A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models[†‡]

H.A. Müller S.R. Tilley M.A. Orgun[♣] B.D. Corrie[♢] N.H. Madhavji[♠]

Department of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
Tel: (604) 721-7294, Fax: (604) 721-7292
E-mail: {hausi, stilley}@csr.uvic.ca

Abstract

Reverse engineering is the process of extracting system abstractions and design information out of existing software systems. This information can then be used for subsequent development, maintenance, re-engineering, or reuse purposes. This process involves the identification of software artifacts in a particular subject system, and the aggregation of these artifacts to form more abstract system representations. This paper describes a reverse engineering environment which uses the *spatial* and *visual* information inherent in graphical representations of software systems to form the basis of a software interconnection model. This information is displayed and manipulated by the reverse engineer using an interactive graph editor to build subsystem structures out of software building blocks. The spatial component constitutes information about the relative positions of the meaningful parts of a software structure, whereas the visual component contains information about how a software structure looks. The coexistence of these two representations is critical to the comprehensive appreciation of the generated data, and greatly benefits subsequent analysis, processing, and decision-making.

[†]This work was supported in part by the IRIS Federal Centre of Excellence, the Natural Sciences and Engineering Research Council of Canada, the British Columbia Advanced Systems Institute, Science Council of British Columbia, the University of Victoria, and IBM Canada Ltd.

[‡]The positions expressed herein are solely the views of the authors and are not a reflection of IBM Canada Ltd.'s position.

1 Introduction

Several areas have been identified as critical to improving software maintenance; *recapture* technologies are one of them. Recapture technologies attempt to recover the original design in an existing software system by using reverse engineering and various program-understanding tools. As today's software ages, the task of maintaining it becomes both more complex and more expensive. Software engineers must spend an inordinate amount of time attempting to create an abstract representation of the system's high-level architecture by exploring its low-level source code. With software maintenance routinely consuming upwards of 50% of a software product's life-cycle and overall budget [1], there is a very tangible advantage to improving it: according to [2], "By 1995, a 20% improvement in software productivity will be worth $90 billion worldwide."

Software maintenance cannot be performed without a complete understanding of a subject system. This is a difficult task for software that is 10-25 years old and generally in poor condition. Contributing factors include the lack of accurate documentation, the sheer size of the system, the unstructured programming methods used in the system's design, the fact that the original system designers and programmers are no longer available, and the complication that the software has been changed several times since its first release, and has thus *evolved* into something different than the original [3]. Through reverse engineering, the overall structure of the subject system can be determined and some of its architectural design information can be recovered. This recovered information can then be used for subsequent development, maintenance, re-engineering, or reuse purposes.

Over the past five years we have been developing *Rigi*,[1] a versatile system and framework for discovering and

[1]Rigi is named after a mountain in central Switzerland.

analyzing the structure of large software systems. In particular, we concentrated on investigating algorithms, methods, techniques, and tools for the composition, analysis, presentation, and representation of subsystem structures. Some of the early work resulted in a graph model for software structures and a graph editor supporting the model [4]. The model can be effectively used at all stages of a software product's life-cycle, in the subsystem decomposition process, modeling software evolution, change analysis and so on.

By software structure we mean a collection of artifacts that software engineers use to form mental models when designing, implementing, integrating, inspecting, or analyzing software systems. Artifacts include software *components* such as procedures, modules, subsystems, and interfaces; *dependencies* among components such as supplier-client, composition, and control-flow relations; and *attributes* such as component type, interface size, and interconnection strength. The artifacts are stored in an underlying graph database, and the graph editor allows users to edit, maintain, and explore the objects stored in the repository.

More recently, our investigations have been focused on methods and algorithms for summarizing software structures by building hierarchies of subsystems [5]. One of the results is a *semi-automatic* reverse engineering method that can serve as a precursor for maintenance and re-engineering applications, as a front-end for conceptual modeling and design recovery tools, and as input to project decision-making processes.

Discovering and building subsystem structures is an art. Our work is based on the premise that an experienced software engineer will always be able to produce a "better" subsystem decomposition than a fully automatic procedure — given sufficient time. However, the human designer needs expert assistance from the programming environment for the tedious and arduous tasks involved in the composition process, and Rigi provides such help.

We have also experimented extensively with user interface issues regarding the effective interaction with and presentation of subsystem structures. It is difficult to convey and communicate the wealth of information generated as a result of reverse-engineering a subject system. This problem is exacerbated by the necessary coexistence of *spatial* and *visual* data.

♣ School of Mathematics, Physics, Computing, and Electronics, Macquarie University, North Ryde, NSW 2109, Australia.

◇ Department of Computer Science, Australian National University, GPO Box 4, Canberra, ACT 2601, Australia. E-mail: bcorrie@cs.anu.edu.au.

♠ School of Computer Science, McGill University, 3480 University St., Montréal, QC, Canada H3A 2A7. E-mail: madhavji@opus.cs.mcgill.ca.

The spatial component constitutes information about the relative positions of the meaningful parts of a software structure (*i.e.*, where things are relative to one another) that allow for the systematic exploration of the software structures (*e.g.*, identification of all the clients or suppliers of a component). Visual graph representations (*i.e.*, rendering of nodes and arcs in various formats in a workstation window) aim to exploit the ability of the human visual system to recognize and appreciate patterns and motifs (*e.g.*, central, fringe, or isolated components).

We have found that an effective interaction with and presentation of *both* kinds of information are vital in the reverse engineering process. This information forms the basis of a software interconnection model: a graphical method, augmented with measurements and program-understanding aids, used to construct subsystem structures out of software building blocks.

