An approach to recovering data flow oriented design of a software system

Arun Lakhotia

The Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
(318) 231-6766, -5791 (Fax)
arun@cacs.usl.edu

Abstract

This paper describes an approach for recovering data flow oriented design of a software system from its source code. Data flow diagrams are used for analysis and design in software development in variations of Structured Analysis techniques. A tool that extracts designs based on these diagrams from the code will be useful in maintaining the consistency of design document with its code, migrating old software code to emerging Computer Aided Software Engineering (CASE) technology, and for understanding large software systems.

The approach proposed employs reverse engineering techniques that create hierarchical clusters of functions and procedures to identify the “bubbles” at various levels in the hierarchy of DFD’s. It uses results from interprocedural flow analysis to compute the “logical” flow of data between these bubbles. And it uses information about data types provided with the source code to create the data dictionary.

The paper also identifies the open problems whose solutions would enable the recovery of data flow oriented designs. Our current research effort is focussed on solving these problems.

Abstract

This paper describes an approach for recovering data flow oriented design of a software system from its source code. Data flow diagrams are used for analysis and design in software development in variations of Structured Analysis techniques. A tool that extracts designs based on these diagrams from the code will be useful in maintaining the consistency of design document with its code, migrating old software code to emerging Computer Aided Software Engineering (CASE) technology, and for understanding large software systems.

The approach proposed employs reverse engineering techniques that create hierarchical clusters of functions and procedures to identify the “bubbles” at various levels in the hierarchy of DFD’s. It uses results from interprocedural flow analysis to compute the “logical” flow of data between these bubbles. And it uses information about data types provided with the source code to create the data dictionary.

The paper also identifies the open problems whose solutions would enable the recovery of data flow oriented designs. Our current research effort is focussed on solving these problems.

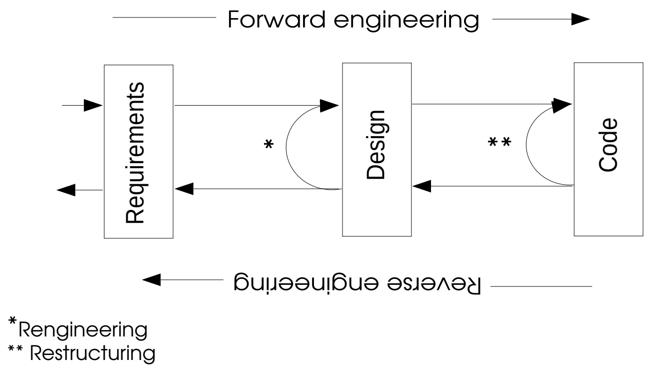

Figure 1 Schematic diagram showing relationship between forward engineering, reverse engineering, reengineering, and restructuring (as defined by Chikofsky and Cross [9]).

1 Introduction

Design recovery, according to Chikofsky and Cross [9], is the identification of “meaningful higher level abstractions beyond those obtained directly by examining [a software] system itself”. According to Biggerstaff, the information so recovered would facilitate understanding “what a program does, how it does it, why it does it, and so forth” [4]. This recognition has led to an interest in creating higher levels of abstractions (such as requirements and design) from the source code. This is termed as *reverse engineering* as opposed to *forward engineering* - the development of more concrete artifact from an abstract one, see Figure 1.

The need for reverse engineering has been very eloquently elaborated upon by [9] and other researchers. The specific benefit of recovering data flow oriented design may be emphasized from two points. First, in order that practitioners use a reverse engineering approach it is important that the recovered design match one they use in forward engineering. There are several requirements analysis and design methods that are based on *data flow diagrams* (DFD), for instance De Marco [12], Gane and Sarson [14], Ross and Brackett [36]. It would be preferred if the design recovered from a system developed using these techniques be expressed using DFDs.

The second benefit is in organizations introducing CASE tools in their development methods. These organization would invariably like a path to migrate their existing software to CASE tool(s) of their choice. Since several requirement analysis and design tools are based on methods listed earlier, a tool that recovered data flow oriented design becomes an important technology in enabling the transition.

To the best of our knowledge, and as is also observed by [39], there has been no work in recovering data flow design of a software system from its source code. The next section describes the approach we propose for recovering data flow oriented design. Section 3 surveys work related to the proposed approach. Section 4 concludes the paper with a list of open research problems, the focus of our current research efforts.

