Using Virtual Subsystems in Project Management[†]

Scott R. Tilley Hausi A. Müller

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}@csr.uvic.ca

Abstract

Software project management cannot be performed without a sufficient understanding of the entire software system. When it comes to making informed project-related decisions, management personnel 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 risk analysis and project management can be improved through the use of virtual subsystems created through reverse engineering.*

Keywords: project management, program understanding, reverse engineering, risk analysis.

1 Introduction

System comprehension is the most important prerequisite for maintaining and managing a software system. 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. This is a difficult task for managing legacy systems: software systems that are 10-25 years old and often 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 from the original [1]. Because of the lack of high-level information, risk analyses and high-level project management decisions are often based on whatever data can be gleaned from the low-level source code—not an easy task for multimillion line systems.

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 risk analysis and 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 helping them.

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, reengineering, 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 to reduce complexity. Through reverse engineering, the overall structure of the subject system can be de-

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

Copyright © 1993 IEEE. Reprinted, with permission, from *CASE '93: Proceedings of the Sixth International Conference on Computer Aided Software Engineering*, (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993), pages 144–153, IEEE Computer Society Press (Order Number 3480-02).

termined, 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 [2]. This approach is supported by *Rigi*[1] [3], a versatile system and framework under development at the University of Victoria for discovering and analyzing the structure of large software systems.

The hierarchical subsystem structures created by the reverse engineer can be used to impose logical structure on legacy systems. Because prolonged maintenance tends to degrade software structure, it is sometimes advantageous to disregard the existing modularization based on the source code's physical structure. Instead, one can construct aggregations of software artifacts based on whatever clustering and selection criteria are deemed appropriate for enhancing the understanding of the entire system or of selected subsystems of relevance. Different views of the software system may be produced for diverse audiences by using different clustering guidelines. Moreover, these views may coexist simultaneously and are kept up-to-date automatically. The result of this logical structuring is *virtual subsystems*: multiple abstract representations of a software system's architecture.

This paper describes how project management can benefit from reverse engineering.[2] In particular, it focuses on how virtual subsystems can be used to aid management decisions related to large software projects. The next section briefly outlines our reverse engineering process. Section 3 describes virtual subsystems. Section 4 explains how these structures can be used for risk analysis in project management.

2 The reverse engineering process

The goal of this paper is to describe how one uses the information produced during the reverse engineering of a software system, not to detail the reverse engineering process itself. In fact, there are many diverse approaches to reverse engineering. 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 [4]. A more detailed description of our reverse engineering environment can be found in [5].

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

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

During the process of reverse engineering, the source code is not altered, although additional information about the system 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 that alters the subject system's source code.

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 [7]. 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 [8]. 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 that generate hierarchies of subsystems [9]. 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* [10, 11]. We have also formulated software quality criteria and measures based on exact interfaces, and have established software engineering principles to evaluate the resultant subsystem structures [12, 13, 14]. These measures are not meant to be absolute; they are used to judge the relative merits of one system decomposition with respect to another.

The generated structures embody *visual* and *spatial* information that serve as organizational axes for the exploration and presentation of the composed

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

[2] Specifically, *authorized* reverse engineering of one's own code. We do not advocate improper reverse engineering of source code when one does not have the right to do so.

subsystem structures. These structures can be augmented with *views*: textual (and potentially hypermedia) annotations that highlight different aspects of the software system under investigation [15]. 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 Virtual subsystems

Traditional imperative, procedural programming languages provide nesting schemes and/or separate modules as structuring mechanisms [16]. Nesting schemes can be used to construct hierarchical structures, while separate modules can be used to specify arbitrary structures, as long as the language supports explicit provision and requirement specifications. Legacy software systems are often written in languages that support neither of these simple structuring mechanisms. Consequently, any structure they may have originally had was implicit in the design, and not necessarily reflected in the code's physical structure. The designers may have worked with an architecture that included modules with implied restrictions and requirements, but as software evolves these implicit design decisions may be lost. Maintenance fixes tend to reduce structure because the reason for the fix is often not adequately documented before the programmer who performed the fix leaves [17]. This has the effect of increasing the complexity of the software system, making maintenance and project management difficult. While software does not physically deteriorate, it may deteriorate logically (in terms of its structural integrity) [18, 19].

In addition, the cumulative affect of changes made to a software system tend to produce change patterns that cross module boundaries. Changes often reflect customer requirements; these are not necessarily reflected by the software system's architecture. They may be better expressed in terms of the application domain [20]. A simple change (from the user's perspective) may involve altering code scattered across multiple modules, whether or not the module boundaries are implicit or explicit.

