Manipulating and Documenting Software Structures Using SHriMP Views[†]

Margaret-Anne D. Storey

School of Computing Science
Simon Fraser University
Burnaby BC
Canada V5A 1S6
mstorey@csr.uvic.ca

Hausi A. Müller

Department of Computer Science
University of Victoria
Victoria, BC
Canada V8W 3P6
hausi@csr.uvic.ca

Abstract

An effective approach to program understanding involves browsing, exploring, and creating views that document software structures at different levels of abstraction. While exploring the myriad of relationships in a multi-million line legacy system, one can easily loose context. One approach to alleviate this problem is to visualize these structures using fisheye techniques.

This paper introduces Simple Hierarchical Multi-Perspective views (SHriMPs). The SHriMP visualization technique has been incorporated into the Rigi reverse engineering system. This greatly enhances Rigi's capabilities for documenting design patterns and architectural diagrams that span multiple levels of abstraction. The applicability and usefulness of SHriMPs is illustrated with selected program understanding tasks.

Keywords: program understanding, reverse engineering, re-engineering, software visualization, fisheye views.

1 Introduction

"Clutter and confusion are failures of design, not attributes of information."
Edward R. Tufte, Envisioning Information.

Effectively presenting large amounts of information in any form is challenging. Although the computer screen is relatively small, it is easy to fill it with so much information and detail that it completely overwhelms the user. It is not the amount of information that is relevant, but rather how it is displayed[14]. Careful consideration must therefore be given on how to present information so that it can be used effectively. A crucial step in this process, is determining the purpose of the visualization. This problem is particularly acute in the process of understanding software systems using reverse engineering tools.

The visualization and user interface communities have suggested many approaches for visualizing large information spaces. Approaches based on the fisheye lens paradigm seem well suited to the task of visualizing software. These techniques allow users to create views that span different levels of abstractions. For example, a high-level architectural diagram might include detailed information at strategic points to highlight pertinent information or to illustrate a bottleneck. Architectural styles[6], or design patterns[5] often include entities at various levels of detail. However, usually information spaces of this kind are modeled as graphs and displayed using a set of tiled windows. It is easy to loose context because the relationships among windows are typically implicit.

This paper describes techniques for visualizing software structures modeled as nested graphs, using fisheye views. Nested graphs are used for visualizing the structure and organization of the software. Nodes represent artifacts in the software, such as functions or data variables. Arcs represent dependencies among these artifacts, such as call dependencies. Composite nodes correspond to subsystems in the software.

[†]This work was supported in part by the British Columbia Advanced Systems Institute, the IBM Software Solutions Toronto Laboratory Centre for Advanced Studies, the IRIS Federal Centres of Excellence, the Natural Sciences and Engineering Research Council of Canada, the Science Council of British Columbia, the University of Victoria and Simon Fraser University.

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE

Composite arcs represent a collection of dependencies. Composite nodes may contain other composite nodes and arcs as well as atomic nodes and arcs. This nesting feature of nodes communicates the hierarchical structure of the software (e.g. subsystem or class hierarchies). Nested graphs offer assistance in a reengineering phase of software maintenance when multiple levels of abstraction need to be visualized concurrently.

The fisheye views emphasize detail of current interest within the context of the overall software structure. They provide an alternative to scrolling through graphs that are too large to be displayed in their entirety on the screen. A user browses the graph by selectively enlarging nodes within an area of interest while simultaneously shrinking the rest of the graph. A software engineer can more easily identify structures in the software by enlarging sets of nodes which may not be adjacent in the graph. In addition, the source code of a function or data type may be displayed by zooming the representative node. This provides a mechanism for a software maintainer to seamlessly switch between the implementation and the documentation of a system.

The Rigi reverse engineering system is designed to analyze and summarize the structure of large software systems. It is intended to document the structure of multi-million line legacy software systems[15]. While exploring the myriad of relationships in a multi-million line legacy system, one can easily loose context. One approach to alleviate this problem is to visualize these structures using fisheye techniques. The SHriMP (Simple Hierarchical Multi-Perspective) visualization technique presented in this paper has been incorporated into the Rigi reverse engineering system. This greatly enhances Rigi's capabilities for identifying and documenting design patterns and architectural diagrams that span multiple levels of abstraction.

Section 2 provides background on Rigi and reverse engineering. Section 3 describes several deficiencies encountered when visualizing and navigating large software structures in Rigi. Section 4 describes the SHriMP tool for visualizing large information spaces. Section 5 illustrates the applicability and usefulness of SHriMPs when reverse engineering legacy software systems. Section 6 discusses the benefits of this approach. Section 7 draws some conclusions.

