Understanding Software Systems Using Reverse Engineering Technology†

Hausi A. Müller Kenny Wong 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: {hausi, kenw, stilley}@csr.uvic.ca

Abstract

Software engineering research has focused primarily on software construction, neglecting software maintenance and evolution. Observed is a shift in research from synthesis to analysis. The process of reverse engineering is introduced as an aid in program understanding. This process is concerned with the analysis of existing software systems to make them more understandable for maintenance, re-engineering, and evolution purposes. Presented is reverse engineering technology developed as part of the Rigi project. The Rigi approach involves the identification of software artifacts in the subject system and the aggregation of these artifacts to form more abstract system representations. Early industrial experience has shown that software engineers using Rigi can quickly build mental models from the discovered abstractions that are compatible with the mental models formed by the maintainers of the underlying software.

Keywords: Legacy software, program understanding, reverse engineering, software engineering education, software evolution, structural redocumentation.

†This work was supported in part by the British Columbia Advanced Systems Institute, IBM Canada Ltd., the IBM Centre for Advanced Studies, 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.

Colloquium on Object Orientation in Databases and Software Engineering; The 62nd Congress of "L'Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS)"; (May 16–17, 1994, Montreal, Quebec, Canada).

1 Introduction

Suppose we could turn back time to 1968—to the first software engineering conference held in Garmisch, Germany. This NATO conference, which was held in response to the perceived software crisis, introduced the term *software engineering* and significantly influenced research and practice in the years to follow. What advice would we give to those software pioneers, given our experience and the state-of-the-art in software engineering we have today? This advice might have changed history in the process. Are there software engineering problems we face today that did not exist in 1968?

These software pioneers did not anticipate that their software, constructed in the 1960's and early 1970's, would still be used and modified twenty-five years later. Today's developers inherit a huge legacy of such *heritage* software systems. Such software includes telephone switching systems, banking systems, health information systems, avionics systems, and pervasive computer vendor products. In switching systems, new functionality is added periodically to reflect the latest market needs. Banks have to update their systems regularly to implement new or changed business rules and tax laws. Health information systems must adapt to rapidly changing technology and additional services. Computer vendors are often committed to supporting their products (for example, database management systems) indefinitely, regardless of age. Evolution is a natural phenomena for such systems; it is not that someone forgot a requirement—the requirements inevitably change. There will always be old software, as new software written today will be in use for decades to come.

These systems are inherently difficult to understand and maintain due, in part, to their size and complexity, as well as their evolution history. The average Fortune 100 company maintains 35 million lines of code and adds an additional ten percent each year just in enhancements, updates, and other maintenance. As a result of maintenance alone, software inventories will double in size every seven years.

Since legacy systems cannot be replaced without reliving their entire history, managing long-term software evolution is critical. These systems embody substantial corporate knowledge such as requirements, design decisions, and business rules that have evolved over many years and are difficult to obtain elsewhere. This knowledge constitutes significant corporate assets totalling billions of dollars. Consequently, long-term software maintenance and evolution are as important (if not more) as new software construction—especially if we consider the economic impact of these legacy systems.

The next section discusses the need for increased emphasis on software analysis. Section 3 describes an approach to aiding program analysis and understanding using reverse engineering technologies. Section 4 outlines the Rigi approach to reverse engineering. Section 5 summarizes the paper.

2 Balancing act

Our main advice to the software pioneers of 1968 would be to carefully balance *software analysis* and *software construction* efforts in both research and education. This advice, which might have changed history, is still valid today.

Over the past three decades, software engineering research has focused mainly on new software construction and has neglected software maintenance and evolution. For example, numerous successful tools and methodologies have been developed for the early phases of the software life cycle, including requirement specifications, design methodologies, programming languages, and programming environments. As such, there has arisen a dramatic imbalance in software engineering education, for both academia and industry, of favoring original program, algorithm, and data structure construction. There is a need to focus more on understanding.