2 Proposed approach

A *data flow design* consists of a “hierarchy” of DFDs ordered by increasing information flow and processing detail. The top most diagram, the DFD 0, in this hierarchy consists of one bubble representing the entire system and “arrows” showing flow of information from this bubble to “external elements”. Each

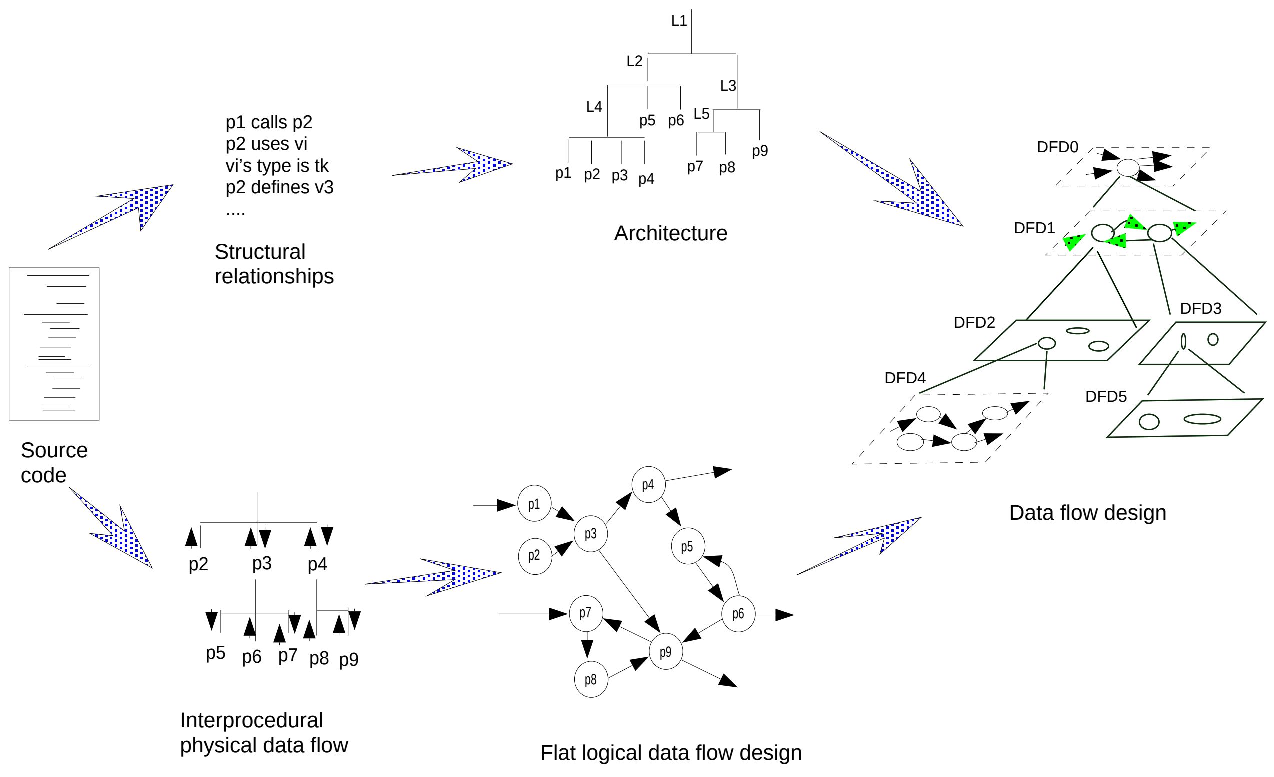

Figure 2 Schematic diagram of the steps proposed in recovering the data flow design of a software system from its source code. Algorithms and tools for activities on the right hand side of the dotted line will be investigated in this research.

DFD in the hierarchy "decomposes" a bubble of a parent DFD into bubbles denoting its subfunctions. The bubbles that have no decomposition are referred to as "terminal" or "leaf" bubbles.

