Understanding Software Systems Using Reverse Engineering Technology Perspectives from the Rigi Project[†]

Hausi A. Müller   Scott R. Tilley   Kenny Wong

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, stilley, kenw}@csr.uvic.ca

Abstract

Software engineering research has focused mainly on software construction and has neglected software maintenance and evolution. Proposed is a shift in research from synthesis to analysis. Reverse engineering is introduced as a possible solution to program understanding and software analysis. 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 architectural models. Reported are some analyses on the source code of SQL/DS, performed by the authors while visiting the Program Understanding project at the IBM Centre for Advanced Studies in Toronto.

Keywords: Legacy software, software evolution, program understanding, reverse engineering.

[†]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.

Reprinted from *Proceedings of CASCON '93*, (Toronto, Ontario; October 25-28, 1993), pages 217-226.

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 the software engineering background and experience we have today?

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, these systems are often referred to as *legacy* or *heritage* systems. They include telephone switching systems, banking systems, health information systems, avionics systems, and many 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 increased demands. Computer vendors are often committed to supporting their products (for example, database management systems) indefinitely, regardless of age.

Such legacy systems cannot be replaced without re-living their entire history. They 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. As a result, long-term software maintenance and evolution are as important as software construction—especially if we consider the economic impact of these systems.

2 Analysis and synthesis

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.

Over the past three decades, software engineering research has focused mainly on 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.

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 and relevant to analysis. The study of 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.

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

> Programmers have become part historian, part detective, and part clairvoyant.
>
> — T.A. Corbi, IBM

One of the most promising approaches to the problem of software evolution is *program understanding* technology. It has been estimated that fifty to ninety percent of evolution work is devoted to program comprehension or understanding [3]. 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 early work on the theory of domain bridging [4, 5, 6] 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.

For large legacy systems, the manual matching of such plans is difficult. One way of augmenting the program understanding process is through *reverse engineering*. Although there are many forms of reverse engineering, the common goal is to extract information from existing software systems. This knowledge can then be used to improve subsequent development, ease maintenance and re-engineering, and aid project management [7].

3.1 Reverse engineering process

The process of reverse engineering a subject system involves the identification of the system's current components and their dependencies followed by the extraction of system abstractions and design information. During this 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 or re-implementation phase that alters the subject system.

There is a large number of commercial reverse and re-engineering tools available; [8] lists over one 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.

3.2 Reverse engineering approaches

Reverse engineering consists of many diverse approaches, including: formal transformations [9], meaning-preserving restructuring [10], pattern recognition [11], function abstraction [12], information abstraction [13, 14], maverick identification [15], graph queries [16], and reuse-oriented methods [17].

Three specific approaches are briefly described below. Each approach is supported by one (or more) research groups in the PU project at CAS. One goal of this project is to exploit all three to produce a more comprehensive reverse engineering toolset.

3.2.1 Defect filtering

The work by Buss and Henshaw at IBM CAS explores design recovery and knowledge re-engineering of large legacy software, with an emphasis on software written for the SQL/DS[1] product. SQL/DS is typical of many legacy systems: old, highly modified, popular, successful, and large. Higher quality standards and increased productivity goals motivated this exploration.

Their work in the PU project is mainly concerned with *defect filtering* in SQL/DS [18] and other IBM products. They use the commercial Software Refinery product (REFINE) [19] to parse the source code into a form suitable for analysis. This work applies the experience of domain experts and the results of causal analysis to create REFINE "rules" to find certain families of defects in the subject software. These defects include programming language violations (overloaded keywords, poor data typing), implementation domain errors (data coupling, addressability), and application domain errors (coding standards, business rules).

[1]SQL/DS is a trademark of the International Business Machines Corporation.

3.2.2 Pattern matching

Four research groups affiliated with the IBM CAS PU project focus on pattern-matching approaches at various levels: textual (characters), lexical (token stream), syntactic (parse tree), semantic (meaning), and structural (architecture).

Johnson and Gentleman study redundancy at the textual level, with some work at the lexical level. A number of uses are relevant to the SQL/DS product: looking for code reused by cut-and-paste, building a simplified model for macro processing based on actual use, and providing overviews of information content in absolute or relative (version or variant) terms.

Paul and Prakash use programming language constructs as a type of plan in the SCRUPLE system [20]. Instead of looking for low-level textual patterns or very high-level semantic constructs, SCRUPLE looks for code clichés. The cliché pattern is user-defined. This approach is a natural progression from simple textual scanning techniques; instead of using character strings as the search criteria, programming-language constructs are used.

The work by Kontogiannis concerns semantic or behavioral pattern-matching [21]. A transformational approach is used to simplify syntactic programming structures and expressions, such as `while` loops, by translating them to simpler canonical forms. A canonical form is a semantic abstraction that improves understanding while providing a unified representation for similar constructs. These canonical forms can reduce the number of plans that need to be stored.

Finally, structural patterns are investigated as part of the Rigi[2] [22] project.

3.2.3 Structural redocumentation

Software structure refers to a collection of artifacts that software engineers use to form mental models when designing, documenting, implementing, 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 structure of a system is the organization and interaction of these artifacts [23].

For a large software system, the reconstruction of the structural aspects of its architecture is beneficial. This process may be termed *structural redocumentation*. It involves the identification of the software artifacts in the subject system and the organization of these artifacts into more abstract models to reduce complexity. As a result, the overall structure of the subject system can be derived and some of its architectural design information can be recaptured.

This process is supported by Rigi, a flexible environment under development at the University of Victoria for discovering and analyzing the structure of large software systems. It provides the following desirable components of a reverse engineering environment:

- a variety of parsers to support the common programming languages of legacy software;[3]
- a repository to store the information extracted from the source code [24]; and
- an interactive graph editor to manipulate program representations.

In the Rigi approach, the first phase of the structural redocumentation process is automatic and language-dependent. It involves parsing the source code of the subject system and storing the extracted artifacts in a repository. The second phase involves human interaction and features language-independent subsystem composition methods that generate bottom-up, layered hierarchies [25, 26]. Subsystem composition is the iterative process of aggregating building blocks such as data types, procedures, and subsystems into composite subsystems. The process is guided by partitioning the resource-flow graphs of the source code via equivalence relations that embody software engineering principles concerning module

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

[3]Such software is typically written in procedural or imperative programming languages such as C, COBOL, Fortran, and PL/I.

interactions such as *low coupling* and *strong cohesion* [27, 28]. The Rigi project also devised software quality measures, based on exact interfaces and established software engineering principles, to evaluate the generated subsystem hierarchies [29, 30, 31].

4 The Rigi project

The main goal of the Rigi project is to investigate frameworks and environments for program understanding, reverse engineering, and software analysis (all in-the-large). The most recent results of the project include a reverse engineering environment consisting of a parsing subsystem, a repository, and a graph editor [32]; a reverse engineering methodology based on subsystem composition [25]; a documentation strategy using views [33, 34, 35]; a structured hypertext layer [36]; and an extension mechanism via a scripting language [37]. These results have been applied to several industrial software systems to validate and evaluate the Rigi approach to program understanding [38, 39, 40]. Early experience has shown that we can produce views that are compatible with the mental models used by the maintainers of the subject software. Over the past year, we analyzed the source code of the SQL/DS system as part of the IBM CAS PU collaborative project.

4.1 Scalability, flexibility, and extensibility

To compete with commercial tools and to inspire the current state-of-practice, we need to train our program understanding tools and methodologies on large software systems. Techniques that work on toy projects typically do not scale up. Our current scale objective is to analyze systems consisting of up to five million lines of code.

Because program understanding has so many different facets and applications, it is wise to make our approach as flexible as possible for use in many different domains. Most reverse engineering tools provide a fixed set of extraction, selection, filtering, organization, documentation, and representation techniques. We provide a scripting language that allows users to customize, combine, and automate these activities in novel ways.

4.2 Involving the user

Much work on program understanding still makes heavy use of human cognitive abilities. There is a tradeoff 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. 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.

4.3 Summarizing software structure

Subsystem composition is the methodology used in Rigi for generating layered hierarchies of subsystems, thereby reducing the cognitive complexity of understanding large software systems. The Rigi environment supports a partitioning of the resource-flow graph based on established software engineering principles. However, because the user is in charge, the partition can easily be based on other criteria, such as business rules, tax laws, message paths, or other semantic or domain information. Moreover, alternate *decompositions* may co-exist under the software structure representation supported by Rigi.

4.4 Documenting with views

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. The Rigi environment eases the task of redocumenting the subject software by presenting the results using interactive *views*. A view is a bundle of visual and textual frames that contain, for example, call graphs, overviews, projections, exact interfaces, and annotations. A view is a dynamic snapshot that reflects the current reverse engineering state. As such, a view remains up-todate. Views can accurately capture co-existing architectural decompositions, providing many different perspectives for later inspection.

5 Analyzing SQL/DS

In 1992/93, the first author spent a ten-month sabbatical at IBM CAS. The other two authors joined him for four months and together we analyzed the source code of SQL/DS using the Rigi environment.

5.1 Scaling up

SQL/DS contains about three million lines of PL/AS source code, excluding comments and blank lines. PL/AS is an internal IBM programming language—a mix of PL/I and assembly language. When we started, the Rigi system did not include a PL/AS parser. Moreover, we had never analyzed a system over 120 thousand lines of code. Thus, this analysis presented a real challenge and an excellent test of whether our methodology and environment would scale up to the million line range.

Many commercial reverse engineering tools store entire parse trees in their repositories. For a multi-million line program, this can require several hundred megabytes of storage. While this level of detail may be necessary for tasks such as data and control-flow analyses or code optimization and generation, it is not necessary for understanding the architecture. For program understanding, it is important to build abstractions that emphasize important themes and suppress irrelevant details; deciding what to include and what to ignore is still an art.

The Rigi parsing subsystem can extract a variety of software artifacts at various levels of detail. Thus, we can reduce the repository size for a multi-million line program significantly, making a major difference when retrieving data interactively. For example, the Rigi database for SQL/DS is under two megabytes.

Displaying graphs of twenty to fifty thousand vertices and their edges for a multi-million line program is another problem of scale. For smaller graphs, it is feasible to update a window after every event or command. This strategy fails for very large graphs on current display technology. We needed to tune the user interface, redesigning it to allow the user to batch sequences of operations and to specify when to update a window.

5.2 Identifying a target audience

After tuning the Rigi environment to handle multi-million line programs, we had to identify a target audience for our program understanding experiments. Both the CAS PU groups and the SQL/DS development and management teams were extremely helpful.

On our first try, we summarized the entire call graph of SQL/DS without considering any domain knowledge. The result was not encouraging because the developers did not recognize the structures we generated, making it difficult to give constructive feedback. On our second try, we considered the naming conventions used by the developers. This time, they readily recognized our subsystem decomposition.

We then focused on one large subsystem of SQL/DS, the relational data system (RDS), which contains about one million lines of source code. With help from a domain expert, RDS was further decomposed into four main subsystems: (1) run-time access generator; (2) optimizer pass one and two; (3) optimizer path selection; and (4) executive, interpreter, and authorization. Four distinct development teams were in charge of these subsystems.

5.3 Developer feedback

For the individual subsystems, we proceeded to analyze their call graphs, summarizing them in a set of views depicting different architectural perspectives. We then presented these views to the development teams with a series of carefully designed one-hour demonstrations.

A demonstration consisted of four phases: (1) highlighting the main features of the Rigi user interface; (2) exhibiting the structural views of the subsystems pertinent to the particular development group; (3) allowing the audience to interact with their software structures using views as starting points; and (4) allowing individual developers to create new views on the fly to reflect and record specific domain knowledge.

While our prepared views did not uncover the exact mental model of each developer, the audience readily recognized the presented structures. There were two main reasons for this. First, these developers knew their subsystems intimately. Second, and more importantly, the views represented the right level of abstraction. Most satisfying for us was when the individual developers used their specific domain knowledge to design additional views to reflect their mental model more closely. This was usually done by emphasizing important components and filtering irrelevant information. Invariably, after the demonstrations, the developers came back to try out and document additional domain knowledge and perspectives.

6 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.

With the focus changing to software evolution, program understanding tools and methodologies that effectively aid software engineers in understanding large and complex software systems can have a significant impact. This is critical for keeping up with the varied demands of the information industry.

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 documenting layered subsystem hierarchies. Critical to the usability of the Rigi system is the ability to store and retrieve views—snapshots of reverse engineering states. The views are used to transfer information about the abstractions to the software engineers.

While the Rigi system is now primarily a reverse engineering environment, it was originally conceived as a design tool. In fact, the system can be used for both reverse and forward engineering, helping to complete the cycle of software evolution.

6.1 Future work

We are currently designing and developing a more ambitious reverse engineering environment based on seven years of experience gained with the Rigi project. This new environment is supported by an NSERC CRD (Collaborative Research and Development) grant and involves three universities (McGill University, the University of Toronto, and the University of Victoria) with IBM as the industrial partner. Collaboration is the main theme as the universities walk a fine line between pure research and developing an integrated system for addressing real-world program understanding problems in industry.

McGill University will extend the structural pattern matching capabilities of the Rigi system to support syntactic, semantic, functional, and behavioral search patterns. The University of Toronto will deliver a more flexible and powerful repository for storing software artifacts, pattern matching rules, and software engineering knowledge. The University of Victoria will make the Rigi system more flexible, scalable, extensible, by developing a scripting language for users to design their own high-level operations by composing and tailoring existing operations. IBM will provide an industrial perspective on architectural issues of the new environment, characterize real-world re-engineering applications, and contribute legacy code for testing the new environment. The results of the project will be disseminated in papers, demonstrations, and tutorials.

We see re-engineering as the next logical step after reverse engineering. We are seeking to apply the results of the NSERC CRD work to other large legacy systems.

About the authors

Dr. Hausi A. Müller is an Associate Professor of Computer Science at the University of Victoria, where he has been since 1986. From 1979 to 1982 he worked as a software engineer for Brown Boveri & Cie in Baden, Switzerland (now called ASEA Brown Boveri). In 1992/93 he was on sabbatical at the IBM Centre for Advanced Studies in Toronto working with the program understanding group. His research interests include software engineering, software analysis, program understanding, reverse engineering, re-engineering, programming-in-the-large, software metrics, and computational geometry. His Internet address is `hausi@csr.uvic.ca`.

Scott R. Tilley is currently on leave from IBM Canada Ltd., pursuing a Ph.D. in the Department of Computer Science at the University of Victoria. His field of research is software engineering in general, and program understanding, software maintenance, and reverse engineering in particular. He can be reached at the University of Victoria, or at the IBM PRGS Toronto Laboratory, 844 Don Mills Rd., 22/121/844/TOR, North York, ON, Canada M3C 1V7. His e-mail addresses are `stilley@csr.uvic.ca` at UVic, and `stilley@vnet.ibm.com` or `TOROLAB6(TILLEY)` (VNet) at IBM.

Kenny Wong is a Ph.D. student in the Department of Computer Science at the University of Victoria. His research interests include program understanding, user interfaces, and software design. He is a member of the ACM, USENIX, and the Planetary Society. His Internet address is `kenw@sanjuan.uvic.ca`.

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 cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 9:737–751, 1977.

[5] R. Brooks. Using a behavioral theory of program comprehension in software engineering. In *ICSE'3: Proceedings of the 3rd International Conference on Software Engineering,* (Atlanta, Georgia; May 10-12, 1978), pages 196–201, May 1978.

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

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

[8] C. Sittenauer, M. Olsem, and D. Murdock. Re-engineering tools report. Technical Report STSC-Rev B, Software Technology Support Center; Hill Air Force Base, July 1992.

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

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

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

[12] 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.

[13] Y. Chen, M. Nishimoto, and C. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

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

[15] 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.

[16] 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.

[17] T. J. Biggerstaff, B. G. Mitbander, and D. Webster. The concept assignment problem in program undertsanding. In *WCRE '93: Proceedings of the 1993 Working Conference on Reverse Engineering,* (Baltimore, Maryland; May 21-23, 1993), pages 27–43. IEEE Computer Society Press (Order Number 3780-02), November 1992.

[18] E. Buss and J. Henshaw. Experiences in program understanding. Technical Report TR-74.105, IBM Canada Ltd. Centre for Advanced Studies, July 1992.

[19] G. Kotik and L. Markosian. Program transformation: The key to automating software maintenance and re-engineering. Technical report, Reasoning Systems, Inc., 1991.

[20] S. Paul. SCRUPLE: A reengineer's tool for source code search. In *CASCON'92: Proceedings of the 1992 CAS Conference,* (Toronto, Ontario; November 9-12, 1992), pages 329–345. IBM Canada Ltd., November, 1992.

[21] K. Kontogiannis. Toward program representation and program understanding using process algebras. In *CASCON'92: Proceedings of the 1992 CAS Conference,* (Toronto, Ontario; November 9-12, 1992), pages 299–317. IBM Canada Ltd., November 1992.

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

[23] 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.

[24] N. Kiesel, A. Schürr, and B. Westfechtel. GRAS: A graph-oriented database system for (software) engineering applications. In *CASE '93: The Sixth International Conference on Computer-Aided Software Engineering,* (Institute of Systems Science, National University of Singapore, Singapore; July 19-23, 1993), pages 272–286, July 1993. IEEE Computer Society Press (Order Number 3480-02).

[25] 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).

[26] 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,* 1993. In press.

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

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

[29] H. Müller. Verifying software quality criteria using an interactive graph editor. In *Proceedings of the Eighth Annual Pacific Northwest Software Quality Conference,* (Portland, Oregon; October 29-31, 1990), pages 228–241, October 1990. ACM Order Number 613920.

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

[31] 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.

[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 *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).

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

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

[35] 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.

[36] S. R. Tilley, M. J. Whitney, H. A. Müller, and M.-A. D. Storey. Personalized information structures. *SIGDOC '93: The 11th Annual International Conference on Systems Documentation,* (Waterloo, Ontario; October 5-8,

1993), pages 325–337, October 1993. ACM Order Number 6139330.

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

[38] 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.

[39] S. R. Tilley. Management decision support through reverse engineering technology. In *Proceedings of CASCON '92,* (Toronto, Ontario; November 9-11, 1992), pages 319–328, November, 1992.

[40] S. R. Tilley and H. A. Müller. Using virtual subsystems in project management. In *CASE '93: 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, July 1993. IEEE Computer Society Press (Order Number 3480-02).