To restore some semblance of order to such a software system, restructuring may be carried out. However, instead of physically restructuring the code, it might be better to simply provide multiple abstract views of the software system as it currently exists. These views would represent different aspects of the software system, and provide a dynamic and virtual structuring mechanism that can be used to organize the underlying code in a variety of ways. This mechanism would provide the ability to specify arbitrary subsystem decompositions and document them in a graphical manner. Rigi provides such an environment: one can decompose a software system into multiple, virtual subsystems that can be tailored to a variety of uses and target audiences.

Because these subsystem structures are virtual, they may be altered at any time. They represent different views of the underlying software system. Rigi's underlying semantic network data model [21] provides a structuring facility based on $(k, 2)$-partite graphs that enable the reverse engineer to produce multiple views of arbitrary objects. Figure 1 illustrates two different views of a single subsystem. Subsystems S1 and S2 share an object from subsystem S3. The arcs connecting Level_i to Level_{i+1} are *composite* arcs; they represent structural relationships (the logical aggregation of components). Arcs within a level are *resource flow* arcs; they represent referential relationships. Each view is a subsystem in itself, but includes different modules from the subsystem below. Hence, one might also call these views *semantic slices* of the software system.

4 Project management support

As the work on existing application systems consists largely of continued development [22], one way to view maintenance is as "reuse-oriented software development" [23]. There have been several areas identified as critical to improving software maintenance and (re)development, and *recapture* technologies are one area. Simply put, recapture technologies attempt to recover some of the original design in an existing software system by using reverse engineering and various program understanding tools. This knowledge can then be used for further maintenance. With the cost of software maintenance routinely consuming upwards of 50% of a product's life-cycle and budget [24], 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.

Reverse engineering can benefit many people involved in software production (for example, maintainers, developers, documenters, managers, and testers).

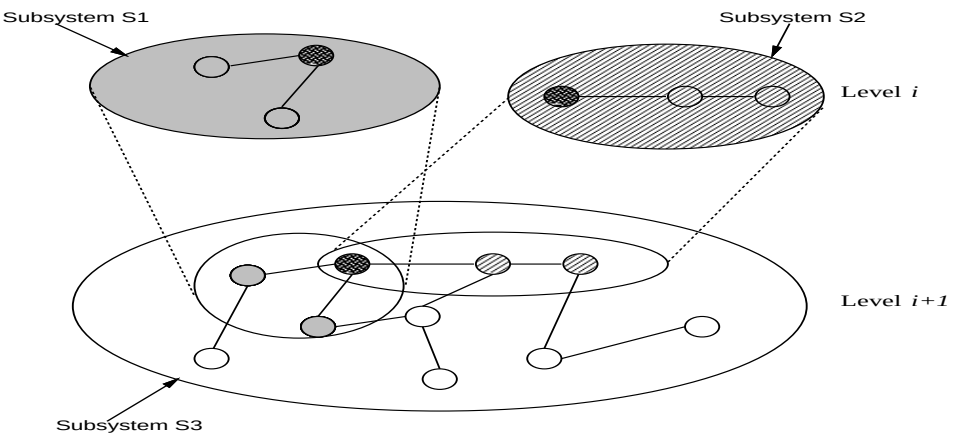

Figure 1: Multiple views of a subsystem

However, some of the greatest benefits of reverse engineering a software system can be realized by personnel involved in risk analysis related to project management. Risk is defined in Webster's online dictionary as "the possibility of loss or injury." From a software management perspective, the meaning of loss or injury is clear: loss of revenue, injury to company and career, and so on.

Project management and planning at most corporations is a complicated process. The software systems which managers are responsible for exist in various lifecycle 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 tasks 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.

Risk management involves two primary steps [25]: risk assessment and risk control. Risk assessment involves the identification, analysis, and prioritization of risks, while risk control involves risk management planning, resolution, and monitoring. To illustrate how the virtual subsystem structures produced through our reverse engineering process can be used to address the challenges of risk management, we will use three representative views of a subject system: `yacc`; a parser generator provided with the UNIX[3] system [26]. The `yacc` program consists of 5 modules and roughly 2500 lines of C. It is complex enough to require some effort to understand, yet simple enough to highlight some of the advantages to management personnel of reverse engineering. The views were produced using Rigi during the reverse engineering of `yacc`.

