a second 12-line Tcl procedure that opens a inspection GUI window (as shown in Fig. 6 middle). This procedure is activated on the changing of the highlight selection in the detail viewer, i.e. when one clicks on the desired node in this viewer.

Overall, our RE tool proved more flexible than Rigi or VANISH for the same case data. Among the most positive points reported by end users were the possibility to write custom selections, metrics, and glyph factories in a few Tcl lines and to test them interactively. The slowest and most delicate to tune part was the layout computation, especially for graphs over 800 nodes. For such graphs, we had to design, usually on the fly, several selection and/or aggregation operations to reduce the data size prior to layout and visualization or alternatively use splatting on the whole graph.

7. Discussion and Future Work

We have presented a new toolkit for prototyping RE data exploration which has several advantages as compared to similar systems we have worked with. Our aim was to strike a balance between producing a too complex to learn and maintain (yet versatile) system and building a too rigid and specialized (yet simple to use) one. Overall, our toolkit has currently about 10000 C++ and 1500 Tcl lines grouped in around 50 classes and took five man-months development time. The toolkit implements around 60 operations (8 data readers, 4 data writers, 20 structure editing and metrics operations, 8 layout operations, and about 15 mapping operations). The GUI-based application built atop of the toolkit adds around 500 Tcl lines and proved in our daily practice easier to customize than specialized systems such as [21, 9, 10]. For most visualization scenarios imagined by our users, writing (or adapting) a few small Tcl scripts of under 50 lines was enough. This was definitely not the case with other RE systems we worked with. Although our focus is reverse engineering, our toolkit can be directly used as a prototyping platform for various graph-based visualization applications.

We further aim to develop and integrate several domain-specific operations, such as graph simplification, layout, and glyph mapping, for the software domain models used at Nokia. Our RE system will thus serve both as a tool for understanding our mobile telephony software and as a testbed for prototyping new information visualization techniques.

References

1. T. Biggerstaff, B. Mittbrander, D Webster, *The Concept Assignment Problem in Program Understanding*, Proc. WCRE '93, IEEE CS Press, 1993.
2. S. Card, J. Mackinlay, B. Shneiderman, *Readings in Information Visualization*, M. Kaufmann, 1999.
3. S. Eick and G. Wills, *Navigating large Networks with Hierarchies*, in *Readings in Inf. Vis.* [2].
4. A. Frick, A. Ludwig and H. Mehldau, *A fast adaptive layout algorithm for undirected graphs*, Proc. Graph Drawing '94, Springer, 1995.
5. E.R. Gansner, S.C. North, *An open graph visualization system and its applications to software engineering*, Software-Practice and Experience, John Wiley & Sons, (S1) 1-5, 1999.
6. Herman, G. Melancon, M.S. Marshall, *Graph Visualization and Navigation in Information Visualization: a Survey*, IEEE TVCG, 2000.
7. M. Himsolt, *GraphEd user manual*, Technical report, Fakultat fur Informatik, Universitat Passau, 1992.
8. M. Himsolt, *Graphlet: Design and Implementation of a Graph Editor*, Software – Practice & Experience, no. 30, pp. 1303–1324, 2000
9. R. Kazman, J. Carriere, *Rapid Prototyping of Information Visualizations using VANISH*, Proc. IEEE InfoVis '95, IEEE CS Press, 1995.
10. M. S. Marshall, I. Herman, G. Melancon, *An Object-Oriented Design for Graph Visualization*, Software: Practice & Experience, 31, pp. 439–756, 2001
11. A. Mendelzon, J. Sametinger, *Reverse Engineering by Visualizing and Querying*, internal report, Computer Systems Research Institute, Univ. of Toronto, Canada, 1997.
12. S. C. North, E. Koutsofios, *DOT and NEATO User's Guide*, AT&T Bell Labs Reports, http://www.research.att.com, 1996.
13. R. van Liere, *Studies in Interactive Visualization*, PhD thesis, CWI, Amsterdam, 2001.
14. J. Rohrich, *Graph Attribution with Multiple Attribute Grammars*, ACM SIGPLAN 22 (11), pp.55-70, 1987.
15. S. Tilley, *A Reverse-Engineering Environment Framework*, CMU/SEI-98-TR-005, Carnegie-Mellon, 1998.
16. W. Schroeder, K. Martin, B. Lorensen, *The Visualization Toolkit*, 2nd edition, Prentice Hall, 1998
17. J. Stasko, J. Domingue, M. H. Brown, B. A. Price, *Software Visualization - Programming as a Multimedia Experience*, MIT Press, 1998.
18. K. Sugiyama, S. Tagawa, M. Toda, *Methods for Visual Understanding of Hierarchical Systems Structure*, IEEE Trans. Systems, Man, and Cybernetics, Vol. 11, No. 2, pp.109-125, 1989.
19. J. Wernecke, *The Inventor Mentor: Programming Object-Oriented 3D Graphics*, Addison-Wesley, 1993.

20. K. WONG, S. TILLEY, H. MULLER, M. STOREY, *Structural Redocumentation: A Case Study*, IEEE Software 12 (1), 1995, pp. 46-50.

21. K. WONG, *Rigi User's Manual version 5.4.4*, Dept. of Computer Science, Univ. of Victoria, Canada.

22. P. YOUNG *Program Comprehension*, Centre for Software Maintenance, Univ. of Durham, 1996.

23. MATLAB, *Matlab Reference Guide*, The Math Works Inc., 1992.

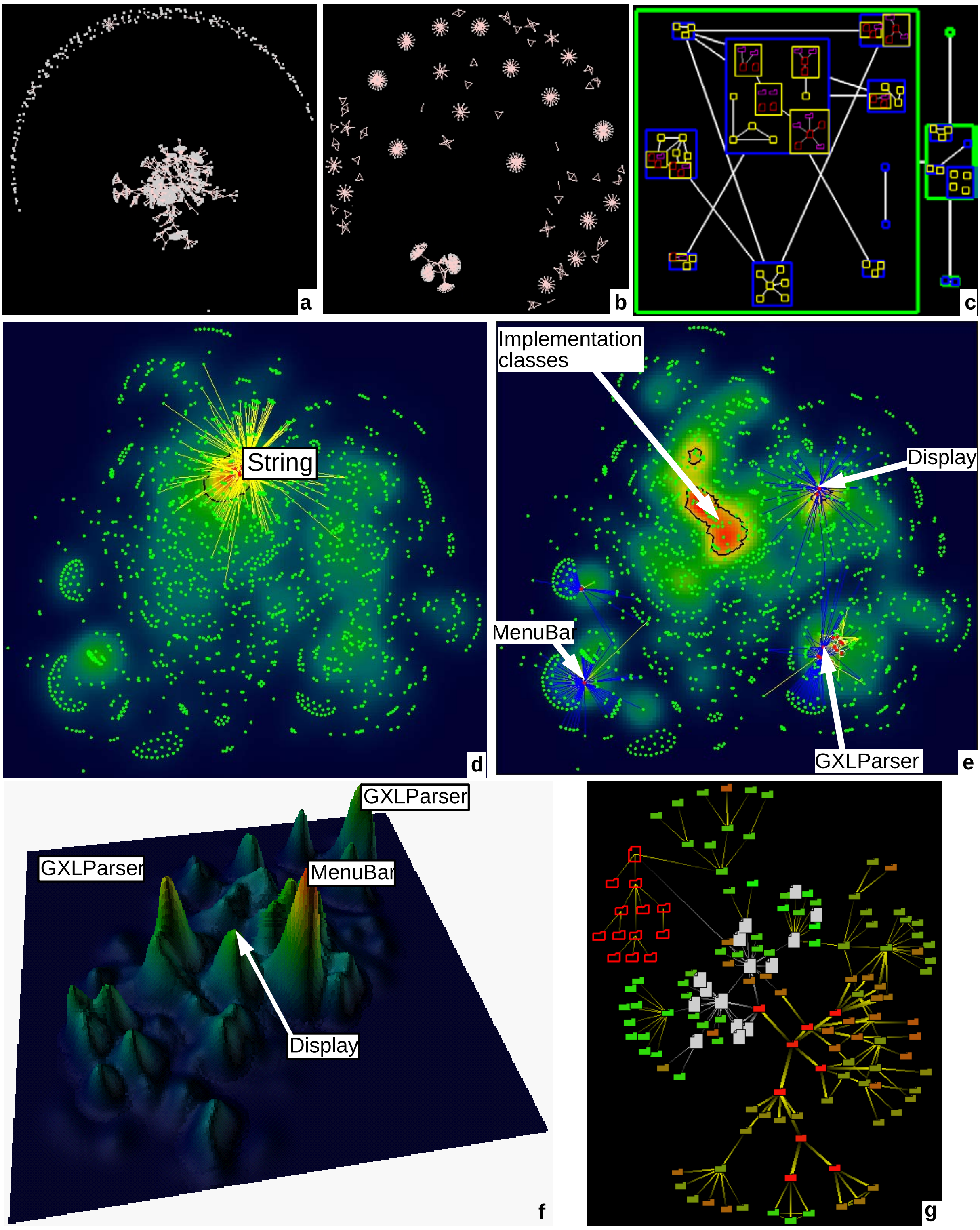


Figure 11: *Software visualizations in reverse engineering applications*