A Reverse Engineering Approach To Subsystem Structure Identification

HAUSI A. MÜLLER[†]
Department of Computer Science
University of Victoria
P.O.Box 3055, Victoria, B.C.
Canada V8W 3P6
Tel: (604) 721-7630, Fax: (604) 721-7292
E-mail: hausi@csr.uvic.ca

MEHMET A. ORGUN
Department of Computing
Macquarie University
Sydney, NSW 2109,
Australia

SCOTT R. TILLEY AND JAMES S. UHL
Department of Computer Science
University of Victoria
P.O.Box 3055, Victoria, B.C.
Canada V8W 3P6

[†]Address for correspondence.

A Reverse Engineering Approach To Subsystem Structure Identification[1]

SUMMARY

Reverse engineering is the process of extracting system abstractions and design information out of existing software systems. This process involves the identification of software artifacts in a particular subject system, the exploration of how these artifacts interact with one another, and their aggregation to form more abstract system representations that facilitate program understanding.

This paper describes our approach to creating higher-level abstract representations of a subject system, which involves the identification of related components and dependencies, the construction of layered subsystem structures, and the computation of exact interfaces among subsystems. We show how top-down decompositions of a subject system can be (re)constructed via bottom-up subsystem composition. This process involves identifying groups of building blocks (e.g., variables, procedures, modules, and subsystems) using composition operations based on software engineering principles such as low coupling and high cohesion. The result is an architecture of layered subsystem structures.

The structures are manipulated and recorded using the Rigi system, which consists of a distributed graph editor and a parsing system with a central repository. The editor provides graph filters and clustering operations to build and explore subsystem hierarchies interactively. The paper concludes with a detailed, step-by-step analysis of a 30-module software system using Rigi.

KEYWORDS: Software maintenance, reverse engineering, program understanding, software engineering principles, resource-flow graphs, subsystem hierarchies, subsystem composition, exact interfaces, re-engineering, change analysis.

[1]This work has been 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, the Science Council of British Columbia, the University of Victoria, and IBM Canada Ltd.

1 INTRODUCTION

Software maintenance is defined as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment (ANSI/IEEE, 1983). In an ideal world, software projects are well prepared for such maintenance tasks. The design information is readily available to the maintainers, the documentation accurately reflects the source code (including the cumulative effects of all the evolutionary modifications), and the system is structured in an intuitive and predictable manner that facilitates understanding and allows maintenance to be performed with confidence. Unfortunately, evolving software projects rarely reflect this ideal scenario.

During long-term maintenance of large software projects, rationales for design decisions are often not available because the people who might be able to trace the change histories are no longer with the project. Similarly, design documents may be inconsistent with respect to the actual source code due to undocumented corrections, improvements, customizations, or enhancements. This kind of evolution is common and can complicate maintenance tasks considerably. The source code is often the only reliable means for a software maintainer to decide how a software system is to be modified when implementing a desired enhancement or eradicating an undesired feature. However, the sheer size of the source text often prevents a thorough and exhaustive analysis of all potentially affected components and dependencies. A seemingly inconsequential change may have unforeseen and sometimes devastating results even when only a small portion of the code is actually modified. Therefore, the effects of any modification must be considered carefully, not only locally, but throughout the system.

A good first step in analyzing large amounts of source code is to identify system abstractions and, in particular, to determine overall structure and to recover architectural design information. This recovered information can then be used to break down the source code into understandable and manageable subsystems at various levels of abstraction. These subsystem structures, in turn, can then serve as organizational axes for program understanding as well as risk, change, and impact analyses. *Reverse engineering* has emerged as a promising technology for recognizing such architectural features in source texts (Forte, 1992). Note that our approach concentrates on structural abstractions as opposed to functional abstractions as for example in the REDO project (Breuer *et al.,* 1991).

The process of reverse engineering a subject system is usually defined as consisting of two distinct phases (Arnold, 1990; Chikofsky and Cross II, 1990): (1) the identification of the system's current components and their interrelationships, and (2) the extraction of system abstractions and design information. The information produced as a result of this process can then serve as a basis for system comprehension and analysis.

The aim of our approach to extracting system abstractions out of a subject system is to expose its overall structure. In particular, it involves the identification of subsystems, the construction of hierarchies of subsystem structures, and the computation of exact interfaces among subsystems. Once the structural properties of the system are identified using this reverse engineering approach, the effects of local changes can be traced effectively by analyzing the resources exchanged among subsystems at various levels of detail.

Identifying subsystems is a precursor to creating hierarchical subsystem structures and is performed many times over the life span of a software project. During the design phase, subsystem structures are often used to split the project into work assignments to manage the design and implementation of the project. At integration time, subsystem decompositions may serve as testing and integration plans. Thus, constructing hierarchical composition structures can not only be beneficial for long-term maintenance, but also during the early design stages.

Discovering subsystem structures is an art. Our work is based on the premise that, *given sufficient time*, an experienced software engineer is usually able to decompose a system better than an automatic procedure can. However, the human designer needs assistance from the development environment for the tedious and arduous tasks involved in the decomposition process. A designer may call upon the environment to produce alternative clusterings of a given set of modules and then decide on strategies to form compositions. As hierarchies of subsystems are being built, the software engineer can interactively modify the layers and possibly undo some of the clusters if they are later deemed inappropriate. Thus, the role of the human software engineer constitutes a central and integral part of our approach to subsystem identification and composition. In short, we subscribe to the motto "automate as much as possible, but never fully automate."

This paper presents techniques for discovering, restructuring, and analyzing subsystem structures in software systems using Rigi,[2] a system for reverse engineering. Section 2 outlines our subsystem composition methodology. Section 3 introduces software structure models for modeling multiple subsystem hierarchies and discusses the relations used for composing subsystem structures. In Section 4, we present measures for identifying subsystem structures based on established software engineering principles. The features of the Rigi system that are relevant for the subsystem identification and composition process are described in Section 5. In Section 6, we analyze a real software system to demonstrate how Rigi's composition methods can be utilized to recover architectural design information. This section also outlines subsystem composition strategies that can be realized by the software engineers using the composition operations provided by the Rigi system. Section 7 discusses some of the lessons learned from reverse engineering this particular subject system. Related work is summarized in Section 8. The paper concludes with a brief report on our early experience with the Rigi system.

2 SUBSYSTEM COMPOSITION METHODOLOGY

Reverse engineering generally involves extracting design artifacts out of the source code and building or synthesizing abstractions that are less implementation-dependent than the source code (Chikofsky and Cross II, 1990). Note that in this process the source code of the subject system is *not* altered, although additional information about the system is constructed and generated. In contrast, the process of *re-engineering* typically consists of a reverse engineering phase followed by a forward engineering or re-implementation phase that alters the source text.

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

The first phase of the reverse engineering process, the extraction of software artifacts, is language-dependent and essentially involves parsing the source code and storing the artifacts in a repository. Our parsing system currently supports the programming languages C, COBOL, and a proprietary IBM system-programming language. We use GRAS, a database specifically designed to represent graph structures, as a central repository to store the parsed artifacts (Brandes and Lewerentz, 1985). The software engineer can then manipulate the stored artifacts through an interactive graph editor. The interaction among the tools of the Rigi system is depicted in Figure 1.

Our approach to the second phase features a semi-automatic language-independent subsystem composition methodology to construct hierarchical subsystem structures (Müller and Uhl, 1990). This process involves the interactive construction of aggregate software components out of building blocks (e.g., variables, procedures, modules, and subsystems) using the Rigi system. 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 *small and few interfaces between subsystems* and *high strength within subsystems*. The resulting composition structures are layered graphs called $(k, 2)$-partite graphs (Müller and Uhl, 1990; Mata-Montero, 1990).

An important ingredient in our composition methodology is software quality measures based on exact interfaces and established software engineering principles to evaluate subsystem structures (Müller, 1990). These 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 *fire walls* among subsystems.

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

In summary, our subsystem composition methodology involves the following stages:

1. The extraction of relevant system components and dependencies out of the subject system's source code to form resource-flow graphs;
2. The semi-automatic composition of subsystem hierarchies using the Rigi system on top of these resource-flow graphs;
3. The computation of exact interfaces among the constructed subsystems;
4. The evaluation of the (re)constructed subsystem structures using measures based on established software engineering principles; and finally,
5. The capturing of pertinent view sequences (snapshots of the reverse engineering environment) for target audiences, which can be recalled, inspected, played back, and can serve as a basis for further investigations and explorations.

The following discussion focuses on software structure models for multiple subsystem hierarchies, composition measures, operations provided by the Rigi system to support different composition strategies, and, in particular, the implementation of the first three stages of the above outlined subsystem composition methodology.

3 SOFTWARE STRUCTURE MODELS

Software structures such as control flow, data flow, resource flow, as well as aggregation and generalization hierarchies, are often modeled by directed weighted graphs. The nodes and directed edges (arcs) of these graphs represent system components and their dependencies. The weights on the arcs vary with the application. We use a special class of directed graphs, called $(k, 2)$-partite graphs, for representing the structure of software systems. In this section, we show how resource-flow graphs and multiple subsystem hierarchies can be modeled using these layered graphs.

3.1 Resource Relations

The primary models used to describe, represent, and manage software structure are the *unit interconnection model* and the *syntactic interconnection model* (Prieto-Diaz and Neighbors, 1986; Perry, 1987). The main distinction between the unit and the syntactic models is the granularity of interconnection ranging from *files*, *subsystems, classes,* and *modules* in the unit model to the nameable entities of programming languages such as *procedures*, *functions*, *constants*, or *variables* in the syntactic model. Below, we extend these two models to form arbitrary subsets of system components and dependencies. The graphical representations of these software structures are captured by a new type of software interconnection model based on spatial and visual imagery (Müller *et al.*, 1992).