2 Rigi

Rigi is a system for analyzing, visualizing, documenting, and recording the structure of evolving software systems. Software structure refers to a collection of artifacts that software engineers use to form mental models when designing or understanding software systems. Artifacts include software components such as subsystems, procedures, and interfaces; dependencies among components such as client-supplier, composition, inheritance, or control and data-flow relations; and attributes such as component type, interface size, and interconnection strength.

In the Rigi reverse engineering system, artifacts are stored in an underlying repository and manipulated using a graph editor that supports editing, manipulation, annotation, hypertext, and exploration capabilities. Software hierarchies are visualized with overlapping windows and overview windows. A user travels through the hierarchy by opening a window to show the next layer in the hierarchy. An overview window provides context to the individual windows.

Reverse engineering a system involves information extraction and information abstraction. One objective of a reverse engineer is to obtain a mental model of the structure of a subject software system and to communicate this model effectively. This process can be automated to a certain extent but the perceptual abilities and domain knowledge of the reverse engineer play a central role.

Rigi is end-user programmable through the RCL (Rigi Command Language) which is based on the Tcl/Tk scripting language[10]. The reverse engineering methodology can be easily adapted and tailored to diverse program understanding scenarios and selected target domains by writing RCL scripts. Moreover, Rigi can easily be integrated with other tools which incorporate the Tcl/Tk language. As a result, extending the user interface with new visualization techniques such as SHriMP, is feasible.

3 Deficiencies with current approach

Visualization tasks can be divided into two categories corresponding to the reverse engineering and reengineering phases of software maintenance. Tasks performed by the reverse engineer when composing a representation of a mental model of the software differ significantly from those of a maintainer or project manager browsing such a representation.

The reverse engineering phase is one of discovery where a reverse engineer uses visualization techniques to facilitate the identification of candidate subsystems and to assist in the visualization of structures and patterns in the graph. For larger graphs consisting of thousands of nodes and arcs, the ability to inspect smaller groups of nodes and arcs in more detail is

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE

needed. A scrollable window can be used, but only one portion of the graph is visible at any one time. Ideally, the reverse engineer should be able to focus on parts of the system without losing sight of the whole.

Other users, in contrast to the reverse engineer, may merely wish to browse and customize the software hierarchy created by the reverse engineer. For large software systems, it is preferable to obtain an understanding of the overall architecture of the software before proceeding in a top-down fashion to the lower-level details of the software[15]. When trying to understand smaller substructures, it is desirable to retain sight of the overall architecture and to see where the module under investigation is with respect to the rest of the software. In Rigi, hierarchies are visualized with overlapping windows and overview windows. The problem with this approach is that the user is forced to mentally integrate two views.

For larger systems, the hierarchy may be very deep and many windows may be opened to expose the desired information. The user has to manage these windows by tediously resizing them to keep pertinent information visible, and closing them when they are no longer useful. Windows consume screen space and it is too easy for the user to become disoriented as they open further windows. Users of hypertext systems encounter similar problems[13].

During the reverse engineering and reengineering phases, the source code often needs to be inspected in detail. Currently, source code relating to a particular atomic node is displayed in a text editor window. Better visual links between the source code and documentation describing the architecture of the software may assist in program understanding.

Rigi is a sophisticated visualization tool for navigating and manipulating software structure. However, more sophisticated methods are required for visualizing software structures in large legacy systems. For the purposes of program understanding, users must be able to see micro and macro views of the program: they must be able to see the architecture of the program as well as smaller parts of the program in detail, right down to the code level. While looking at smaller portions, the *big picture* should also be maintained. With these requirements in mind, the next two sections discuss an alternative display method which directly addresses these issues.

4 SHriMP Views

SHriMP, a tool for visualizing large graphs, uses the nested graph formalism and a fisheye view algorithm for manipulating large graphs. A basic incentive for writing this tool is to provide a mechanism for visualizing detail of a large information space and at the same time to provide contextual cues concerning its context. When visualizing any large information space, it is necessary to be able to create different views of the information where each one provides a different perspective. SHriMP goes one step further by providing a mechanism to create views that can show multiple perspectives concurrently.

SHriMP is implemented in the Tcl/Tk language and is currently a library that can easily be integrated into systems that have the Tcl/Tk language available in it. The following subsections provide some background on the nested graph formalism and the fisheye view paradigm used by it.

