Management Decision Support Through Reverse Engineering Technology†‡

Scott R. Tilley

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@csr.uvic.ca`

Abstract

Managers of large software systems face enormous challenges when it comes to making informed project-related decisions. They require a high-level understanding of the entire system *and* in-depth information on selected components. Unfortunately, many software systems are so complex and/or old that such information is not readily available. Reverse engineering—the process of extracting system abstractions and design information from existing software systems—can provide some of this missing information. This paper outlines how a software system can benefit from reverse engineering, and describes how management personnel can use the information provided by this process as an aid in making informed decisions related to large software projects.

†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, the Science Council of British Columbia, the University of Victoria, and IBM Canada Ltd.

‡This paper is based on material contained in IBM Technical Report TR-74.086.

Reprinted from *Proceedings of CASCON '92*, (Toronto, Ontario; November 9-11, 1992), pp. 319-328.

1 Introduction

Project management, like software maintenance, cannot be performed without a sufficient understanding of the entire software system. This is a difficult task for software that is 10-25 years old and generally in poor condition. Contributing factors include the lack of accurate documentation, the sheer size of the system, the unstructured programming methods used in the system's design, the fact that the original system designers, managers, and programmers may no longer be available, and the complication that the software has been changed several times since its first release, and thus has *evolved* into something different than the original [4].

This problem is exacerbated for management personnel because they may lack in-depth technical knowledge of the product(s) they are managing. They rely on data provided by senior members of their department, "gut" feelings, and experience. Anything that can increase their understanding of the software system(s) they are responsible for and aid them in making important project-related decisions—such as where to allocate precious funds, where to place key personnel, and where to concentrate effort for maximum pay-back—would be beneficial. Reverse engineering is one way of doing this.

Reverse engineering is the process of extracting system abstractions and design information from existing software systems. This information can then be used for subsequent development, maintenance, re-engineering, and project management purposes. This process involves the identification of software artifacts in the subject system, and the aggregation of these artifacts to form more abstract system representations. Through reverse engineering, the overall structure of the subject system can be determined and some of its architectural design information recovered. An approach to reverse engineering based on building hierarchies of subsystem structures out of software building blocks is outlined in [9]. This approach is supported by *Rigi*[1] [15], a versatile system and framework under development at the University of Victoria for discovering and analyzing the structure of large software systems.

This paper describes how a software system can benefit from reverse engineering. In particular, it focuses on how software analysis tools such as Rigi can be used to aid management decisions related to large software projects. The next section briefly outlines the reverse engineering process. Section 3 describes how the information produced by this process can be used to aid in management decisions. Finally, Section 4 reports on some of our early experience of applying and using Rigi on real-world software systems.

2 The reverse engineering process

The goal of this paper is to describe how one uses the information produced by reverse engineering a software system, not to detail the reverse engineering process itself. Nevertheless, some background on the reverse engineering process helps one understand how the generated information is used. A survey of several state-of-the-art program understanding techniques is given in [21]. A description of our reverse engineering environment can be found in [13].

The process of *reverse engineering* a subject system involves two distinct phases [1]:

1. The identification of the system's current components and their dependencies.

2. The extraction of system abstractions and design information.

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

During the process of reverse engineering, the subject system is not altered, although additional information about it is generated. In contrast, the process of re-engineering typically consists of a reverse engineering phase, followed by a forward engineering or re-implementation phase which alters the subject system.

In our approach, the first phase of the reverse engineering process—the extraction of software artifacts—is automatic and language-dependent. It essentially involves parsing of the subject system and storing the artifacts in a repository. Some of our early work resulted in a graph model for software structures and a graph editor supporting the model [11]. By software structure we mean a collection of artifacts that software engineers use to form mental models when designing, implementing, integrating, inspecting, or analyzing software systems. Artifacts include software *components* such as procedures, modules, subsystems, and interfaces; *dependencies* among components such as supplier-client, composition, and control-flow relations; and *attributes* such as component type, interface size, and interconnection strength. The artifacts are stored in an underlying database specifically designed to represent graph structures [3]. The Rigi graph editor allows the users to edit, maintain, and explore the objects stored in the repository.

Our approach to the second phase is semi-automatic and features language-independent subsystem composition methods which generate hierarchies of subsystems [14, 10]. Subsystem composition is the process of constructing composite software components out of building blocks such as variables, procedures, modules, and subsystems. Hierarchical subsystem structures are formed by imposing equivalence relations on the resource-flow graphs of the source code. These relations embody software engineering principles concerning module interactions such as *low coupling* and *strong cohesion* [18]. We have also formulated software quality criteria and measures based on exact interfaces and established software engineering principles to evaluate the resultant subsystem structures [16, 17, 20]. These measures are not meant to be absolute; they are used to judge the relative merits of one system decomposition with respect to another.[2]

The generated structures embody *visual* and *spatial* information which serve as organizational axes for the exploration and presentation of the composed subsystem structures. These structures can be augmented with *views*: textual (and potentially hypermedia) annotations that highlight different aspects of the software system under investigation [25]. 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.

3 Management decision support

If one views maintenance as "reuse-oriented software development" [2], reverse engineering can benefit everyone involved in software production (e.g., maintainers, developers, documenters, managers, and testers). There have been several areas identified as critical to improving software maintenance and (re)development, and *recapture* technologies are one of them. Simply put, recapture technologies attempt to recover the original design in an existing software system by using reverse engineering and various program understanding tools. This knowledge can then be reused for further maintenance. With the cost of software maintenance routinely consuming upwards of 50% of a product's life-cycle and budget [26], any savings in maintenance will have a significant impact on lowering the overall project cost. It can also affect the quality of the software by reusing tested components, domain knowledge, and information—something that is becoming increasingly important in today's competitive marketplace.

Some of the greatest benefits of reverse engineering a software system can be realized by management personnel. Project management and planning at most corporations is a complicated process. The software systems for which they are responsible exist in various life-cycle stages: new product development, testing, maintenance of existing code, and different versions. They 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, impact on other products and development groups, and market analysis. All of these things make 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.

Benefits produced by reverse engineering include aiding system comprehension, change analysis, and recovering lost information. To illustrate these benefits, we will use three representative views of a subject system: `yacc`; a parser generator in the UNIX[3] system [7]. The `yacc` program consists of 5 modules and roughly 2500 lines of C. It is complex enough to require some effort to understand (if one had only the source code to look at), yet simple enough to highlight some of the advantages to management personnel of reverse engineering. The views were produced using Rigi during `yacc`'s reverse engineering, which was completed by the author in roughly two hours.

3.1 Aiding system comprehension

When faced with the task of maintaining a software system, system comprehension is typically the most important prerequisite. Managers require a high-level understanding of the entire system. They may also need in-depth information on selected parts of the system to aid them in making decisions related to project management. The educating of new members of the development team is a never-ending and important aspect of most software projects. These education sessions are often held informally, by having new employees sit down with experienced developers who then explain the code to

[2] We plan on augmenting these measures with metrics commonly found in many CASE packages, such as functions points and cyclomatic complexity, in the future.

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

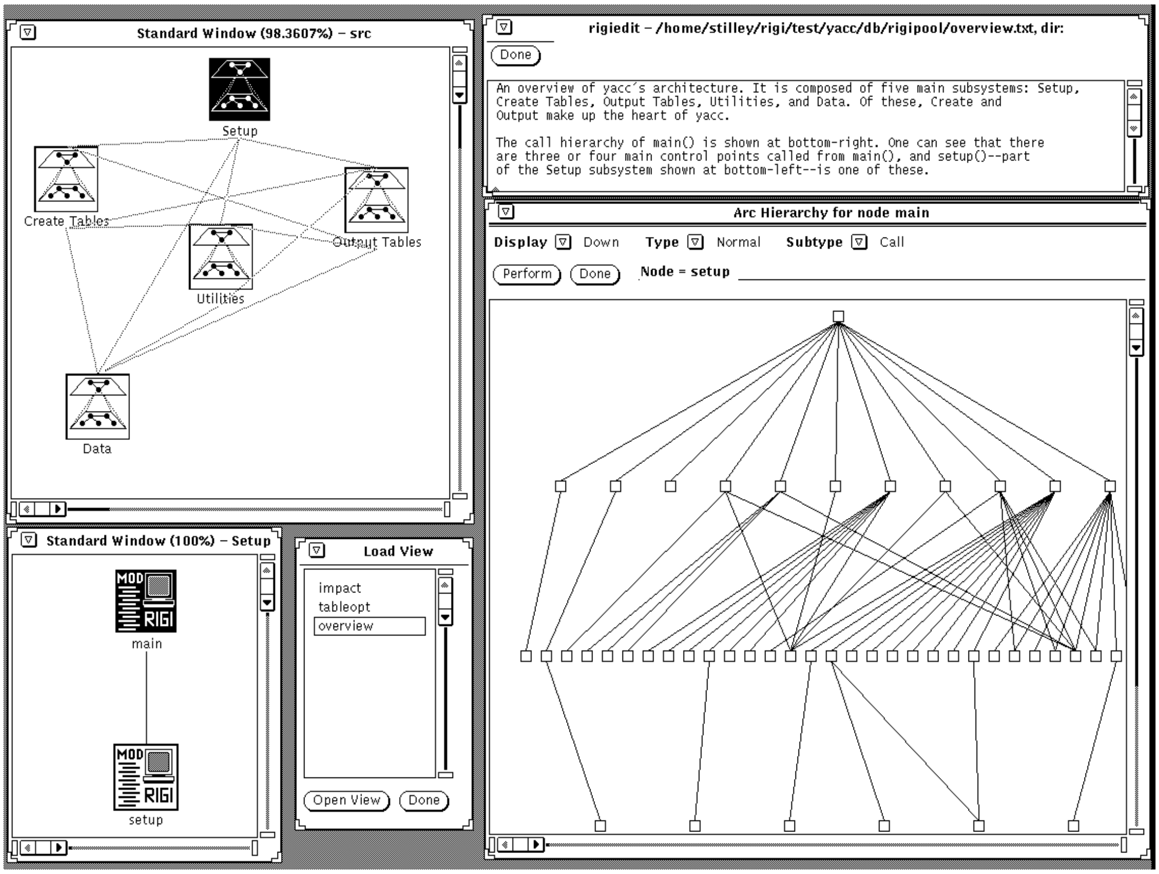
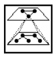


Figure 1: An overview of yacc's architecture and call structure.

them. Rigi offers a semi-automated alternative to these problems.

The Rigi system uses *views* to direct the focus on visual data and guide the exploration of spatial data. A view represents a particular state and display of a constructed software model. Different views of the same software model can be used to address a variety of target audiences and applications. Hence the same software system can be used by all those involved in the project, including development, testing, communications, and management.

For example, Figure 1 represents a high-level overview of `yacc`'s architecture and call structure. Such a view might be used by management to gain an overall understanding of the entire software system. Icons represent different artifacts of the software system: the icon  Utilities represents the `Utilities` subsystem, and 

main represents the function `main()`.

Arcs connecting icons represent resource-flow relationships between artifacts. A thin line represents a call dependency, a dashed line a data dependency, and a dotted line a composite dependency. If detailed knowledge concerning any particular subsystem is required, the user can open any icon to explore the underlying subsystem, or select any arc between icons and obtain exact dependency information. For example, Figure 2 shows the details of the `Table Optimization` subsystem, which is part of the `Output Tables` subsystem.