It is convenient for us to think of such an interconnection model as a directed weighted *resource-flow graph* (RFG). The vertices and edges of an RFG represent system components and resource supplier-client relations, respectively. A directed edge from component v to w indicates that component v provides a set of syntactic objects to component w. In other words, w is a client of v, and v is a supplier of w. Depending on the application, the edge weights may be a set of resource names (e.g., functions, data types, modules and subsystems), the cardinality of the resource set, or even absent.

More formally, a resource-flow graph $G = (V, E)$ consists of two components: (1) V is a set of system components, and (2) $E \subseteq V \times V$ is a set of binary tuples of the form $< v, w >$ representing supplier-client relations between components (i.e., a directed edge from v to w). In the following discussion, we assume that the edge weights (EW) are a list of resource names (e.g., $EW(v, w) = \{fa, fb, dt\}$).

The following terminology is also used in subsequent sections when referring to RFGs: the two sets of all syntactic objects *provided* and *required* by a component $v \in V$, denoted by $Prv(v)$ and $Req(v)$, respectively, are called the *provisions* and *requisitions* of the component. These resource sets are defined in terms of edge weights:

$$Prv(v) = \bigcup_{x \in V} EW(v, x),$$

$$Req(v) = \bigcup_{x \in V} EW(x, v).$$

3.2 Composition Relations

Programming languages offer modules and classes to aggregate and encapsulate sets of related routines, data types, variables, and constants. Files and directories can then be used to group related modules or classes to form subsystems. In turn, subsystems can be further composed to form (multiple) hierarchies of subsystems. The containment relationships of these resulting hierarchical structures constitute *composition dependency graphs* (CDGs). The nodes of a CDG represent system components or subsystems and the arcs signify aggregation or composition relationships. In the sequel, the nodes of a CDG are referred to as subsystems regardless of whether they denote functions, data types, classes, modules, or high-level subsystems.

If the composition relation is constrained so that a given node can only appear in one subsystem, then the relation induces a strict tree hierarchy. In our model this restriction is not enforced, resulting in CDGs that are directed, acyclic graphs. The main reason for this structural relaxation is to be able to represent multiple subsystem hierarchies within the same model.

The graphs resulting from a combination of the graphs induced by resource and composition relations are termed $(k, 2)$-*partite graphs.* As depicted in Figure 2, a $(k, 2)$-partite graph consists of a series of graph *layers* $G_1, \ldots, G_n$, modeling a series of subsystem layers or RFGs. Layers are connected by means of *vertical edges*; however, vertical edges may only connect adjacent layers (i.e., at most two layers or adjacent sets in the sequence $V_1, \ldots, V_n$). Moreover, the number of nodes per layer is bounded by k for theoretical reasons (Mata-Montero, 1990).

3.3 Exact Interfaces

At the unit level, most programming languages are imprecise with respect to requisition and provision of resources (Wolf *et al.*, 1988). For example, modules often import and export entire interfaces rather than specific objects. Given composition and resource relations as outlined in the preceding sections, one can compute *exact* syntactic interfaces or *exact* cross-references among any subsets of system components (Uhl, 1989). Exact interface information at various levels of abstraction is extremely useful for risk, change, and impact analyses (Tilley, 1992).

Let $v, w \in V$ be components in a resource-flow graph $G = (V, E)$. The exact interface between v and w is a pair consisting of a set of *exact requisitions* of v from w, $ER(v, w)$, and a set of *exact provisions* of v to w, $EP(v, w)$, which are defined as follows:

$$ER(v, w) = Req(v) \cap Prv(w),$$

$$EP(v, w) = Prv(v) \cap Req(w).$$

Thus, $ER(v, w)$ is defined as the intersection of the requirements of v and the provisions of w, and $EP(v, w)$ is the intersection of the provisions of v and the requisitions of w. These definitions can easily be extended to entire subgraphs of V.

4 COMPOSITION MEASURES

There are many ways in which composition relations can be imposed on resource-flow graphs. However, during the process of program understanding, we are particularly interested in subsystem structures that expose the overall architecture of the subject system, help recover original design decisions, and facilitate system comprehension. The methods and strategies that have been used in software development for subsystem decomposition can thus be used as a guide during discovery and reconstruction of subsystem structures.

This section introduces two sets of software composition measures for RFGs based on the established software engineering concepts of *coupling* and *cohesion* (Myers, 1975). High coupling among subsystems and low cohesion within subsystems is indicative of a lack of information hiding, which can complicate program understanding and maintenance efforts considerably. Low coupling among subsystems and high cohesion within subsystems minimizes the number of paths through which changes and errors can be propagated throughout the system, and localizes both change and impact analyses.

The intended purpose of the first set of similarity measures is to capture the two software engineering principles *high strength within a component* and *low coupling among components.* The intention of the second set is to identify loosely coupled components having *common clients* or *common suppliers.* The latter measure captures the software engineering principle *few interfaces.* Similarity measures have been studied extensively in many other areas to devise clustering methods and taxonomic hierarchies (Dunn and Everit, 1982).

4.1 Interconnection Strength Measures

The *interconnection strength* $IS(v,w)$ of two nodes, v and w, in an RFG is defined as the *exact* number of syntactic objects exchanged between the two nodes. More formally,

$$IS(v,w) = |ER(v,w)| + |EP(v,w)|.$$

Two components are said to be *strongly coupled* if their interconnection strength is greater than the *high-strength threshold* T_h and *loosely coupled* if their interconnection strength is less than the *low-strength threshold* T_l. The thresholds T_l and T_h can be increased and decreased in a stepwise fashion to obtain alternative compositions and partitions and to control the sizes of the interfaces among subsystems.

Subsystems with *high cohesion* can be found by identifying strongly coupled components. Subsystems with low coupling among them can be found by *separating* loosely coupled components or by using a graph partitioning algorithm for computing *articulation points.* If the removal of a node v disconnects a connected graph, then v is said to be an articulation point (Aho *et al.*, 1974, p. 179). The *internal strength* of a given component v can be computed as the sum of (the cardinality of) the weights of all the edges in the subgraph subsumed by v through composition dependencies.

4.2 Common Clients/Suppliers Measures

The common clients/suppliers measures are intended for the identification of loosely coupled components having *common clients* or *common suppliers*. These measures satisfy the software engineering principle of *few interfaces among components*, because merging components that have common clients or suppliers reduces the number of interfaces among the components involved.

Two components are similar with respect to their clients if and only if they provide objects to similar sets of clients. Analogously, two components are similar with respect to their suppliers if and only if they require objects from similar sets of suppliers. Thus, the common client and supplier subsets of a set M of system components in an RFG $G = (V, E)$, $CS(M)$ and $SS(M)$, respectively, are defined by the following equations:

$$CS(M) = \bigcap_{x \in M} \{v \in V | < x, v > \in E\},$$

$$SS(M) = \bigcap_{x \in M} \{v \in V | < v, x > \in E\}.$$

Two nodes, v and w, are said to be *neighbors* or *siblings* with respect to their clients (suppliers) if and only if the cardinality of their client (supplier) subset $|CS(\{v, w\})|$ ($|SS(\{v, w\})|$) is greater than the client (supplier) threshold T_c (T_s).

Sibling nodes are often found in libraries: library routines may implement a set of primitives with analogous functionality (i.e., they are logically related), but are not directly implemented in terms of each other. Examples of such libraries under UNIX[3] include the standard C library `stdlib` and the mathematics library `math`. Thus, if a routine out of one of these libraries is used by a given client c, it is likely that similar library routines are also used by c.

[3]UNIX is a trademark of Unix Systems Laboratories, Inc.

5 RIGI EDITOR

The user interface of the Rigi system is a distributed, window-based, graph editor that allows the users to edit, maintain, and explore the objects stored in an underlying repository representing a software system (Müller and Klashinsky, 1988). This section outlines the salient operations of the graph editor for discovering, composing, exploring, visualizing, and analyzing layered subsystem structures.

5.1 Basic Operations

The user interacts with the Rigi editor via the mouse and the keyboard in a multi-window environment such as Sunview, OpenLook, or Motif. Most of the operations provided involve a group of *selected* objects (e.g., modules or subsystems) and/or dependencies (e.g., function calls or data type references) in a $(k, 2)$-partite graph and which are normally displayed in a single window. The editor provides navigation facilities to explore software systems both laterally (at the same level of detail) using scroll bars and vertically (at increasing or decreasing levels of detail) by opening and closing documents. Individual layers as well as hierarchical cross-sections of a $(k, 2)$-partite graph (or parts thereof) can be displayed in a variety of ways for visual inspection purposes. A collection of windows with hypertext-like annotations (snapshots of the user interface) can be saved, recalled, and played back for further inspection and documentation purposes (Tilley *et al.*, 1992).

5.1.1 *Basic graph editing*

Entire subgraphs and hierarchies of subgraphs displayed in a single window may be duplicated and deleted using the operations *Cut*, *Copy*, *Paste*, and *Clear*. For instance, applying the *Copy* operation to a subsystem node involves the copying of all objects and dependencies associated with that subsystem including modules, interfaces, versions, and source documents. The *Copy* operation enables a subgraph to exist in multiple subsystems simultaneously, as is permitted in the $(k, 2)$-partite graph model. With a simple *Cut* operation, some system components and their dependencies that are immaterial for the current investigation can be removed from the current database. Thus, these basic editing operations affect the underlying database.