4.1 Nested Graphs

Nested graphs were first introduced by David Harel in 1988[7]. Nested graphs, in addition to nodes and arcs, contain *composite nodes* which are used for denoting set inclusion. The containment or nesting feature of composite nodes implicitly communicates the parent-child relationships in a hierarchy. In SHriMP a non-leaf node is *open* when its children are visible and *closed* when its children are hidden from view.

Due to limited screen space, nodes and composite nodes need to be resized as information needs change. The following describes an automatic strategy to zoom (enlarge or shrink) nodes which will assist in managing the screen space available.

4.2 Fisheye Views

Visualizing large information spaces is a focus for current research. Commonly large knowledge bases are represented using graphs. However, manipulating large graphs on a small screen can be very problematic. Because of this, various methods have been proposed for displaying and manipulating large graphs.

One approach partitions the graph into pieces, and then displays one piece at a time in a separate window. This was the original approach taken by Rigi, see figure 2. However, context is lost as detail is increased. Another approach makes the entire drawing of the graph smaller, thus preserving context, but the smaller details become difficult to see as the scale is reduced. A combination view can be given by providing context in one window and detail in another but this requires that the user mentally integrate the two; not always an easy task.

Techniques have been developed to view and navigate detailed information while providing the user

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE

with important contextual cues[1]. Fisheye views, an approach proposed by Furnas in 1986[4], provides context and detail in one view. This display method is based on the fisheye lens metaphor where objects in the center of the view are magnified and objects further from the center are reduced in size. In Furnas' formulation, each point in the structure is assigned a *priority* that is calculated using a *degree of interest (DOI)* function. Objects with a priority below a certain threshold are filtered from the view.

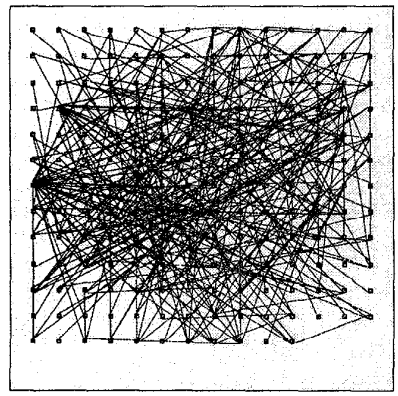
In order to deemphasize information of lower interest, several variations on this theme have been developed that use size, position and colour in addition to filtering. For example, SemNet uses three-dimensional point perspective that display close objects larger than objects further away[3]. Treemaps are used to display hierarchies by representing each object as a rectangle, where children are drawn inside their parents. The size of each rectangle is determined by the weight assigned to it by the user, with the constraint that the weight is greater than or equal to the sum of the weights of its children.

Graphical Fisheye Views, a technique developed by Sarkar and Brown, magnify points of greater interest and correspondingly demagnify vertices of lower interest by distorting the space surrounding the focal point[11]. The continuous zoom algorithm, suitable for interactively displaying hierarchically-organized, two-dimensional networks[2], allows users to view and navigate nested graphs by expanding and shrinking nodes. A survey of these approaches and others such as Perspective Wall and Cone Trees are described in [9].

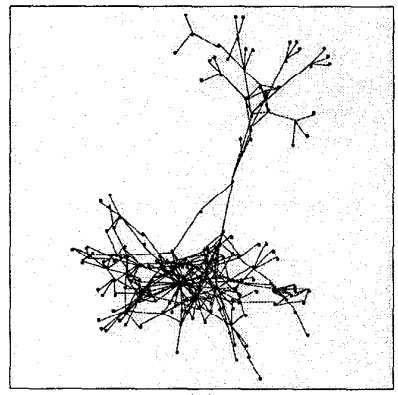
4.3 SHriMP Fisheye View Algorithm

The fisheye algorithm used by SHriMP has several nice features as follows. The zooming technique is highly interactive, even for very large graphs. When one node is enlarged, the other nodes smoothly decrease in size to make room for the selected node, similarly to the continuous zoom algorithm[2]. The zoom-out operation is the inverse operation of the zoom-in operation. Therefore different areas of the graph may be inspected without permanently altering the layout of the graph. A user may zoom multiple focal points and focal areas in the graph.

