Documenting Software Systems with Views[†‡]

Scott R. Tilley Hausi A. Müller Mehmet A. Orgun

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: {stilley, hausi, morgun}@csr.uvic.ca

Abstract

Software professionals rely on internal documentation as an aid in understanding programs. Unfortunately, the documentation for most programs is usually out-of-date and cannot be trusted. Without it, the only reliable and objective information is the source code itself. Personnel must spend an inordinate amount of time exploring the system by looking at low-level source code to gain an understanding of its functionality. One way of producing accurate documentation for an existing software system is through *reverse engineering*. This paper outlines a reverse engineering methodology for building subsystem structures out of software building blocks, and describes how documenting a software system with *views* created by this process can produce numerous benefits. It addresses primarily the needs of the software engineer and technical manager as document users.

Key words: Software documentation, reverse engineering, software maintenance.

1 Introduction

As today's software ages, the task of maintaining it becomes both more complex and more expensive. A contributing factor to this cost increase is the generally poor condition of the software. This can be partly attributed to the lack of accurate documentation, 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 thus has evolved into something different from the original [1, 2, 3, 4].

Of these deficiencies, the lack of detailed, accurate, and up-to-date program documentation is critical for software engineers and technical managers responsible for the maintenance of existing software systems. Without it, the only reliable and objective information is the source code itself [5]. Maintenance personnel must spend an inordinate amount of time attempting to create an abstract representation of the system's high-level functionality by exploring its low-level source code. With software maintenance routinely consuming upwards of 50% of a product's lifecycle and budget [6], any improvement in documenting a program's evolution and overall architecture would ease the tasks of the maintenance team. One way of producing accurate documentation for an existing software system is through *reverse engineering*.

Reverse engineering is the process of extracting system abstractions and design information out of existing software systems for maintenance, re-engineering, and reuse. This process involves identifying software artifacts in a particular representation of a subject system via mental pattern recognition by the reverse engineer, and the aggregation of these artifacts to form more abstract system representations. A reverse engineering methodology is outlined in [7] for building subsystem structures out of software building blocks. This methodology is supported by *Rigi*,[1] a system and framework for analyzing large software systems [8].

[†]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.

Copyright © 1992 Association for Computing Machinery, Inc. Reprinted, with permission, from *SIGDOC '92: Proceedings of the 10th International Conference on Systems Documentation*, (Ottawa, Ontario; October 13-16, 1992), pp. 211–219. ACM Order Number 613920.

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

We have concentrated on investigating algorithms, techniques, and tools for the composition, analysis, and presentation of subsystem structures. In particular, we have focused on methods and algorithms for summarizing software structures by building hierarchies of subsystems [9]. The generated structures embody *visual* and *spatial* information that serve as organizational axes for the exploration and presentation of the composed subsystem structures. These structures can be augmented with *views*: hypertext (and potentially multimedia) annotations that highlight different aspects of the software system under investigation. 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.

This paper shows how documenting a software system with views created through reverse engineering can be used to provide numerous benefits. The next section describes some of the deficiencies of traditional methods of software documentation. Section 3 explains why visual and spatial information play a crucial role in program understanding, and how Rigi uses views for software documentation. Section 4 highlights some of the benefits that reverse engineering and documenting a software system with views can bring. Section 5 reports on some of our early experience of applying and using Rigi on real-world software systems.

2 Deficiencies in traditional documentation techniques

The documentation for a typical project must serve a diverse group of readers. Two very distinct audiences are authors and end-users. The former includes both software engineers and technical managers: they require an internal view of the system to understand how it actually works. The latter requires an external view, which may include a description of the program's functionality and a tutorial on how to use it. While the external view of a program may change very little over its lifetime—even if its functionality is enhanced—the program's internal architecture might change dramatically. In an ideal situation, the internal documentation would be continually updated to reflect frequent changes in the source code. In practice, documentation and source code are not always synchronized. This section concentrates on documentation deficiencies as they relate to the first group: authors.

High-quality documentation is widely recognized as an important part of any software system [10]. It has a significant effect on program understanding. Software engineers and technical managers base many of their project-related decisions on their understanding of the architecture of the software systems they are responsible for. While they rely on original design documents, maintenance histories, and experienced project members (if they are available) to help them understand how a program works, internal documentation is often their primary source of information. Hence, the most obvious way to support program comprehension is to produce and maintain adequate documentation [11]. Regrettably, documentation for most older programs is sadly out-of-date.