Computer science and computer engineering programs prepare future software engineers with a background that encourages fresh creation or synthesis. Concepts such as architecture, consistency and completeness, efficiency, robustness, and abstraction are usually taught with a bias toward synthesis, even though these concepts are equally applicable to analysis. The study of existing, real-world software systems is often overlooked. Instructors rarely provide assignments that model the normal mode of operation in industry: analyzing, understanding, and building upon existing systems. Contrast this situation with electrical or civil engineering education, where the study of existing systems and architectures constitutes a major component of the curriculum.

Knowledge of architectural concepts in large software systems is key to understanding legacy software and to designing new software. These concepts include: subsystem structures; layered structures; aggregation, generalization, specialization, and inheritance hierarchies; resource-flow graphs; component and dependency classification; event handling strategies; pipes and filters; user interface separation; and distributed and client-server architectures. The importance of architecture is now recognized; courses on the foundations of software architecture [1] have recently emerged at several universities.

The bias toward synthesis has also resulted in a lack of tools for the software maintainer. To correct the imbalance, a shift in research from synthesis to analysis is needed. This would allow the software maintenance and evolution community to catch up. Once the repertoire of tools and methodologies for analysis becomes as refined as that for synthesis, software maintenance and evolution will become more tractable. Experimental software engineering should concentrate on software evolution. The development of software evolution strategies is a great challenge—and a great opportunity.

In 1990, the Computer Science Technology Board of the U.S. proposed a research agenda for software engineering [2]. Their report concluded that progress in developing complex software systems was hampered by differing perspectives and experiences of the research community in academia and of software engineering practitioners in industry. The report recommended nurturing a collaboration between academia and industry, and legitimizing academic exploration of complex software systems directly in government and industry. Software engineering researchers should test and validate their ideas on large, real-world software systems. An excellent example of providing such arrangements in Canada is the IBM Centre for Advanced Studies (CAS) in Toronto, where there is a Program Understanding (PU) project.

3 Program understanding via reverse engineering

One of the most promising approaches to the problem of software evolution is *program understanding*. It has been estimated that fifty to ninety percent of evolution work is devoted to program comprehension or understanding [3]. Hence, easing the understanding process can have significant economic savings.

Programmers use programming knowledge, domain knowledge, and comprehension strategies when trying to understand a program. For example, one might extract syntactic knowledge from the source code and rely on programming knowledge to form semantic abstractions. Brooks's work on the theory of domain bridging [4] describes the programming process as one of constructing mappings from a problem domain to an implementation domain, possibly through multiple levels. Program understanding then involves reconstructing part or all of these mappings. Moreover, the programming process is a cognitive one involving the assembly of programming plans—implementation techniques that realize goals in another domain. Thus, program understanding also tries to pattern match between a set of known plans (or mental models) and the source code of the subject software.

3.1 Reverse engineering

For large legacy systems, the manual matching of such plans is difficult. One way of augmenting the program understanding process is through computer-aided *reverse engineering*. Although there are many forms of reverse engineering, the common goal is to extract information from existing software systems to better understand them. The subject software system is represented in a form where many of its structural and functional characteristics can be analyzed. This knowledge can then be used to improve subsequent development, ease maintenance and re-engineering, and aid project management [5]. This knowledge can help defend against brittle software systems that resist graceful change. Problems can be exposed and corrected if reverse engineering is applied preventatively during evolution. As maintenance and re-engineering costs for large legacy software systems increase, the importance of reverse engineering will grow accordingly.

The reverse engineering process involves two distinct phases [6]. The first identifies the system's current components and captures their dependencies; the second discovers design information and generates system abstractions. The second, discovery phase of reverse engineering is a highly interactive and cognitive activity. The user may build up hierarchical subsystem components that embody software engineering principles such as *low coupling* and *high cohesion* [7]. Discovery can also include the reconstruction of design and requirements specifications (often referred to as the "domain model") and the correlation of this model to the code. One classic use of this information is to redocument a software system whose documentation is missing or out-of-date.