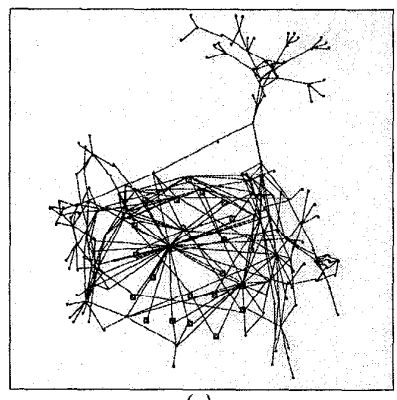
Many fisheye algorithms, such as Brown and Sarkar's, are based on distorting the area surrounding the focal point(s). For visualizations of many information spaces, there is no notion of geometric distance. Nodes that are close to the focal point, are no more important than nodes far away. The SHriMP fisheye



(a)



(b)



(c)

Figure 1: (a) The flat graph representative of the source code for the ray tracer program. (b) The ray tracer graph, after applying the spring layout algorithm. Note the very busy area in the center of the graph. (c) The area in the ray tracer graph is enlarged to show more detail. A single node appears to be the cause of the majority of this complexity.

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE

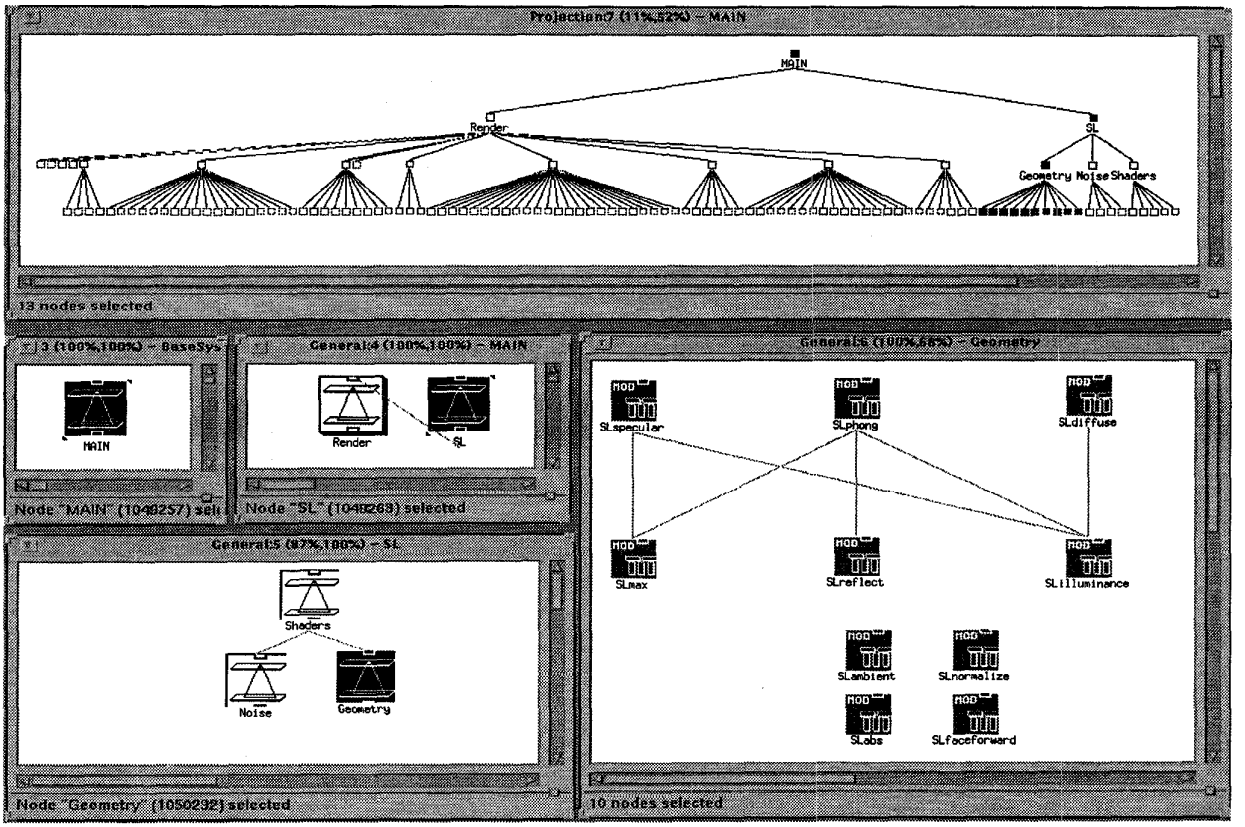


Figure 2: The top window in this screendump shows the overview of the main subsystem in a Ray Tracer program. The arcs in the overview window are level arcs, representing the parent-child relationships between nodes. The MAIN subsystem when opened, displays a window with its two children nodes, Render and SL. The SL subsystem node has also been opened, and its children are displayed in the bottom left window. Finally the Geometry subsystem node is opened in the bottom right window. Nodes highlighted in the overview window are those nodes which are visible in the bottom windows.

algorithm uniformly resizes nodes when more or less screen space is requested.