Graphical representations have long been ac-

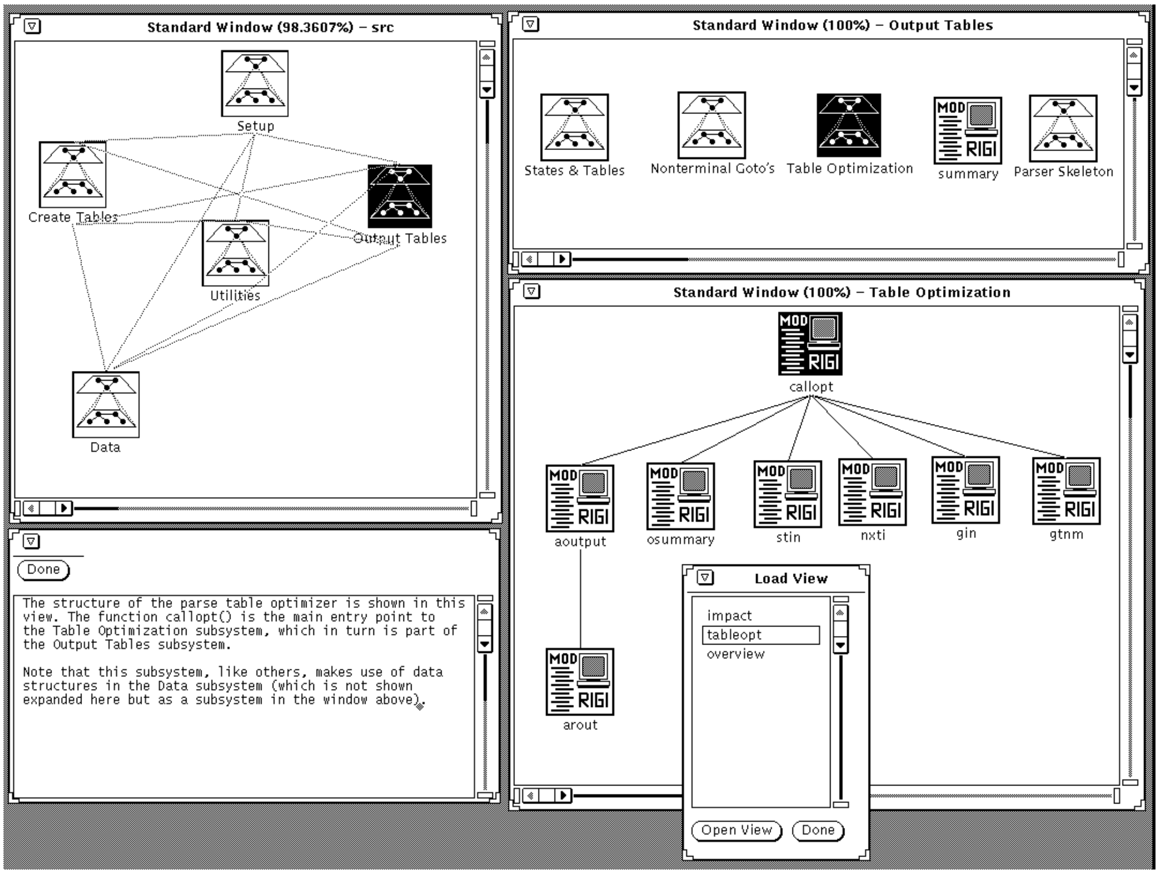


Figure 2: A detailed view of the `Table Optimization` subsystem.

cepted as comprehension aids [5]. 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 [24] to guide them in understanding the software system. As they learn about the system by examining the source code and the tutorial views, they can record the 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. Such a facility is considered essential in modern hypermedia systems [8].

Rigi also provides textual information (e.g., software quality measures) 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 could be disseminated throughout the organization, and would facilitate reuse.

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 (e.g., project priorities or personnel assignments).

For example, visible in all three figures is the `Load View` window. This window contains a scrollable list of views that may be investigated at the user's convenience. Taken together, these views form an introductory tutorial describing some of `yacc`'s operational design. Such a tutorial might be used as an aid to program understanding.

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. The views generated by reverse engineering can greatly reduce the overall cost of software by lessening the time required to understand the system. When programmers are assigned particular maintenance tasks, often they have little knowledge of the overall system design; they cannot see the forest for the trees. Managers may have the opposite problem: they may understand the overall system architecture, but do not know how specific parts of the system function. A system such as Rigi addresses this problem by providing views at various levels of detail.

3.2 Change analysis

Changing existing components is a common problem in large, evolving software systems, and occurs in both the development and maintenance phases of the software life cycle. Programmers become unwilling to alter a low-level component (also termed *basic interface*) since they are unable to estimate the effects of the change on the entire system. In the development of large systems, it is important to know what has changed, and what effect the change has on the system, because those effects can be far reaching and perhaps unanticipated. Management also becomes unwilling to allow the alteration of central components because of their potential effects on the rest of the system. Software maintenance performed without sufficient understanding of its impact is unreliable and untestable.

A graphical representation of the reverse engineered software system, such as that provided by the Rigi editor, can provide crucial information for such investigations. Many development managers still rely on SLOC[4] measurements when assessing the amount of work needed to perform a particular task. This number is usually not indicative of the actual amount of work done, and does not encourage reuse, because a smaller SLOC count is interpreted as less work by many managers. A better estimate of the work required can be provided by Rigi's presentation of affected subsystems and modules.

Reverse engineering exposes 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. Managers can quickly see where key experienced personnel should be placed (on the central components), and where newer team members can work (on fringe components). In this way, the optimum pay-off is achieved.