Our approach to reverse engineering is summarized in the next section. Operations for the presentation of and interaction with representations of the generated software structures are presented in Section 3. Section 4 introduces our notion of a *view*, a means for effectively manipulating and storing these representations. Section 5 illustrates some of the introduced concepts by presenting a tour through four representative views of a graphics program. The tour outlines how the results of this approach can serve as input for software maintenance, re-engineering, and project management. Section 6 reports on some early findings with our reverse-engineering approach.

2 Reverse engineering approach

The process of *reverse engineering* a subject system involves two distinct phases [6]:

1. The identification of the system's current components and their dependencies.
2. The extraction of system abstractions and design information.

During the process of reverse engineering the subject system is not altered, although additional information about it is generated. In contrast, the process of re-engineering typically consists of a reverse engineering phase, followed by a forward engineering or re-implementation phase which alters the subject system. A survey of several state-of-the-art program understanding techniques is given in [7]. Here we present a brief overview of our approach.

The first phase of the reverse engineering process—the automatic extraction of software artifacts—is language-dependent and essentially involves parsing of the subject system and storing the artifacts in a repository. Full parsing is not normally required, since we are only interested in extracting the pertinent information for the discovery and identification of the structure and properties of the subject system, and for the subsequent analysis. Our repository, GRAS, is a database that is specifically designed to represent graph structures [8]. A graph editor allows the users to edit, maintain, and explore the objects stored in the repository. The parsing system currently supports the programming languages C and COBOL; support for C++ is under development.

The Rigi environment models the underlying software system as a tree with two branches. One branch is automatically constructed by the parsing system, and is not directly manipulated by the reverse engineer. It contains information at the unit and syntactic interconnection levels [9]. The other branch is constructed semi-automatically during the second phase of reverse engineering. It represents logical views of the first branch. As such, it contains information at the semantic interconnection level.

Our approach to the second phase features language-independent subsystem composition algorithms, which generate hierarchies of subsystems [5]. Subsystem composition is the process of constructing composite software components out of building blocks such as variables, procedures, modules, and subsystems, which also involves the computation of exact interfaces for the composed objects. Hierarchical subsystem structures are formed by imposing equivalence relations on the resource-flow graphs of the source code. The relations embody software engineering principles that concern module interactions such as *low coupling* and *strong cohesion* [10].

The generated structures embody *visual* and *spatial* information which serve as organizational axes for the exploration and presentation of the composed subsystem structures. These structures can be augmented with *views*: textual (and potentially hypermedia) annotations that highlight different aspects of the software system under investigation [11]. Our semi-automatic reverse engineering methodology can serve as a precursor for maintenance and re-engineering applications, as a front-end for conceptual modeling and design recovery tools, as a documentation and program-understanding aid for large software systems, and as input to project decision-making processes.

We have also formulated software quality criteria and measures based on exact interfaces and established software engineering principles to evaluate subsystem structures [12, 13, 14]. The quality measures quantify the *encapsulation effect* of individual modules or subsystems, as well as the effectiveness of module or subsystem compositions with respect to *separating concerns*. In other words, we measure the strength or cohesion of subsystems and the thickness of the *firewalls* among them.

Using these subsystem composition methods, which are supported by the graph editor, software structures such as call graphs, module graphs, include file dependency graphs, and directory hierarchies can all be summarized, analyzed, evaluated, and optimized subject to software engineering principles. Being able to retrieve, browse, and trace these structures effectively is a key to system comprehension.

In summary, our reverse engineering approach involves:

1. The extraction of relevant system components and dependencies out of source text, thereby forming resource-flow graphs.
2. The composition of subsystem hierarchies on top of these resource-flow graphs, using an interactive graph editor.
3. The computation of the exact interfaces among the constructed subsystems by analyzing and propagating the dependencies extracted from the source code.
4. The evaluation of re-constructed subsystem structures with respect to established software engineering principles such as *small and few interfaces between subsystems* and *high strength within subsystems*.
5. The capturing of pertinent views for target audiences which can be recalled, inspected, played back, and serve as a basis for further investigations and explorations.

3 Spatial and visual software interconnection models

The process of reverse engineering involves the identification of software artifacts in a particular representation of a subject system via mental pattern recognition on the part of the reverse engineer, and the aggregation of these artifacts to form more abstract system representations. Graphs are segmented into features, which are then pattern-matched against individual recollections of expected structural motifs. The success of this process of system reconstruction depends crucially on the individual's recollection of existing structural knowledge

Spatial	Visual
Identification of strongly connected components	Overviews of component dependencies
Identifying the common clients of a component	Overviews of part-of relationships
Identifying the common suppliers of a component	Projections of subsystems
Exploration of software structures	Identification of central components
Orientation and navigation	Identification of congestion
Identifying the neighborhood of a component	Identification of disconnected components
Change and impact analysis with respect to a component	Quality of modularizations

Figure 1: Applications of spatial and visual data representations

and on his or her ability to recognize its presence in a noisy map.

Theories of cognition suggest that imagery involves both *descriptive* and *depictive* information [15]. In the realm of software structure modeling and analysis, *spatial* and *visual* representations of artifacts seem to be the key to forming mental models of software structures. The reverse engineer exploits both spatial and visual information when identifying components and building abstractions. Software maintainers and project managers, who interpret the results of the reverse engineering process, also use spatial and visual information to understand and analyze subject systems.