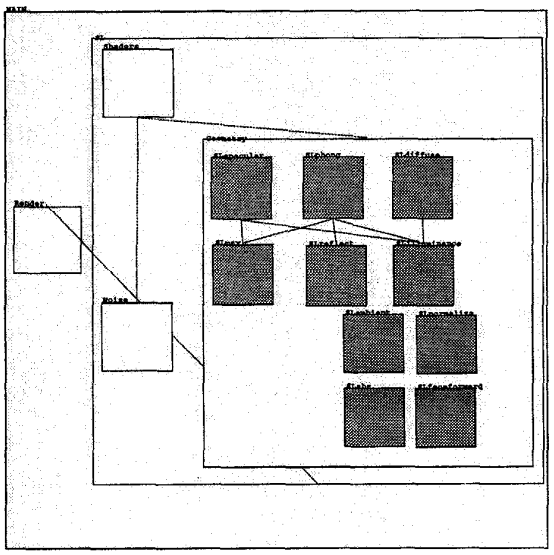
The SHriMP algorithm is flexible in its distortion technique. For a grid or tree layout, nodes that are parallel remain parallel in the distorted view. However, in other layouts, where nodes adjacencies are important, the proximity of nodes is maintained. This algorithm is fully described in [12].

The next section presents two examples where SHriMP has been integrated in the Rigi system and how it is used to visualize and software navigate structures.

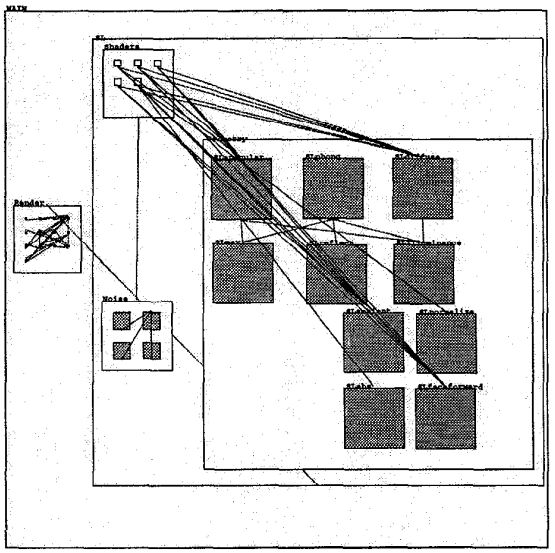
5 Documenting Software Structures using SHriMP Views

This section presents two examples where SHriMP is used to visualize software graphs created by Rigi. Since Rigi is end-user programmable, it is easy to integrate the visualization techniques available in SHriMP with those in Rigi. The usefulness of this approach is demonstrated with a variety of programming tasks applied to two systems.

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE



(a)



(b)

Figure 3: (a) The Geometry subsystem is shown in detail and in context with the rest of the hierarchy. The information displayed here is roughly the same as the information portrayed in Figure 2. (b) Several subsystems have been opened to show more detail. In addition, the composite arc between the Shaders and Geometry subsystems has been opened to show the lower level arcs that it represents.

5.1 The Ray Tracer

The Ray Tracer is a graphics program consisting of approximately thirty modules. This program was written in C following structured programming techniques. Figure 1(a) shows a SHriMP view of the flat graph of the artifacts and dependencies extracted by the Rigi C parser. A spring layout algorithm has been applied to the graph in part (b). This algorithm places highly connected nodes closer together. There is a complex area in the center of the graph that has been magnified using the SHRIMP fisheye view algorithm in Figure 1(c).

The magnification of this area exposes that a single node is the cause of much of this complexity. This node represents a print error routine that is called by many functions. Since an error routine does not provide very much information when trying to understand the structure of the system, the reverse engineer may choose to hide this node to reduce the complexity of this region.

In the following figures, the flat graph shown in Figure 1 has been reverse engineered using the techniques described in [8]. Figure 2 shows the hierarchy imposed on this flat graph. This figure was created using the multiple window approach in Rigi and its purpose is to show detail in the Geometry subsystem and the context of this subsystem with respect to its hierarchy.

The top window is an overview of the hierarchy rooted at the main subsystem. A separate window is opened to represent each level in the hierarchy. Here, the user has opened the main node, which displays a window labeled main containing the subsystems Render and SL. The SL window contains SL's children: Noise, Geometry and Shaders. Finally, the Geometry subsystem has been opened to display the functions and data types of this subsystem, and the dependencies between them. These windows have been manually resized and positioned.