Views such as that shown in Figure 3 might be used to assign work to personnel in areas of the system best suited to their knowledge and experience, based on visual information provided by the spatial relationships of the central and fringe components, and on the textual reports indicating subsystem resource interconnections. Experienced programmers can be assured of working on the more critical aspects of the system (those with the most dependencies, or the most importance), while less experienced personnel can work on more isolated parts of the system. For example, the Exact Interface report shown in Figure 3 indicates that the `Create Tables` and `Data` subsystems exchange just one object. This means that changes made to either of these two subsystems will not affect the other—unless it is the `looksets` data structure that is being changed in the `Data` subsystem.

The information contained in the report can also be used by the supporting environment—for example by the compiler to limit recompilation. The CMI (*Changing Module Interfaces*)

[4] SLOC refers to "source lines of code."

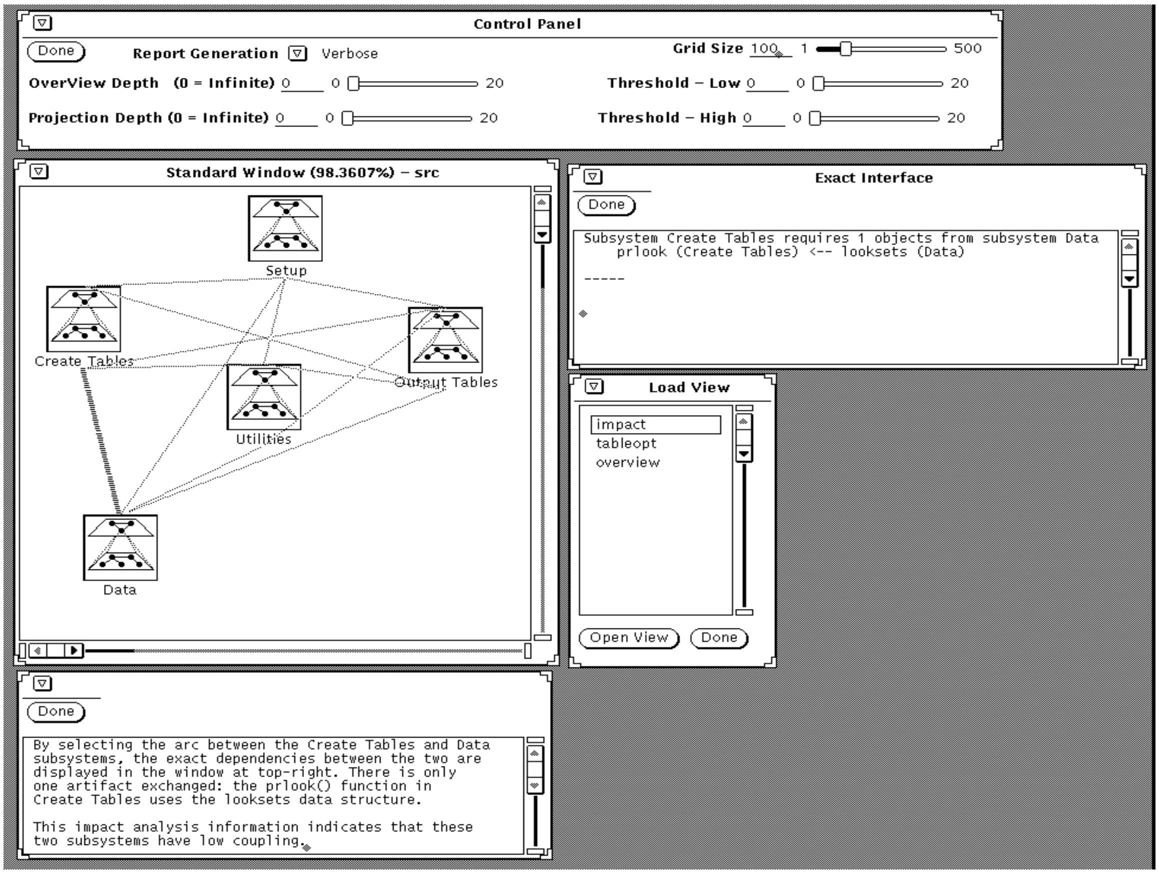


Figure 3: Exact dependency between two subsystems.

implementation [22, 23] of the *global interface analysis algorithms* [6] analyze, predict, and limit the effects of a change to a basic interface in a software system. CMI is part of the Rigi project, but has not yet been integrated into the Rigi environment.

