MANIPULATING AND DOCUMENTING SOFTWARE STRUCTURES

Margaret-Anne D. Storey
School of Computing Science,
Simon Fraser University,
Burnaby BC, Canada V5A 1S6
`mstorey@cs.sfu.ca`

Hausi A. Müller, Kenny Wong
Department of Computer Science,
University of Victoria,
Victoria BC, Canada V8W 3P6
{`hausi`,`kenw`}`@csr.uvic.ca`

An effective approach to program understanding involves browsing, exploring, and creating views that document software structures at multiple levels of abstraction. While exploring the many relationships in a multi-million line legacy software system, one can easily lose context. One approach to alleviate this problem is to visualize these structures using fisheye-view techniques. This chapter introduces Simple Hierarchical Multi-Perspective (SHriMP) views. The SHriMP visualization technique has been incorporated into the Rigi reverse engineering system, greatly enhancing its capabilities for documenting software abstractions. The applicability and usefulness of SHriMP views are illustrated with selected software visualization tasks.

1 Introduction

"Clutter and confusion are failures of design, not attributes of information."
Edward R. Tufte, Envisioning Information.[1]

The term software visualization is used to describe many different processes. According to the taxonomy of software visualization by Price *et al.*,[2] this term has been used to describe broad research areas such as algorithm visualization, program visualization, and visual programming. To visually present software artifacts and structure, graphs are particularly suitable. Nodes in the graph typically represent system components, and directed arcs represent dependencies among those components. Nevertheless, as the size of software systems increase, so too do their representations as graphs. Advanced graphics and abstraction techniques are needed to manage the visual complexity of these large graphs.

The Rigi[3] reverse engineering system is designed to extract, navigate, analyze, and document the static structure of large software systems (to aid software maintenance and reengineering activities). 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 containment, function call, and data access; and attributes such as component type and source-code location.

While exploring the many relationships in a multi-million line system, one can easily lose context. Visualization techniques developed for browsing large information spaces in other fields can be applied to the problem of understanding large software systems. The Simple Hierarchical Multi-Perspective (SHriMP) visualization technique presents software structures using fisheye views of nested (inclusion) graphs. This technique provides a mechanism for presenting detail of a large information space while also displaying contextual cues at the same time. The technique has been incorporated into the Rigi reverse engineering system. This chapter describes software visualization using the SHriMP technique and outlines the benefits of applying this technique for understanding large software systems.

Section 2 describes the Rigi reverse engineering system. An overview of the SHriMP technique appears in Section 3. Section 4 illustrates how SHriMP views are used for visualizing software structures. Section 5 discusses the applicability and usefulness of SHriMP views when reverse engineering large software systems. Section 6 draws some conclusions.

2 The Rigi System

Rigi is a reverse engineering system developed to extract, navigate, analyze, and document the structure of evolving software systems. The Rigi system is centered around a language-independent graph editor for presenting software artifacts. The first phase of reverse engineering a subject software system is fully automatic and involves parsing the software and storing the extracted artifacts. Rigi has parsers for several imperative languages, including C and COBOL. This first phase results in a flat resource-flow graph which can be manipulated using the Rigi editor.

The next phase is semi-automatic, where the objective for the reverse engineer is to obtain a mental model of the structure of the software system and then build abstractions on the flat graph to capture this model. To manage the complexity of large software systems, the second phase involves pattern-recognition skills and features subsystem-composition techniques to generate multiple, layered hierarchies of higher-level abstractions.[4] In this discovery

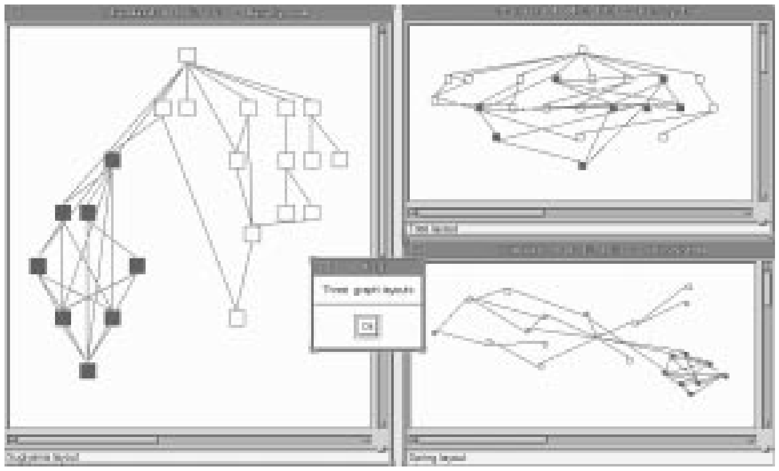


Figure 1: Three graph layouts of the same COBOL program. The left window displays a Sugiyama layout, the top-right window shows a tree layout, and the bottom-right window displays a spring layout.

phase, the reverse engineer employs various visualization aids to recognize patterns, identify candidate subsystems, and understand software structures in the graph. This discovery phase can be partially automated, but the perceptual abilities and domain knowledge of the reverse engineer play a central role.

Various visualization tools are available in the Rigi editor to aid the reverse engineer in discovering and documenting system design information. Some of these tools include selection algorithms, filtering (elision) algorithms, software metrics, and graph layout algorithms.[5,6] For example, in Fig. 1, the resource-flow graph of a small COBOL example is displayed using three graph layouts: spring,[7] Sugiyama,[8] and a tree layout.[9] These three layouts present three different views of the software structure. A single set of nodes is selected (highlighted) in each of the graphs. Note that these nodes are placed close to one another (forming a cluster) in both the Sugiyama and spring layouts. This kind of visual information gives strong evidence to the reverse engineer that such a cluster is a good candidate for a subsystem abstraction.