The spatial component constitutes information about the relative positions of the meaningful parts of a software structure (*i.e.*, where things are relative to one another). It provides low-level, detailed information concerning the immediate neighborhood of a software artifact in a graphical representation. Spatial information is the graphical analogy of the syntactic interconnection model as proposed by Perry [9]. Borrowing the syntax used in [9] to describe the unit, syntactic, and semantic interconnection models, the spatial interconnection model has as its set of objects software artifacts, and has as its set of relations graphical neighborhood information such as adjacent components, common suppliers, and so on, which support the operations outlined in Figure 1.

$$\mathit{Spatial\ IM} = (\{\mathit{artifacts}\}, \{\mathit{neighborhood\ info}\})$$

The visual component preserves information about how a software structure looks (*e.g.*, size, shape, or density). It provides a high-level view of the system; the *gestalt* of the entire image. Visual information is the graphical analogy of the unit interconnection model. The visual interconnection model has as its set of objects *images* of clusters of software artifacts, and bases its set of relations on information such as the size, shape, and density of the image, which support the operations outlined in Figure 1.

$$\mathit{Visual\ IM} = (\{\mathit{images}\}, \{\mathit{size,\ shape,\ \ldots}\})$$

Taken together, these two graphical representations form a new type of software interconnection model based on spatial and visual imagery. The reverse engineer uses this model, augmented with semantic information provided by documentation or software engineers, and other information provided by the Rigi environment such as measurements of interconnection strength, to compose subsystem structures.

The graph editor supports a variety of operations to manipulate *visual* and *spatial* representations of graph structures. Figure 1 summarizes and contrasts some of these applications. The most important visual representations are *overviews* and *projections*. Structural relations such as function, module, subsystem, and composition dependencies can all be displayed as Rigi graphs. Such graph or subgraph overviews can be depicted in various patterns and arrangements using grouping, scaling, layout, and filtering operations. Overviews at different levels of detail can be obtained by means of projections.

For example, given two high-level subsystems, their function or module dependencies can be generated by simply adjusting the *projection-detail* parameter. Such subsystem overviews can be used to identify structural bottlenecks and anomalies. An example of a structural bottleneck is a component that has a high number of incident arcs. Such bottlenecks can also be identified by selecting those nodes whose number of incident arcs exceeds the *incident threshold*. Fringe or isolated components can be recognized in a similar fashion. Unnecessary structural complexity (*e.g.*, components that are in the wrong module or subsystem) can also be recognized visually.

Graph structures are ideally suited for representing spatial relationships. The Rigi system provides operations and user interface tools to follow paths and explore neighborhoods in graphs and hierarchies. For example, one can identify the set of nodes that are reachable or are impacted by a given component, or one can select all the functions that operate on a particular data type. To investigate the cohesion and coupling properties of subgraphs, Rigi provides a variety of selection operations. For example, one can identify node pairs with *high* or *low coupling* or node pairs that have *common clients* and/or *suppliers*. The set of common clients/suppliers of two nodes, say x and y, includes all those nodes that require/provide resources from/to x *and* y. Again thresholds, which can be adjusted by the user, are used to guide these identification processes.

4 Documenting and analyzing software systems with views

The Rigi system uses *views* to direct the focus on visual data and guide the exploration of spatial data. Such a view represents a particular state and display of a software model. The exact configuration and contents of a set of Rigi windows displayed on a workstation can be named and saved as a view to a system or user directory. Subsequently, individual views or sequences of views can be loaded to explore or analyze particular aspects of a subject software system.

Different views of the same software model can be used to address a variety of target audiences and applications. A view can make a point, highlight pertinent data, show relevant data while hiding immaterial information, sharpen the focus of the reverse engineer, clarify an issue, illustrate a problem, display cross sections of the underlying software model, or simply save the current working environment. Views can be collected into sequences to form related sets of documentation, to represent guided tours for tutorial purposed, to highlight system components that need to be analyzed and understood when performing specific maintenance or reengineering tasks, to summarize change, impact, or performance analyses, to describe critical paths that ought to be followed during testing and integration, or to annotate critical sections with measurements that serve as input to decision-making (for example, project priorities or personnel assignments).

Figure 2 shows an overview of a ray tracing program used to illustrate the Rigi environment in Section 5. Such a view might be used by management personnel to gain an understanding of the overall architecture and subsystem interaction, or by new employees just learning the system. Without such views, these users would be forced to read through many thousands of lines of source code to even begin to understand the system's functionality and architecture. Several views of the same system, at different levels of detail targeted to diverse users and requirements, are documented and depicted in Section 5.

Any number of views can be attached to a particular software model. Views can easily be combined and tailored to form new or alternate views, or manipulated interactively to explore details of spatial and visual information that is captured. Using the View Browser, which simply lists the names of the saved views, sequences of related views can conveniently be played back and forth.

Some operations in Rigi directly affect the underlying software model (*e.g.*, insertions of nodes and arcs), while others only affect the current view (*e.g.*, rendering formats of icons). A Rigi view consists of one or more windows as well as some global state information. The list below describes the information that is saved in order to restore the view completely. This information includes the window and icon formats, the current selection, and the node and arc filters.

View documentation
: A view is identified by means of a name, and may be annotated with some free text to explain the view's purpose.[2] A browser widget, which lists the saved views alphabetically, is used to store and load individual views.

Window format
: The window format information includes the window's contents, type, size, position (in both world and screen space), scroll bar positions, and scale factor with which the data are being displayed.

Icon format
: The format of an icon specifies how it is rendered in a window. In particular, an icon may be displayed with or without its name and with or without its icon type. Icon names and types may be enabled and disabled for all the icons in a window, or for just a subset thereof.

Selection
: A view may include a selection; that is, a set of highlighted icons.

Data shown and hidden
: When incremental loading is used, of those objects only the components and dependencies that

[2] We hope to augment this textual annotation with multimedia in the future.