Improved understanding of the software is beneficial to project management processes [8]. These processes are complex and involve human elements, funding, schedules, politics, trends, and marketing. The technical and organizational complexity of the project can threaten to overwhelm even the most prepared managers. To make informed decisions, managers need both a high-level perspective of the entire system and in-depth information on selected components. This information is particularly important for risk analysis: where to allocate scarce resources, where to place personnel, what is the impact of changes, and where to focus effort for maximal benefit. It is better to depend on objective data (verifiable against the actual source code) than relying only on gut feelings and experience. The required information entails another perspective on the software system (complementing the programmer's view).

During the reverse engineering process, the source code 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 phase that moves from high-level abstractions to physical implementations and reconstitutes the subject system into a new form. Moreover, re-engineering consists of many feedback loops between the old and new versions—opportunities for understanding to occur. Re-engineering can help an organization to recoup or extend its software investment and salvage corporate knowledge.

In comparison, business process re-engineering re-examines and streamlines the way businesses work. Information technology is often introduced into the workplace to simply automate old ways of doing business. Such simple optimizations do not lead to major benefits. There is a need to rethink these tasks in an organization-wide setting to better exploit the computer technology. This form of re-engineering involves changing an organization's fundamental work processes and is necessary for businesses to survive in an increasingly competitive world.

Software re-engineering involves many risks. Before embarking on a significant re-engineering project, the goals must be very clear. What characterizes a successful re-engineering process? What is a "good" re-engineering technology? What kinds of applications are amenable to re-engineering? Choosing the right toolset is critical; one must avoid subscribing to obsolete, incomplete, or incompatible methodologies.

3.2 Reverse engineering approaches

Many research groups have focused their efforts on the development of tools and techniques for program understanding. The major research issues involve the need for formalisms to represent program behavior and visualize program execution. Reverse engineering has many supporting aspects. It may focus on features such as control flows, global variables, data structures, and resource exchanges. At a higher semantic level, it may focus on behavioral features such as memory usage, uninitialized variables, value ranges, and algorithmic plans. At an even higher level of abstraction, it may focus on business rules, policies, and responsibilities. Each of these points of investigation must be addressed differently.

There are many commercial reverse and re-engineering tools available; catalogs such as [9, 10] describe several hundred such packages. Most commercial systems focus on source-code analysis and simple code restructuring, and use the most common form of reverse engineering: information abstraction via program analysis. Such capabilities extend computer-aided software engineering toolsets.

Research in reverse engineering consists of many diverse approaches. Three sample ones are formal transformations, pattern recognition, and reuse-oriented approaches. Formal transformation approaches include: concept recognition and transformation [11], least common abstractions [12], program refinements and transformations [13], and meaning-preserving restructuring [14]. Pattern recognition approaches look for matching patterns and include: defect filtering [15], syntactic clichés (SCRUPLE) [16], user interface analysis [17], graph parsing [18], characterizing design decisions [19], function abstraction [20], information abstraction (CIA, CIA++) [21, 22], maverick identification [23], and graph queries (GraphLog, G+) [24]. Approaches based on reuse include: reuse-oriented software development [25], design recovery (DESIRE) [26], and teleological maintenance (IRENE) [27].

4 The Rigi project

Rigi [28] is a framework under development at the University of Victoria for program understanding, software analysis, reverse engineering, and programming-in-the-large. One major goal is to extract abstractions from software representations and transfer this information into the minds of software engineers for software evolution purposes. The chief benefit is to reduce maintenance and evolution costs. The focus is on summarizing, querying, representing, visualizing, and evaluating the structure of large, evolving software systems. One end-product is better documentation.

Reconstructing the design of existing software is especially important for complex legacy systems. Documentation has always played an important role in program understanding. There are, however, great differences in documentation needs for software systems of 1,000 lines versus those of 1,000,000 lines. Typical software documentation is *in-the-small*, describing the program in terms of isolated algorithms and data structures. Moreover, the documentation is often scattered about on different media. The maintainers have to resort to browsing the source code and piecing disparate information together to form higher-level structural models. This process is always arduous; creating the necessary documents from multiple perspectives is often impossible. Yet it is exactly this sort of *in-the-large* documentation that is needed to expose the overall architecture of large software systems.

For a large software system, the reconstruction of the structural aspects of its architecture is beneficial. This process may be termed *structural redocumentation*. As a result, the overall structure of the subject system can be derived and some of its architectural design information can be recaptured. Software structure is the collection of artifacts used by software engineers when forming mental models of software systems. These artifacts include software *components* such as procedures, modules, and interfaces; *dependencies* among components such as client-supplier, inheritance, and control-flow; and *attributes* such as component type, interface size, and interconnection strength. The structure of a system is the organiza-

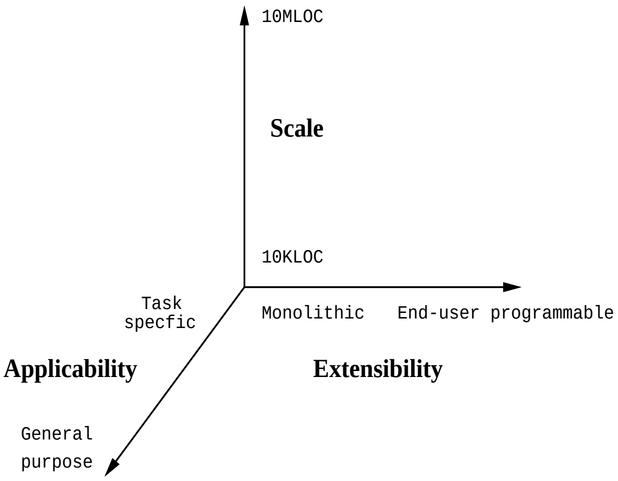

Figure 1: Reverse engineering design space

tion and interaction of these artifacts [29].

4.1 Key requirements

To be considered successful, a reverse engineering tool must be flexible with respect to its applicability to multiple domains, it must be extensibile with respect to its functionality, and it must support the analysis of large ($O(10^6)$ LOC) programs. These three requirements form a *design space* [30] for reverse engineering tools, as illustrated in Figure 1.

Flexibility: Because program understanding involves many different scenarios and target domains, it is wise to make our approach as flexible as possible for use in many different domains. An approach is flexible if it can be easily adapted to a variety of situations. In particular, we mean extensibility, tailorability, and configurability of the reverse engineering methodology and supporting environment. Most reverse engineering tools provide a fixed palette of extraction, selection, filtering, organization, documentation, and representation techniques. One important need is to provide a way for users to extend the tool's functionality. This may involve user-defined algorithms or integration with other external tools. Moreover, by making the tool user-programmable, it becomes domain-retargetable rather than domain-specific—allowing the user to tailor the tool to fully exploit aspects of the problem that make its solution easier.

Scalability: Effective approaches to program understanding must be applicable to huge, multi-million line software systems. Such scale and complexity necessitates fundamentally different approaches. Program representation, search strategies, and human-computer interfaces that work on systems "in-the-small" often do not scale up. For very large systems, the information accumulated during program understanding can be over-whelming. Current repository technology does not easily manage complete and detailed program representations for huge systems. It may be perfectly acceptable to ignore certain details for program understanding tasks. To obtain manageable repositories, coarser-grained artifacts can be extracted, partial systems can be incrementally investigated, and irrelevant parts can be ignored. To gain useful knowledge, the information must be effectively summarized and abstracted. In a sense, a key to program understanding is deciding what information is material and what is immaterial: knowing what to look for—and what to ignore [31].

4.2 Rigi framework

The most recent results of the Rigi project include: a reverse engineering environment consisting of a parsing subsystem, a distributed, multi-user repository, and an interactive graph editor [32]; a representation for software structure based on $(k, 2)$-partite graphs [33]; a reverse engineering methodology [34]; measures for evaluating the quality of structural abstractions [35]; a documentation strategy using up-to-date views [36]; a facility to understanding document structure [37]; and an extension mechanism via a scripting language [38]. Output from this environment can also serve as input into conceptual modeling, design recovery, and project management processes.

Much work on program understanding still makes heavy use of human cognitive abilities. There are tradeoffs in program understanding environments between what can be automated and what should (or must) be left to humans. The best solution seems to lie in a combination of the two. The extraction phase in Rigi is initially automatic and involves parsing the source code of the subject system and storing the extracted artifacts in the repository. This produces a flat resource-flow graph of the software. To manage the complexity, this phase is followed by a largely semi-automatic one that exploits human pattern recognition skills and features language-independent subsystem composition techniques. Rigi depends heavily on the experience and domain knowledge of the software engineer using it; the user makes all the important decisions. Nevertheless, the process is one of synergy as the user also learns and discovers interesting relationships by exploring software systems with the Rigi environment.

Subsystem composition is the methodology used in Rigi for generating layered hierarchies of subsystems, thereby reducing the cognitive complexity of understanding large software systems. It is a recursive process whereby building blocks such as data types, procedures, and subsystems are grouped into composite subsystems. This builds multiple, layered hierarchies for higher-level abstractions [39]. The criteria for identifying these "clusters" depends on the purpose, audience, and domain. For program understanding purposes, the process can be guided by dividing the resource-flow graph using established modularity principles such as low coupling and strong cohesion. Exact interfaces between subsystems and modularity/encapsulation quality measures can be used to evaluate the generated software hierarchies and assess the extent of changes. One goal is to expose properties and anomalies of the software structure for managing risk and deciding personnel assignments. For example, highly complex "central" components (as opposed to simpler "fringe" components) are best handled by the senior maintainers. The partition can also be based on other criteria such as business rules, tax laws, message paths, personnel assignment, or other semantic information.

Software engineers rely heavily on internal documentation to help understand programs. Unfortunately, this documentation is typically out-of-date and software engineers end up referring to the source code. There is a need to link the documentation and source code together. The Rigi environment eases the task of redocumenting the subject software by presenting the results using interactive *views* (somewhat similar to database views) [40]. The focus is to construct readable, accurate, and up-to-date system documentation. A view is a bundle of visual and textual frames that contain, for example, call graphs, overviews, projections, exact interfaces, and annotations [41]. A view is an adaptive snapshot that reflects the reverse engineering state contained in the graph model, repository, and editor user interface; retrieving a view reconstitutes a particular reverse engineering state from which to highlight pertinent maintenance constraints or to help illustrate problems. Moreover, this state captures the actual, operational structure of the software. Thus, a view remains up-to-date.

The economic cost of understanding a software system is significant, especially for every time a new person must learn the system. The created views can lessen the time required to understand the system. Programmers are often assigned to specific components and lack the big picture; managers may understand the overall design but lack details on specific parts. Views can accurately capture co-existing architectural decompositions, providing many different perspectives at various levels of detail for later inspection.

Reverse engineering approaches are diverse and depend on aspects of the application domain, the implementation domain, and the reverse engineering domain. It is impossible to know in advance all that will be needed to understand a software system. What is needed is flexibility and an approach that allows users to adapt the environment to exploit domain-specific knowledge themselves. Rigi supports a scripting language that allows users to customize, combine, and automate reverse engineering activities in novel ways. The language allows an analyst to complement the built-in operations with external, possibly application-specific, algorithms for graph layout, complexity measures, pattern matching, slicing, and clustering. Complex analysis tasks can be automated for more consistency and repeatability. Efforts are proceeding to make the user interface and configuration settings more user-customizable [42]. This approach permits analysts to tailor the environment to better suit their needs, providing a smooth transition between automatic and semi-automatic reverse engineering [43].

These results have been applied to several industrial software systems to validate and evaluate the Rigi approach to program understanding. In 1990, we analyzed the *Practice Manager*, a 57000 line COBOL program for managing physician's practices in British Columbia [44]. The main purpose of the analysis was to build up-to-date logical subsystems, assess the maintainability of the system, and identify components for re-engineering. In 1991, we analyzed a 82000 line C program for the isotope separator experiment at TRIUMF (TRI-University Meson Facility) in Vancouver. Early experience has shown that we can produce views that are compatible with the mental models used by the maintainers of the subject software. That is, the views we create are presenting information about the system at the right level of abstraction. Over the past year, we analyzed the source code of the SQL/DS system as part of the IBM CAS PU project [45, 46]. The goal was to apply reverse engineering technology to improve the quality of subsequent maintenance.

5 Summary

There will always be old software that needs to be understood. It is critical for the software industry to deal effectively with the problems of software evolution and the understanding of legacy software systems. Since the primary focus of the industry is changing from completely new software construction to software maintenance and evolution, software engineering research and education must make some major adjustments. In particular, more resources should be devoted to software analysis in balance with software construction.

Program understanding tools and methodologies address the problems of software evolution by helping software engineers to understanding large and complex software systems. Effective reverse engineering technologies can have a significant impact on the maintenance and evolution of these systems.

The Rigi environment focuses on the architectural aspects of the subject software under analysis. The environment provides many ways to identify, explore, summarize, evaluate, and represent software structures. More specifically, it supports a reverse engineering methodology for identifying, building, and evaluating layered subsystem hierarchies. These hierarchies are documented through views—snapshots of reverse engineering states. The views are used to transfer information about the abstractions to the minds of the software engineers. A scripting language permits analysts to adapt the environment to their needs and support automatic reverse engineering. This methodology of building subsystems, creating views, and writing scripts is applicable to other instances of structural understanding (for example, large technical documents and hypertext). The results of the Rigi project have been successfully applied to several industrial software systems.

References

[1] 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.

[2] CSTB. Scaling up: A research agenda for software engineering. *Communications of the ACM*, 33(9):281–293, March 1990.

[3] T. A. Standish. An essay on software reuse. *IEEE Transactions on Software Engineering*, SE-10(5):494–497, September 1984.

[4] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[5] R. Arnold. *Software Reengineering*. IEEE Computer Society Press, 1993.

[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] G. Myers. *Reliable Software Through Composite Design*. Petrocelli/Charter, 1975.

[8] S. R. Tilley and H. A. Müller. Using virtual subsystems in project management. In *Proceedings of the Sixth International Conference on Computer-Aided Software Engineering (CASE '93),* (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993), pages 144–153, July 1993. IEEE Computer Society Press (Order Number 3480-02).

[9] M. R. Olsem and C. Sittenauer. Reengineering technology report (Volume I). Technical report, Software Technology Support Center, August 1993.

[10] N. Zvegintzov, editor. *Software Management Technology Reference Guide*. Software Management News Inc., 4.2 edition, 1994.

[11] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, December 1992.

[12] G. Arango, I. Baxter, P. Freeman, and C. Pidgeon. TMM: Software maintenance by transformation. *IEEE Software*, 3(3):27–39, May 1986.

[13] M. Ward. *Proving Program Refinements and Transformations*. PhD thesis, Oxford University, 1988.

[14] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[15] E. Buss and J. Henshaw. A software reverse engineering experience. In *Proceedings of CASCON '91,* (Toronto, Ontario; October 28-30, 1991), pages 55–73. IBM Canada Ltd., October 1991.

[16] S. Paul and A. Prakash. Source code retrieval using program patterns. In *Proceedings of the Fifth International Workshop on Computer-Aided Software Engineering (CASE '92),* (Montréal, Québec; July 6-10, 1992), pages 95–105, July 1992.

[17] E. Merlo, J. Girard, K. Kontogiannis, P. Panangaden, and R. D. Mori. Reverse engineering of user interfaces. In *WCRE '93: Proceedings of the 1993 Working Conference on Reverse Engineering,* (Baltimore, Maryland; May 21-23, 1993), pages 171–179. IEEE Computer Society Press (Order Number 3780-02), May 1993.

[18] C. Rich and L. M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, 7(1):82–89, January 1990.

[19] S. Rugaber, S. B. Ornburn, and R. J. L. Jr. Recognizing design decisions in programs. *IEEE Software*, 7(1):46–54, January 1990.

[20] P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner. Using function abstraction to understand program behavior. *IEEE Software*, 7(1):55–63, January 1990.

[21] Y. Chen, M. Nishimoto, and C. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[22] J. E. Grass. Object-oriented design archaeology with CIA++. *Computing Systems*, 5(1):5–67, Winter 1992.

[23] R. Schwanke, R. Altucher, and M. Platoff. Discovering, visualizing, and controlling software structure. *ACM SIGSOFT Software Engineering Notes*, 14(3):147–150, May 1989. Proceedings of the Fifth International Workshop on Software Specification and Design.

[24] 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.

[25] V. R. Basili. Maintenance = reuse-oriented software development. In *Proceedings of the IEEE 1988 Conference on Software Maintenance*, pages 3–4, October 1988.

[26] T. J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Software*, 22(7):36–49, July 1989.

[27] V. Karakostas. Modelling and maintenance software systems at the teleological level. *Journal of Software Maintenance: Research and Practice*, 2:47–59, 1990.

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

[29] H. L. Ossher. A mechanism for specifying the structure of large, layered systems. In B. D. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 219–252. MIT Press, 1987.

[30] T. G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU/SEI-90-TR-18, Software Engineering Institute; Carnegie-Mellon University, November 1990.

[31] M. Shaw. Larger scale systems require higher-level abstractions. *ACM SIGSOFT Software Engineering Notes*, 14(3):143–146, May 1989. Proceedings of the Fifth International Workshop on Software Specification and Design.

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

[33] 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.

[34] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

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

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

[37] S. R. Tilley, M. J. Whitney, H. A. Müller, and M.-A. D. Storey. Personalized information structures. In *Proceedings of the 11th Annual International Conference on Systems Documentation (SIGDOC '93),* (Waterloo, Ontario; October 5-8, 1993), pages 325–337. ACM (Order Number 6139330), October 1993.

[38] S. R. Tilley, H. A. Müller, M. J. Whitney, and K. Wong. Domain-retargetable reverse engineering. In *Proceedings of the 1993 International Conference on Software Maintenance (CSM '93),* (Montréal, Québec; September 27-30, 1993), pages 142–151. IEEE Computer Society Press (Order Number 4600-02), September 1993.

[39] H. Müller and J. Uhl. Composing subsystem structures using $(k, 2)$-partite graphs. In *Proceedings of the 1990 Conference on Software Maintenance (CSM '90),* (San Diego, California; November 26-29, 1990), pages 12–19, November 1990. IEEE Computer Society Press (Order Number 2091).

[40] S. R. Tilley. Documenting-in-the-large vs. documenting-in-the-small. In *Proceedings of the 1993 IBM/NRC CAS Conference (CASCON '93),* (Toronto, Ontario; October 25-28, 1993), pages 1083–1090, October 1993.

[41] K. Wong. Managing views in a program understanding tool. In *Proceedings of the 1993 IBM/NRC CAS Conference (CASCON '93),* (Toronto, Ontario; October 25-28, 1993), pages 244–249, October 1993.

[42] S. R. Tilley. Domain-retargetable reverse engineering II: Personalized user interfaces. In *International Con-*

ference on Software Maintenance (ICSM '94), (Victoria, BC; September 19-23, 1994), pages 336–342. IEEE Computer Society Press (Order Number 6330-02), September 1994.

[43] S. R. Tilley, K. Wong, M.-A. D. Storey, and H. A. Müller. Programmable reverse engineering. To appear in the *International Journal of Software Engineering and Knowledge Enginering*, 4(4), December 1994.

[44] H. A. Müller, J. R. Möhr, and J. G. McDaniel. Applying software re-engineering techniques to health information systems. In T. Timmers and B. Blums, editors, *Software Engineering in Medical Informatics*, pages 91–110. Elsevier North Holland, 1991.

[45] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Structural redocumentation: A case study. To appear in *IEEE Software*, January 1995.

[46] E. Buss, R. D. Mori, W. M. Gentleman, J. Henshaw, H. Johnson, K. Kontogiannis, E. Merlo, H. A. Müller, J. Mylopoulos, S. Paul, A. Prakash, M. Stanley, S. R. Tilley, J. Troster, and K. Wong. Investigating reverse engineering technologies for the CAS program understanding project. *IBM Systems Journal*, 33(3):477–500, 1994.