Figure 3(a) presents a SHriMP view of the same subsystem presented in the previous figure. Figure 3(b) follows from Figure 3(a) where some of the subsystems have been opened to show more detail. In addition, a *composite arc*, which is similar in functionality to a composite node, has been expanded to display the lower level dependencies between the Geometry and Shaders subsystems.

In Figure 4, the SLphong and SLreflect functions in the Geometry subsystem have been magnified so that their source code can be displayed. The code for these functions is stored in separate C files.

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE

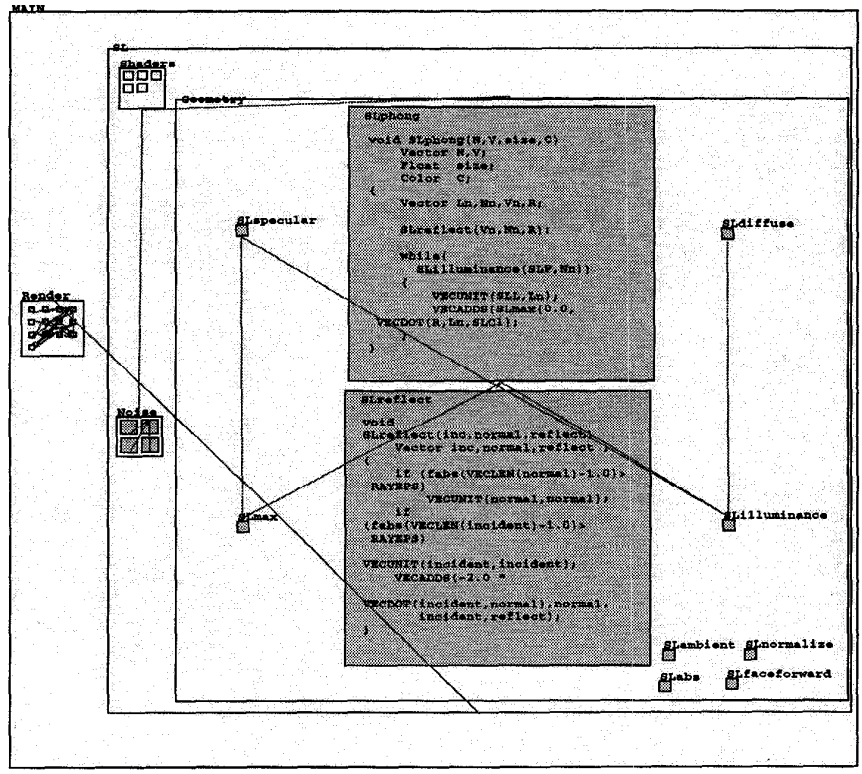


Figure 4: Browsing source code using SHriMP Views

5.2 SQL/DS

The Structured Query Language/Data System is a large, relational database-management system written in PL/AS, a proprietary IBM systems-programming language. SQL/DS consists of about 1,300 compilation units, roughly split into three large systems and several smaller ones. Due to its size and complexity, no individual alone can understand the entire program[15]. Rigi has been applied to this system to ease software maintenance by providing up-to-date, high-level perspectives of the system structure.

The SHriMP views were designed for the visualization of large graphs and are therefore ideal when manipulating and documenting large system structures. Figure 5(a) shows the flat graph of a subsystem in the SQL/DS program using SHriMP. This graph contains 691 nodes and 2917 arcs. A spring layout algorithm has been applied to this graph, and several groups of nodes on the fringe of the graph are easily identifiable as possible candidates for subsystems.

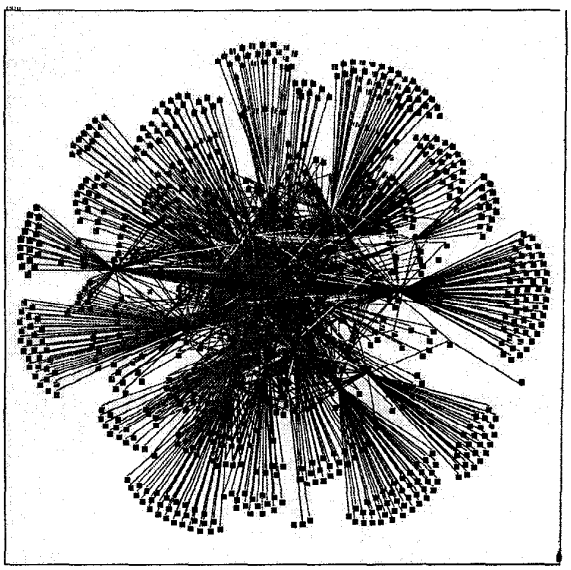
Figure 5(b) shows the result of using SHriMP to select and zoom nodes in the forward dependency tree of the ARIXI20 module. This set of nodes is a good subsystem candidate since each of them only call the ARIXI20 module and no other module. This structure has been emphasized by uniformly magnifying the nodes selected. The nodes are enlarged so that their labels are visible.