One reason for the general lack of quality documentation may be that preparing software documentation is usually one of the last activities to take place during program development. Software documentation seems to be of little interest to the development community; it is the "poor stepchild" of most software development efforts [12]. If is often considered something to be put off until the last possible moment; in many cases, it is postponed indefinitely. While there do exist model programmers who carefully document design and implementation decisions while developing a piece of code, usually documentation is tacked on as an after-thought. Other pressures of day-to-day development seem to take precedence over documentation. These may include the following:

- Getting versions of the software to the testing department or customer beta-test sites.
- Managers feeling that the project's schedule is slipping and hence all "extras" must be eliminated to meet the deadline.
- Programmers moving on to new projects.

Another contributing factor to the inadequacy of most software documentation is the age and maintenance history of the software. Managing complexity and supporting evolution are two fundamental problems with large-scale software systems [13]. Keeping the documentation in synchronization with the program's evolution is given a lower priority than getting the code fixed or enhanced, and is often left "until later" — which may mean never at all. The philosophy of "get it working quickly," without keeping the documentation up-to-date, is doomed to failure. As changes to the program continue, the original documentation quickly becomes useless. A direct consequence is the increased time required by software engineers to understand the system. The short-term gain of getting the program fixed and shipped as soon as possible will be overwhelmed by the long-term increase in costs.

It is a common misconception that maintenance is easier than new development. Consequently, new programmers are often placed in maintenance situations when they start a new job. Since they have not been involved in the product's development prior to maintaining it, they become heavily reliant on support documentation.

Software documentation has many different audiences. Different levels of documentation are required for the casual user of a program, for the developer familiar with the code, for the maintainer unfamiliar with the system, for testers and technical writers trying to understand its functionality, and project management personnel looking for "the big picture": an external view of the system's architecture and history [14, 15, 16]. A person browsing a file attempting to get an overview of its functionality may be satisfied with short, descriptive comments concerning algorithms, data structures, and design decisions that affected the file. A fellow developer may require detailed information about a particular function.

With traditional source-code documentation, it is very difficult to provide different levels of information to such a diverse community. Although each person may have different objectives, everyone will see the same thing: inline and block commentary with the source code, and original design documents and maintenance logs.[2] It would be far better to be able to provide each user with a view of the program that suited their needs. Rigi accomplishes this goal with *views*.

3 Using views for program documentation

The Rigi system uses views to direct the user's focus on visual data and to guide the exploration of spatial data to support program documentation and understanding. Such a view represents a particular state and display of a software model. 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 [17]. The spatial component constitutes information about the relative positions of the meaningful parts of a software structure, whereas the visual component supplies information about how a software structure looks. The reverse engineer exploits both spatial and visual information when identifying components and building abstractions.

[2] Assuming these documents exist at all.

The Rigi editor is an interactive graph editor that allows the reverse engineer to maintain software structures stored in a graph database [18]. It supports a variety of operations to manipulate visual and spatial representations of graph structures. Among Rigi's most important visual representations are *overviews* and *projections*. System overviews can be depicted in various patterns and arrangements using graph editor operations such as grouping, scaling, layout, and filtering. Projections are overviews at different levels of detail.

Different views of the same software model can be used to address a variety of target audiences and applications. Views can be collected into sequences to form related sets of documentation, to represent guided tours for tutorial purposes, to highlight system components that need to be analyzed and understood when performing specific maintenance or re-engineering tasks, to summarize change, impact, or performance analyses, or to annotate critical sections with measurements that serve as input to decision-making (for example, project priorities or personnel assignments).

Figure 1 shows an overview of tbl: a document formatting preprocessor for *troff* which enables tables to be typeset in the UNIX[3] system [19]. The tbl program is written in C and consists of 22 modules. 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, are described in Section 4.

4 Benefits

Reverse engineering a software system, and subsequently documenting it using views, can provide numerous benefits. The views can be used to aid management decisions, recover lost information, and improve system comprehension. Each of these benefits are discussed below.

4.1 Aiding management decisions

One of the biggest advantages of reverse engineering a software system with Rigi can be realized by management personnel. Project management and planning

[3] UNIX is a trademark of AT&T Bell Labs.

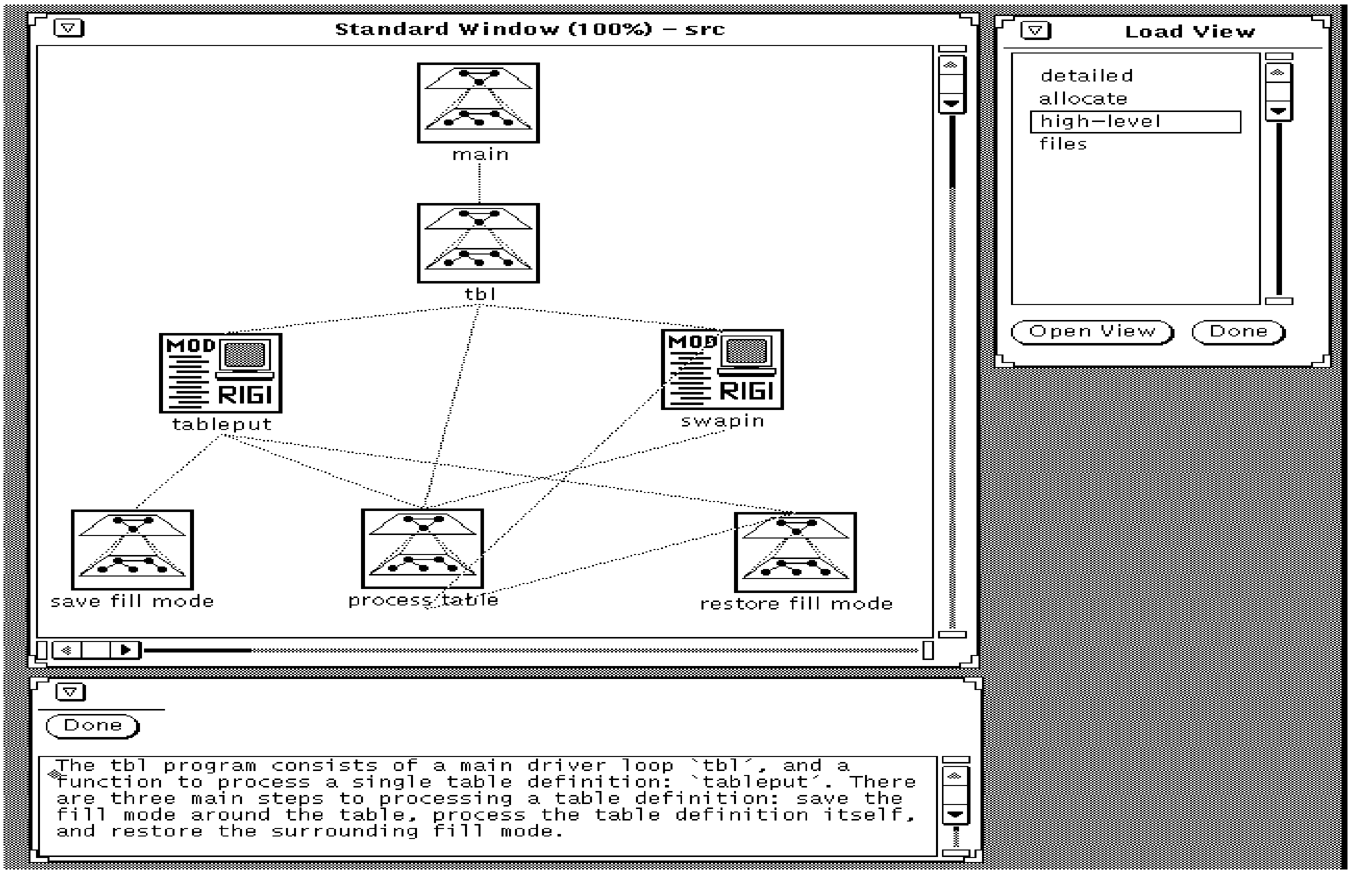

Figure 1: A high-level view of the tbl program's architecture

at most corporations is a complicated process. The software systems they are responsible for exist in various life-cycle stages: new product development, testing, maintenance of existing code, and different versions. Management personnel must also manage the human element of the project: identify the strengths of team members, allocate resources based on various needs (both personal and financial), incorporate new personnel into the project, and compensate for the departure of experienced staff. Other considerations include funding, experience and talents of the people available, schedules, effect on other products and development groups, and market analysis. All of these things make the task of management very difficult. This problem is exacerbated when the complexity of the project, both technical and organizational, threatens to overwhelm even the most prepared managerial personnel.

Because of this complexity, many managers rely on input from other sources to help make their decisions. Since they may lack in-depth technical knowledge of the products they are managing, they must rely on data provided by members of their team, "gut" feelings, and experience. For these reasons, software analysis tools such as Rigi can aid the project manager in making important management decisions that will affect the outcome of the project. Views can aid in making decisions such as where to allocate precious funds, where to place key personnel, and where to concentrate effort for maximum pay-back by exposing system structure and module dependencies. The graphical representation of the system makes *central* and *fringe* components immediately obvious. One can then tailor activities such as testing, monetary and personnel allocation, and subsequent development efforts on the components desired. Dead code is quickly identified and can be eliminated from the system, thereby reducing overall complexity and maintenance.

The view shown in Figure 1 can be used by managers as a guide to the high-level architecture of the tbl program. The view in Figure 2—which depicts the same program but at a different level of detail—might be used to assign work to personnel in areas of the system best suited to their knowledge and experience, based on the visual information provided by the spatial relationships of central and fringe components. The highlighted nodes are central components, hence modifying them should be done with care, preferably by experienced personnel.

4.2 Recovering lost information

Reverse engineering can produce consistent and accurate documentation. As large systems evolve over their product life-cycle, information concerning the original design is lost. Even if the system was designed using modern software engineering principles of modularization and information hiding, the original design becomes compromised during maintenance. "Bug" fixes and enhancements that seem small at the time soon resemble patches instead of a smooth extension of the original code. A side-effect of these changes is that documentation is usually not kept up-to-date.

Even worse is documentation that no longer reflects reality; the code has changed but the documentation has not. One often relies heavily on programmers who know the system intimately, or one invests substantial amounts of time for maintainers to explore and learn the system. Given time, most programmers will attempt to keep in-line documentation and source code synchronized, but project work books and higher level design documents are rarely updated to reflect maintenance. If they are, the updates resemble appendices to the original, and the documentation quickly becomes difficult to follow. The reverse engineering facilities provided by Rigi allows one to produce an accurate "operational" design document describing the architecture of the software system's current state—not that of the original system before numerous maintenance changes were made.

Many end-user documents are becoming available online, either as a replacement of, or in addition to, traditional hardcopy manuals. For existing systems, work is also going on to move the "traditional" documentation from hardcopy to online and hypertext mediums (for example, [20]). The methodology presented here provides a two-fold benefit that is analogous to the modernization of end-user documentation. Firstly, the reverse engineering process produces accurate and up-to-date documentation; it does not directly rely on existing source-code annotations. Secondly, the views created are not just textual; they can serve as hypertext documentation that augments traditional source-code commentary. However, there is no reason to limit the view concept to textual and two-dimensional graphical images. The online documentation produced by the reverse engineer can (in theory) be true multimedia, including graphics, images, voice, and even video [21], in which the original software engineer explains the program.[4]

New software development projects may use and produce a variety of technical documents, such as design specifications, performance goals, functional specifications, design decisions, and maintenance logs. These may be found in-line with the source code, as traditional hardcopy documents, or (in the most modern systems) online in various hypertext and multimedia formats. Unfortunately, older software systems rarely provide such a wide range of documentation. Typically all that is available is a single document that is used to represent the entire system.

However, during reverse engineering, a variety of documents and graphical representations of the system can be generated by Rigi. These views of the system can be saved and replayed at a later date, serving as tutorials for other team members, as operational design documents, or as system overviews for management personnel and external documentation. For example, a more detailed view describing the three high-level susbsytems of the `tbl` program is shown in Figure 3. This view might be used by the maintainer responsible for this part of the system; it provides a description (both graphical and textual) of the subsystem's structure. Hence, the programming team can rely less on chief (or original) programmers—who may not be available—and more on automated tools to provide them with the knowledge they need to better understand the system.

Reverse engineering the system can recover some of the original design decisions. The salvaging of "corporate knowledge" from earlier projects can greatly improve the quality of new projects, as well as reduce cycle time from design to delivery. Full design recovery is the next step after reverse engineering, and semi-automatic tools such as Rigi are a step in this direction.

4.3 Improving system comprehension

When faced with the task of maintaining a software system, system comprehension is typically the most important prerequisite. Educating new members of the development team is a never-ending and important aspect of most software development companies. These education sessions are often held informally, by having new employees sit down with experienced developers who then explain the code to them. Rigi offers a semi-automated alternative to this education problem.

For example, visible in the bottom-right corner of Figure 2 is the `Load View` window. This window contains

[4] Multimedia program documentation is an area of research we are planning on pursuing in the future.

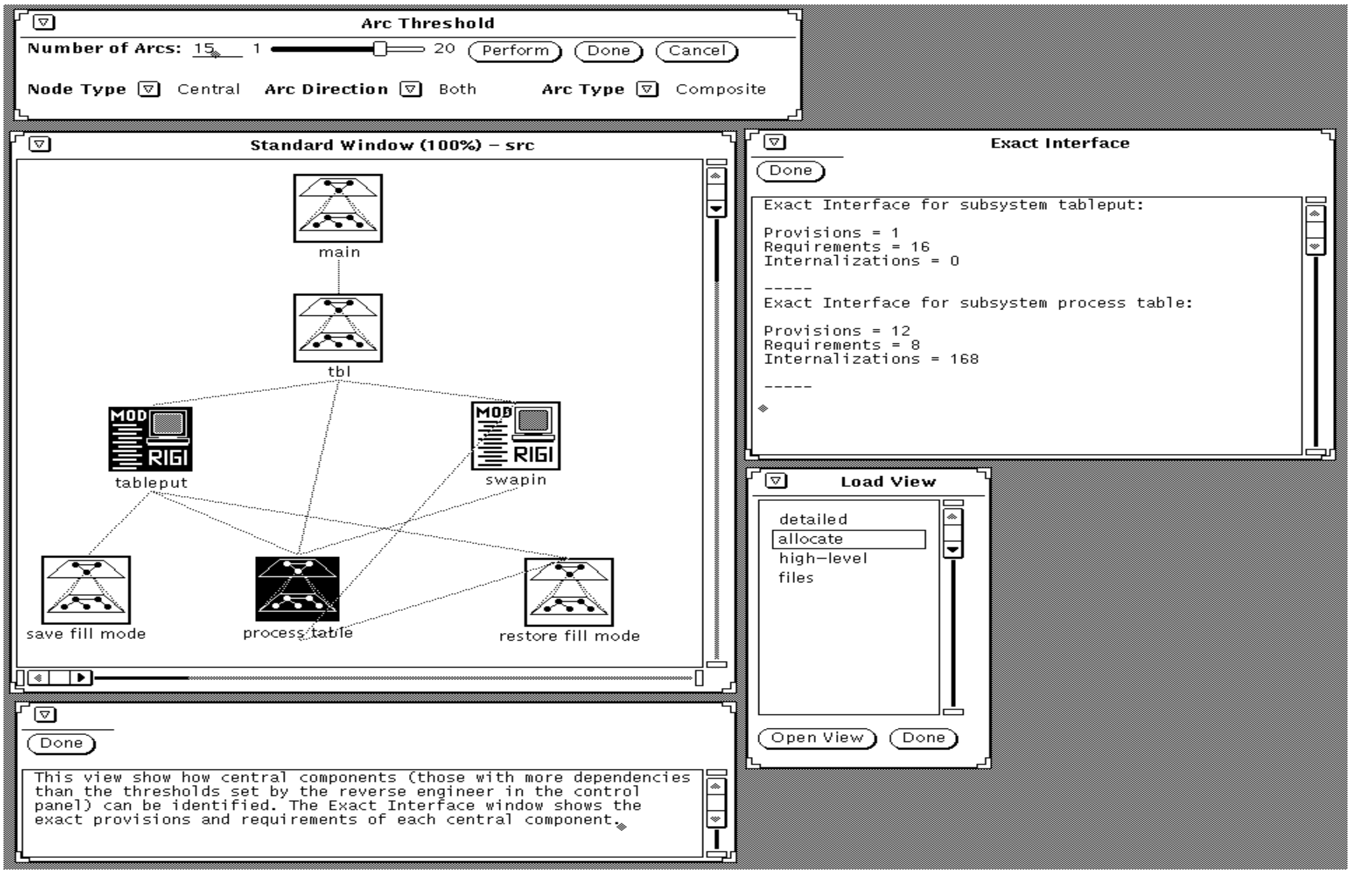

Figure 2: Allocating personnel based on component location

a scrollable list of views that may be investigated at the user's convenience. Taken together, these views form an introductory tutorial describing all of the tbl program's architecture and call structure. Such a tutorial might be used by new personnel as an aid in program understanding.

Graphical representations have long been accepted as comprehension aids. The Rigi system allows the software system to be viewed in a variety of ways. This graphical representation of information can greatly increase one's understanding of the software system. The new employees can work with the code and explore the entire system on their own, using the spatial and visual information, views, and linked documentation [22] to guide them in understanding the software system. As they gain knowledge about the system from examination of the source code and the tutorial views, they can record this information by creating new views, either for themselves or for the whole maintenance team. Rigi allows views to be either *local* or *global*, which enables the user to save views of the software system that they find useful for their particular task, yet still be able to look at different views of the same software system created by other members of the team.

Rigi also provides textual information (for example, software quality measures [23]) that augment the graphical displays. The same mechanism can be used to present an overview (or a detailed discussion) of some or all of the software system to other management personnel, other departments, or other development teams. In this way, a high-level understanding of the system is disseminated throughout the organization.

Each user will have his/her own requirements for system comprehension. Managerial personnel might use the high-level view shown in Figure 1. Software engineers unfamiliar with the system might use the tutorial views as shown in Figure 3. Experienced personnel which require detailed knowledge of subsystem architecture might also use some of the views available in the tutorial, or a lower-level view describing the hierarchical structure of any subsystem. Figure 4 depicts just such a view: the arc hierarchy of the `process table` subsystem.

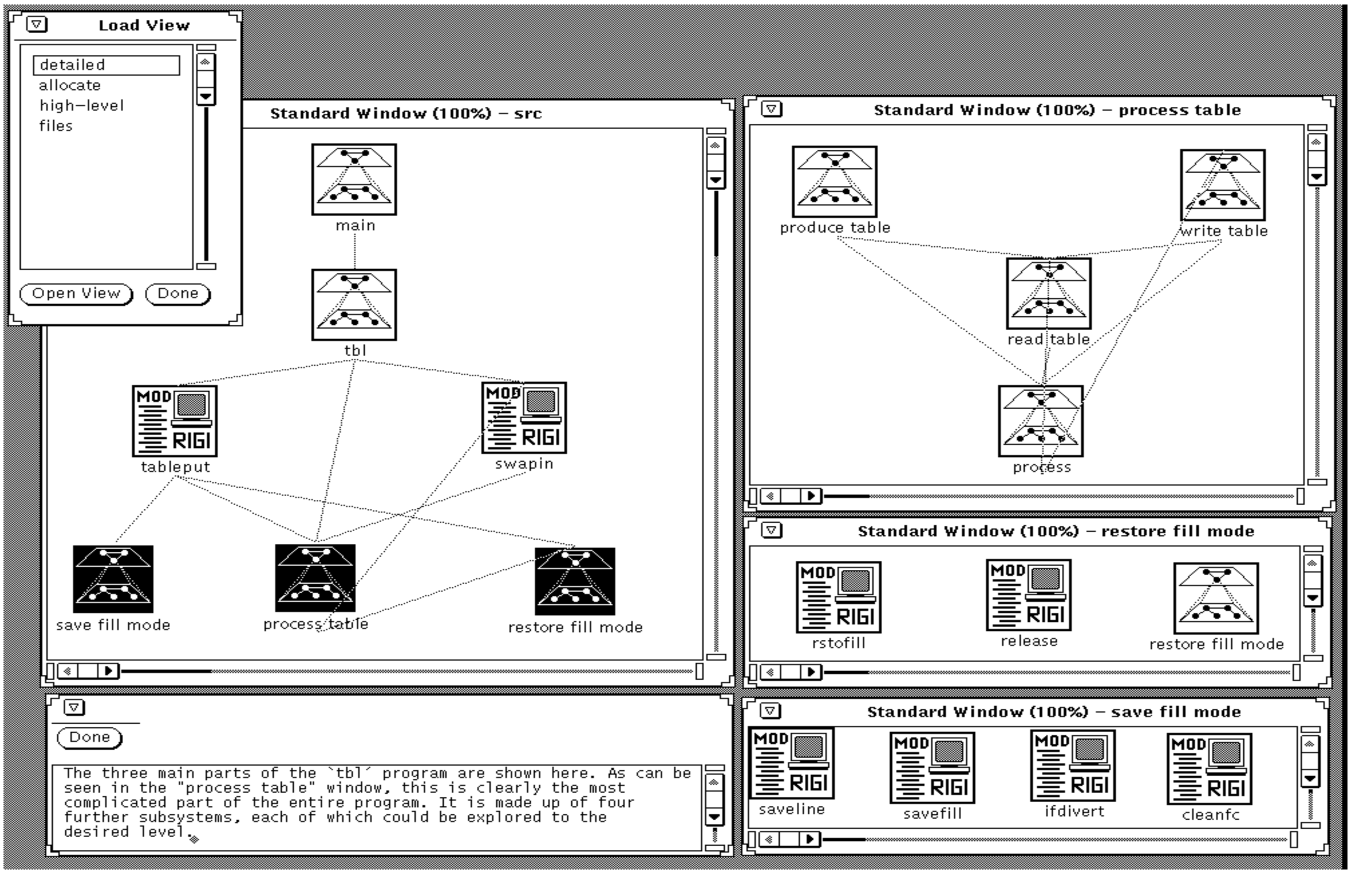

Figure 3: A detailed view of the three main subsystems

It is not just time that is spent (re)learning the system. The monetary cost of understanding software is significant, and it is multiplied every time a new person must learn the system anew. The views generated by reverse engineering can greatly reduce the overall cost of software by lessening the time required to understand the system. When a programmer is assigned a particular maintenance task, often he/she has little knowledge of the overall system design. A system such as Rigi can provide (sub)system overviews at various levels of detail.

5 Conclusions

Rigi is a versatile framework for analyzing large software systems. Many of its capabilities can be used to document existing software systems for program understanding and maintenance purposes, and by management personnel to support some of the complex decisions they face in project management. These include resource allocation, personnel placement, system comprehension, investigations into reuse potential, and information recovery.

We have successfully applied our reverse engineering methodology to several real-world software systems. In 1990, we analyzed the *Practice Manager*: a 57,000 line COBOL program by Osler Management Inc. of Victoria [24]. It is a comprehensive software system for the management of physician's practices in British Columbia. The main 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.

In 1991, we analyzed an 82,000 line C program for the isotope separator experiment at TRIUMF in Vancouver. The main objective of the 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.