One way to limit the time it takes to build system abstractions on top of an RFG is to prune the initial database as much as possible using a sequence of *Cut* or *Clear* operations.

5.1.2 *Zooming and filtering*

The Rigi system provides a variety of *zooming*, *filtering*, and *grouping* facilities allowing users to navigate swiftly through the myriad objects and dependencies and to identify and aggregate pertinent information threads quickly. For instance, the Rigi editor offers *node* and *edge type filters* for focusing on individual semantic networks or contexts (i.e., a graph whose nodes and edges are members of a single node and/or edge type, respectively) or combinations of individual contexts. Thus, one can easily focus on the dependencies of a certain type of nodes (e.g., *data types*) by invoking a node type filter.

The zooming and filtering operations do not affect the underlying database. They simply allow the user to impose different views on the entities of the repository, filter unnecessary details, and alter focus.

5.1.3 *Searching*

The search commands include *grepping* (string search using regular expressions as supported by the UNIX operating system) for string patterns in node labels. The graph editor supports the logical renaming of node labels; their initial value is extracted from the source code by the parsing system during the first stage of reverse engineering (for low level artifacts such as functions and data types), or is supplied by the user interactively. Filters and graph traversal techniques are also used to constrain the search space. For example, the search space can be restricted to a fixed set of node classes (e.g., modules and subsystems) or to the forward and/or backward dependency trees starting from a designated node.

5.1.4 *Collapsing*

Collapsing is a form of graph transformation for defining and composing subsystem structures. The *collapse* operation essentially replaces a subgraph (a set of components at some level of a given $(k, 2)$-partite graph) by a single node (a subsystem). All the collapsed components are pushed to the next (more detailed) level of the subsystem hierarchy. Its inverse operation restores the original graph. To make this operation completely reversible, the subgraph and the edges between the subgraph and the remaining graph have to be restored.

After collapsing takes place, the exact interfaces are propagated as follows: for each collapsed node v, compute its exact interface by computing the exact interface of the subgraph that was collapsed to form v. For instance, if S is the collapsed subgraph, then the pair $ER(S, V - S)$ and $EP(S, V - S)$ is computed. Note that the exact provisions of a subsystem are often a subset of the objects provided by all of its components. Consequently, subsystems can encapsulate large interfaces and provide a considerably smaller set of objects to the remainder of the system (i.e., a *small interface*).

5.2 Subsystem Composition Operations

We now have sufficient machinery to describe some of the subsystem composition operations provided by the Rigi system. These graph operations are self-contained and efficient and can therefore be invoked individually and interactively through the Rigi user interface. Given a (layered) resource-flow graph, composition operations can be used to identify subgraphs with desired properties. Such a subgraph can then be *collapsed*: replaced by a newly created node representing a subsystem. The parameterized composition measures introduced in Section 4 are usually used in the composition process and are designed to improve the overall quality of subsystem structures by promoting low coupling and strong cohesion. Note that the composition operations are not transitive in general and thus different subsystem structures may result depending on the order in which composition and collapse operations are applied. However, since subsystem identification and subsystem composition are two distinct steps, the collapsing of individual subsystems can be delayed until the user is satisfied with the arrangement of a particular layer.

5.2.1 *Remove omnipresent nodes*

This is a parameterized operation designed to filter the noise at the initial stages of the subsystem composition process. For each node $v \in V$ in a resource-flow graph $G = (V, E)$, let $c(v)$ be the number of direct clients of v. If $c(v)$ is greater than the *omnipresent threshold* T_{op}, then v is said to be *omnipresent.* Because omnipresent components obscure system structure, they are often removed from an RFG together with all their incident edges (by cutting or filtering). An example of an omnipresent node is a debugging module containing debug variables or routines that are referenced by most other modules.

5.2.2 *Compose by interconnection strength*

This composition operation is based on the interconnection strength measure $IS(v, w)$ defined between any two subsystems v and w. The user can interactively adjust two parameters, T_h and T_l, which represent high and low-strength threshold values, respectively, to guide the identification process. Depending on the value of $IS(v, w)$, one of the following three clauses applies.

- $IS(v, w) \geq T_h$

  In this case, v and w are strongly coupled and thus can be collapsed into the same subsystem.

- $IS(v, w) \leq T_l$

  In this case, v and w are loosely coupled and thus can be assigned to two separate subsystems.

- $T_l < IS(v, w) < T_h$

  Node pairs in this category are neither strongly nor loosely coupled; they can be assigned to the same subsystem or separate subsystems.

5.2.3 *Compose by common clients/suppliers*

These composition operations are based on the common clients/suppliers measures defined above. For each node pair $v, w \in V$ in an RFG $G = (V, E)$, let $CS(\{v, w\})$ and $SS(\{v, w\})$ be the common client and the common supplier subset, respectively. If the cardinalities of these subsets are greater than or equal to their respective thresholds T_c and T_s (i.e., v and w have either similar clients or similar suppliers and are thus siblings), then v and w are assigned to the same subsystem.

5.2.4 *Compose by centricity*

This composition operation is parameterized by T_k and T_f, representing the *connectivity threshold* values for identifying central (key) and fringe components. Given a component $v \in V$ in an RFG $G = (V, E)$, the *external strength* of v, $ES(v)$, is defined as the sum of (the cardinality of) the weights of all the edges between v and all the other components in the RFG that are not subsumed by v through composition dependencies. A node v is said to be a *central* component if $ES(v) \geq T_k$ and similarly a *fringe* component if $ES(v) \leq T_f$. Central and fringe components are normally assigned to separate subsystems. Identifying central components is critical for change and risk analyses, because a small chance in a central component may affect a large number of other components.

5.2.5 *Compose by name*

This composition operation is implemented using the standard Unix regular expression pattern matching engine and is used to identify objects with similar names. For each node $v \in V$ in an RFG $G = (V, E)$, search its module, file, and/or path name for common substrings (e.g., a common prefix). Composition by name is particularly useful during reverse engineering if the designers of the subject system have adhered to naming conventions which might have been recorded in a design document. However, this composition strategy should be used with care, as it can also identify sets of totally unrelated components due to accidental string matches.

6 A CASE STUDY

In this section, we illustrate our reverse engineering approach by analyzing a 30-module software system written in C, a ray-tracing rendering system (Corrie, 1990). The ray tracer creates a two-dimensional image of a three-dimensional scene by simulating how light interacts with the objects located in the scene. The purpose of this analysis is to show how the Rigi system can be used on real-world software systems to identify and compose subsystems, maximize data and control encapsulation qualities within subsystems, and to minimize the number and size of the interfaces among subsystems.

To generate an initial resource-flow graph (RFG), the C source code of the ray tracer, which is kept in multiple directories, is parsed to extract data type and function dependencies automatically. The subject system consists of 265 C objects (functions and data types) and 369 dependencies among these objects (data dependencies and function calls). The initial RFG is stored in the repository as a $(k, 2)$-partite graph and is then maintained as such by the database server. The user then browses, modifies, and analyzes the graph structures through the Rigi editor. Using the composition operations provided by the Rigi editor, subsystem hierarchies are then built interactively on top of the flat RFG and exact interfaces are computed.

The threshold values presented above can be combined in numerous ways to obtain a variety of subsystem composition alternatives. When discriminating among alternatives, the Rigi software quality measures can be used as a guide (Müller, 1990). The rough compositions can then be finely tuned by adjusting individual nodes and edges while observing the effect on the quality measures. However, in the following analysis the focus is on subsystem composition methods and how they are used in a bottom-up fashion to expose the overall architecture of the ray tracer.

6.1 The Overall Picture

The top two levels of the subsystem hierarchy corresponding to the top two layers in the constructed $(k, 2)$-partite graph are depicted in Figure 3. The ray tracer consists of two main subsystems: `Ray Tracer` and `Shader Library`. `Ray Tracer` performs the ray tracing rendering process whereas `Shader Library` provides the basic library routines for shading geometric objects.

The subsystem `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; `Ray`—the basic ray tracing and rendering subsystem; and `Utilities`—all the basic data types, their access functions, and some other primitive operations.

The subsystem `Shader Library` also consists of four subsystems: `SL Shader`—the shading of objects with different surface characteristics; `SL Light`—operations on lighting models; `SL Primitives`—the primitive data types and their access functions required by the shading operation; and `SL Utilities`—auxiliary shading operations. Our analysis has revealed that a considerable number of functions in `Shader Library` are not actually being used by `Ray Tracer`; the system is still under development, and the shading method is not quite so sophisticated as to require all of the functionality provided by `Shader Library`. The designer of the ray tracer verified our findings.

The complete subsystem hierarchy of `Ray Tracer` is depicted in Figure 4. The four nodes at the second layer correspond to the four main subsystems of `Ray Tracer` as described above, and the leaf nodes correspond to functions and data types that are defined in the source code. Our analysis has shown that there are few cross references between the subsystems `Ray Tracer` and `Shader Library`, and that the system is almost tree-structured.

The subsystem hierarchy of `Ray Tracer` consists of six layers. At the lowest level are the primitive data types and their access functions (methods). At the fourth and fifth layers, leaf nodes represent a number of common library routines and system-defined data types. The subsystem hierarchy of `Shader Library` consists of only four layers. Again at the lowest level are the primitive data types and their access functions for shading geometric objects. In total, 170 objects and over 287 dependencies are encapsulated in `Ray Tracer`, and approximately 95 objects and 72 dependencies are encapsulated in `Shader Library`.

The number of layers required in the hierarchy for `Ray Tracer` (together with the number of objects and dependencies absorbed in it) is also indicative of the complexity of the ray-tracing rendering process (the heart of the ray tracer.) We found that a shallow subsystem hierarchy with four layers was enough to fully understand and convey the overall structure and functionality of `Shader Library`, which is evidently less complex than that of `Ray Tracer`. Moreover, `Shader Library` has an almost strict tree hierarchy with little interaction among its loosely coupled components.

With the Rigi editor, exact interfaces can be automatically computed for any collection of system components and/or dependencies. For instance, the computed exact interface for the subsystem `Ray Tracer` is shown in Figure 5. Note that the object under scrutiny is highlighted in reverse video. The exact interface for `Shader Library` is also exposed in the figure because of the symmetry between the two subsystems. `Ray Tracer` requires five objects from `Shader Library` and provides five objects to it; a total of ten objects (seven functions and three data types) is exchanged between the two subsystems. `Ray Tracer` also internalizes 287 objects (i.e., it localizes 287 dependencies among its components).

As shown in the Control Panel window at the top of Figure 5, we set the high threshold value for interconnection strength to five, which means that two components at the second level of the subsystem hierarchy are considered strongly (loosely) coupled if their interconnection strength is greater (less) than five. This setting indicated that there are two medium strength interfaces, but no high strength or low strength interfaces. Although at higher levels of the subsystem hierarchy interfaces generally tend to absorb more than five dependencies, we felt that for a medium-sized software system such as the 30-module ray tracer, five was a realistic number for interconnection strength. Therefore, we can claim that the software engineering principle *small interfaces among components* is satisfied. Moreover, most of the control and data dependencies are absorbed within the two subsystems, indicating the software engineering principle *high strength within subsystems* is also satisfied.

6.2 Composition Process

The subsystem composition methodology as supported by the Rigi system is designed to be flexible. The software engineer can experiment with and realize various identification and composition strategies by applying the operations in different orders and by adjusting the various thresholds. In this section, we give a more detailed, step-by-step account of how the final subsystem structure of the ray tracer outlined in the preceding section was derived.

6.2.1 *Filtering the noise*

The starting point is the initial RFG of the ray tracer generated by the parsing system. The entire graph is displayed in one window with object and dependency types represented by different icon and line patterns. This graph looks more complicated than it really is because of the noise introduced by dead code and some omnipresent objects such as debugging and error-reporting functions. The dead code is easily identified by adjusting the fringe component threshold T_f to zero. This connectivity threshold is used to identify those components whose connectivity (the total number of direct suppliers and/or clients) is less than or equal to the threshold value. Figure 6 highlights the unused code in the `Ray Tracer` subsystem. A total of 68 functions and data types is identified as unconnected and thus as unused objects in the `Ray Tracer` and `Shader Library` subsystems as a result.

The error-reporting function `rayerror` is identified by adjusting the omnipresent threshold T_{op} to a fairly high value. All the identified objects do not contribute much to the actual structure of the system. However, it is most likely that some central components are also going to be identified in a similar fashion. Therefore, we cannot blindly remove all the identified components; further inspection is needed. In our case, the functionality of `rayerror` is clear, and hence we can discard it with confidence.

After removing the dead code along with `rayerror` and its incident arcs (all 33 of them), we arrive at a much cleaner graph. Note that dead code is not physically removed from the database of the source code, just its representation in the current view.

6.2.2 *Composition by name*

We observe that a significant number of data types and functions have the common prefix `SL`. A cursory examination of the source text of these objects reveals that they all share the common feature of providing the basic functionality required for shading geometric objects. Therefore, all functions and data types whose names begin with either an "`SL`" or "`sl`" prefix (a total of 95 objects) are collapsed into the `Shader Library` subsystem. The remaining objects including their dependencies are collapsed to form subsystem `Ray Tracer`.

After carefully examining the resulting subsystem structure and inspecting the source code, we did not detect any accidental couplings of objects that do not seem to belong to subsystem `Shader Library`. Encouraged with this observation and the fact that the two main subsystems communicate through a small-sized interface, we conclude that the designer of the ray tracer had in fact two main subsystems in mind. In short, we were able to identify a likely top-down design decision by means of a bottom-up strategy; we created a system decomposition by subsystem composition.

The two main subsystems are further decomposed into smaller subsystems using additional composition-by-name operations.

6.2.3 *Encapsulating data types*

To facilitate data encapsulation at the lower levels of the subsystem hierarchy, we try to identify the basic data types provided in the system, such as primitive geometric objects, and group these data types with their access functions. This strategy is based on the assumption that the functionality of a given data type is better understood in the proper semantic context.

For example, we can select the `Polygon` data type and apply the *select incoming neighbors* operation on data arcs. As exhibited in Figure 7, this operation identifies all the functions and data types that depend on `Polygon`. Note that all direct clients of `Polygon` are highlighted. The visual inspection of the selected data types and functions as well as their source code suggests that these objects really form an abstract data type and thus constitute a subsystem. However, there might be additional functions that logically belong to this abstract data type, but syntactically do not access the data type. Indeed, three extra functions are found in the dead code with a common prefix "`Polygon`." These functions are currently not used by any other function, but have apparently have been written so that the abstract data type provides a logically complete set of operations. There are two more functions whose names contain the string "`Polygon`," but further inspection of the source text reveals that they do not actually belong to the `Polygon` subsystem.

We also identify two other subsystems that are very similar to `Polygon` with respect to structure (i.e., data type with analogous access functions): `Sphere` and `Quadric`. This investigation shows nicely how the system can be extended to support the rendering of additional geometric primitives (e.g., `SuperQuadric`).

However, we realize that encapsulating data types is not always a winning composition strategy, especially for those data types that are used by many unrelated functions without any uniform pattern or for system-defined data types. Without the aid of the browsing and analyzing capabilities of the Rigi editor, it would be tedious to verify that these data types and their dependencies obscure the overall structure of the system and collapsing them into subsystems would not be worthwhile for a better understanding of the system. In fact, these data types and their dependent functions are candidates for re-engineering for the purpose of simplifying the design of the ray tracer (especially that of the `Ray Tracer` subsystem.)

6.2.4 *Composition by common clients/suppliers*

We observe that some of the functions are not tied together by any data types. Thus, as a next step we look for functions that are used by common clients or have common suppliers. By adjusting the corresponding threshold values T_c and T_s, we can search for potential subsystems and inspect their structures.

For instance, when the threshold values are quite small (two or three), a number of functions that perform some matrix operations also share the common prefix "`Matrix`." Further inspection reveals that these operations depend on a few vector operations that are also identified, but not required by any other object in the system. The vector operations are apparently implemented as lower-level library routines for matrix operations. The effect of subsystem identification by common clients/suppliers measures is shown in Figure 8.

By collapsing the vector operations as a new subsystem into the `Matrix` subsystem, more control dependencies are absorbed and the number of interfaces is reduced. Some other functions identified by the common prefix `Matrix` (shown in the lower left corner in Figure 8) are also collapsed into `Matrix`. However, we observe that the number of dependencies within `Matrix` is small; therefore, `Matrix` has low internal strength. Note that not all of the identified functions are collapsed into `Matrix` (e.g., `ReadView`, which belongs to an Input/Output library, is not included).

6.2.5 *Finishing touches*

As we repeat the composition strategies discussed above (and some others that have not been mentioned in this paper), `Shader Library` takes its final form. As for `Ray Tracer`, we use composition by interconnection strength and composition by centricity at the top levels (as well as visual inspection of graphs and source text in some cases) to group relatively small subsystems together. They form higher-level subsystems that reduce the number of interfaces among subsystems. Again, we continually inspect the resulting subsystem structures to avoid anomalies while adjusting the threshold values to arrive at the final software structure.

7 DISCUSSION

Our experiments with the ray tracer demonstrate that there is usually not a unique static subsystem structure that serves all purposes. While the initial resource-flow graph generated by the parsing system is static and unique, the various investigation and composition facilities provided by the Rigi system allow the user to explore many alternative subsystem hierarchies and to simulate a variety of what-if scenarios.

For instance, the very first subsystem structure we came up with was not satisfactory; the interfaces were large, and the heart of `Ray Tracer`, the `Voxel` subsystem, was not structured with consideration to its interconnection strength, but rather to the data encapsulation principle. `Voxel` contains a number of client functions and data types that depend on other primitive objects and utilities. As a result, the overall subsystem structure was quite poor, and did not provide much help for fully understanding the architectural design of the ray tracer. In the end, we realized that the composition of `Voxel` should be delayed until after all the basic subsystems are formed, and aside from a few related primitive operations and small subsystems, interconnection strength should be used in determining the perimeter of `Voxel` (i.e., what objects and dependencies belonged to it outside of the `Voxel`'s core objects.)

In our opinion, a user who is unfamiliar with the ray tracer, but familiar with the Rigi editor, can easily build a subsystem hierarchy comparable to the one described in this article. Other members of the Rigi project repeated the described experiment and arrived at comparable subsystem hierarchies.

While the extraction of the resource-flow graphs and the computation of the exact interfaces is fast, the semi-automatic, interactive subsystem composition may take from a few hours for a 30-module system, such as the ray tracer, to a few days for a system consisting of a hundred modules. Nevertheless, considering the size of the source code for the ray tracer (approximately 12,000 lines) this semi-automatic procedure is considerably faster than a comparable manual construction of subsystem structures.