The next section discusses the advantages and disadvantages observed while using this approach.

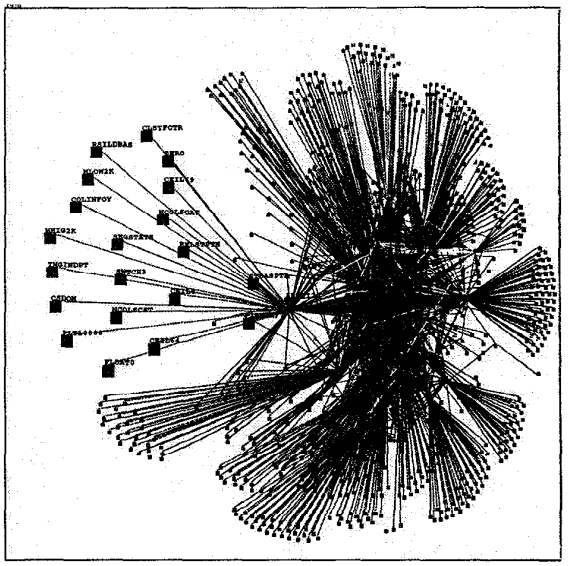
6 Discussion

Both the multiple window technique and the single window fisheye view technique provided by SHriMP have advantages and disadvantages. Using the examples presented in the last section, the two techniques are compared.

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE



(a)



(b)

Figure 5: (a) The spring layout algorithm has been applied to the SQL/DS software system. This algorithm creates clusters of nodes on the fringe of the graph, which are candidates for subsystems. (b) One of the clusters of nodes is enlarged, so that it can be examined in more detail. By enlarging these nodes, the node labels are now visible.

6.1 Detail In Context

For large software systems, understanding the structural aspects of a system's architecture is initially more important than understanding any single component[15]. The nested graph formalism is particularly well suited to showing different levels of abstraction in a system's architecture concurrently. The user incrementally exposes the structure of the software by magnifying subsystems of current interest.

Figure 3(a) of the Ray Tracer example provides detail of the Geometry subsystem while simultaneously displaying the structure of the program. The multiple window approach in Figure 2 depicts the same information, but the user must mentally synthesize an architectural model from information in different windows.

Figure 3(b), demonstrates how additional information concerning the exact nature of the interface between the Shaders and Geometry subsystems can easily be displayed in the SHriMP view. The Shaders subsystem has been opened to show more detail, and a composite arc between the two subsystems has also been opened. The maintainer can integrate this information into a more comprehensive conceptual model.

6.2 Visualizing Software Structures

For larger systems, the SHriMP views are particularly well suited to exposing structures in the software. The zooming mechanism provides an alternative to scrolling by enlarging nodes in a user defined area of interest and concurrently deemphasizing, but not hiding, the remainder of the graph. By zooming on different portions of a large graph, a reverse engineer can quickly identify highly connected nodes, candidate subsystems and other important features.

Alternatively, a user can select a group of nodes which are not necessarily adjacent in the graph, and then zoom these nodes for further scrutiny. Figure 5(b) shows the result of selecting and zooming the nodes in the call forward dependency tree of the ARIXI20 module. By enlarging related but distributed sets of nodes, structures in the graph, such as design patterns, can be emphasized without adversely affecting the general layout of the graph.

By concurrently zooming multiple structures, a software maintainer can see their relative locations in the overall structure, examine their similarities and differences, and visualize any dependencies between them.

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE

6.3 Visualizing Source Code

For software maintainers, an understanding of the architecture is often a prerequisite to understanding the code of the modules or functions. The fisheye view in SHriMP provides a mechanism for a maintainer to read source code while retaining sight of the software architecture. Figure 4 demonstrates that the source code can become an integral part of the architecture documentation, as opposed to being a separate entity which is normally the case. Feedback indicates that this functionality will increase the maintainer's understanding considerably.

6.4 Navigating Software Hierarchies

As with any large information space, the navigation of large software systems is non-trivial. In the multiple window approach, a user travels through the hierarchy by opening windows as they move from one level of abstraction to the next. It is not unusual for users to become "lost" as they move deeper in the hierarchy. The SHriMP view technique provides better contextual cues for the visualizer as they navigate through the hierarchy. All steps in the path traveled are visible, in the form of the nested nodes. A user can elect to return to any subsystem in the branch traveled, and elide the information contained in that subsystem. By using the nested graph formalism in a single fisheye view, manual operations to open, close, resize and reposition windows are performed by the fisheye view algorithm automatically.

However, the multiple, overlapping window approach originally provided by Rigi may be the desired approach in certain situations. For example, in a very large project, a maintainer may only be interested in one small part of the system. Using a catch-all SHriMP view may retain unnecessary information about higher levels of abstraction. The Rigi overview window feature which displays containment hierarchies is effective at presenting a tree or dag-like view of a hierarchy. This may be a more familiar visualization of a hierarchy than SHriMP views.

Therefore combinations of both display techniques may be the best approach. For example, a maintainer can open separate windows until the subsystem of current interest is reached, and then use a SHriMP view from then on.

7 Conclusions

This paper has demonstrated how structures of large software systems at various levels of abstraction can be effectively explored and documented using SHriMP views with the programmability and extensibility features in Rigi. SHriMPs help reverse engineers in the discovery phase by allowing them to *see* detailed structures and patterns, but still look at these structures within the context of the overall architecture. The containment or nesting feature of subsystem nodes implicitly communicates the parent-child relationships and readily exposes the structure of the hierarchy. For maintainers and managers wishing to understand the structure of the software, this approach provides the mechanism to visualize the architecture of the system and simultaneously browse the implementation. Architectural styles and patterns spanning several levels of abstractions can be effectively documented. In addition, SHriMP views are also ideally suited for documenting program slicing results.

Early observations indicate that users adopt SHriMP views quickly and easily exploit the relative advantages of this software visualization technique. Further studies will evaluate its effectiveness and compare it to other techniques.

Acknowledgments

The authors would like to thank Bryan Gilbert and James McDaniel for their editing suggestions and comments, and Brian Corrie for assisting in the incorporation of SHriMP views into Rigi.

References

[1] M.M. Burnett, M.J. Baker, C. Bohus, P. Carlson, S.Yang, and P. van Zee. Scaling up visual programming languages. *IEEE Computer, Special Issue on Visual Languages*, 28(3), March 1995.

[2] J. Dill, L. Bartram, A. Ho, and F. Henigman. A continuously variable zoom for navigating large hierarchical networks. In *Proceedings of the 1994 IEEE Conference on Systems, Man and Cybernetics*, 1994.

[3] K.M. Fairchild, S.E. Poltrock, and G.W. Furnas. Semnet: Three-dimensional graphic representations of large knowledge bases. In Raymonde Guindon, editor, *Cognitive Science and its Applications for Human-Computer Interaction*. Lawrence Erlbaum Associates, Publishers, 1988.

[4] G.W. Furnas. Generalized fisheye views. In *Proceedings of ACM CHI'86,* (Boston, MA), pages 16–23, April, 1986.

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE

[5] E. Gamma, R. Helm, R. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[6] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of 17th International Conferences on Software Engineering,* (Seattle, Washington, U.S.A.), pages 179–185, April 23-30, 1995.

[7] D. Harel. On visual formalisms. *Communications of the ACM,* 31(5), May 1988.

[8] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice,* 5(4):181–204, December 1993.

[9] E.G. Noik. A space of presentation emphasis techniques for visualizing graphs. In *Proceedings of Graphics Interface '94,* (Banff, Alberta: 18-20 May 1994), pages 225–233, May 1994.

[10] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.

[11] M. Sarkar and M.H. Brown. Graphical fisheye views. *Communications of the ACM,* 37(12), December, 1994.

[12] M.-A.D. Storey and H.A. Müller. Graph layout adjustment strategies. In *Proceedings of Graph Drawing 1995,* (Passau, Germany, September 20 - 22, 1995). Springer Verlag, 1995. Lecture Notes in Computer Science. *To appear December 1995.*

[13] S.R. Tilley, M.J. Whitney, H.A. Müller, and M.-A.D. Storey. Personalized information structures. In *Proceedings of the 11th Annual International Conference on Systems Documentation (SIGDOC '93),* (Waterloo, Ontario; October 5-8, 1993), pages 325–337. ACM (Order Number 6139330), October 1993.

[14] E.R. Tufte. *Envisioning Information.* Graphics Press, 1990.

[15] K. Wong, S.R. Tilley, H.A. Müller, and M.-A.D. Storey. Structural redocumentation: A case study. *IEEE Software,* 12(1):46–54, January 1995.

Proceedings of the International Conference on Software Maintenance (ICSM '95)
1063-6773 /95 $10.00 © 1995 IEEE