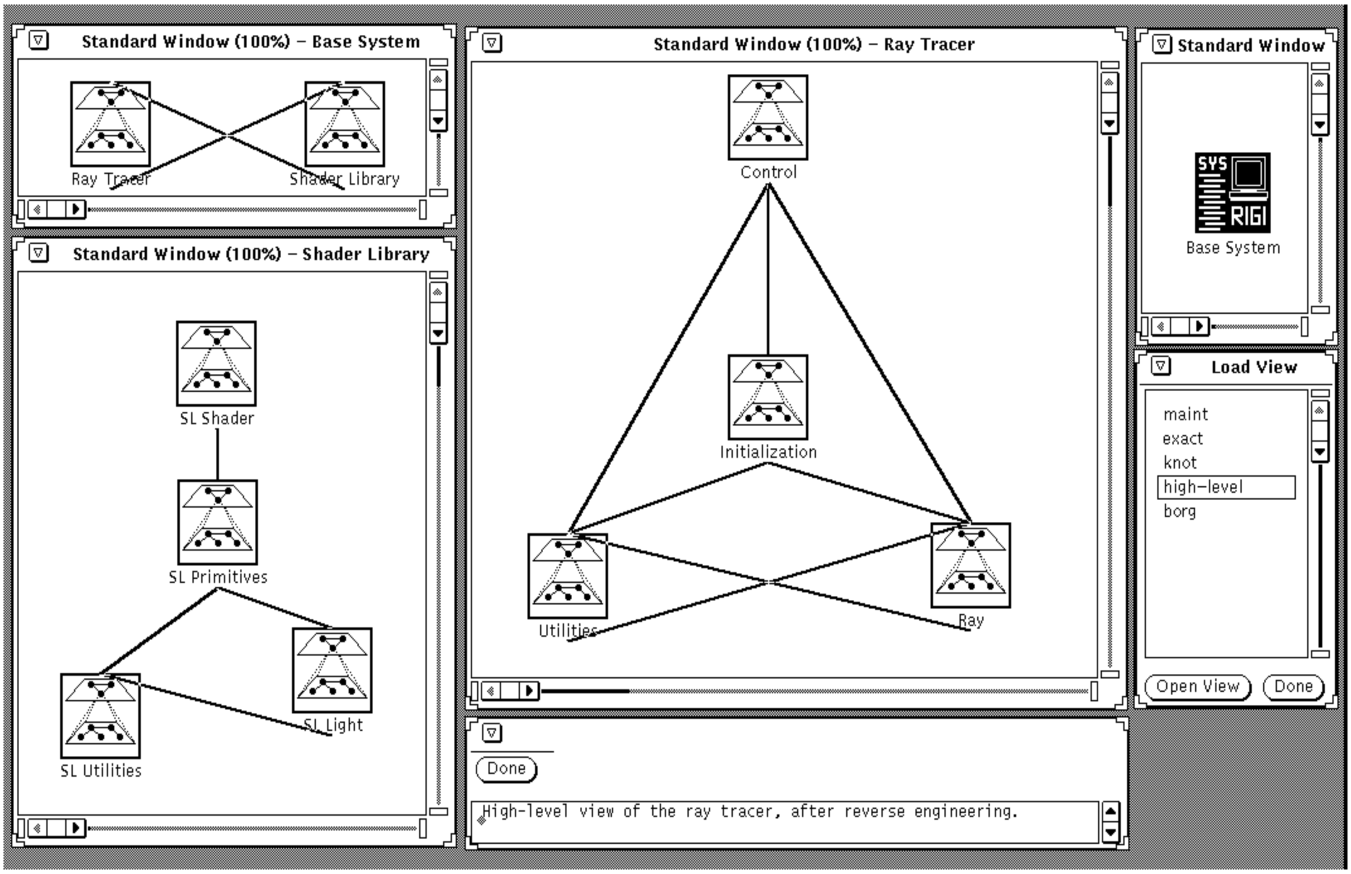

Figure 2: System overview

are loaded from the database into memory are displayed. Moreover, only those components and dependencies whose icon and arc types are enabled (via menu selections) are shown. In addition, selected individual icons and their attached dependencies may be hidden.

5 A tour through Rigi

User interaction with the Rigi system is difficult to portray without the actual implementation of the system at hand. Nevertheless, we attempt to illustrate this interaction by describing four representative views of a subject system. The system under scrutiny is a graphics program; a ray tracer written in C consisting of 30 modules. The views are intended to capture different applications scenarios for this software system, and to illustrate some of the representations and concepts introduced in the preceding sections.

To generate an initial call graph of the ray tracer, we first parse the C source code, which is kept in multiple directories, to extract software artifacts such as data types, functions and dependencies. The call graph of the system is then stored and maintained in the graph database. Using the subsystem identification and composition operations provided by the Rigi editor, subsystem hierarchies are then built on top of the initial call graph, and exact interfaces of the constructed subsystems are computed.

The web depicted in Figure 3 represents the call graph of the ray tracer after preliminary reverse engineering. Such a Gordian knot usually cannot be untied with a single cut as Alexander the Great so cleverly did; however, a sequence of cuts, which separate sets of related nodes and arcs, thereby forming subsystems, will normally do the trick. There are a variety of tools—besides a sword—for organizing this web of software structures. Rigi does not offer a fully automatic procedure but rather a set of operations, which are applied interactively in any given order by a software engineer or reverse engineer, for searching the graph and identifying meaningful subgraphs.