Rigi also provides incremental algorithms so that new and updated modules can easily be integrated with the current software model. Therefore, once a system is reverse engineered using Rigi, the recovered system abstractions can quickly be brought up-to-date by only processing the changed modules, and by analyzing the affected modules through exact interfaces. This process is fully automatic; components and dependencies are updated correctly without destroying the created subsystem structures. Hence, the database and the software structures can be kept current at minimal time and cost.

During the process of subsystem composition, composition strategies based on software engineering principles should be followed; it is assumed that the same principles are adhered to during software development. However, the strategies should be constantly revised and finely tuned to the different requirements at different levels of abstraction in the subsystem hierarchy. For instance, in our experiment, encapsulating data types is naturally used as a composition strategy at the lower levels of the hierarchy. Moreover, what works for one system does not necessarily work for another, and the choice of the correct strategy depends on the problem domain as well. Therefore, one should be prepared to experiment with flexible tools such as Rigi and use all available information that might help in discovering the structural and functional properties of a given system.

8 RELATED WORK

Reported module or subsystem partitioning and clustering methods typically embody some form of inter-module measures. A partitioning method based on interconnection strength was proposed by Choi and Scacchi (1990). The objective in their approach is to obtain a subsystem decomposition with minimal coupling and minimal alteration-distance among modules. They compute the articulation points and the biconnected components of the module graph and then build a composite, hierarchical directed acyclic graph by assigning a subsystem to each detected articulation point and each biconnected component. Their partitioning method is incorporated in SOFTMAN, an environment for forward and reverse engineering (Choi and Scacchi, 1991).

Kaiser, Maarek, and Perry use partitioning and clustering methods for change analysis in the Infuse system (Kaiser and Perry, 1987; Perry and Kaiser, 1988; Maarek and Kaiser, 1988). Infuse clusters the set of modules involved in a change into a hierarchy of experimental databases where the hierarchy controls the integration of changes. Their rationale is based on the premise that the probability of an interface error between two modules is proportional to the modules' interconnection strength.

Belady and Evangelisti (1982) use data bindings to form a flat module graph out of procedures. A data binding is an ordered triple (p, x, q) where p and q are procedures and x is a variable within the static scope of both p and q. Hutchens and Basili (1985) extended this approach to produce dendograms (hierarchies of modules). Selby and Basili (1988) also use cluster analysis based on data bindings to localize errors and to identify error-prone system structures.

Schwanke and Platoff (1989) recently outlined a clustering measure based on shared neighbors for their ARCH environment. They use this measure for summarizing call graphs, splitting large include files, and improving system modularity (Schwanke, 1991).

Newbery and Tichy (1990) developed EDGE, an extendible directed graph editor, which is similar to our Rigi editor. Newbery (1989) also proposed a graph-theoretic approach to the problem of reducing the complexity of a directed graph. She computes edge clusters as opposed to node clusters.

Rigi incorporates many features typically found in hypertext systems. Fletton and Munro (1988) exploit hypertext technology for (re)documenting software systems. Garg and Scacchi (1990) use a hypertext system to manage all software life-cycle documents. Instead of $(k, 2)$-partite graphs, Ossher (1984) uses *Grids* as a skeleton for representing software systems.

9 CONCLUSIONS

When composing subsystem structures, software engineers make intuitive or subjective decisions based on experience, skill, and insight which cannot and should not be fully automated.

However, subsystem composition methods are objective with respect to a given similarity measure, but usually take only one measure into account at a time. Providing an expert designer with a selected set of clustering methods through a flexible tool such as Rigi therefore presents a viable solution.

In 1990, we applied our reverse engineering methodology to a 57,000 line COBOL program, the Practice Manager by Osler Management Inc. of Victoria (Müller *et al.*, 1990). The Practice Manager is a comprehensive system for the management of physicians' practices in British Columbia, Canada. The purpose of the analysis was to build up-to-date subsystem structures, to assess the quality of the entire system with respect to ease of maintenance, and to identify subsystems that are candidates for re-engineering.

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 Facility) in Vancouver, British Columbia. The main objective for this analysis was to identify components for re-engineering; in particular, abstract data types.

We are currently analyzing a large commercial database management system in cooperation with the Centre for Advanced Studies, IBM Toronto Laboratory. The goal of this investigation is to construct current architectural diagrams at various levels of detail of this multi-million line system.

Acknowledgments

The authors would like to thank all those students and research associates who contributed to the Rigi project. In particular, Craig Sinclair designed the parsing system; Brian Corrie implemented the user interface and converted the Rigi system from C to C++; Jacek Walkowicz and Margaret-Anne Storey improved the user interface and ported the system from Sunview to OpenLook and Motif; Ken Wong served as code librarian and organized the development environment. Thanks are also due to two anonymous referees for their helpful comments and suggestions on an earlier draft of this paper and to Ann Gawman of IBM

Canada Ltd.'s Information Development Department for her editing expertise.

References

Aho, A. V., Hopcroft, J. E. and Ullman, J. D. (1974) *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

ANSI/IEEE (1983) 'IEEE Standard Glossary of Software Engineering Terminology', ANSI/IEEE Standard 729–1983.

Arnold, R. S. (1990) 'Tutorial on Software Reengineering', *Conference on Software Maintenance—1990*, (San Diego, California, November 26–29), IEEE Computer Society Press (Order number 2091).

Belady, L. A. and Evangelisti, C. J. (1982) 'System partitioning and its measure', *Journal of Systems and Software*, 2(1), pp. 23-29.

Brandes, T. and Lewerentz, K. (1985) 'GRAS: A non-standard database system within a software development environment', *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-large* (Harwichport, Mass., June 9–12), GTE Laboratories Inc., pp. 113-121.

Breuer, P. T. and Lano, K. (1991) 'Creating specifications from code: reverse engineering techniques', *Journal of Software Maintenance: Research and Practice*, 3(3), pp. 145-162.

Chikofsky, E. J. and Cross II, J. H. (1990) 'Reverse engineering and design recovery: a taxonomy', *IEEE Software* 7(1), pp. 13–17.

Choi, A. C. and Scacchi, W. (1990) 'Extracting and restructuring the design of large software systems', *IEEE Software*, 7(1), pp. 66-71.

Choi, A. C. and Scacchi, W. (1991) 'SOFTMAN: environment for forward and reverse CASE', *Information and Software Technology*, 33(9), pp. 664-674.

Corrie, B. (1990) *A Workbench for Realistic Image Synthesis,* M.Sc. Thesis, Department of Computer Science, University of Victoria.

Dunn, D. and Everitt, B. S. (1982) *An Introduction to Mathematical Taxonomy,* Cambridge University Press.

Fletton, N. T. and Munro, M. (1988) 'Redocumenting software systems using hypertext technology', *Proceedings of Conference on Software Maintenance 1988,* (Phoenix, Arizona, October 24-27), IEEE Computer Society Press (Order Number 879), pp. 54-59.

Forte, G. (1992) 'Reverse engineering tools', *CASE OUTLOOK*, March-April, pp. 5-28.

Garg, P. K. and Scacchi, W. (1990) 'A hypertext system to manage software life-cycle documents', *IEEE Software*, 7(3), pp. 90-98.

Hutchins, D. H. and Basili, V. R. (1985) 'System structure analysis: clustering with data bindings', *IEEE Transactions on Software Engineering*, SE-11(8), pp. 749-757.

Kaiser, G. E. and Perry, D. E. (1987) 'Workspaces and experimental databases: automated support for software maintenance and evolution', *Proceedings of Conference on Software Maintenance 1987*, (Austin, Texas, September 21-24), IEEE Computer Society Press (Order Number 800), pp. 108-114.

Maarek, Y. S. and Kaiser, G. E. (1988) 'Change management for very large software systems', *Proceedings of Phoenix Conference on Computers and Communications* (Scottsdale, Arizona, March 16-18), pp. 280-285.

Mata-Montero, M. (1990) *Algorithms for Treewidth-Bounded Graphs*, Ph.D. Dissertation, Department of Computer Science, University of Victoria, Victoria, B.C.

Müller, H. A. (1990) 'Verifying software quality criteria using an interactive graph editor', *Proceedings of the Eighth Annual Pacific Northwest Software Quality Conference* (Portland, Oregon, October 29-31), pp. 228–241.

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

Müller, H. A., Möhr, J. R. and McDaniel, J. G. (1990) 'Applying software re-engineering techniques to health information systems', *Proceedings of the IMIA Working Conference on Software Engineering in Medical Informatics (SEMI)* (Amsterdam, October 8-10, 1990), Editor, T. Timmers and B. I. Blum, Elsevier Science Publishers, pp. 91–110.

Müller, H. A., Tilley, S. R., Orgun, M. A., Corrie, B. D., and Madhavji, N. H. (1992) 'A reverse engineering environment based on spatial and visual software interconnection models', *SIGSOFT'92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments* (Tyson's Corner, Virginia, December 9-11), ACM Software Engineering Notes, 17(5), pp. 88-98.

Müller, H. A. and Uhl, J. S. (1990) 'Composing subsystem structures using $(k, 2)$-partite graphs', *Proceedings of Conference on Software Maintenance 1990* (San Diego, California, November 26-29), IEEE Computer Society Press (Order Number 2091), pp. 12-19.

Myers, G. L. (1975) *Reliable Software Through Composite Design*, Petrocelli/Charter, New York.

Newbery, F. J. (1989) 'Edge concentration: a method for clustering directed graphs', *Proceedings of the 2nd International Workshop on Software Configuration Management* (Princeton, New Jersey, October 24). *ACM SIGSOFT Software Engineering Notes*, 17(7), pp. 76-85.

Newbery, F. J. and Tichy, W. F. (1990) 'EDGE: an extendible graph editor', *Software—Practice & Experience*, 20(1), pp. 63-88.

Ossher, H. L. (1984) 'Grids: a new program structuring mechanism based on layered graphs', *Proceedings of the Eleventh Annual Symposium on Principles of Programming Lan-*

guages (Salt Lake City, Utah. January 15-18), ACM (Order Number 549840), pp. 11-22.

Perry, D. E. (1987) 'Software interconnection models', *Proceedings of the 9th International Conference on Software Engineering* (Monterey, CA, March 30 - April 2), IEEE Computer Society Press (Order Number 767), pp. 61-69.

Perry, D. E. and Kaiser, G. E. (1987) 'Infuse: a tool for automatically managing and coordinating source changes in large systems', *Proceedings of the ACM Fifteenth Annual Computer Science Conference* (St. Louis, Missouri), ACM (Order Number 404870), pp. 292-299.

Prieto-Diaz, R. and Neighbors, J. M. (1986) 'Module interconnection languages', *Journal of Systems and Software*, 6(1986), pp. 307–334.

Schwanke, R. W. and Platoff, M. A. (1989) 'Cross references are features', *Proceedings of the 2nd International Workshop on Software Configuration Management* (Princeton, New Jersey, October 24). *ACM SIGSOFT Software Engineering Notes*, 17(7), pp. 86-95.

Schwanke, R.W. (1991) 'An intelligent tool for re-engineering software modularity', *Proceedings of the 13th International Conference on Software Engineering (ICSE)* (Austin, Texas, May 13-17), IEEE Computer Society Press (Order Number 2140), pp. 83-92.

Tilley, S. R. (1992) 'Management decision support through reverse engineering technology', *Proceedings of CASCON'92* (Toronto, Ontario, November 9-12), pp. 319-328.

Tilley, S. R., Müller, H. A. and Orgun, M. A. (1992) 'Documenting software systems With views', *Proceedings of ACM SIGDOC'92* (Ottawa, Ontario, October 13-16), ACM (Order Number 613920), pp. 211-219.

Selby, R. W. and Basili, V. R. (1988) 'Error localization during software maintenance: generating hierarchical descriptions from source code alone', *Proceedings of Conference* *on Software Maintenance 1988* (Phoenix, Arizona, October 24-27), IEEE Computer Society Press (Order Number 879), pp. 192-197.

Uhl, J. S. (1989) *Discovering Structure in Large Software Systems,* M.Sc. Thesis, Department of Computer Science, University of Victoria, Victoria, B.C.

Wolf, A. L., Clarke, L. A. and Wileden, J. C. (1988) 'A model of visibility control', *IEEE Transactions on Software Engineering*, SE-14(4), pp. 512-520.

