A Toolset for the Reengineering of Complex Computer Systems*

Franz J. Kurfess
Mrinalini Lankala
Ashok Vantipalli

Computer and Information Science
New Jersey Institute of Technology
Newark, NJ 07102
franz@cis.njit.edu

Lonnie R. Welch

Computer Science and Engineering
University of Texas
Arlington, TX
welch@cse.uta.edu

Abstract

This paper describes a set of tools for the reengineering of computer-based systems, in particular software. The toolset is based on an abstract intermediate representation (IR) which incorporates the system software architecture at five levels of granularity: program level, task level, package/object instance level, subprogram level and statement level. The toolset provides a graphical user interface that allows various views of a software architecture, including call graph, rendezvous graph, call-rendezvous graph, call-data-rendezvous graph, control flow graph and dependence graphs. The information captured by the toolset is useful in software structure, flow and interaction analysis, tasks commonly performed manually during maintenance and reengineering. This information is also helpful for understanding the software design to guide software transformation, and for porting software to distributed platforms.

1 Introduction

The development of large software systems does not end with the installation of the executables on the target system: Apart from bug fixes, changes will have to be made to accomodate new functionalities, or to port the system to a new operating system or hardware platform. A complete redevelopment often is economically infeasible and unnecessary. In a situation that requires major modifications, e.g. the transition from a mainframe to a client-server environment, a decision has to be made which parts of the software system should be kept, which ones should be modified, and which ones have to be completely rewritten. The basis for such a decision relies to some degree on strategic factors, (the trustworthiness of a program, for example) but should also consider aspects reflecting the quality of the system with respect to current software engineering practices. Whereas it is not really clear what exactly determines the quality of a program, a number of metrics have been developed that express certain properties of a program in a numerical way based on quantifiable features of the program. A well-known example of such a metric is the McCabe complexity [8]; others are described in other publications [19].

The set of tools described here is centered around an intermediate, language-independent representation of the essential characteristics of such a system, and uses graphical displays of various interdependencies between program components for easier understanding of legacy software. In the following, the software reengineering pro-

*Supported in part by The U.S. NSWC (N60921-93-M-1912 and N60921-94-M-G096), by the U.S. ONR (N00014-92-J-1367), and by the State of New Jersey (SBR-421290 and SBR-431330).

0-8186-7889-5/97 $10.00 © 1997 IEEE

cess will be discussed briefly, followed by a description of the intermediate representation. After that, the individual components of the toolset will be presented in the form of a short tutorial. At the end, we provide an outlook into future work.

2 Software Reengineering

Some essential aspects of software re-engineering are the extraction of essential information from legacy programs; a language-independent format, the intermediate representation (IR); analysis and modification of the system, and transformation into the target language.

The reengineering process itself starts with the legacy system as input, then applies several steps with intermediate representations, metrics, and new configurations as intermediate goals, and finally integrates new requirements and objectives in order to produce the new system. The legacy system is the system to be reengineered (consisting of hardware, human and software elements) and all of its artifacts. Legacy system metrics are used to achieve a concise characterization of important aspects of the legacy system. The reengineering decision must answer to the question "Which components from the legacy system should be reengineered?". The first intermediate representation (IR1) contains an abstract representation of the legacy system, in machine-processable form. New requirements and objectives may have to be considered during the reengineering process. They provide a description of the constraints and desirable properties that the reengineered system is to have. The second intermediate representation (IR2) is an abstract representation of the new system, in machine-processable form. The new system metrics describe important aspects of the new system. The new configuration finally is a description of the interactions of the hardware, operating system, application software and human elements of the new system.

2.1 Reverse Engineering

The purpose of reverse engineering is to provide an understanding of the important aspects of the legacy system, like hardware, software design, and operating system.

The first step is a decision on the translation of the software. It is based on technical factors like properties of the legacy system, measured by metrics of some kind, as well as strategic and administrative reasons. