Software Visualization for Reverse Engineering

Rainer Koschke

University of Stuttgart,
Breitwiesenstr. 20-22,
70565 Stuttgart, Germany
`koschke@informatik.uni-stuttgart.de`,
`http://www.informatik.uni-stuttgart/ifi/ps/rainer`

Abstract.

This article describes the Bauhaus tool suite as a concrete example for software visualization in reverse engineering, re-engineering, and software maintenance. Results from a recent survey on software visualization in these domains are reported. According to this survey, Bauhaus can indeed be considered a typical representative of these domains regarding the way software artifacts are visualized. Specific requirements for software visualizations are drawn from both the specific example and the survey.

1 Introduction

Reverse engineering is the process of analyzing a subject system to identify the system's components and their relationships and create representation of the system in another form or at a higher level of abstraction, whereas *re-engineering* is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form. Re-engineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring [Chikofsky & Cross, 1990].

Research in reverse engineering focuses on extracting, storing, presenting, and browsing information, reducing the amount of unnecessary information for a particular task, analyzing the extracted data and to build useful abstractions of the system under analysis. Because reverse engineering is generally a highly interactive and incremental process, in which results of automatic analyses need to be presented to the reverse engineer that are then validated, augmented, and fed back to following automatic analyses, software visualization plays a key role in reverse engineering. Presenting the data to the reverse engineer in a suitable manner is a main issue here and the reverse engineering research community struggles with finding solutions to this problem.

In the next section, the Bauhaus tool suite is described as one concrete example of software visualization for reverse engineering. The example is not a particularly advanced use of software visualization. As a matter of fact, I rather believe that there is still enough room for improvements. The example was chosen because it can be considered typical for the domain according to a survey that I recently conducted [Koschke, 2001]. Bauhaus is rather an example for the state of the practice than for the state of the art with respect to its software visualization capability. With respect to its analytical reverse engineering capabilities, it is more advanced. Yet, its exact analyses are beyond the scope of this article. The results of the more general survey are presented in Section 3.

2 Architecture Recovery in Bauhaus

The *Bauhaus project* researches reverse engineering techniques to help program understanding of legacy code [Bauhaus, 2001]. Bauhaus has support for frequent maintenance tasks that involve program-understanding-in-the-small (points-to and side effect analyses, detection of uses of uninitialized variables and dead code, program slicing, etc.) and re-engineering tasks that require knowledge of the system's architecture and hence are more oriented toward program-understanding-in-the-large.

This section describes the software visualization in Bauhaus. However, we will start with some background information on reverse engineering.

2.1 Reverse Engineering Background

Analyzing a system, we can roughly distinguish three different levels of abstraction:

– The *lower level* represents the source code in a way that contains *all necessary details* of syntactic, semantic, control and data flow information.
– The *middle level* only contains *global* information that can be automatically extracted from source code, like global variables, functions, user-defined types and their relationships. The middle level is the seam between the lower level and the next upper level, namely, the architectural level.
– The *architectural level* contains architectural information.

Bauhaus seeks to recover architectural descriptions from source code. According to the *IEEE Standard on Recommended Practice for Architectural Description of Software-Intensive Systems* [IEEE-Std-1471-2000], an *architectural description* is a collection of products to document an architecture. An *architecture* is the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution. A *view* is a representation of a whole system from the perspective of a related set of concerns. A *viewpoint* is a specification of the conventions for constructing and using a view. A viewpoint acts as a pattern or template from which to develop individual views by establishing the purposes and audience for a view and the techniques for its creation and analysis [IEEE-Std-1471-2000].

2.2 Extracted Artifacts

Information about the system is exclusively extracted from the source code in Bauhaus – because this is the only reliable source of information – and represented at each level as graphs with varying granularity. The maintenance analyses are based on information at the lower level only and are fully automatic. Their results are generally represented as links to the original source code. On the other hand, architecture recovery with Bauhaus is a semi-automatic process that involves the Bauhaus user (presumably a software maintainer or auditor). Due to the highly interactive nature of this process, software visualization is an important part. While Bauhaus uses marked-up text and hypertext for the results of the more source-code oriented maintenance analyses at the lower level, the architecture recovery tasks have higher demands on software visualization.

At the middle level, information extracted from the source code is represented as a resource graph (RG) [Koschke, 2000]. The resource graph is a coarse-grained intermediate representation that represents concrete as well as conceptual information in form of entities and their relationships. Concrete information constitutes a *base view* and is global information that can be directly extracted from the source code, yet abstracts from a particular source language, such as call, type, and use relations. The conceptual information is added by the Bauhaus user based on results from automatic reverse engineering analyses. Currently, the Bauhaus toolkit offers techniques for component and connector recovery. In the following, we will focus on the component recovery part of Bauhaus. The semi-automatic process of component recovery will be described in Section 2.4.

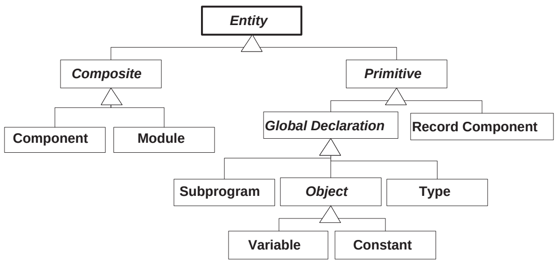


Fig. 1. Entities of the Resource Graph

The entities of the RG are programming language constructs and recovered abstract concepts. The entities, which are represented as nodes in the RG, are shown by the UML inheritance model in Figure 1. Relationships are represented as edges in the RG. Examples of relationships range from information that can be directly extracted from the source code (e.g., function calls, variable references, function signatures) – as shown in Figure 2 as *base viewpoint* – to more abstract concepts (e.g., the *conceptual component decomposition* as shown in Figure 2). The RG can be visualized and manipulated in Bauhaus with our extension of the graph editor Rigi [Müller, 1992]. The RG is the shared knowledge base – so-to-speak – between automatic analyses and the Bauhaus user.

Even though the RG is a coarse-grained representation of the system, the RG gets large for large systems. Figure 3 shows the number of nodes and edges in the RG that represent extracted information for five C systems (Concepts, Aero, Bash, CVS, and Mosaic) in the range of 7 to 52 KLOC (thousand lines of code). Interestingly enough, as Figure 3 shows, the RG size is not necessarily a monotonic function of the program size in terms of commented lines of codes (Figure 3 only contains the entities and relationships of the base viewpoint as described in Figure 2). The RG for the system Aero, which has about 28 KLOC, has more nodes and edges than the RG for the three larger systems (except for the number of edges for the largest system, namely Mosaic). The number of nodes per lines of code depends on the programming style, in particular the number of lines of code per subprogram and the use of global variables. However, the graph density in terms of number of edges per node seems to be a constant factor of about four edges per node at the middle level.

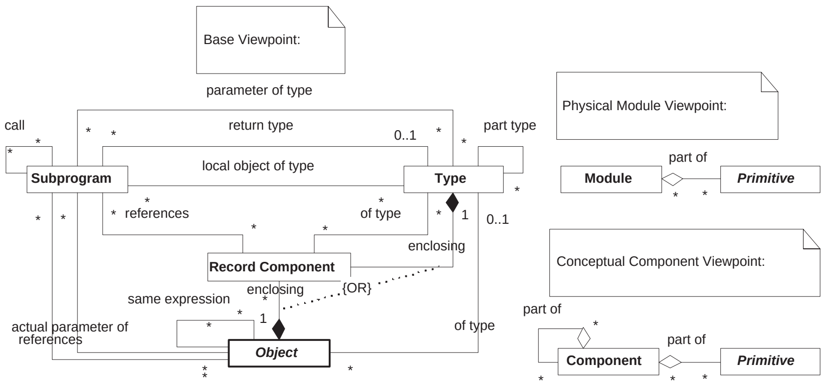


Fig. 2. Base, Physical Module, and Conceptual Component Viewpoints

2.3 Recovered Conceptual Artifacts

One particularly developed part of Bauhaus supports component recovery for procedural programming languages, specifically for C. *Components* are cohesive groupings of related global declarations of subprograms, variables, and user-defined types. The decomposition of the system into components is a structural viewpoint of the conceptual architecture. Component views are needed for re-modularization, migration to object-oriented languages, and program understanding in general. The component view gives a maintainer a conceptual perspective of the system and is in contrast to the *physical module view* that shows how the system is decomposed into physical modules (or files in the programming language C). The latter view can trivially be derived by looking at the existing files that make up the system. However, the physical module view shows the system as built and may not necessarily reflect how the decomposition of the system should be decomposed into cohesive groupings. Due to ad-hoc maintenance tasks the (originally well-designed) physical module decomposition may be deteriorated, i.e., show high coupling between modules and low cohesion within modules. Conceptual components, by definition, are always cohesive.

Figure 2 shows the description of the *physical module view* consisting of modules, program primitives, and their part-of relationships and the *conceptual component view* consisting of modules, program primitives, and their part-of relationships. Note that components can be hierarchical whereas modules that model C header (include files with suffix *.h*) and body files (that contain function definitions with suffix *.c*) cannot contain other modules.

The process of recovering the component view is described in the following section.

2.4 Component Recovery Process

Bauhaus supports an iterative semi-automatic method to detect *components*, also known as *logical modules* or *objects*. The analysis cycle consists of the following steps (see Figure 4) [Koschke, 2000]:

1. The Bauhaus user selects one ore more fully automatic techniques. Currently, 15 different techniques are available, many of them have additional variants. The description of the techniques is beyond the scope of this article. A detailed description can be found in [Koschke, 2000].

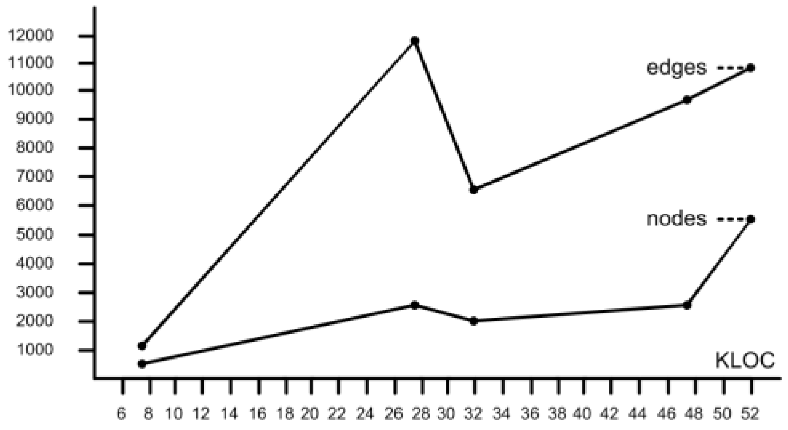


Fig. 3. Number of Nodes and Edges at the Middle Level

2. The selected techniques use as input the base view that contains the program primitives and their relationships extracted from the system (see Figure 2) and the so-called *user view* that contains the components that have been found so far. In the beginning, the user view is empty.
3. Each analysis application yields one component view, which can be assessed by certain metrics (e.g., number of elements, name similarities, cohesion, etc.). Multiple component views of different analyses may be combined by intersection, union, and difference based on fuzzy sets.
4. The resulting combined component view may then be validated by the user. The user can reject components in part and as a whole and may add additional entities to existing components. Accepted components are moved to the user view.

The user controls the detection process by selecting analyses and metrics and by validating the candidates proposed by the automatic techniques. The task of the computer comprises the automatic analyses, computation of the metrics for the proposed candidates, presentation of the results, and bookkeeping of the user decisions.

An analysis selected by the user takes into consideration the components that were previously confirmed by the user (in the first iteration there are none). Thus, the analyses are applied incrementally. In each iteration, the user selects and combines different analyses to find components that could not be found by previous analyses. The process ends when the found components are sufficient for the task at hand or no further component can be found anymore. Each intermediate and resulting view can be visualized as a graph as described in the following section.

2.5 Software Visualization in Bauhaus

Suitable visualization is an important issue in the above process of architecture recovery. The user should be able to quickly grasp the represented information. Moreover, he or she should also be able to quickly derive other information that might be needed, yet not forseen by our analyses. Hence, beyond pure visualization, browsing capabilities for the large information space need to be offered, too. Since our research focus in on the analytic parts of architecture recovery rather than visualization or browsing of architectural views and developing an own visualization and browsing tool is a major effort, we used and extended the generic graph editor, Rigi, originally developed by Hausi Müller and his team at the university of Victoria in Canada [Müller, 1992].

Rigi is an interactive, visual graph editor designed to help to better understand and redocument software. The underlying multi-graph consists of typed attributed nodes and directed edges. Special level edges allow for hierarchical graphs. Rigi provides selection, filtering, and editing operations, dependency and change impact reports, overview and projection perspectives, metrics for cohesion and coupling, views to capture interesting perspectives, a scripting language and command library, annotations of nodes and edges, and a customizable

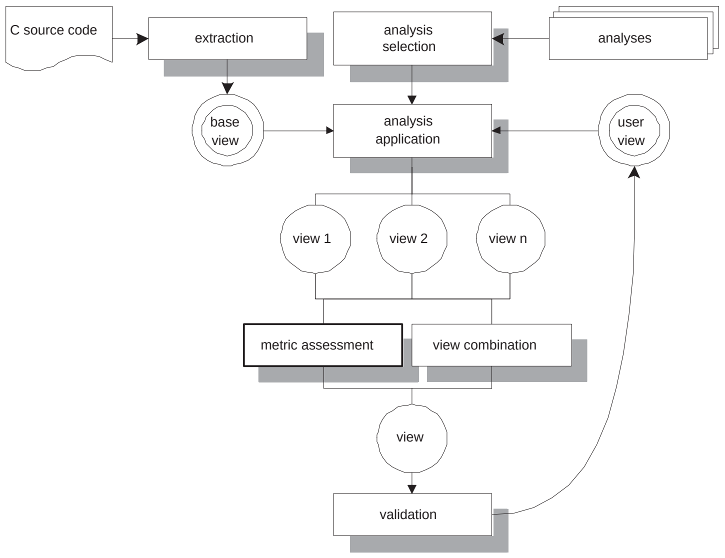
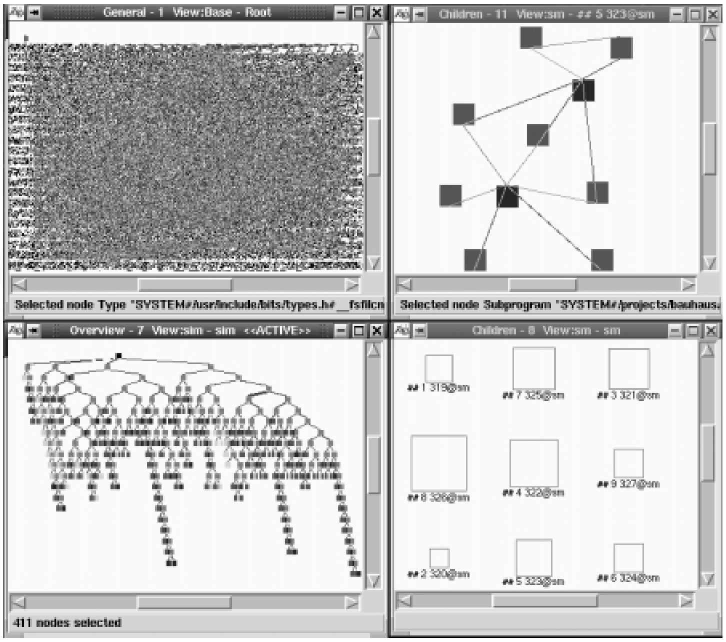


Fig. 4. Process of Component Recovery in Bauhaus

user interface. Nodes may be linked to source positions and a user-defined text viewer may be opened to show the source text when the user clicks on the node. Rigi is adaptable to different languages and purposes.

Rigi also offers different automatic graph layouts. Only some of them are built-in layouts, namely, those for trees and grids. For more advanced layouts, `Graphed` is used as an external layouter for spring embedder and Sugiyama's layout [GraphEd]. Our group has integrated `Graphlet` as an additional external layouter [Graphlet]. Tree layouts are used in Bauhaus/Rigi for the dominance tree, which shows local functions in the call graph and subsystems in the dependency graph, and for the result of hierarchical clustering techniques for component recovery. Spring embedder and Sugiyama's algorithm are used for call graphs and type and reference relationships.

Figure 5 shows a few visualization examples with Bauhaus/Rigi. The left upper window shows all nodes and edges of the base view (the base view is described in Figure 2) for a 7.5 KLOC system, overlapping each other. Obviously, this visualization is useless. In order to understand all the relationships that are there, filtering and selection mechanisms are needed. The upper right window shows the contents of a component that contains two types and eight subprograms. The visualization immediately shows that three subprograms are connected to both types and all others are connected to one type only. The lower left window shows the result of a hierarchical clustering technique. The iterative hierarchical clustering technique groups in each step the two most similar subtrees based on a similarity metric in a bottom-up fashion. The visualization shows the order of this clustering, i.e., the most similar entities can be found in lower subtrees. The higher a subtree, the less similar are its elements. Each inner node (which represents a grouping) is annotated with its similarity value. This visualization is an important guidance for a user's validation. The user can start at the leaves and then climb up the tree until a grouping gets doubtful. In principal, the same information could also be "visualized" as a matrix of similarity values, but the superiority of the visualization as tree is obvious. The visualization is even active in the sense that a user can reject and accept groupings within the visualization and obtain the source code of contained programming primitives by one mouse click. The lower right window shows an example on how components can be assessed with metrics. The metric value is expressed as the size of the nodes that represent components. This way the relation between the components with respect to the given metric is easy to grasp.


General - 1 View:Base - Root
Children - 11 View:sm - ## 5 323@sm
Overview - 7 View:sim - sim <<ACTIVE>>
Children - 8 View:sm - sm
## 1 319@sm
## 7 325@sm
## 3 321@sm
## 8 326@sm
## 4 322@sm
## 9 327@sm
## 2 320@sm
## 5 323@sm
## 6 324@sm

Selected node Type "SYSTEM#/usr/include/bits/types.h#__fsfilcn

Selected node Subprogram "SYSTEM#/projects/bauhaus.

411 nodes selected

Fig. 5. Different Kinds of Visualization with Bauhaus/Rigi

The Bauhaus GUI, which is based on Rigi, has a few unpleasant properties. It is relatively slow, which can cause noticeable waiting periods for large graphs and hence sometimes disrupts the fluent use of the tool. Graphs with more than 500 nodes, graphs with multiple edges between the same nodes, and disconnected graphs cannot be automatically layouted due to limitations of the external layouter `Graphed`. `Graphlet` does not have these principal restrictions, but it is much slower than `Graphed` and its layouts cannot be used for larger graphs in an interactive application.

The visualization of nodes and edges is limited, too. All nodes have the same shape of a rectangle. Semantics of nodes and edges is encoded by colors only, making it difficult to distinguish nodes and edges if one has many different types. Nodes have only two ports to which edges can be connected – one for incoming, one for outgoing edges. If two nodes have multiple shared edges, edges lay on top of each other. Edges are only straight lines and may not have bends, which basically makes use of planar graph layouts impossible.

3 Software Visualization for Reverse Engineering in General

Software visualization in Bauhaus is rather typical for the domain of reverse engineering as can be seen by a survey recently conducted [Koschke, 2001]. This section describes the findings of the survey.

The survey was conducted by way of a questionnaire sent via email to researchers in the areas of software maintenance, reverse engineering, and re-engineering. The list of researchers was compiled from several lists of attendees and PC members of conferences related to these fields, namely, the *International Conference on Software Maintenance* (ICSM), the *Working Conference on Reverse Engineering* (WCRE), the *International Workshop on Program Comprehension* (IWPC), and the *European Conference on Software Maintenance and Re-engineering* (CSMR). The list contained about 580 different email addresses. The response rate of the survey was about 20 percent. Out of the 111 answers, 83 researchers confirmed that they are active researchers in the area of software maintenance (absolute number: 56), reverse engineering (52), re-engineering (36), metrics (25), and related domains (6), where multiple selections were possible.

Roman and Cox define *program visualization* as the mapping from programs to graphical representations [Roman & Cox, 1992]. The definition is general enough that we can widen it to other kinds of software artifacts (including programs). Consequently, we define *software visualization* as the mapping from software to graphical representations. Different categories of software visualization may be distinguished according to the following criteria with respect to this mapping [Roman & Cox, 1992]:

– Scope: what aspect of the software is visualized?
– Abstraction: what kind of information is conveyed by the visualization?
– Specification Method: how is the visualization constructed?
– Technique: how is the graphical representation used to convey information?

The survey on software visualization in the area of software maintenance, reverse engineering, and re-engineering has shown that the scope of the visualization is quite diverse [Koschke, 2001]. It ranges from module and subsystem dependencies, call graphs, object models, software architectures, web artifacts, semantic nets and ontologies, control and data flow, database schemas, directory structures to source text.

In terms of Roman and Cox's taxonomy, the abstraction mechanism embodied in common reverse engineering tools is usually a structural representation that is obtained by concealing or encapsulating some of the details associated with the software or its execution and using a direct representation of the remaining information [Roman & Cox, 1992]. Graphs are typically used to depict program structures, dependencies, control and data flow. The information presented to the viewer is present in the program, although simply obscured by details. The representation simply conveys the information in a more economical way by suppressing aspects not relevant to the viewer. For instance, a call graph shows aspects of the software's global control structure, but at the same time it suppresses detailed aspects of the call sites, like the conditions that must hold for the calls to happen and the exact number and order of multiple calls in the same function body.

The specification method is generally predefined, i.e., the viewer cannot really influence the way the information is presented. Many systems allow the viewer to specify colors or shapes for the visualization or select different kinds of automatic graph layouts, but the principle way of visualizing is generally fixed.

The techniques most maintenance, reverse engineering, and re-engineering tools use to visualize information are centered around graphs and text, as shown by the recent survey [Koschke, 2001]. Among the selected representations, graphs are used in 52% of the cases (many of them are hierarchical graphs). In 18% of the cases, UML diagrams are used. Text and hypertext are used in 18% of the cases, but we may assume that a textual representation is actually much more often used than responded – many people do not perceive text as a way of software visualization. Other, less frequently used ways of representation are scatter plots, charts, process flows, database models, and tables.

Animation is rarely used. Only 12.5% of the respondents said that they are using animated representations. Interestingly enough, 40% do believe that animation is useful (in particular for dynamic information) and 34% responded that it might be useful – whereas only 15% believe it is not useful at all (11% did not answer the question).

Since graphs are the dominant way of visualization, the question is raised whether automatic graph layout algorithms are used. As a matter of fact, 71% of the respondents use automatic graph layouters (12% did not answer the question). Among these, surprisingly many have implemented their own graph layout algorithms (28%). Only 41% use the readily available non-commercial or commercial graph layout packages (31% did not answer the question). The most frequently used layout package for graphs is the `GraphViz` system by AT&T (12 in absolute numbers) [GraphViz]. `GraphEd` [GraphEd] and its successor, `Graphlet` [Graphlet], together amount to 7 users. Another 9 use commercial UML tools for rendering UML models, like Rational Rose or Together. `VCG` is used by 5 [VCG], whereas the commercial layout package by Tom Sawyer Software is used by 2. Three people are using Rigi (these people also use the integrated `GraphEd` implicitly; the number of users given for `GraphEd` includes the Rigi users). The remaining 10 people use other packages such as Java2D, Java3D, Microsoft Visio, daVinci, and others.

The most frequently used class of layout algorithms is those for trees (10 in absolute numbers); 9 people use a Sugiyama algorithm, and 7 a spring embedder. Only 2 people use planar graph layout algorithms. The remaining 8 use other, less known layouts like Tunkelang, Minbackward, Barymedian, Manhattan Edges, X-Dags, SequoiaView, and others.

The encouraging message for researchers in the area of software visualization is that 40% of the interviewed people believe software visualization is absolutely necessary for software maintenance, reverse engineering, and re-engineering and still 42% think software visualization is important but not critical. 7% think that it is at least relevant and 6% that they can do without but it is nice to have. Only 1% (actually, a single person) believes software visualization is not an issue at all (4% did not answer the question).

Even though most people acknowledge the importance of software visualization, relatively few people consider it their primary research (11%) or at least a substantial part of their research (18%). Many people are doing research in software visualization every now and then (20%). The relative majority is primarily using or integrating existing software visualization tools developed by others (33%). 11% do not deal with software visualization at all. 7% did not answer the question.

4 Conclusion

We conclude with listing problems of software visualization for software maintenance, reverse engineering, and re-engineering drawn from our own experience in Bauhaus and the survey. I do not think that they are all specific to the mentioned domains, but will arise for all domains in which large and semantically rich information spaces are to be visualized.

Semantics. Graphs are frequently used to represent information. However, these graphs do have semantics and automatic layouts should take the semantics of nodes and edges into account and also the conventions used to draw such graphs manually. For instance, in a UML class model, one would expect to direct all inheritance relationships in one direction; all other kinds of relationships are subordinated.

Size. The amount of data that need to be visualized can be rather large; graphs with 4,000 nodes and more are typical. One may argue that graphs at this size should not be visualized at once since they cannot be understood at this size anyway. In fact, one needs additional navigation, selection, and filtering mechanisms. However, even if a larger call graph with more than 1,000 nodes (still a small system) should be presented in excerpts, the layout for the whole graph needs to be computed in advance. Then a "lense perspective" could be used to browse the large graph showing only subgraphs – where the nodes in the whole graph as well as in the mental map of the viewer would keep their position while the viewer moves the lense.

Evolution. Maintenance and reverse engineering activities require weeks, months, or even years, and usually one cannot afford to freeze normal development. Consequently, the system is permanently under change and, hence, there is not just one graph, but many graphs that are derived from each other. Visualizations may evolve and one has to keep track of this evolution

Multiple users. Large maintenance and reverse engineering projects require team-work and, hence, visualizations need to support multiple users that may work at the same system at the same time at possibly different locations.

Multiple views. Maintenance and reverse engineering involve different stakeholders and, thus, require multiple perspectives from which a system may be viewed. Moreover, different dimensions of the data need to be visualized, like the time dimension or level of abstraction. Multiple views raise the questions of how to integrate these views, how to navigate within and between views, and how to preserve the context during navigation?

Static and dynamic visualization. Most kinds of visualization in maintenance, reverse engineering, and re-engineering are static. However, for dynamic aspects animated visualization would be useful.

Cognitive models. There is a lack of cognitive models for visual understanding and empirical evidence for appropriate visualizations, i.e., we currently do not really know how maintainers grasp visualized artifacts and which kinds of visualization work best for a specific problem. If we knew answers to the latter question, we could also try to automatically select the right kind of visualization depending upon criteria of the input data to be visualized.

Interoperability. No single tool alone can solve the manifold and complex problems of reverse engineering. Consequently, several tools need to be integrated, which requires a high degree of interoperability among tools. Currently, the reverse engineering community works on interoperability issues, in particular, on data exchange and standard schemas. GXL has been evolved to a standard vehicle for data exchange among reverse engineering research tools [GXL]. GXL basically allows one to transfer graphs. It would be advantageous to the reverse engineering and software visualization community to agree upon a joint exchange format.

The Dagstuhl seminar on software visualization has brought together many researchers from very different areas of software visualization. Most of their ideas on software visualization are specifically interesting for the domain of maintenance, reverse engineering, and re-engineering and may help to overcome some of the problems mentioned above. Strangely enough, there is surprisingly little overlap between the communities for reverse engineering and software visualization in terms of people despite of the large overlap in terms of topics. It is high time for our communities to team up since our common goal is to help programmers understand programs.

References

[Bauhaus, 2001] Bauhaus, http://www.bauhaus-stuttgart.de.

[Chikofsky & Cross, 1990] Chikofsky, E.J.; Cross II, J. H.: Reverse Engineering and Design Recovery. IEEE Software, pp. 13-17, January, 1990.

[Graphlet] Graphlet, http://www.graphlet.de.

[GraphViz] GraphViz, http://www.research.att.com/sw/tools/graphviz

[GXL] GXL, http://www.gupro.de/GXL. See also the paper by Andreas Winter et al. in these proceedings.

[GraphEd] GraphEd, http://www.uni-passau.de/GraphEd

[IEEE-Std-1471-2000] IEEE-Std-1471-2000: Recommended Practice for Architectural Description of Software-Intensive Systems, IEEE, 2000.

[Koschke, 2000] Koschke, R.: Atomic Architectural Component Recovery for Program Understanding and Evolution. Dissertation, Institute of Computer Science, University of Stuttgart, Germany, 2000.

[Koschke, 2001] Koschke, R.: Survey on Software Visualization for Software Maintenance, Re-Engineering, and Reverse Engineering, http://www.informatik.uni-stuttgart.de/ifi/ps/rainer/softviz

[Müller, 1992] Müller, H.; Wong, K.; Tilley, S.: A Reverse Engineering Environment Spatial and Visual Software Interconnection Models. Proc. of ACM SIGSOFT Symposium Software Development Environments, pp. 88-98, December, 1992.

[Roman & Cox, 1992] Roman, G.-C.; Cox, K. C.: Program Visualization: The Art of Mapping Programs to Pictures. Proc. of the International Conference on Software Engineering, Association of Computing Machinery, 1992.

[VCG] VCG, http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html