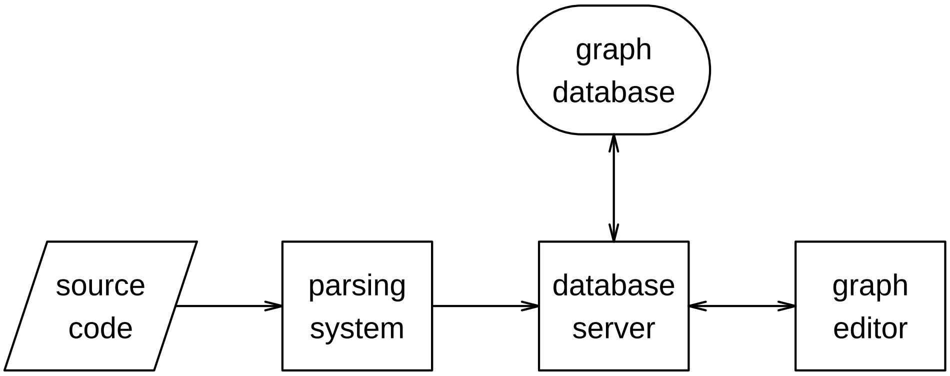

Figure 1: Architecture of the Rigi system

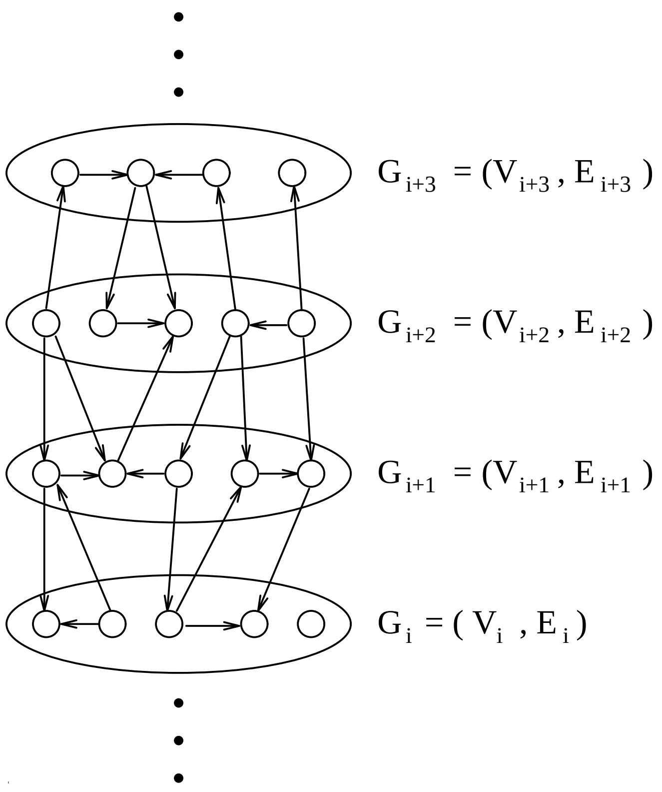

Figure 2: Directed $(k, 2)$-partite graph

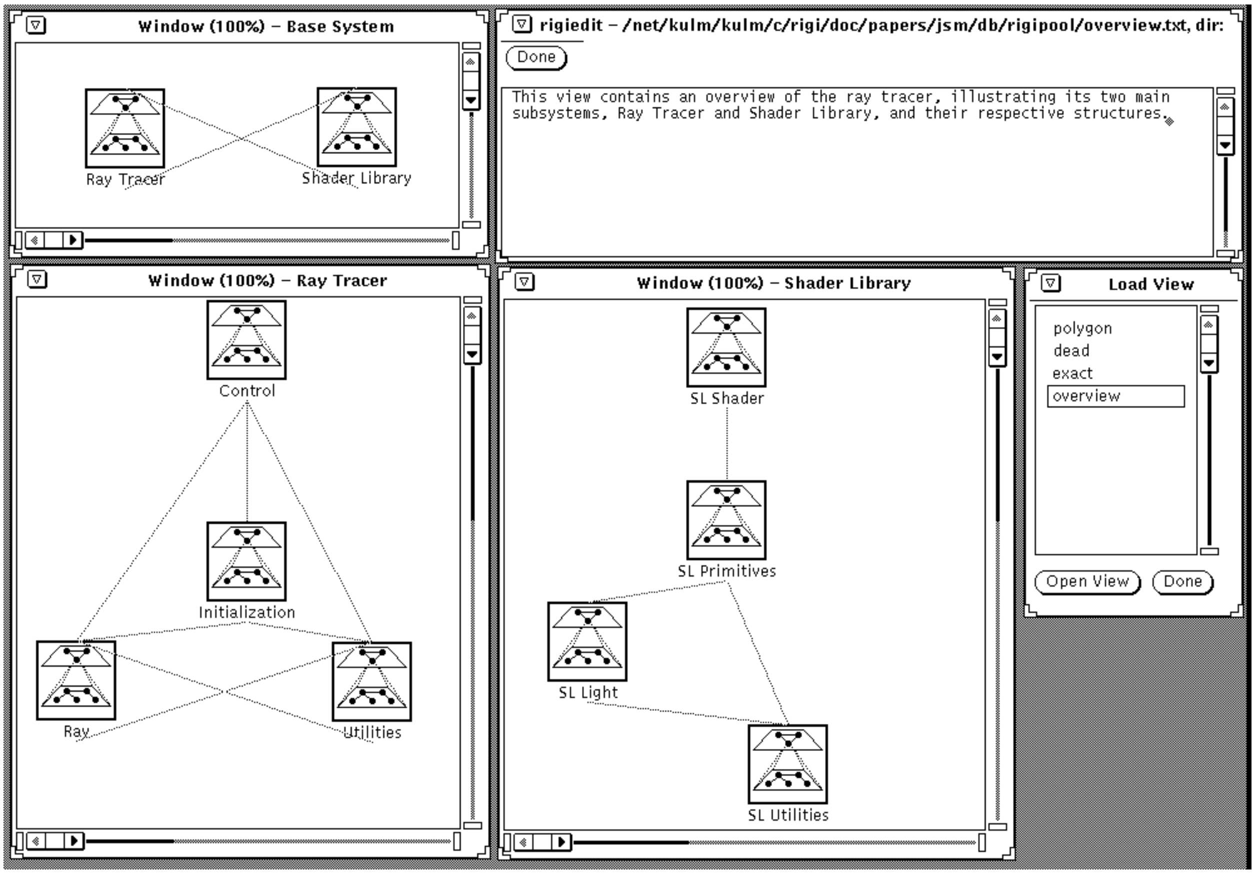

Figure 3: An overview of the ray tracer

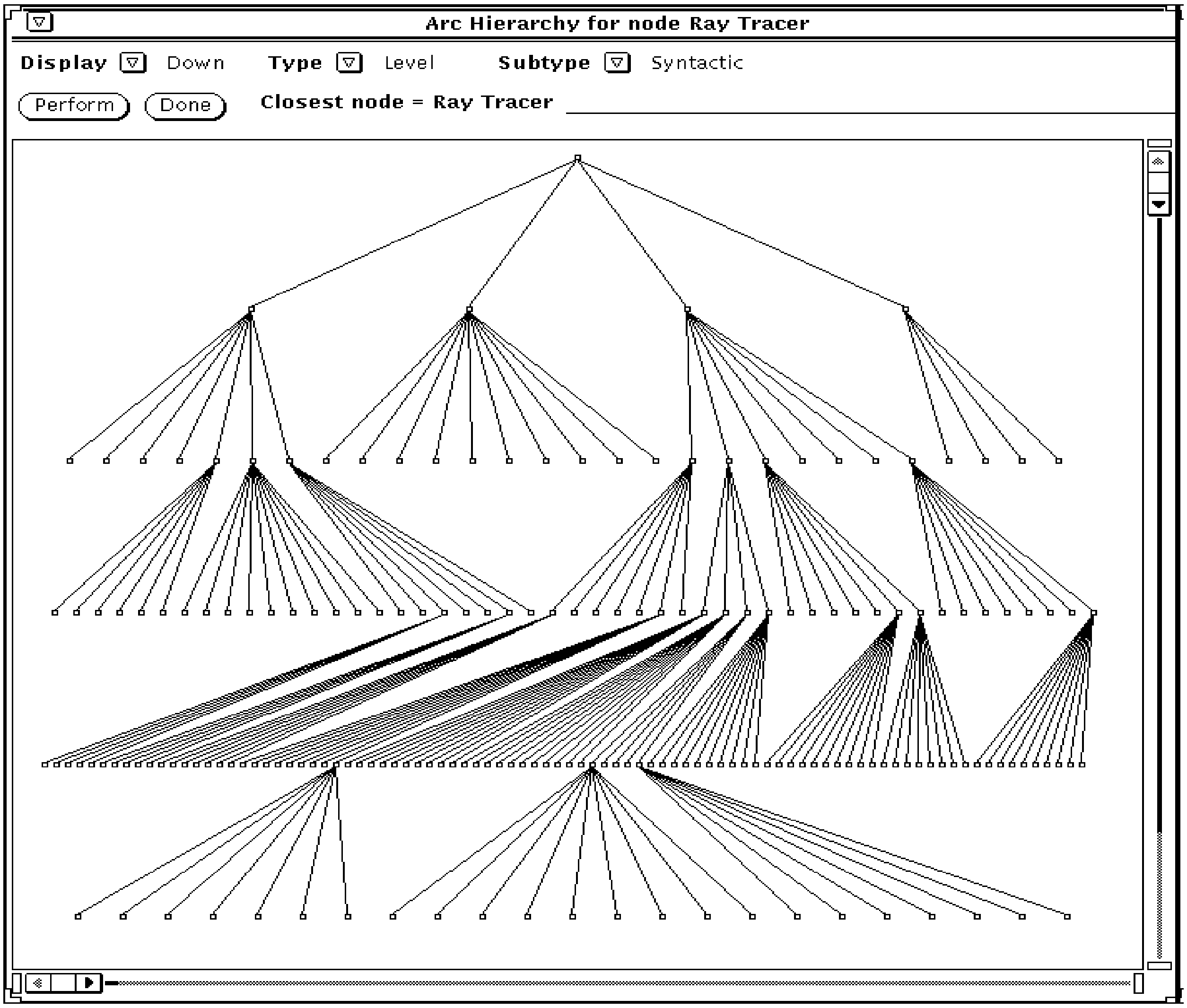

Figure 4: The subsystem hierarchy for subsystem `Ray Tracer`

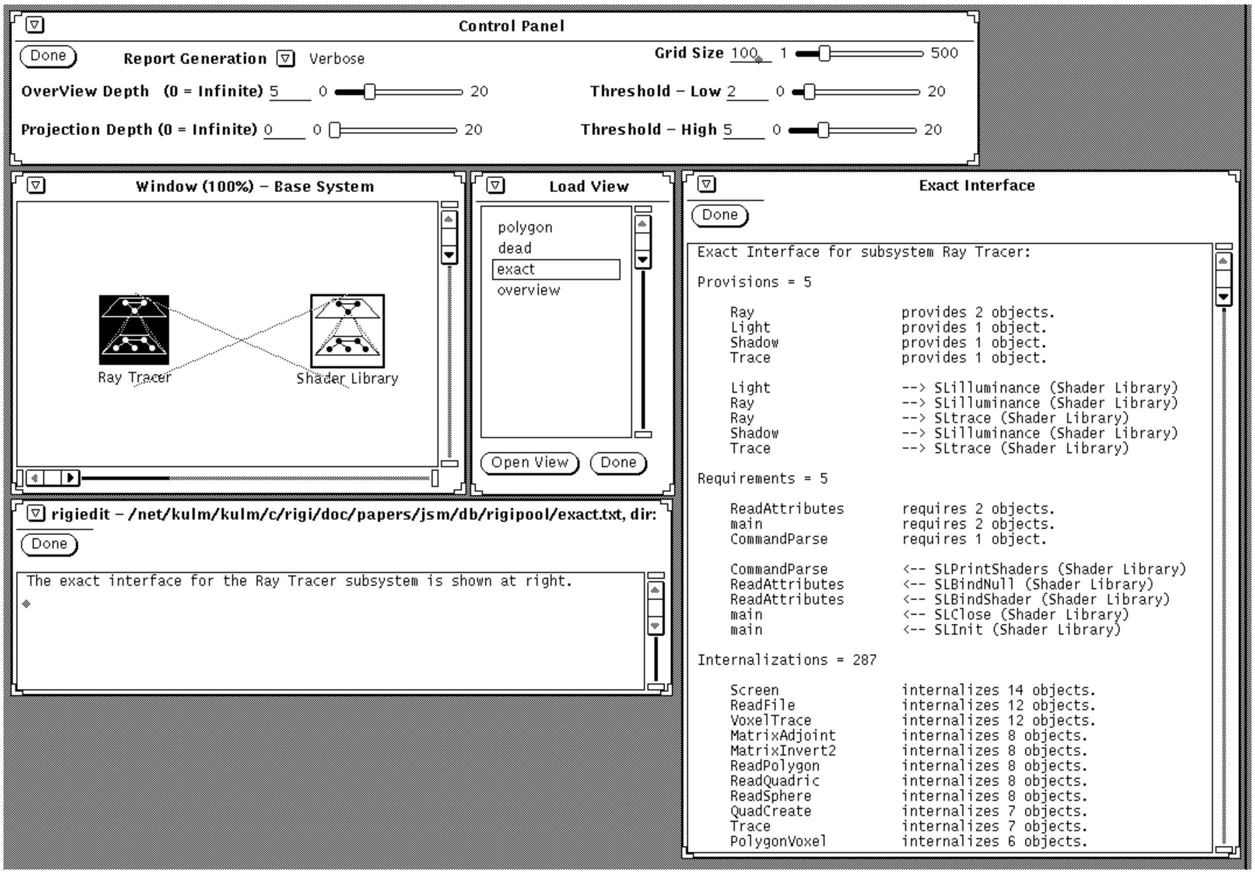

Figure 5: The exact interface for subsystem `Ray Tracer`

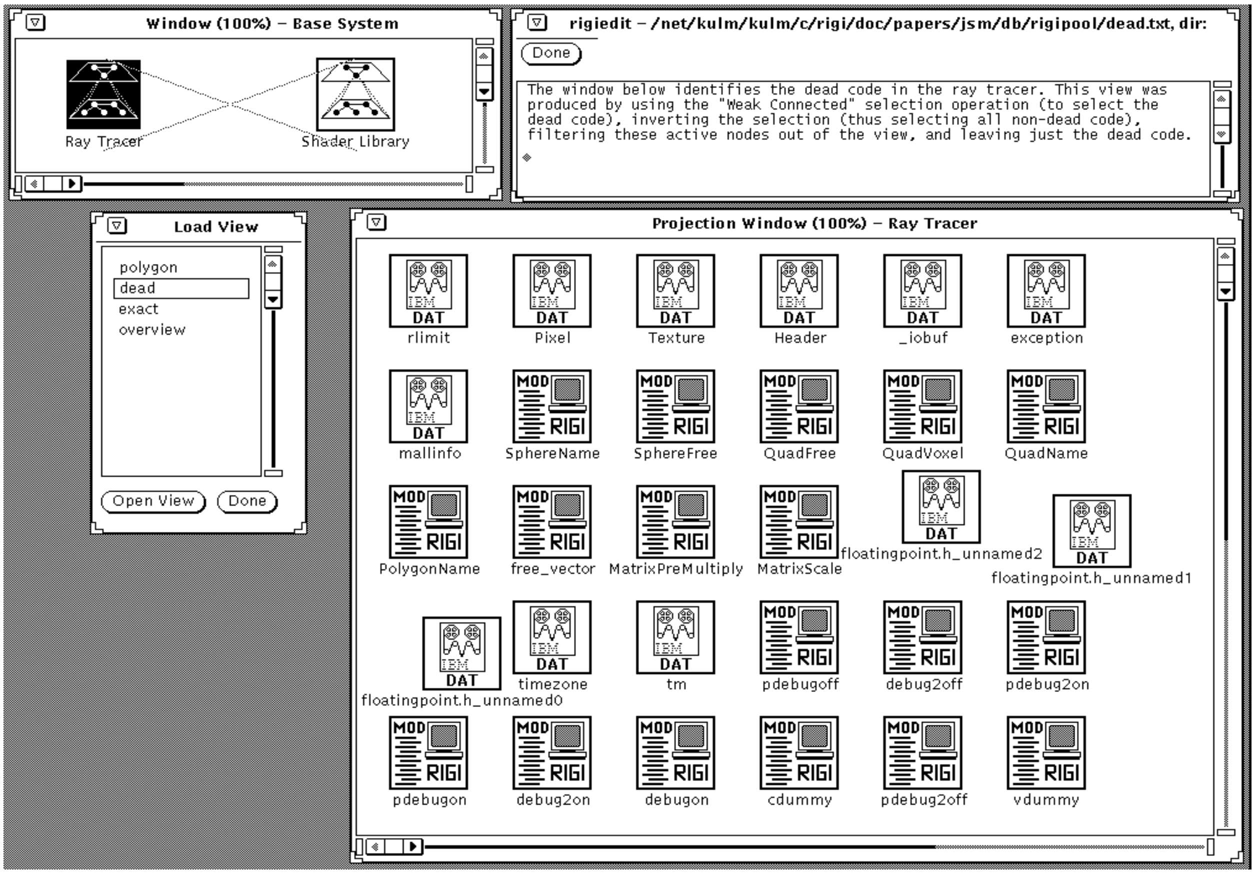

Figure 6: Dead code in subsystem `Ray Tracer`

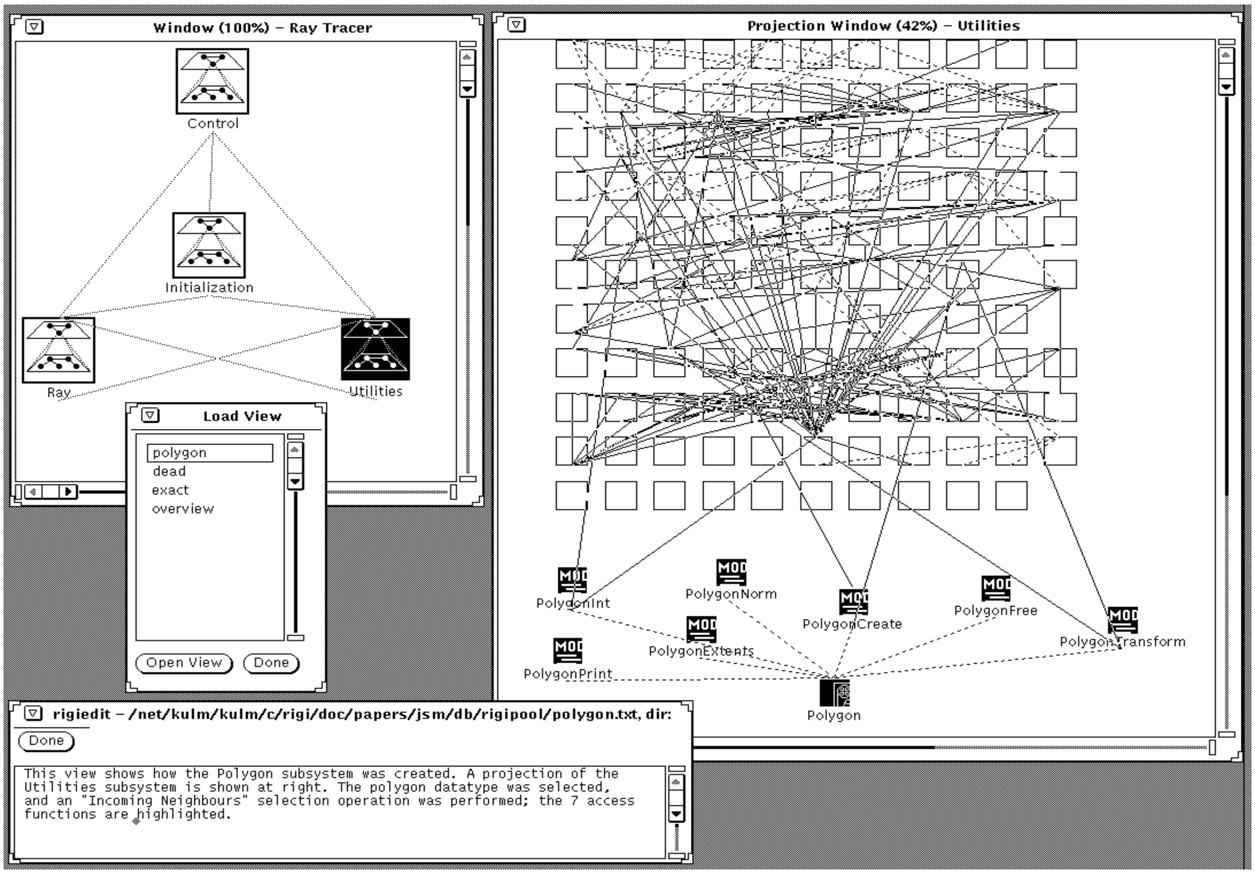

Figure 7: Abstract data type `Polygon`

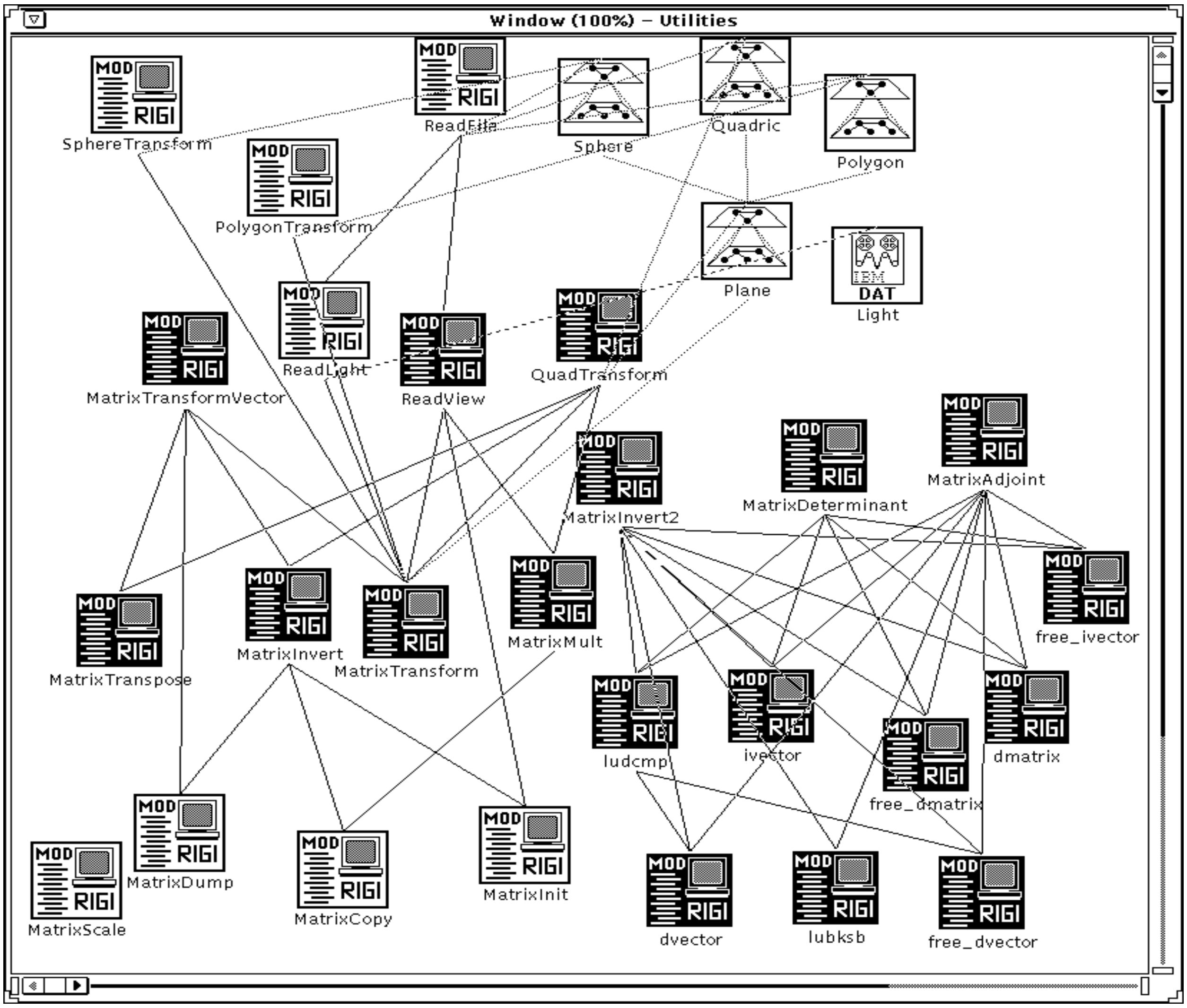

Figure 8: Subsystem identification by common clients/suppliers measures