Since all programs process data, DFD's can be used to abstract their data flow behavior. Figure 2 gives a schematic diagram of the approach proposed for recovering data flow design. The various terms used in the figure will be elaborated upon in the following discussion. Our discussion is restricted to DFDs that only have "bubbles" (transformers) and "arrows" (flow of data). The mechanisms to implement "data stores" and access to "external elements" (part of De Marco's DFDs) differ significantly between procedural languages. For instance, in COBOL the physical names of files processed is available in the header of a program; which is not the case in C. Hence we do not include information about data stores and external elements in our formulation of the problem.

The problem: *develop algorithms and tools to recover the data flow design of a software system from its source code*, may be subdivided into the subproblems of:

1. create a hierarchy of "processing elements" (bubbles) with *functions* and *procedures* of the software system at the "leaf", and
2. identify the content of the information that flows between processing elements at each level in the hierarchy.

such that:

1. the resulting data flow design is a *correct* abstraction of the actual system,
2. the hierarchy maintains *information flow continuity*, and

3. the decomposition at each level in the hierarchy is *satisfactory*.

Semi-formal definitions of the phrases *correct*, *information flow continuity*, and *optimal* are given below.

Definition: *Information flow continuity* [34] is the condition when the inputs and outputs of a bubble and the DFD decomposing it are identical.

We use the term *procedure* to denote both *function* and *procedure* (as used in Pascal).

Definition: A simple assignment of value of one variable to another is not considered "generation" or "use". A variable is *used* only if it appears on the right hand side of an assignment expression and that expression is not an identity function. And a value is *generated* if a variable appears on the left hand side of such an assignment expression.

Here "generated" and "use" have a more restricted meaning then that used in program flow analysis [2]. The definitions are used to introduce the notion of *logical* and *physical* data flow.

Definition: There is a *logical flow of data* from procedure A to procedure B if there is a potential execution path through which a data generated in A is transferred through a sequence of identity assignments to procedure B, and the data is used in B.

In other words, if a procedure only acts as a conduit to transfer data between two procedures, without transforming it, it participates in the *physical* data flow. There is a logical flow of data from a procedure that generates data to a procedure that uses it. This definition of *logical* and *physical* data flow is similar to that used by De Marco in Structured Analysis and System Specification [12]. In a data flow design we would like to extract the logical flow of data and not the physical flow.

Note: *Throughout this document, the term data flow has been used to imply logical data flow. The physical data flow is referred to as data dependence.*

Definition: *Inlining of DFD bubble* is the process of replacing a bubble in a DFD by the DFD that "decomposes" it (and making appropriate connections).

Definition: The data flow design resulting from inlining all the DFD bubbles that are not leaves (starting from DFD 0) is termed *flat data flow design*.

Notice that each bubble in a flat data flow design corresponds to a procedure of the system.

Definition: A *flat* data flow design is *correct* if it has an arrow from bubble A to bubble B iff there is an execution path from procedure A to procedure B and data "generated" in procedure A is "used" in procedure B.

Definition: A data flow design is *correct* iff its corresponding flat data flow design is correct.

To complete the recovered data flow design one also needs to identify the actual information that flows between processing elements. This may be either the name of the variable in the generating procedure whose value is used in the receiving procedure or the "type" of this variable. The various design and analysis methods using DFDs do not specify whether the annotation on the arrows is names of variables or types. Besides, there are trade-offs in either case. The problem, however, may be formulated without difficulty using either the name of a variable or its type. The trade-offs in either case are highlighted in Section 4.

The next section summarizes works in architecture recovery and interprocedural flow analysis relevant to the approach proposed for recovering data flow designs.

3 Survey of related works

Design recovery

The various design recovery techniques[1] may be classified on the basis of the level of abstraction of the recovered information and the method used to recover it.

- *Hypertext and typographic formats* abstract the software system as a book consisting of chapters, sections, etc. and provide links for ease of browsing and cross-reference [15, 31].
- *Resource flow graphs* extract the relationships between various components of a system, display these graphically, and/or provide means to query information about it [5, 8]. The cross-reference information provided by most compilers, call graph (A *calls* B *calls* C, etc.), and structure chart (i.e. call graph + parameters and global variables used in a call) fall in this category. There are several commercial and public domain tools that recover such information.
- In *Architecture recovery* the resource flow graph is analyzed to order structural elements of a system into a hierarchy. These works are of specific interest to this proposal and are analyzed in detail below.
- *Function Abstraction* is the analysis of the source code to create a higher level description of its functional behavior, for instance- formal specifications or programming plans [16, 17, 35]
  - *Knowledge based approach*: Create a knowledge base of programming concepts and determine a programs behavior by identifying the usage of these concepts in the code [16, 35].
  - *Flow analysis*: Combine data analysis, program slicing, and pattern recognition to abstract the function of a program [17].