In addition, the Rigi editor supports editing, manipulation, annotation, hypertext, and exploration capabilities on the graph. The subsystem hierarchies are presented using multiple windows, with overview windows to provide an overall perspective. A user navigates in the hierarchy by opening a window to show the next layer in the hierarchy. An overview window provides context for the individual windows. Figure 2 shows how the hierarchy of a small C program can be visualized and navigated using the Rigi editor.

Rigi is a sophisticated tool for visualizing software structures. Nevertheless, more effective methods are required for visualizing software structures in large legacy systems. While browsing graphs consisting of thousands of nodes and arcs, the user needs to inspect smaller groups of nodes and arcs in more detail. For these larger software systems, it is preferable to obtain an understanding of the overall, high-level structure of the software before proceeding to lower-level details.[10] Ideally, the user should be able to focus on parts of the system without losing sight of the whole. When trying to understand smaller substructures, it is desirable to retain sight of the overall structure and to see how an artifact of interest relates to the rest of the software.

Rigi is end-user programmable through the Rigi Command Language (RCL),[6] which is based on the Tcl/Tk scripting language.[11] As a result, extending the Rigi editor with new visualization techniques, such as SHriMP, is feasible. The SHriMP technique is described in the next section.

3 SHriMP Views

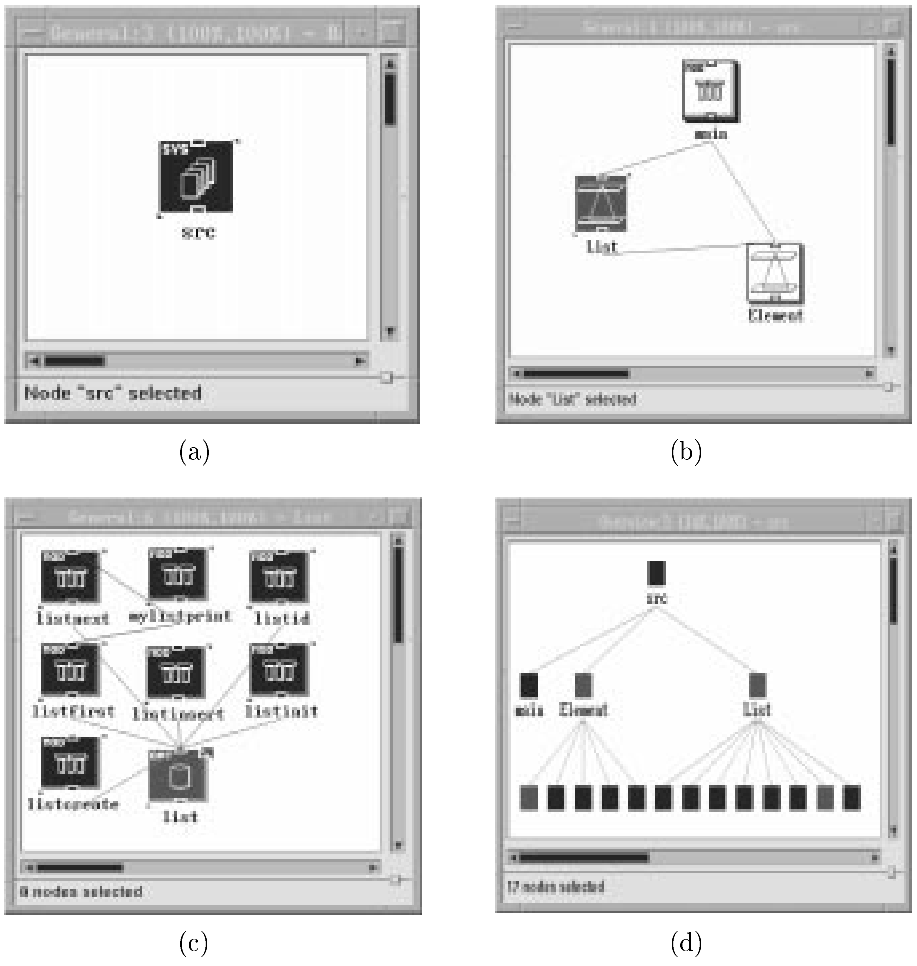
The SHriMP visualization technique uses a nested-graph formalism and a fisheye-view algorithm for manipulating large graphs while providing context and preserving constraints such as orthogonality and proximity. The next four subsections provide some background on nested graphs, dealing with large graphs, fisheye views, and preserving mental maps. Following this material is a brief description of the underlying SHriMP fisheye-view algorithm.

3.1 Nested Graphs

David Harel introduced the concept of a higraph in 1988, a form of nested graph.[12] Nested graphs, in addition to nodes and arcs, contain *composite nodes* that are used to denote set inclusion. The containment or nesting of composite nodes conveys the parent-child relationships in a hierarchy. Figure 3(a) shows a hierarchy that is displayed as a nested graph in Fig. 3(b).

3.2 Dealing with Large Graphs

Visualizing large information spaces is a focus for current research. These information spaces are often represented using large graphs. Since displaying and manipulating large graphs on a small screen is difficult, various approaches have been proposed for dealing with large graphs.



Node "src" selected

Node "List" selected

8 nodes selected

17 nodes selected

Figure 2: (a) This window contains the root node of the program, entitled src. A user may open this node by double clicking on it. (b) This newly opened window contains the children of src: main, List and Element. Arcs in this window are called composite arcs since they relate composite (subsystem) nodes. (c) This window is created when a user opens the List node. The presented nodes are called leaf nodes since they have no children. Arcs in this window represent call and data dependencies in the program. (d) This window is an overview window and provides context for the other windows. It shows the subsystem hierarchy and structure of the program. Arcs between levels in this overview window depict the parent-child relationships among nodes.

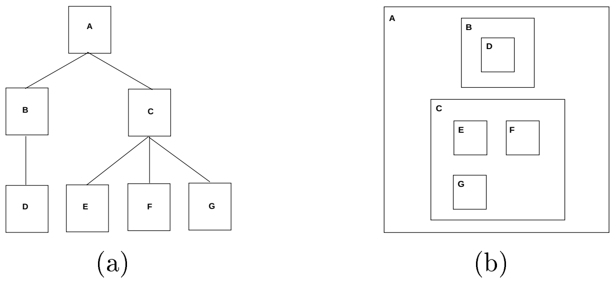


Figure 3: (a) A tree view of a hierarchy. (b) A nested graph view of the same hierarchy.

One approach partitions the graph into pieces, and then displays one piece at a time in separate windows. This *multiple window* approach loses surrounding contextual cues as the focus narrows toward the details. This was the original approach taken by Rigi (see Fig. 2).[13] A second, alternative approach reduces the scale of the entire graph drawing when narrowing the focus, thereby preserving context, but the details become smaller and more difficult to see. In a third approach, a combination view can provide context in one window and detail in another, but this requires that the user mentally integrate the two—not always an easy task. Various techniques have been developed to view and navigate detailed information while providing the user with important contextual cues.[14] The fisheye-view paradigm is one way for accommodating the need to see detail yet maintain contextual cues.

3.3 Fisheye Views

Fisheye views, proposed by Furnas in 1986,[15] provide 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 object in the configuration is assigned a priority that is calculated using a degree-of-interest function. Objects with a priority below a certain threshold are filtered from the view.

To deemphasize information of lesser interest, several variations on this theme have been developed that use size, position, and color in addition to filtering. For example, SemNet uses a three-dimensional point perspective to display nearby objects larger than distant objects.[16] Sarkar and Brown developed a fisheye-view technique that magnifies points of greater interest and correspondingly reduces points of lesser interest by distorting the space surrounding the focal point.[17] A survey of these approaches and others such as treemaps,[18] perspective walls, and cone trees are described by Noik.[19] Related methods include 3DPS[20] and CATGraph.[21]

For visualizations of certain information spaces, however, there is no notion of geometric distance. Nodes that are close to the focal point, are no more important than nodes far away. For these applications, it may be better to *uniformly* reduce the rest of the graph to allow selected regions to increase in size. Although this approach does not strictly follow the fisheye-lens metaphor, it does avoid the problem of causing too much distortion in certain areas of the graph. The continuous zoom algorithm,[22] the biform technique,[23] the rubber sheet orthogonal method,[24] and the bifocal technique[25] uniformly distort the space surrounding the focal points. The first three of these approaches have also been applied to nested graphs. The problem with these techniques, however, is that they are either difficult to implement or their distortion techniques cannot be easily altered to suit different kinds of graph layouts.

3.4 Preserving the Mental Map

Misue *et al.*[26] describe three properties which should be maintained in adjusted layouts to preserve the user's *mental map*: orthogonal ordering, proximity and topology. The orthogonal ordering between nodes is preserved if the horizontal and vertical ordering of nodes is maintained. Proximity is preserved by keeping nodes close in the distorted view if they were close in the original view. The topology is preserved if the distorted view of the graph is a homeomorphism of the original view.

It is impossible to distort a graph without violating one or more of the properties described above. The kind of graph layout and its use should be considered when deciding which properties to preserve. This is particularly important in software visualization where various kinds of graph layouts are used to convey information about different aspects of the software. For instance, in a grid or tree layout, orthogonal ordering of nodes is important to preserve. For other layouts, such as a spring layout to show clusters of highly connected nodes, the proximity relationship among nodes is a more important property.

Graph layouts composed from a combination of layout strategies[27] may also be used to visualize software. For example, the overall structure of the software may be tree-like, with subgraphs laid out in clusters to indicate higher module cohesion. When zooming a node in any part of a graph, the overall layout as well as the subgraph layouts should be maintained.

The fisheye-view algorithm underlying the SHriMP visualization technique is flexible at preserving orthogonality or proximity properties of the graph

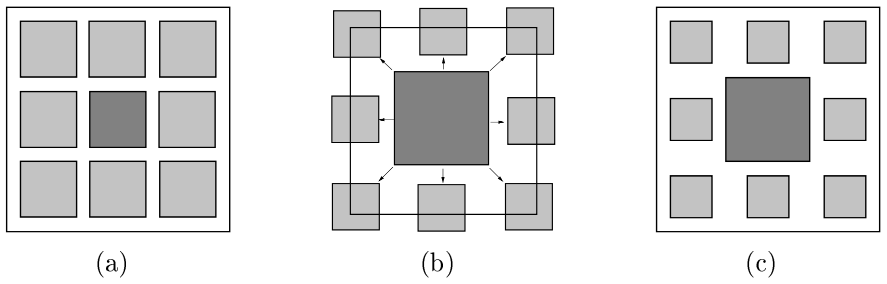


Figure 4: (a) The graph before any scaling is done. A grid of nine nodes is displayed inside a larger box representing the screen. (b) The node of interest (center node) grows by the desired scale factor and pushes its siblings outward, as if there is infinite screen space. (c) Finally, the node and its siblings are scaled to fit inside the screen. This last step is the only step visible to the user of SHriMP, the previous step is shown only to describe the algorithm.

layout. For a grid layout, nodes that are parallel remain parallel in the distorted view. In other layouts, however, where node adjacencies are important, the proximity relationship among nodes is maintained. Tree layouts can also be preserved using a variant of this algorithm.

3.5 The SHriMP Fisheye-View Algorithm

The SHriMP fisheye-view algorithm provides an automatic way to uniformly resize nodes to manage the screen space available.[a] The layout adjustment algorithm for SHriMP fisheye views is elegant in its simplicity.[28] Nodes in the graph uniformly yield screen space to allow the focal node to grow. The focal node grows by pushing its sibling nodes outward so that the node's resized edges maintain some required geometric property with its siblings. Finally, the focal node and its siblings are scaled to fit in the allotted screen space (see Fig. 4). In a nested graph, a node also pushes the boundaries of its parent node outward. The parent in turn pushes its siblings outward in a propagating effect. Finally, all nodes are scaled to fit the screen.

A node grows (shrinks) by pushing (pulling) its sibling nodes outward (inward) along translation vectors. These vectors determine how the sibling nodes are repositioned when a node requires more space. Three methods for setting the magnitude and direction of the vectors are described by Storey and Müller.[28] One layout strategy preserves orthogonal relationships among nodes;

[a]SHriMP was motivated in part by the IGI continuous zoom project at Simon Fraser University.

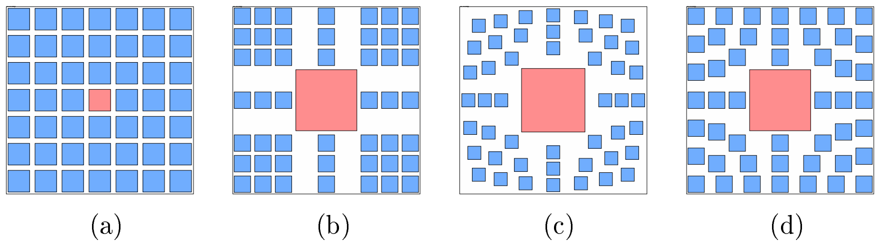


Figure 5: (a) Grid before any scaling is done. (b) The center node is scaled using a layout strategy that preserves orthogonality in the graph. (c) and (d) The center node is scaled using layout strategies that preserve proximity in the graph.

two other strategies, preserve proximity relationships. These three strategies have been used to enlarge the center node of a grid layout in Fig. 5.

The mental map of combination graph layouts can be preserved by applying hybrid strategies. For example, in a tree layout, it is desirable to preserve the orthogonal relationships between levels in the hierarchy, yet keep children close to their parent. Hybrid strategies are straightforward since the method of calculating translation vectors need not be the same for all sibling nodes. In addition, multiple focal points of differing scale factors are also supported.

This flexibility is particularly important when presenting software structures. The next section describes how the SHriMP visualization technique has been incorporated into Rigi to help users at visualizing software structures.

4 Visualizing Software Structures using SHriMP Views

This section describes how the SHriMP visualization technique can be used in the Rigi system. The following provides some examples where the SHriMP technique is used to visualize software graphs created by Rigi. The usefulness of this approach is demonstrated through a variety of software visualization tasks.

4.1 Nested Graphs and Software Hierarchies

In the SHriMP visualization technique, nested graphs are used to present software structures. The nesting of nodes represents the hierarchical structure of the software (e.g., subsystem containment). A small C program, which implements a linked list, is used to demonstrate the SHriMP technique for navigating software hierarchies.

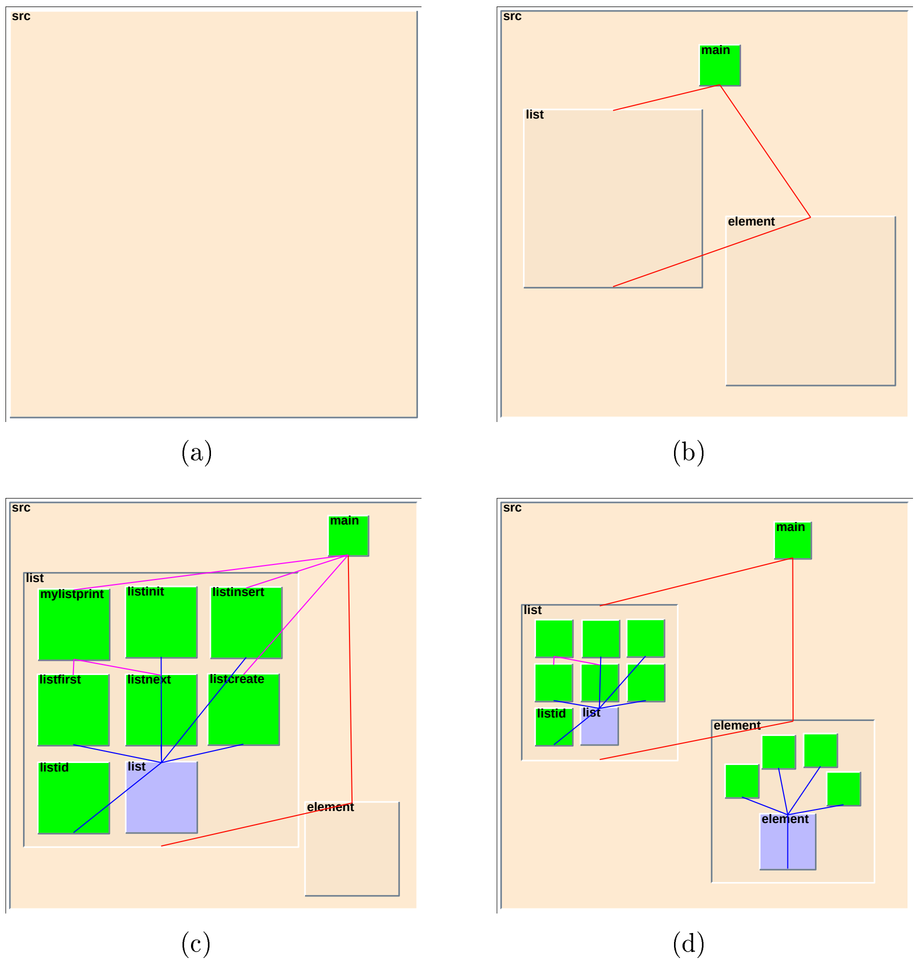


Figure 6: (a) The root node of the program is entitled src. A user may open this parent node to see its children by double clicking on it. (b) The children of src: main, list and element, are displayed within src. (c) The children of list are displayed within list when it is opened. A composite arc between main and list has also been expanded to display lower-level dependencies. (d) The element node has also been opened to display its children. This view now serves a similar function to the Rigi overview window previously shown. The nesting of nodes shows parent-child relationships.

The software hierarchy illustrated in Fig. 6 is equivalent to the hierarchy in Fig. 2. In the SHriMP technique, a user descends through the hierarchy by opening nodes. Nodes and arcs representing the next layer of the hierarchy are displayed inside the opened node, as opposed to being displayed in a separate window. In Fig. 6(a), the src node is displayed as a large box. A user opens this node by double clicking on it. This causes the children of the src node to be displayed inside the src node, as shown in Fig. 6(b). Similarly, in Fig. 6(c), the children of list are displayed inside the list node when it is opened by the user. By opening the element node, Fig. 6(d) shows the same information as the overview window of Fig. 2(d).

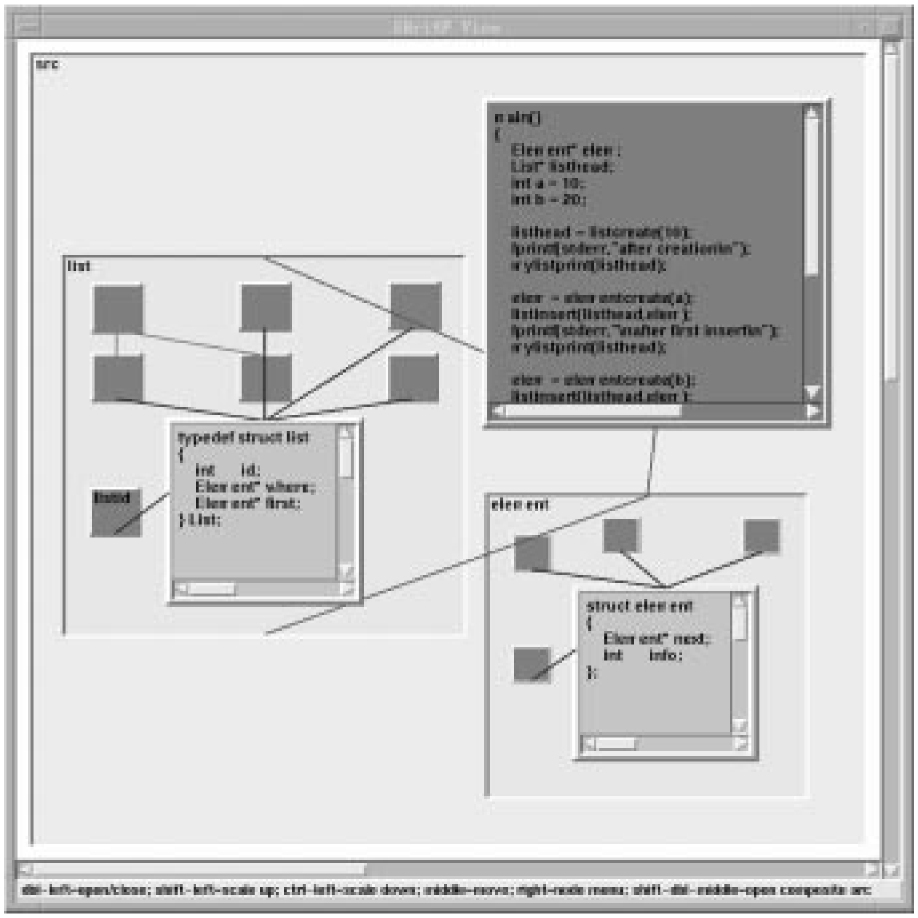
In Fig. 6(c), a *composite arc*, which is similar in functionality to a composite node, has been opened to display the lower-level dependencies between the main and list subsystems. This feature provides details on the particular functions that main calls within the list subsystem.

A leaf node in the hierarchy corresponds to a software artifact extracted by the parser. Using Rigi, a user can select a leaf node and display the file containing the artifact's corresponding source code in a separate text editor. With the SHriMP technique, however, the source code may be displayed directly *inside* the node. Only the relevant section of source code corresponding to the artifact is displayed inside the node, as opposed to displaying the entire file. Figure 7 shows a SHriMP view of the sample program, where three leaf nodes have been opened to display their representative source code.

4.2 Fisheye Views of Software Structures

Figure 8 shows some views of a graphics program consisting of about thirty modules. This program was written in C using a design based on abstract data types. Figure 8(a) shows a grid layout of the initial, flat graph of artifacts and dependencies extracted by the Rigi C parser. A spring layout algorithm has been applied to the graph in Fig. 8(b). This algorithm places highly connected nodes closer together. The complex area in the center of the graph has been magnified using the SHriMP fisheye-view algorithm in Fig. 8(c). The magnification reveals that a single node is causing much of the visual complexity. This node represents an error printing 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 user may choose to hide this node to reduce the complexity of this region. This node has been filtered in Fig. 8(d).

These examples serve to show how SHriMP views can be applied to visualizing software in Rigi. The next section discusses how the SHriMP technique


SHriMP View
src
list
listid
element
dbl-left-open/close; shift-left-scale up; ctrl-left-scale down; middle-move; right-node menu; shift-dbl-middle-open composite arc

```
main()
{
    Element* elem;
    List* listhead;
    int a = 10;
    int b = 20;

    listhead = listcreate(10);
    fprintf(stderr,"after creation\n");
    mylistprint(listhead);

    elem = elementcreate(a);
    listinsert(listhead,elem);
    fprintf(stderr,"\nafter first insert\n");
    mylistprint(listhead);

    elem = elementcreate(b);
    listinsert(listhead,elem);
```

```
typedef struct list
{
    int      id;
    Element* where;
    Element* first;
} List;
```

```
struct element
{
    Element* next;
    int      info;
};
```

Figure 7: The relevant source code for software artifacts represented by leaf nodes is displayed directly inside the nodes in a SHriMP View. This allows the user to browse source code while simultaneously visualizing the location of the code in the software hierarchy.

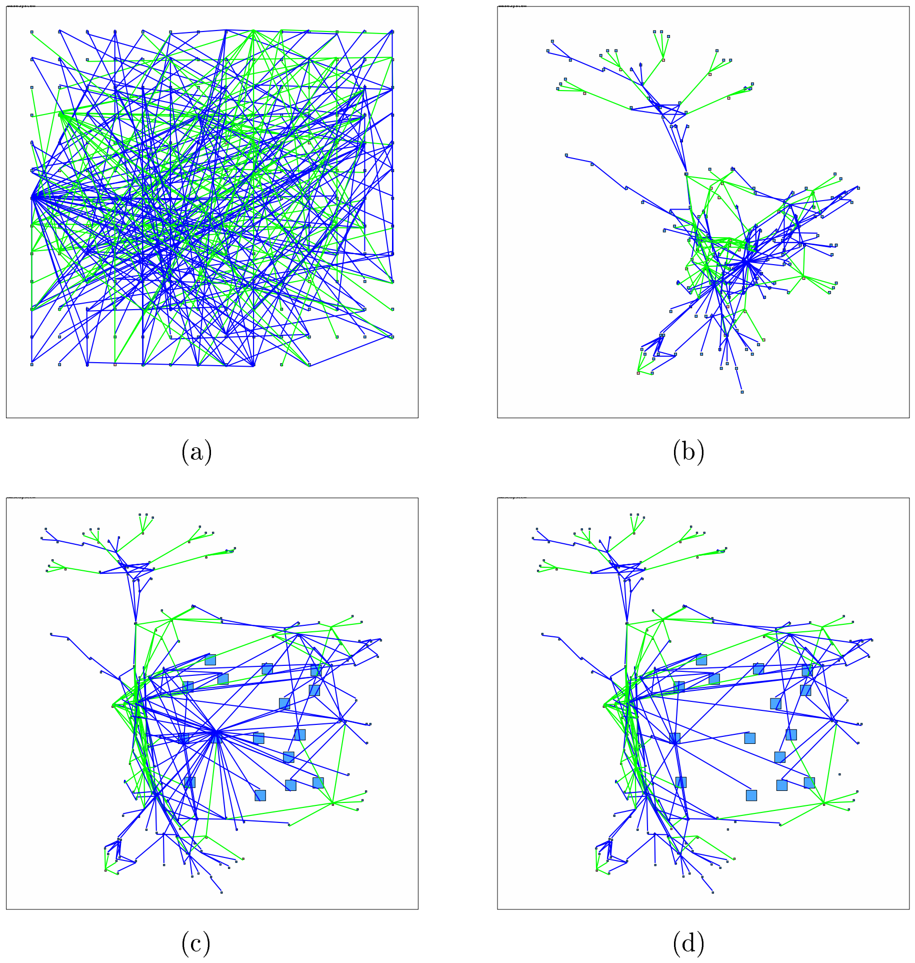


Figure 8: (a) A grid layout of a graphics program written in C. (b) A spring layout of the same graph. (c) The complex area in the center of the graph is magnified using the SHriMP algorithm, exposing a heavily used node. (d) The identified busy node is filtered to reduce the visual complexity.

is helpful for visualizing large legacy software systems, while summarizing the various advantages and disadvantages of this technique.

5 Results

This section discusses the advantages and disadvantages of using the SHriMP visualization technique for manipulating and documenting software structures.

5.1 Detail in Context

For larger software systems, understanding the structural aspects of a system's architecture is initially more important than understanding any single component. The nested graph formalism is particularly suitable for showing several levels of abstraction in a system's architecture at the same time. A single SHriMP view allows the user to focus on smaller details of the software within an overall perspective of the high-level software structure. The user incrementally exposes the structure of the software by opening subsystems of current interest and presenting the children nodes within their parent. This is an improvement over a multiple window approach as the user need not mentally synthesize a mental model from information in different windows.

In addition, the SHriMP technique can present dependencies between subsystems at various levels of abstraction by opening composite arcs to reveal their constituent, lower-level dependencies. Figure 9 shows the structure of a small C program that implements a game. In Fig. 9(a), a high-level view of the major subsystems in this program is shown. One of these subsystems is opened in Fig. 9(b). In Fig. 9(c), further subsystems are opened to show more detail about the program structure. Composite arcs are used to elide details about the lower-level dependencies between the subsystems. In Fig. 9(d), composite arcs have been opened to show lower-level dependencies in the program.

Opening composite arcs provides fine-grained details of the dependencies in the system while the nesting of subsystem nodes concurrently shows a high-level view of the program's overall structure. For larger programs, however, opening many composite arcs can quickly complicate the view. Deciding what to display (or elide) is important for an effective visualization.

5.2 Documenting Software Structures

For larger systems, the SHriMP views are well suited to exposing structures and patterns in the software. The fisheye-view mechanism provides an alternative to scrolling by expanding nodes in a user defined area of interest and

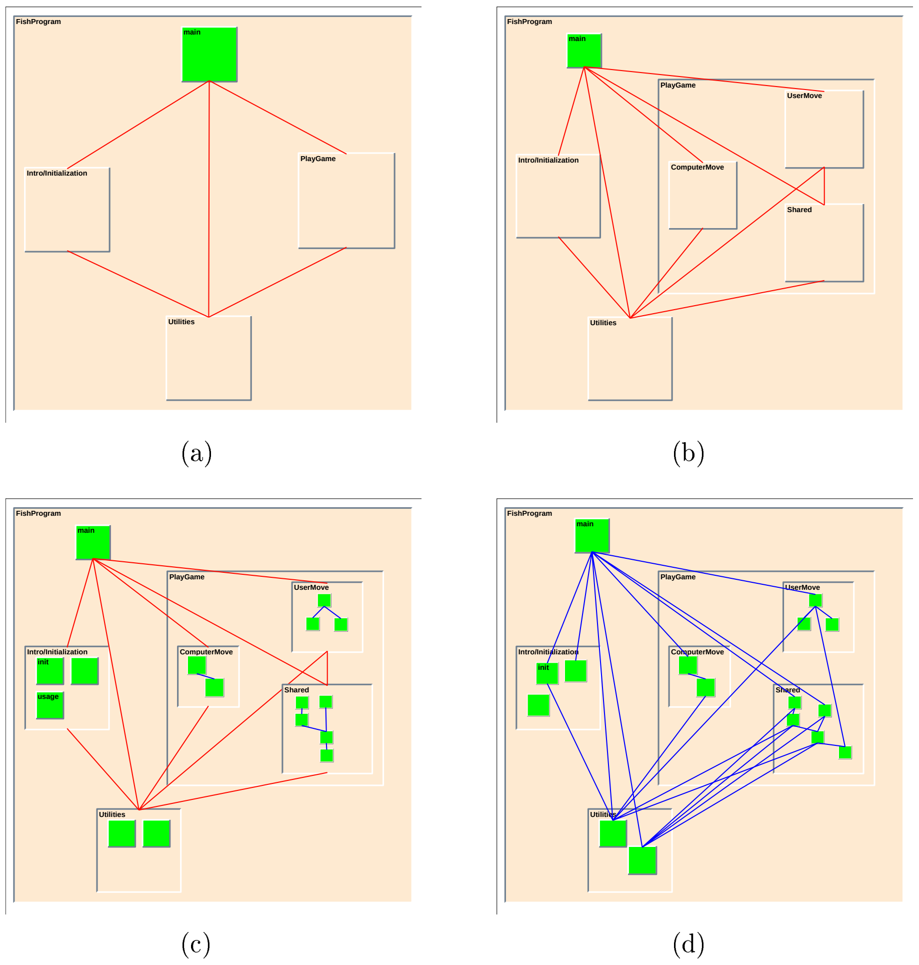
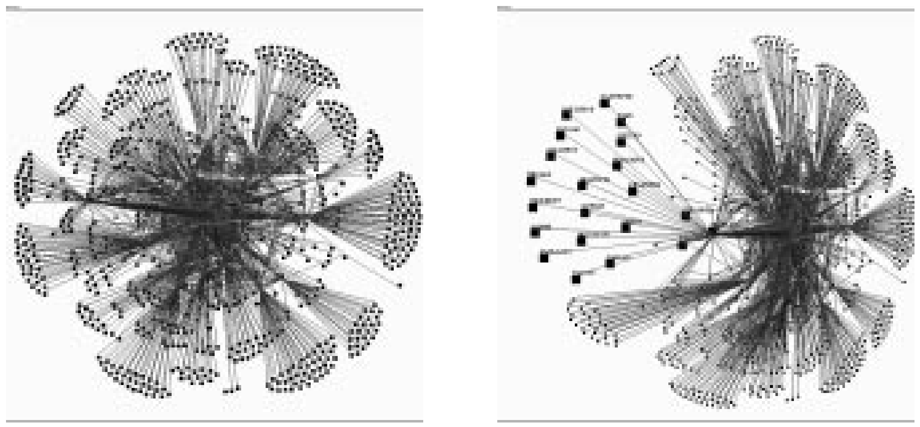


Figure 9: SHriMP views depicting multiple levels of abstraction of a small C program.



(a) (b)

Figure 10: (a) The spring layout algorithm has been applied to the SQL/DS software system. This algorithm helped to expose clusters of nodes on the fringe of the graph, which are candidates for subsystems. (b) One of the clusters of nodes is scaled to show more detail.

concurrently reducing, but not hiding, the remainder of the graph. By zooming in on different portions of a large graph, a user can quickly identify busy nodes, candidate subsystems, and other important features. The filtering of busy nodes can considerably improve the comprehensibility of the graph.

In addition, a user can select a group of nodes which are not necessarily adjacent in the graph, and then scale up these nodes for further study. Figure 10 presents a call graph extracted from SQL/DS, a million-line legacy software system that has evolved over nearly two decades.[10,29] Discovering patterns in such large, complex systems is particularly difficult. Figure 10(b) shows the result of selecting and expanding the nodes in the forward dependency tree of procedure calls from the ARIXI20 module in SQL/DS. By expanding related but distributed sets of nodes, structures in the graph can be emphasized without adversely affecting the general layout of the graph.

By concurrently scaling up several different substructures, a software maintainer can see their relative locations in the overall structure, examine their similarities and differences, and visualize any dependencies among them.

5.3 Browsing Source Code

For software maintainers, an understanding of the high-level structure is often a prerequisite to understanding the code of the modules or functions. With

SHriMP views, the source code becomes an integral part of the structural documentation, as opposed to being a separate entity. Consequently, a software maintainer can seamlessly switch between the implementation and the documentation of a system. Early experiments indicate that this capability will increase the maintainer's understanding considerably. It is not clear, however, if the source code for larger programs can be effectively browsed in such a manner; this issue is currently being investigated.

5.4 Navigating Software Hierarchies

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

Nevertheless, the multiple window approach originally provided by Rigi may be desired in certain situations. For example, in a very large project, a maintainer may only be interested in one small part of the system. A SHriMP view may retain unnecessary information about higher levels of abstraction. Also, the Rigi overview window is effective at presenting a tree-like view of a hierarchy. This may be a more familiar visualization of a hierarchy than SHriMP views for certain users. Therefore, combinations of both SHriMP and traditional Rigi visualization techniques may be the best approach. For example, a software maintainer may choose to open separate Rigi windows until the subsystem of current interest is reached, and then produce a SHriMP view to show the contents of this subsystem.

6 Conclusions

This chapter has demonstrated how structures of large software systems at various levels of abstraction can be effectively explored and documented using SHriMP views. These views help users in the discovery phase of reverse engineering by allowing them to see detailed structures and patterns, but still view these structures within the context of the overall system structure. The nesting feature of subsystem nodes implicitly communicates the parent-child relationships and readily exposes the structure of the subsystem containment hierarchy. For maintainers wishing to understand the structure of the software, this approach provides a mechanism to present the high-level structure of the system and simultaneously browse the implementation.

Early observations show that users quickly adopt SHriMP views and easily exploit the relative advantages of this software visualization technique. The effectiveness of this technique is currently being evaluated through user experiments.

Acknowledgments

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 University of Victoria, and Simon Fraser University.

References

1. E.R. Tufte. *Envisioning Information.* Graphics Press, 1990.
2. B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing*, June 1993.
3. H.A. Müller, K. Wong, and S.R. Tilley. Understanding software systems using reverse engineering technology. In V.S. Alagar and R. Missaoui, editors, *Object-Oriented Technology for Database and Software Systems*, pages 240–252. World Scientific, 1995.
4. 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.
5. H.A. Müller, S.R. Tilley, M.A. Orgun, B.D. Corrie, and N.H. Madhavji. A reverse engineering environment based on spatial and visual software interconnection models. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments (SIGSOFT '92),* (Tyson's Corner, Virginia; December 9-11, 1992), pages 88–98, December 1992. In *ACM Software Engineering Notes*, 17(5).
6. S.R. Tilley, K. Wong, M.-A.D. Storey, and H.A. Müller. Programmable reverse engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4), December 1994.

7. T. Fruchtermann and E. Reingold. Graph drawing by force-directed placement. Technical Report UIUC CDS-R-90-1609, Department of Computer Science, University of Illinois at Urbana-Champaign, 1990.
8. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 11(4):109–125, 1981.
9. E.M. Reingold and J.S. Tilford. Tidier drawing of trees. *IEEE Transactions on Systems, Man, and Cybernetics*, SE-7(2), March 1981.
10. 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.
11. J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, 1994.
12. D. Harel. On visual formalisms. *Communications of the ACM*, 31(5), May 1988.
13. H.A. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering (ICSE '10),* (Raffles City, Singapore; April 11-15, 1988), pages 80–86, April 1988. IEEE Computer Society Press (Order Number 849).
14. 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.
15. G.W. Furnas. Generalized fisheye views. In *Proceedings of ACM CHI'86,* (Boston, MA), pages 16–23, April, 1986.
16. 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.
17. M. Sarkar and M.H. Brown. Graphical fisheye views. *Communications of the ACM*, 37(12), December, 1994.
18. B. Johnson and B. Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings Visualization 91,* (San Diego, California: 22-25 October 1991), pages 284–291, Oct 1991.
19. 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.
20. M. S. T. Carpendale, D. J. Cowperthwaite, and F. D. Fracchia. 3-dimensional pliable surfaces: For effective presentation of visual informa-

tion. In *UIST: Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 217–227, 1995.

21. K. Kaugars, J. Reinfelds, and A. Brazma. A simple algorithm for drawing large graphs on small screens. In *Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*. Springer-Verlag, October 1994.
22. 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.
23. K. Misue and K. Sugiyama. Multi-viewpoint perspective display methods: Formulation and application to compound graphs. In *4th Intl. Conf. on Human-Computer Interaction, Stuttgart, Germany*, volume 1, pages 834–838. Elsevier Science Publishers, September 1991.
24. M. Sarkar, S.S. Snibbe, O.J. Tversky, and S.P. Reiss. Stretching the rubber sheet: A metaphor for viewing large layouts on small screens. In *User Interface Software Technology, 1993*, pages 81–91, November 3-5, 1993.
25. Y.K. Leung, R. Spence, and M.D. Apperley. Applying bifocal displays to topological maps. *International Journal of Human-Computer Interaction*, 7(1):79–98, 1995.
26. K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages and Comput.*, 6(2):183–210, 1995.
27. T.R. Henry and S.E. Hudson. Interactive graph layout. In *UIST, Hilton Head, South Carolina*, pages 55–64, November 11-13, 1991.
28. 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.
29. M.-A. D. Storey and H.A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proceedings of the 1995 International Conference on Software Maintenance (ICSM '95)* (Opio (Nice), France, October 16-20, 1995), 1995.