The same information can also be used to identify components whose maintenance would improve (or impact) system performance the most. For example, enhancement requests from customers can be assessed based on the amount of work and time needed to add the code, and the number of subsystems affected. This type of "predictability" is a desirable aspect in software evolution [19]. Similarly, the impact analysis can be used to identify where regression testing must be done. Many development groups simply run their entire test suite after a change to the system; although this method is thorough, it is tremendously expensive and potentially wasteful. Rigi can identify the affected modules of a change and management can use this information to direct testing efforts. At the same time they can be assured that the testing being done is adequate to cover the enhancement (or bug fix). Providing customers with periodic maintenance fixes for a large software system is expensive, so it should be done right the first time.

3.3 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. Corrective maintenance and enhancements that seem small soon resemble patches instead of smoothly extending 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.

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) on-line 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. The project 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.

Rigi can be effectively used to aid reuse of older code not specifically written for reuse by first reverse engineering it, identifying components based on selected search criteria, and then re-engineering the chosen components if needed. In Rigi these components can be individual functions, aggregate modules, or complete subsystems. Reverse engineering will facilitate the recovery of some of the original design. The salvaging of "corporate knowledge" from earlier projects can greatly improve the quality of new projects, as well as reducing 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 Summary

Rigi is a versatile framework for analyzing large software systems. Many of its capabilities can be utilized to document existing software systems for program understanding and maintenance purposes, and utilized by management personnel to support some of the complex decisions they face in overall project management. These decisions 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 [12]. 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 (TRI-University Meson Facility) in Vancouver. The main objective of the analysis was to identify components for re-engineering.

In late 1992 we will analyze a large commercial database management system in conjunction with IBM Canada Ltd., with the goal of reverse engineering the existing system to improve the quality of subsequent maintenance and development.

References

[1] R.S. Arnold. Tutorial on software reengineering. In *CSM'90: Proceedings of the 1990 Conference on Software Maintenance,* (San Diego, California; November 26-29, 1990). IEEE Computer Society Press (Order Number 2091), November 1990.

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

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

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

[5] Mariano Consens, Alberto Mendelzon, and Arthur Ryman. Visualizing and querying software structures. In *ICSE'14: Proceedings of the 14th International Conference on Software Engineering,* (Melbourne, Australia; May 11-15, 1992), pages 138–156, May 1992.

[6] R. Hood, K. Kennedy, and H. Müller. Efficient recompilation of module interfaces in a software development environment. In *Proceedings of the 2nd ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 180–189, 1987.

[7] Stephen C. Johnson. Yacc: Yet another compiler-compiler. In *UNIX Programmer's Manual*, chapter PS1:15. USENIX Association, 1986.

[8] Hermann Maurer. Why hypermedia systems are important. University of Victoria LACIR invited talk, September 23, 1992.

[9] 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.

[10] H.A. Müller, B.D. Corrie, S.R. Tilley, and M.A. Orgun. Rigi — A system for reverse engineering. VHS videotape, University of Victoria, December 1991.

[11] 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).

[12] 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.

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

[14] 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).

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

[16] Hausi A. Müller. Verifying software quality criteria using an interactive graph editor. Technical Report DCS-139-IR, University of Victoria, August 1990.

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

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

[19] David Notkin. Software evolution. In *Proceedings of the Workshop on Future Directions in Software Engineering,* (Schloss Dagstuhl, Germany), February 1992.

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

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

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

[23] Scott R. Tilley and Hausi A. Müller. Changing module interfaces in a software development environment. In *Proceedings of the Sixth National Conference on Ada Technology,* (Arlington, Virginia; March 14-17, 1988), pages 500–508, March 1988.

[24] 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.

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

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

About the author

Scott R. Tilley is currently on leave from IBM Canada Ltd., pursuing a Ph.D. in Computer Science at the University of Victoria. His main field of research is software engineering in general, and reusability and reverse engineering in particular. His research is carried out under the umbrella of the Rigi project, supervised by Dr. Hausi Muller. Scott's M.Sc. work focused on change analysis and optimal recompilation. His current interests include higher-order abstraction mechanisms, hypermedia, and program understanding.

He can be reached at the University of Victoria, or at the IBM Canada Ltd. Laboratory, 844 Don Mills Rd., 22/121/844/TOR, North York, ON, Canada M3C 1V7. His e-mail addresses at IBM are `tilley@torolab6.vnet.ibm.com` on Internet, and `TOROLAB6(TILLEY)` on VNet.