Architecture recovery techniques

We term the problem of creating a hierarchy of procedures from structural relationships as architecture recovery. This problem has been addressed by other researchers [10, 20, 29, 26, 37]. Its use in recovering data flow design has not been proposed before. These techniques take as input the resource flow graph of a program and order the structural elements (usually procedures) of the program in a hierarchy. The techniques may be studied on the basis of the properties of the hierarchy and the approach used for creating it. The creation of hierarchy involves creating groups of structural elements and associating levels between these groups. The hierarchies created by the various architecture recovery techniques differ on whether their vertices are simple vertices or can they themselves be graphs.

The architecture recovery techniques suggested by Choi and Scacchi [10], Hutchens and Basili [20], and Schwanke [37] create hierarchies with simple vertices. The leaf of these trees consist of procedures (or other structural elements) of the program. The intermediate nodes represent a grouping of all its subtrees. In contrast, Müller and Uhl's technique results into hierarchy [30] in which each node may also be a graph made from the structural elements of the original program. The node of the hierarchy does not abstract the graph it contains. Instead the relationship between the original structural elements are preserved in the hierarchy as well. Müller and Uhl's hierarchy therefore only imposes an ordering and grouping of a program's structural elements.

The algorithms for creating the hierarchy may classified into two categories: graph theoretic or information theoretic. The former interpret the resource flow relationships as a graph and use some graph

[1] The IEEE Software, January 1990, carries a collection of papers on design recovery techniques.

theoretic property to create a hierarchy. Choi and Scacchi's [10] and Müller and Uhl's [30] algorithms are graph theoretic. Those of Hutchens and Basili [20] and Schwanke [37] are information theoretic because they use information theoretic approaches such as hierarchical clustering to create a hierarchy.

To create a hierarchy Choi and Scacchi find the biconnected components of a resource flow graph and replace these components by a new vertex. They then recursively do the same with the subgraphs induced by the biconnected components that were removed. Müller and Uhl split the resource flow graph into a set of bi-partite graphs whose cardinality is bounded by some constant k. The graph created using these bi-partite graphs as vertices, termed as $(k, 2)$-partite graph, is the recovered architecture.

Hutchens and Basili [20] and Schwanke [37] use the resource flow graph to compute the similarity (or dissimilarity) between the structural elements of the system. They then use hierarchical cluster analysis to organize the structural elements in a tree. The two methods differ on the functions they use to compute the similarity matrix. The details of these functions are not of interest to this proposal.

The techniques proposed by Choi and Scacchi, Hutchens and Basili, and Schwanke, in our opinion, are most suited for creating the hierarchy of processing elements. Müller and Uhl's [30] may not be very useful because the nodes of the recovered hierarchy are themselves graphs and hence do not reflect the notion of hierarchy in a data flow design.

Besides finding the logical data flow between elements of a hierarchy we are also interested in creating a hierarchy that is "satisfactory". The work on architecture recovery cited above evaluate the "goodness" of the recovered result by human inspection. Reference [24] gives a goodness metric that on a scale of 0 to 1 evaluates how well a recovered architecture compares with an expected architecture. This metric may be used to develop a quantitative measure of "satisfaction" with a recovered data flow design.

Flow analysis of programs

The flow analysis of programs generally refers to control and data flow analysis used in compilers to perform code optimization. Research in program flow analysis has been in progress, probably, ever since the first compiler was written. As a result it is beyond the scope of this document to perform an exhaustive survey of the field. The reader is referred to [2, 18, 28] for a detailed survey.

The work in interprocedural data flow analysis [3, 6, 7, 11, 25, 27, 38] is of interest from the point of view of this paper. Interprocedural analyses answer questions such as "if the value of a variable is changed at statement s_1 in procedure P_1 will it affect the computation of statement s_2 in procedure P_2". We are only interested in *static* analysis, i.e. the resulting answer should be true for all possible inputs to the program.

The *system dependence graph* (SDG) representation of procedural programs given by Horwitz, Binkley, and Reps' [19] appears to be a very promising starting point for extracting logical data flow. An SDG encodes the data and control dependence within each procedure as a procedure dependence graphs (PDG), a variation of program dependence graphs [13, 21, 32]. The SDG is formed by connecting the call vertices of a PDG to the entry vertices in the PDG. An SDG also contains edges representing data dependence between actual parameters and formal parameters.

The data dependence encoded in a system dependence graph provides the *physical* data flow. The *logical* flow of data may be derived by propagating the generation and use information through the dependence path in the PDG. Algorithms to propagate simple tags through an SDG are given in [19, 23, 22]. These algorithms may be generalized to propagate sets of values indicating the data flow information.

4 Current status and open problems

We have implemented several variations of architecture recovery algorithm proposed by Hutchens and Basili [20]. The variations were introduced to improve the “goodness” of the architecture recovered as measured using the goodness metric proposed in [24]. On a scale of 0 to 1, the best goodness measure we have so far received is 0.58. We are currently designing experiments to relate the goodness of a recovered architecture to various the design method and implementation decisions leading to the specific software system. The results of the experiment will help us in improving the architecture recovery algorithm. The goodness metric of [24] is based on an “expected” architecture, which may assumed to be unavailable when recovering design of a architecture in the field. Relating the quality of the results of architecture recovery algorithms to design decisions will allow one to “estimate” the quality of the results based on knowledge of the methods and conventions used in development of the subject software.

We are also working on algorithms to derive the logical data flow between processing elements from the physical data flow. Our current focus is programs written in a simple strongly-typed procedural language without global variables and aliasing. Following is a list of subproblems that are subject of our current investigation:

1. Develop algorithms to extract the *logical* data flow between procedures from the *physical* data flow (or data dependence) encoded in the system dependence graph. These algorithms will give the *flat data flow design* of a program. The algorithms may be a variation of the algorithms for computing interprocedural summary information [7, 11, 19].
2. Develop algorithms for identifying the “type” of data or “name” of the variable that flows between procedures and to create the data dictionary using information.
3. Develop algorithms to create a hierarchical data flow design by integrating the flat data flow design, the data dictionary, and the hierarchical clusters resulting from architecture recovery.
4. Integrate the data flow design recovered with a tool that may display the graph graphically, for instance EDGE [33].
5. Experimentally characterize the optimality of the data flow designs resulting from the various hierarchical clustering algorithms using my goodness metric and the optimality criterion defined by [1].
6. Design a measure of computing the goodness of a data flow design that takes into account the edge density in each DFD and the in-degree and out-degree of each processing element.

One problem that remains to be resolved is whether to use the “name” of a variable or its “type” to annotate edges of a DFD. The trade-offs are as follows. The name of a variable from a local context usually does not convey much information in a global context and very often conflicts with other variables of the same names. Thus if the variable whose value is “flowing” is a local variable, providing its name outside the context of usage may not be sound. The “type” of a variable, on the other hand, is an information global to the generator and user of data. A variable’s type may be more relevant when abstracting the flow of information between procedures. However, the type of a variable abstracts away too much detail when the variable in question is a global variable.

In our estimate the problem of indentifying the type of a variable flowing between procedure for a dynamically-typed procedural language is harder than that of identifying the name of the variable. But the second problem is equally difficult for both strongly-typed and dynamically-typed languages. The choice of a strongly-typed language for our initial effort simplifies the problem of finding the type of a variable whose value flows between procedures. This simplification allows us to postpone the decision of whether names of variables or their types should be used on the edges of a DFD. In the initial system both the methods may be implemented with a relatively low cost. The final decision may be made on based on the benefits perceived after actually using the system.

Bibliography

[1] M. Adler. An algebra for data flow diagram process decomposition. *IEEE Trans. Softw. Eng.*, Feb. 1988.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] F. E. Allen. Interprocedural data flow analysis. In *Proceedings IFIP Congress, 1974*, pages 398–402. North-Holland, 1974.

[4] T. J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, pages 36–49, July 1989.

[5] T. J. Biggerstaff, J. Hoskins, and D. Webster. DESIRE: A system for design recovery. Technical Report STP-081-89, MCC/Software Technology Program, Apr. 1989.

[6] M. Burke. An interval-based approach to exhaustive and incremental interprocedural analysis. *ACM Trans. Prog. Lang. Syst.*, 12(3), July 1990.

[7] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 47–55, June 1988.

[8] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Trans. Softw. Eng.*, 16(3):325–334, Mar. 1990.

[9] E. J. Chikofsky. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

[10] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, pages 66–71, Jan. 1990.

[11] K. D. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, pages 247–258, June 1984.

[12] T. De Marco. *Structured Analysis and System Specification*. Yourdon Press, New York, 1978.

[13] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, 1987.

[14] C. Gane and T. Sarson. *Structured Systems Analysis: Tools and Techniques*. Prentice-Hall, Englewood Cliffs, NJ, 1979.

[15] P. K. Garg and W. Scacchi. ISHYS designing and intelligent software hypertext system. *IEEE Expert*, 4(3), Fall 1989.

[16] M. T. Harandi and J. Q. Ning. Knowledge-based program analysis. *IEEE Software*, pages 74–81, Jan. 1990.

[17] P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner. Using function abstraction to understand program behaviour. *IEEE Software*, pages 55–65, Jan. 1990.

[18] M. S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.

[19] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, 1990.

[20] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. Softw. Eng.*, pages 749–757, Aug. 1985.

[21] D. J. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in FORTRAN-like programs and their resulting speed-up. *IEEE Transactions on Computers*, C-12(12), December 1972.

[22] A. Lakhotia. Improved interprocedural slicing algorithm. Technical Report CACS-TR-92-5-8, University of Southwestern Louisiana, Lafayette, November 1992 (submitted).

[23] A. Lakhotia. Constructing call multigraphs using dependence graphs. In *ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages (POPL'93)*, Jan. 1993 (to appear).

[24] A. Lakhotia, S. Mohan, and P. Poolkasem. On evaluating the goodness of architecture recovery techniques. Technical Report CACS-TR-92-5-4, University of Southwestern Louisiana, Lafayette, Sept. 1992 (submitted).

[25] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *ACM SIGPLAN'92 Conference on Programming Language Design and Implementation*, pages 235–248, July 1992.

[26] R. Lange and R. Schwanke. Software architecture analysis: A case study. In *Proceedings of the Third International Workshop on Software Configuration Management, Trondheim, Norway*. ACM Press, June 1991.

[27] D. B. Lomet. Data flow analysis in the presence of procedure calls. *IBM Journal of Research and Development*, 21(6):559–571, 1977.

[28] S. Muchnick and N. Jones. *Program Flow Analysis: Theory and Applications*. Prentice-Hall,Inc., 1981.

[29] H. A. Müller, J. R. Mohr, and J. G. McDaniel. Applying software re-engineering techniques to health information systems. *Proceedings of the IMIA Working Conference on Software Engineering in Medical Informatics (SEMI), Amsterdam,* Oct. 1990.

[30] H. A. Müller and J. S. Uhl. Composing subsystem structures using (K,2)–partite graphs. *Proceedings of the Conference on Software Maintenance*, pages 12–19, Nov. 1990.

[31] P. W. Oman and C. R. Cook. The book paradigm for improved maintenance. *Computer*, pages 39–45, July 1989.

[32] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. *ACM SIGPLAN Notices*, 19(5), May 1984.

[33] F. N. Paulisch and W. F. Tichy. EDGE: An extendible graph editor. *Software—Practice and Experience*, 20(S1):63–88, June 1990.

[34] R. S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, third edition, 1992.

[35] C. Rich and L. M. Wills. Recognizing a program's design: A graph parsing approach. *IEEE Software*, pages 82–89, Jan. 1990.

[36] D. T. Ross and J. W. Brackett. An approach to structured analysis. *Comput. Decisions*, 8(9):40–44, September 1976.

[37] R. Schwanke. An intelligent tool for reengineering software modularity. In *Proc. 13th International Conference on Software Engineering*, 1991.

[38] W. E. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables, and label variables. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan. 1980.

[39] N. Zvegintov. Issues in reverse engineering, re-engineering, and conversion. *Software Management News*, 10(7+8):10–12, July 1992.