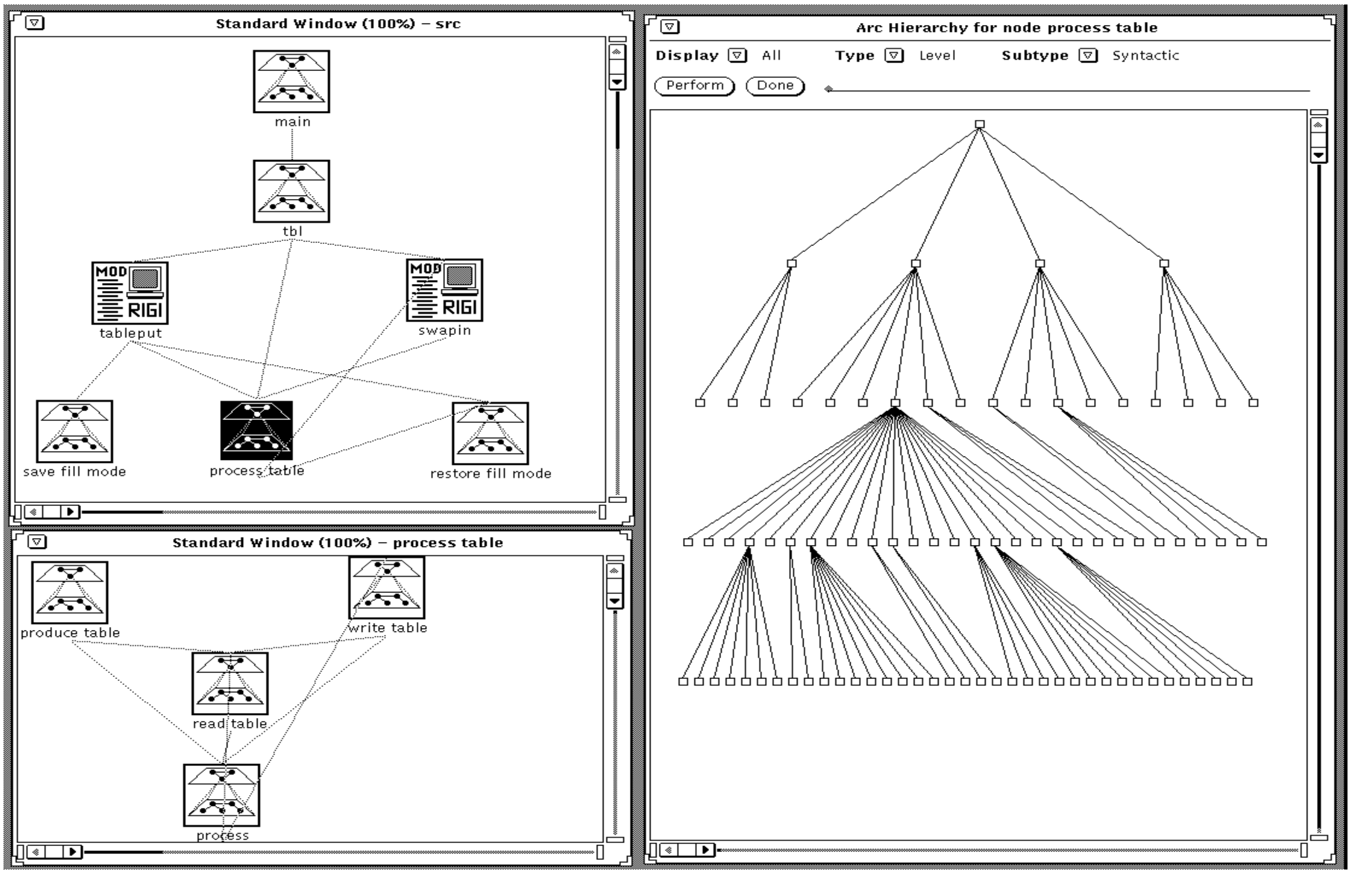

Figure 4: The arc hierarchy for the `process table` subsystem

References

[1] Girish Parikh and Nicholas Zvegintzov, editors. *Tutorial on Software Maintenance.* IEEE Computer Society Press, 1983.

[2] Peter Freeman, editor. *Tutorial: Software Reusability.* IEEE Computer Society Press, 1987.

[3] Will Tracz, editor. *Tutorial: Software Reuse: Emerging Technnology.* IEEE Computer Society Press, 1988.

[4] David H. Longstreet, editor. *Tutorial: Software Maintenance and Computers.* IEEE Computer Society Press, 1990.

[5] Nigel T. Fletton and Malcolm Munro. Redocumenting software systems using hypertext technology. In *CSM'88: Proceedings of the 1988 Conference on Software Maintenance,* (Phoenix, Arizona; October 24-27, 1988), pages 54–59. IEEE Computer Society Press (Order Number 879), October 1988.

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

[7] H.A. Müller, B.D. Corrie, and S.R. Tilley. Spatial and visual representations of software structures: A model for reverse engineering. Technical Report TR-74.086, IBM Canada Ltd., April 1992.

[8] Hausi A. Müller. *Rigi – A Model for Software System Construction, Integration, and Evolution based on Module Interface Specifications.* PhD thesis, Rice University, August 1986.

[9] 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).

[10] Jane E. Huffman and Clifford G. Burgess. Partially automated in-line documentation (PAID): Design and implementation of a software maintenance tool. In *CSM'88: Proceedings of the 1988 Conference on Software Maintenance,* (Phoenix, Arizona; October 24-27, 1988), pages 60–65. IEEE Computer Society Press (Order Number 879), October 1988.