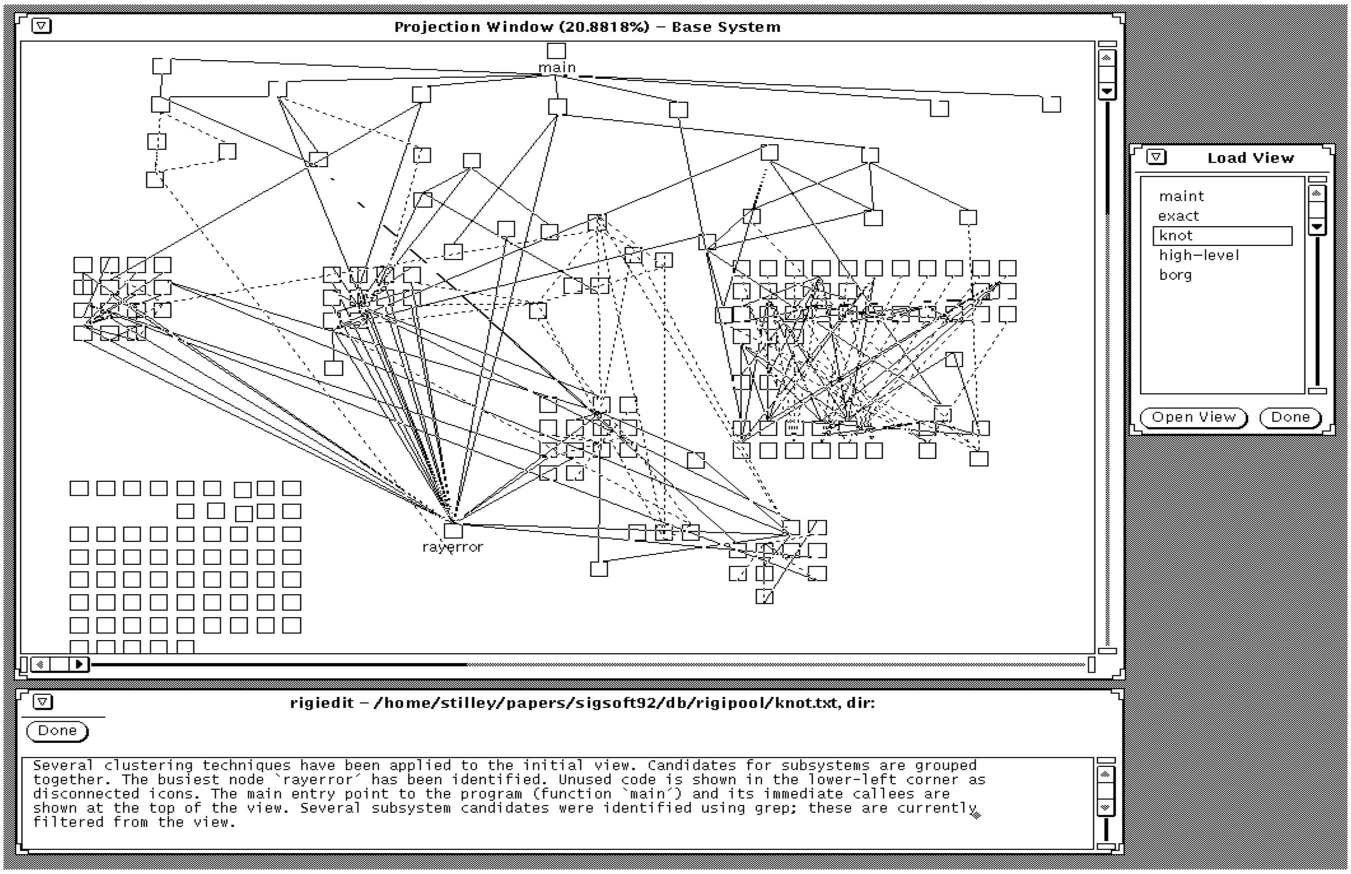

Figure 3: Gordian knot

One possible clustering technique is to search for the main entry point of the program (*e.g.*, function `main`) and then to create a dependency tree of all the routines that are reachable from this node, as depicted in Figure 3. As a side effect, some unused or dead code is typically exposed in the process. This dead code is represented as unattached squares in the lower left-hand corner of Figure 3.

A second tack might be to identify central components; that is, components that are connected to more than, say, n other components (the threshold n can be adjusted by the user). The identified components are then typically used as root nodes for identifying meaningful subtrees. Sets of related nodes are organized as piles and, finally, the piles are collapsed to form subsystems. This process is repeated for the newly created subsystems and their dependencies, thereby forming a hierarchy of subsystems structures. While this process seems tedious—and it is—an experienced reverse engineer can build subsystem structures effectively and reliably.

Figure 2 exhibits the top level of the constructed subsystem hierarchy, a representative overview of the ray tracer. There are numerous strategies in which composition relations can be imposed on the initial call graph. However, we are not interested in just any subsystem composition, but those that expose the overall architecture of the subject system as best as possible, help recover most of the original design decisions, and facilitate system comprehension. The Rigi editor's capabilities such as filtering and searching tools, and parameterized composition operations allow the user to explore a variety of alternative subsystem hierarchies and "what if" scenarios on top of the initial call graph.

The important composition strategies used at the initial stages of the reverse engineering process to construct Figure 2 from Figure 3 are (1) clustering data types with their access functions and methods, (2) identifying and filtering objects that are immaterial for system comprehension such as debugging and error-reporting routines, and (3) identifying and clustering common library routines to form the `Shader Library` subsystem. At the later stages of the reverse engineering process the following composition strategies are used to