4.1 Risk assessment

To assess risks, one must first understand the underlying software system. One way of doing this is to expose its structure, and to impose alternative views on this structure. For project management, 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 Rigi system uses views to depict alternative virtual subsystem structures. A view represents a particular state and display of a constructed software model; in essence, a snapshot of the reverse engineering environment. Different views of the same software model can be used to address a variety of target audiences and applications. As a result, the same software system can be used by all those involved in the project, including development, testing, communications, and management.

For example, Figure 2 represents a high-level overview of `yacc`'s architecture and subsystem structure. Such a view might be used by management to

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

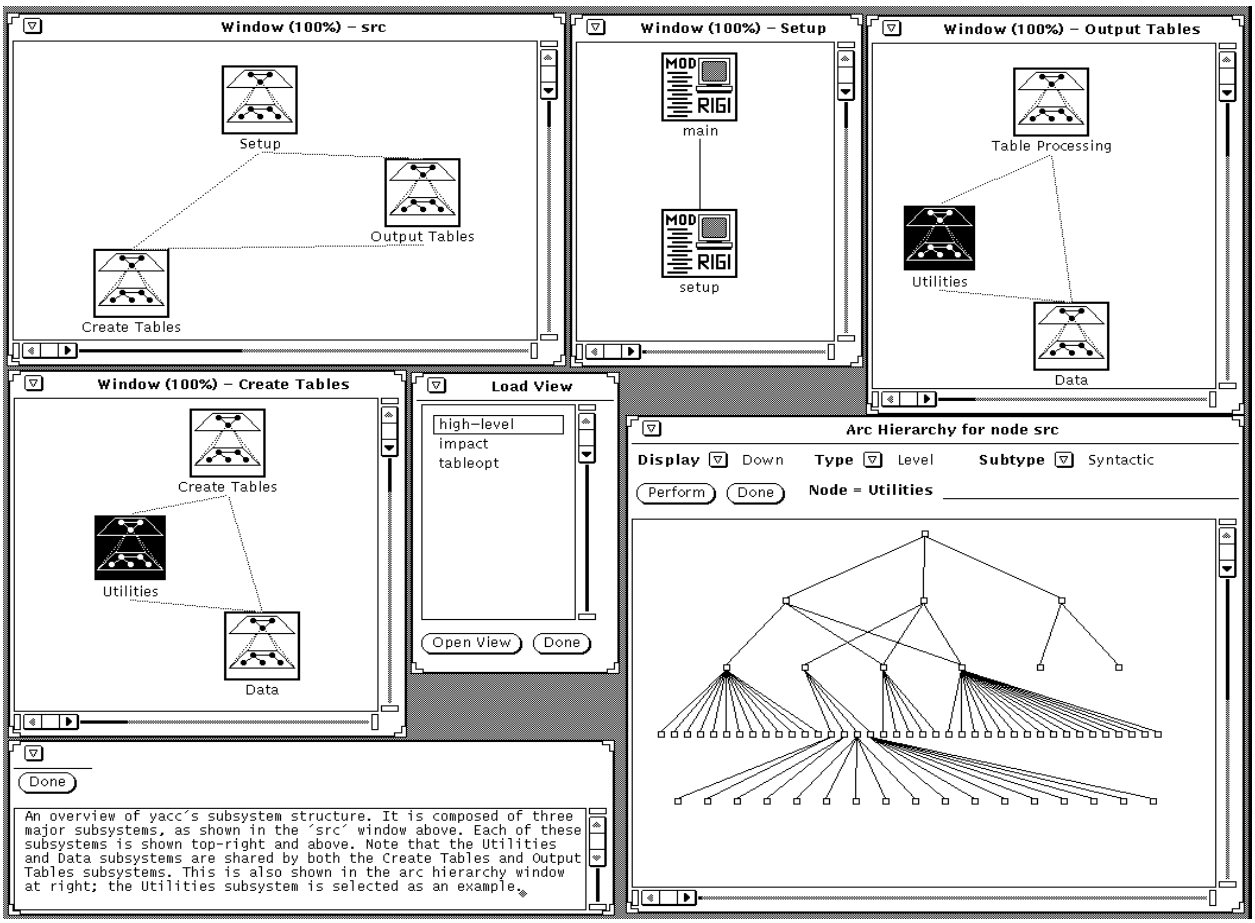

Figure 2: An overview of `yacc`'s architecture and subsystem structure.

gain an overall understanding of the entire software system. In the figure, icons represent different artifacts of the software system: the icon represents the `Utilities` subsystem, and 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 (multiple) dependency. The Arc Hierarchy window displays the virtual subsystem structure of the components of the src subsystem (the directory containing the source to `yacc`). Each level of the hierarchy represents a level in the $(k, 2)$-partite graph underlying the model. 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 3 shows the details of the `Table Optimization` subsystem, which is part of the `Table Processing` subsystem.

An initial decomposition of the subject system can be automatically constructed based on the existing file and directory structure. In this case, a file represents a module, with functions and data types encapsulated within the module. However, because such a modularization may no longer be the best one (based on criteria such as coupling and cohesion), one can choose to ignore the existing structure and decompose the system through bottom-up construction of subsystem structures from the building blocks extracted from the source code (for example, functions and data types in C). The latter approach was taken in the reverse engineering of `yacc`.

Graphical representations have long been accepted

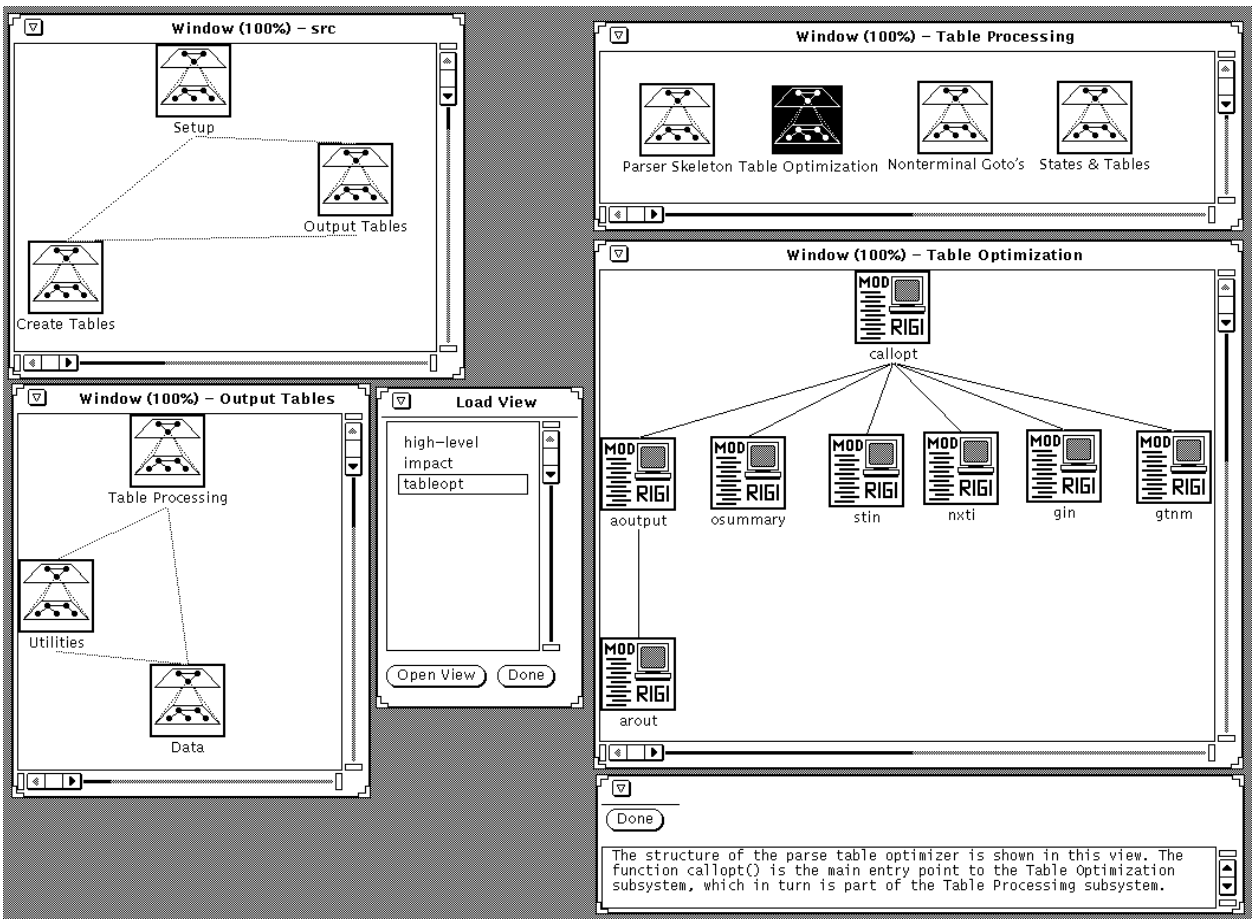

Figure 3: A detailed view of the `Table Optimization` subsystem.

as comprehension aids [27]. 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. One can work with the code and explore the entire system on one's own, using the spatial and visual information inherent in the views as a guide in understanding the software system. As one learns about the system by examining the source code and the views, one can record the information by creating new views, either for oneself or for the whole team. Rigi allows views to be either *local* or *global*, which enables users 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 [28]. In this way, detailed views of a particular subsystem can be constructed by a knowledgable software engineer and used by management when assessing the risks involving changes to the subsystem.

Rigi also provides textual information (for example, 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 can be disseminated throughout the organization, and facilitating 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; and to annotate critical sections with measurements that serve as input to informed decision-making,

such as project priorities or personnel assignments.

For example, visible in Figures 2–4 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.

The problem is not just the 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 and from different perspectives.

4.2 Risk control

Once the risks from a change to a software system have been assessed, the impact of that change needs to be minimized. 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*) because they cannot 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 becomes unwilling to allow the alteration of central components because of their potential effects on the rest of the system. Changing software systems without sufficient understanding of the 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 lines-of-code (LOC) 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 LOC 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. One can select an arbitrary subset of components from any virtual subsystem view and perform "what if" scenarios on them. The impact of a change to the selected components is displayed, allowing one to decide beforehand whether or not to carry out the change.

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. Often much effort is spent on testing nominal or low-risk functions while neglecting to test high-risk cases. Through virtual subsystems, 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 4 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 ones can work on more isolated parts of the system. For example, the figure indicates that the three functions `error()`, `aryfil()`, and `symnam()` are the busiest functions in the `Create Tables` and `Output Tables` subsystem. Projection windows allow the user to project the three-dimensional view of an arbitrary subset of subsystems down to a selected depth. When the depth is infinite, the virtual subsystem structures are logically collapsed down to the bottom-most resource flow graph. A projection of the two chosen subsystems is shown in the Projection Window, with the busy nodes highlighted. The Exact Interface report details which functions depend on these nodes and which virtual subsystem their clients are part of. The information contained in the report can also be used by the supporting environment—for example, by the compiler to limit recompilation.

The same information can also be used to identify components whose maintenance would improve (or degrade) system performance the most. For instance, 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

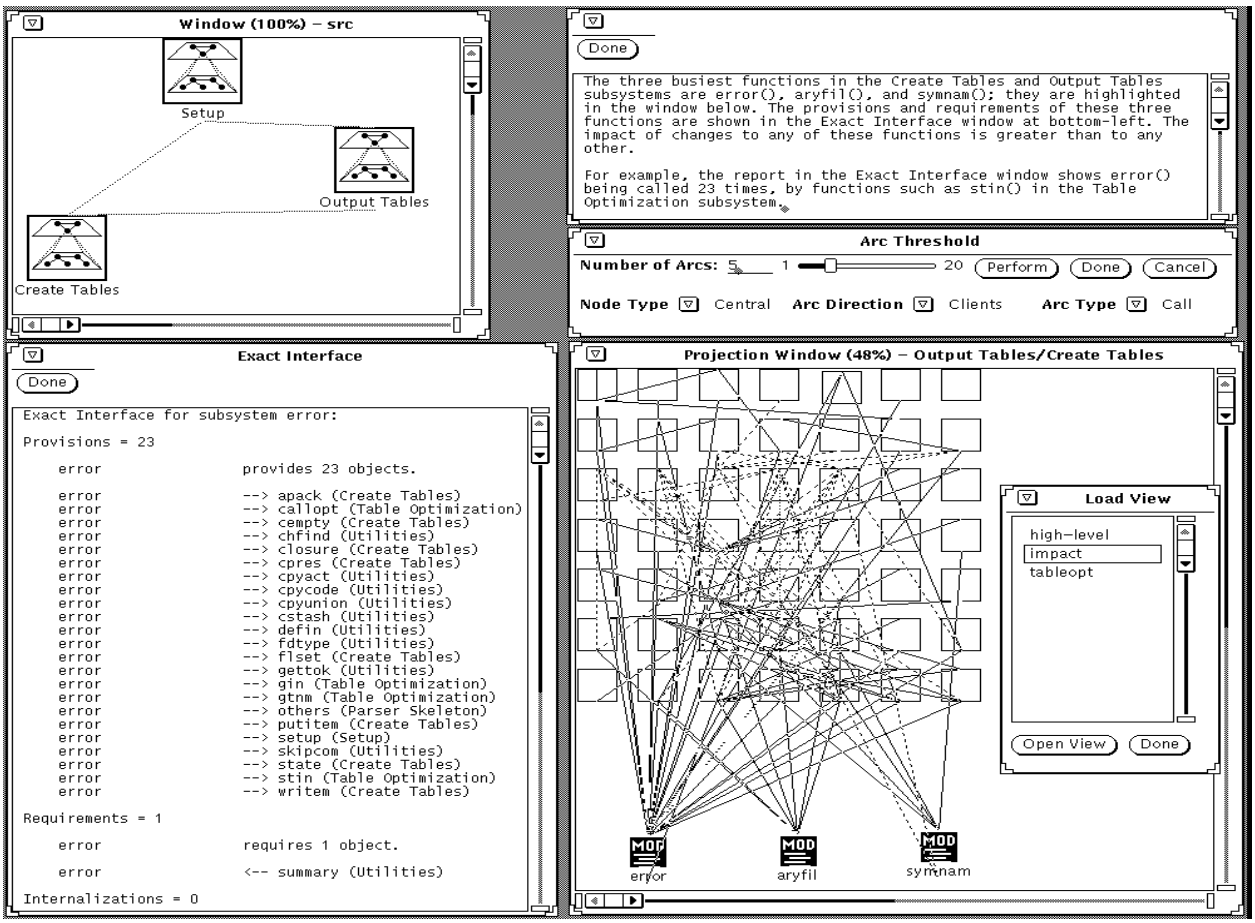

Figure 4: Locating artifacts with maximal impact potential

type of "predictability" is a desirable aspect in software evolution [29]. 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 be used to identify the affected modules of a change, and managers 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 such fixes should be done right the first time.

5 Summary

As large systems evolve over their product lifecycle, information concerning the original design is lost. Even if the system was designed using modern software engineering principles, such as 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 system structure degrades. The reverse engineering facilities provided by Rigi allow one to produce accurate "operational" design documents describing the architecture of the software system's current state—not that of the original system before numerous maintenance changes were made. These documents are part of the virtual subsystem structures constructed during the reverse engineering process. Partial design re-

covery through reverse engineering facilitates the salvaging of "corporate (domain) knowledge" from earlier projects. This can greatly improve the quality of new projects, as well as reduce cycle time from design to delivery.

Virtual subsystems can be utilized to understand and describe existing software systems for risk analysis and project management purposes. Management personnel can use these structures to support some of the complex decisions they face, such as resource allocation, personnel placement, impact analysis, system comprehension, and information recovery.

References

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

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

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

[4] S. Paul, A. Prakash, E. Buss, and J. 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.

[5] H. Müller, S. Tilley, M. Orgun, B. Corrie, and N. 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).

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

[7] H. 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).

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

[9] H. Müller and J. 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] G. Myers. *Reliable software through composite design.* Petrocelli/Charter, 1975.

[11] G. D. Bergland. A guided tour of program design methodologies. *Computer*, 14(10):18–37, October 1981.

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

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

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

[15] S. R. Tilley, H. A. Müller, and M. 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.

[16] H. L. Ossher. *A New Program Structuring Mechanism Based on Layered Graphs.* PhD thesis, Stanford University, 1984.

[17] R. S. Arnold, editor. *Tutorial on software restructuring.* IEEE Computer Society Press (Order Number 680), 1986.

[18] T. Winograd. Beyond programming languages. *Communications of the ACM*, 22(7):391–401, July 1979.

[19] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[20] G. Avellis. CASE support for software evolution: A dependency approach to control the change process. In *CASE'92: Proceedings of the fifth international workshop on computer-aided software engineering,* (Montréal, Québec; July 6-10, 1992), pages 62–73. IEEE Computer Society Press (Order Number 2960), July 1992.

[21] H. A. Müller. $(k,2)$-partite graphs as a structural basis for the construction of hypermedia applications. Technical Report DCS-119-IR, University of Victoria, June 1989.

[22] B. P. Lientz and E. B. Swanson. *Software maintenance management.* Addison-Wesley, 1980.

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

[24] N. Zvegintzov. Nanotrends. *Datamation*, pages 106–116, August 1983.

[25] B. W. Boehm, editor. *Tutorial: Software risk management*. IEEE Computer Society Press (Order Number 1906), 1989.

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

[27] M. Consens, A. Mendelzon, and A. 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.

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

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