[11] Johannes Sametinger. A tool for the maintenance of C++ programs. In *CSM'90: Proceedings of the 1990 Conference on Software Maintenance,* (San Diego, California; November 26-29, 1990), pages 54–59. IEEE Computer Society Press (Order Number 2091), November 1990.

[12] L.D. Landis, P.M. Hyland, A.L. Gilbert, and A.J. Fine. Documentation in a software maintenance environment. In *CSM'88: Proceedings of the 1988 Conference on Software Maintenance,* (Phoenix, Arizona; October 24-27, 1988), pages 66–73. IEEE Computer Society Press (Order Number 879), October 1988.

[13] Frederick P. Brooks Jr. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[14] Frederick P. Brooks Jr. *The Mythical Man-Month.* Addison-Wesley, 1982.

[15] P. J. Brown. Interactive documentation. *Software — Practice and Experience*, 16(3), March 1986.

[16] Duane Ressler and Dee Stribling. Designing and prototyping a portable hypertext application. In *SIGDOC'90 Conference Proceedings*, pages 88–94, October 1990.

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

[18] 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).

[19] M.E. Lesk. Tbl — A program to format tables. Technical report, AT&T Bell Laboratories, October 1986.

[20] Vicki Coleman. Hardcopy to hypertext: Putting a technical manual online. In *Proceedings of SIGDOC'91, (Chicago, IL, Oct. 10-12)*, pages 67–72, October 1991.

[21] Frank A. Cioch. An audiovisual document for software maintenance. In *Proceedings of the IEEE 1988 Conference on Software Maintenance*, pages 390–394, October 1988.

[22] Scott R. Tilley and Hausi A. Müller. INFO: A simple document annotation facility. In *Proceedings of SIGDOC '91: The 9th Annual International Conference on Systems Documentation,* (Chicago, Illinois; October 10-12, 1991), pages 30–36, October 1991.

[23] 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.

[24] 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.