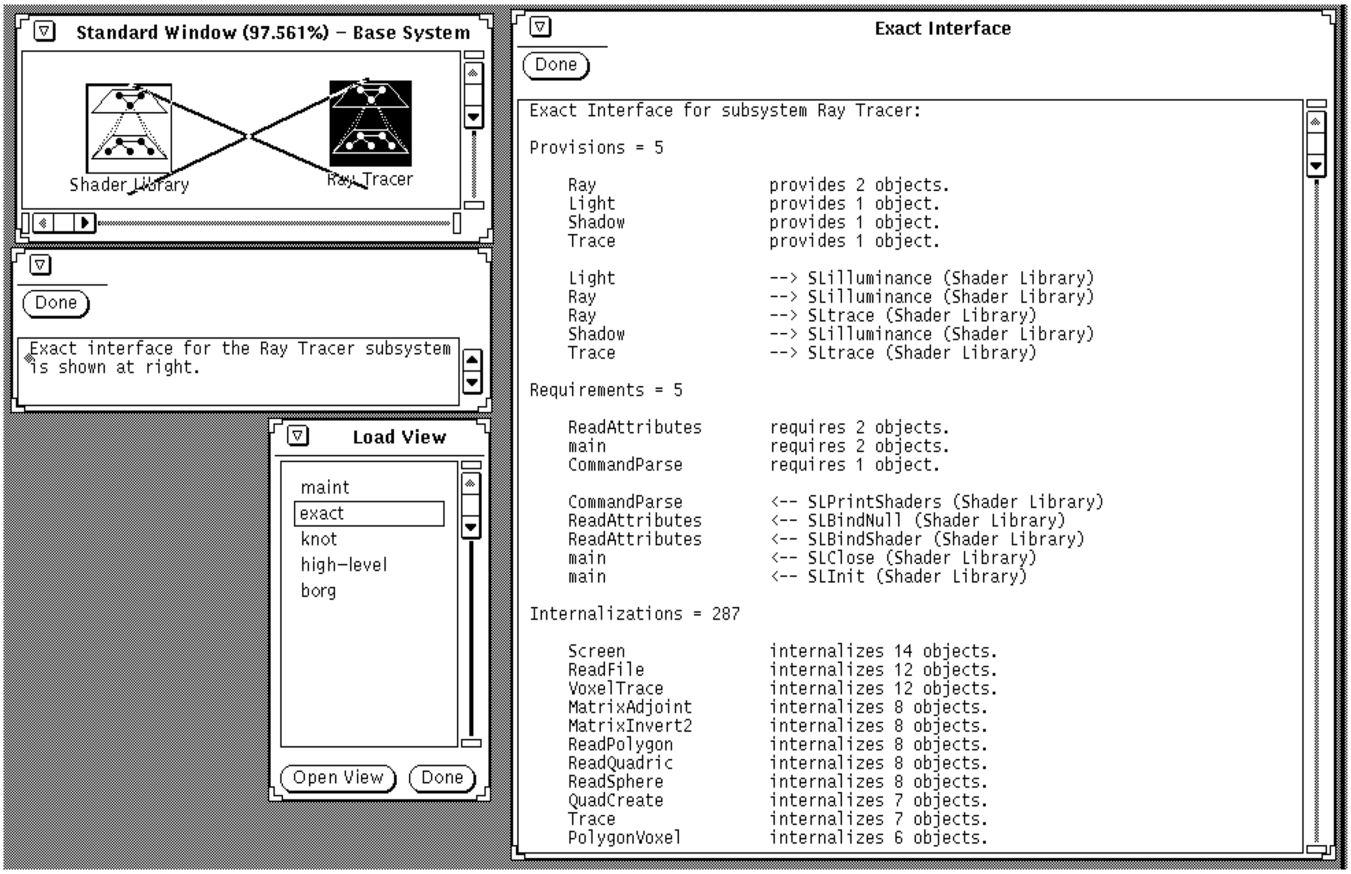

Figure 4: Subsystem interaction

construct higher-level subsystems: (1) composing subsystems around central components, and (2) clustering components with high interconnection strength. The resultant system consists of two main subsystems: `Ray Tracer` and `Shader Library`. The components of subsystem `Ray Tracer` perform the ray tracing rendering process, whereas the `Shader Library` components perform shading of geometric objects.

`Ray Tracer` consists of four subsystems: `Control`—the top level of the function call hierarchy; `Initialization`—a small subsystem for setting up various parameters and starting up the graph editing environment; `Utilities`—all the basic data types, their access functions, and some other primitive operations; and, finally, `Ray`—the basic ray tracing and rendering subsystem.

`Shader Library` also consists of four subsystems: `SL Shader`—shading of objects with different surface characteristics; `SL Primitives`—the primitive data types and their access functions required by the shading operation; `SL Utilities`—auxiliary shading operations; and, finally, `SL Light`—operations on lighting models. This view, combined with a simple textual description such as this one, is a good introduction to the ray tracing system.

The view depicted in Figure 4 shows the interaction between the two main subsystems: `Ray Tracer` and `Shader Library`. The text window shows the exact dependencies between these two subsystems. Note that `Ray Tracer` requires only five objects from and provides only five objects to the `Shader Library`. For example, the `Ray Tracer` subsystem provides the `Ray` data type to the `SLilluminance` function in the `Shader Library` subsystem. Also note that this subsystem internalizes 287 dependencies; that is, there are 287 dependencies that have both the client and supplier in the `Ray Tracer` system. This indicates that the quality of this subsystem decomposition is high (*i.e.*, there are few interfaces between the two subsystems—a good firewall). This kind of interface information is readily available for any collection of components and dependencies in the system, and can be used for change analysis and optimal recompilation strategies [16].

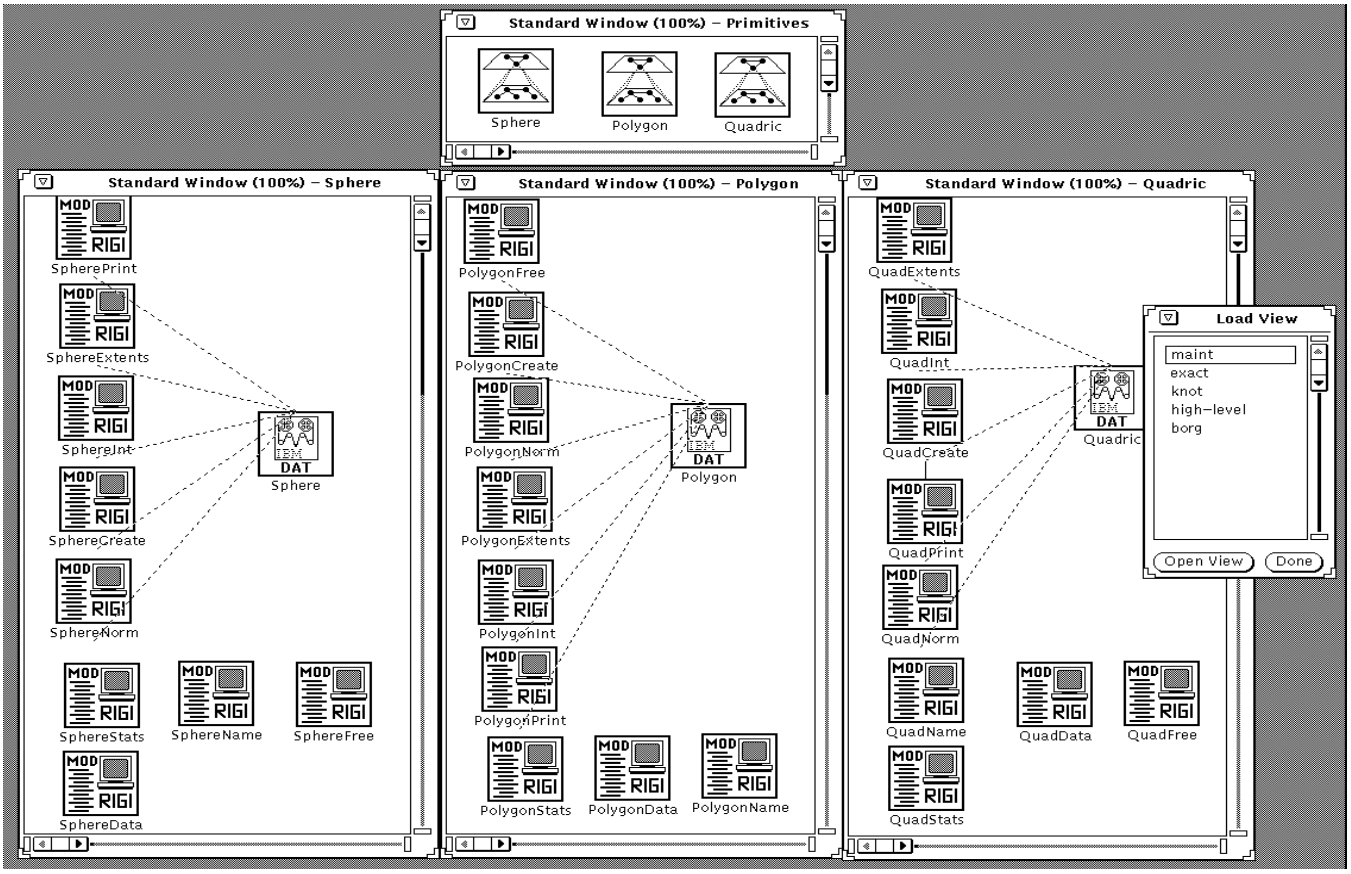

Figure 5: Maintenance view

The `Ray Tracer` subsystem was designed as an extensible subsystem using object-oriented techniques. In Figure 5, note that the subsystems implementing primitive geometric objects (*i.e.*, `Sphere`, `Polygon`, and `Quadric`) have similar access functions for creating and rendering their primitives. Thus, a maintenance programmer can easily recognize how to integrate a new primitive into the current system by simply exploring this view and its interaction with the rest of the system.

The ray tracing system is intended as a development vehicle for graphics research, and as such is in a continuous state of change. Our analysis revealed that a considerable number of functions in `Shader Library` are not actually being used by `Ray Tracer`. The designer of the ray tracer verified our findings. The shading method is not quite as sophisticated as to require more of the functionality provided by `Shader Library`. Moreover, at any one time the system contains a significant amount of code that was used for debugging and testing purposes. When a debugging and testing phase is over, it might be desirable to remove this code from the system. This is not always a simple task, as functions calls or data type references can be deleted in one section of the code without the maintainer realizing that it is the last reference to that portion of the code. The isolated icons shown in Figure 3 represent such dead code.

The four views presented in this section portray various key features of a real-world software system. We believe that these views, combined with the above textual descriptions, provide the reader with a basic understanding of the structure and operation of this system as well as a good starting point for further investigations.

If one views maintenance as *reuse-oriented software development* [17], reverse engineering can benefit everyone involved in software production, including maintainers, developers, documenters, managers, and testers. A recent taxonomy listed key objectives for reverse engineering [18], including coping with complexity, generating alternate views, detecting side effects, recovering lost information, and synthesizing higher abstractions. The views referenced in this section have illustrated how the Rigi environment addresses each of these requirements.

6 Early experience

In 1990 we applied our reverse engineering techniques to a 57,000-line COBOL program, the Practice Manager by Osler Management Inc. of Victoria [19]. The Practice Manager is a comprehensive system for the management of physicians' practices in British Columbia. The purpose of the analysis was to build up-to-date subsystem structures, to assess the quality of the entire system with respect to maintenance, and to identify subsystems that are candidates for re-engineering. The analysis was performed without any foreknowledge of the system and its source code. After completion of the analysis, we presented the derived structures to the chief maintainer of the system. The structures were easily recognized by the maintainer and were consistent with the mental image she had formed of the software over a two-year period. We were also able to show how to split the library into multiple components so that the complexity of the system is significantly reduced. In addition, the information gained on central and fringe components was greatly appreciated by the maintainers of the Practice Manager.

In 1991 we analyzed an 82,000-line physics program, a control and data logging application written in C for the isotope separator experiment at TRIUMF (TRI-University Meson Factory) in Vancouver. The main objective for this analysis was to identify components for re-engineering. In late 1992 we are planning to analyze a large commercial database management system in conjunction with IBM Canada Ltd.

While the first phase—the extraction of the resource-flow graphs and the computation of the exact interfaces—is sufficiently fast, the second phase—the semiautomatic, interactive subsystem composition—may take from a couple of hours for a 30-module system such as the ray tracer, to a few days for a system consisting of a 100 modules. Nevertheless, this semi-automatic procedure is still an order of magnitude faster than manual identification of subsystem structures.

Ideally however, building the subsystem structures of an entire system including its exact interfaces should take no longer than compiling the system, particularly if only a few components or dependencies have been added or deleted since the last time the system was reverse-engineered. To alleviate this problem, incremental algorithms have been implemented so that new and updated modules can easily be integrated with the current software model. Thus, after a change to the source code, the software model can quickly be brought up to date by only processing the changed modules. This process is fully automatic: components and dependencies are updated properly without destroying the created subsystem structures. Hence, the database and the software structures can be kept current at minimal time and cost.

Augmenting the Rigi environment with domain knowledge would enable initial subsystem compositions to be constructed automatically. This would allow the reverse engineer to start at a higher level of abstraction than the call graph (as is currently the case). This generated structure could be rejected by the reverse engineer, enhanced, or altered to suit his or her needs. This area promises to provide a rich area of research in the future.

7 Summary

This paper presented an approach to reverse engineering based on spatial and visual software interconnection models. The Rigi system provides a variety of operations for effective interaction with and presentation of these representations by means of views. Early experience has shown that the coexistence of these two representations is critical to the understanding of the generated data, and greatly benefits subsequent analysis, processing, and decision-making. Using these representations, software engineers can quickly build mental models that are compatible with the ones formed by maintainers of the underlying software. This is a subtle but vital step in the use of reverse engineering technology to meet the challenges of software maintenance and re-engineering.

References

[1] Nicholas Zvegintzov. Nanotrends. *Datamation*, pages 106–116, August 1983.

[2] Barry W. Boehm. Improving software productivity. *Computer*, 20(9):43–57, September 1987.

[3] Robert N. Britcher. Re-engineering software: A case study. *IBM Systems Journal*, 29(4):551–567, 1990.

[4] H.A. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *ICSE'10: Proceedings of the 10th International Conference on Software Engineering,* (Raffles City, Singapore; April 11-15, 1988), pages 80–86, April 1988. IEEE Computer Society Press (Order Number 849).

[5] H.A. Müller and J.S. Uhl. Composing subsystem structures using (k,2)-partite graphs. In *Proceedings of the Conference on Software Maintenance 1990,* (San Diego, California; November 26-29, 1990), pages 12–19, November 1990. IEEE Computer Society Press (Order Number 2091).

[6] R.S. Arnold. Tutorial on software reengineering. In *CSM'90: Proceedings of the 1990 Conference on Software Maintenance,* (San Diego, California; November

26-29, 1990). IEEE Computer Society Press (Order Number 2091), November 1990.

[7] Santanu Paul, Atul Prakash, Erich Buss, and John Henshaw. Theories and techniques of program understanding. In *Proceedings of CASCON'91,* (Toronto, Ontario; October 28-30, 1991), pages 37–53. IBM Canada Ltd., October 1991.

[8] T. Brandes and K. Lewerentz. GRAS: A non-standard database system within a software development environment. In *Proceedings of the Workshop on Software Engineering Environments for programming-in-the-large,* (Harwichport, Maine), pages 113–121, June 1985.

[9] Dewayne E. Perry. Software interconnection models. In *ICSE'9: Proceedings of the 9th International Conference on Software Engineering,* pages 69–69, April 1987.

[10] G.L. Myers. *Reliable software through composite design.* Petrocelli/Charter, 1975.

[11] Scott R. Tilley, Hausi A. Müller, and Mehmet A. Orgun. Documenting software systems with views. In *Proceedings of SIGDOC'92: The 10th International Conference on Systems Documentation,* (Ottawa, Ontario; October 13-16, 1992), pages 211–219, October 1992. ACM Order Number 613920.

[12] H.A. Müller. Verifying software quality criteria using an interactive graph editor. In *Proceedings of the Eighth Annual Pacific Northwest Software Quality Conference,* (Portland, Oregon; October 29-31, 1990), pages 228–241, October 1990. ACM Order Number 613920.

[13] Hausi A. Müller and Brian D. Corrie. Measuring the quality of subsystem structures. Technical Report DCS-193-IR, University of Victoria, November 1991.

[14] Mehmet A. Orgun, Hausi A. Müller, and Scott R. Tilley. Discovering and evaluating subsystem structures. Technical Report DCS-194-IR, University of Victoria, April 1992.

[15] S.M. Kosslyn. *Image and Mind.* Harvard University Press, 1980.

[16] Scott R. Tilley. Changing module interfaces. Master's thesis, University of Victoria, May 1989.

[17] Victor R. Basili. Viewing maintenance as reuse-oriented software development. *IEEE Software*, 7(1):19–25, January 1990.

[18] Eliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, January 1990.

[19] H.A. Müller, J.R. Möhr, and J.G. McDaniel. Applying software re-engineering techniques to health information systems. In *Proceedings of the IMIA Working Conference on Software Engineering in Medical Informatics (SEMI),* (Amsterdam; October 8-10, 1990), October 1990.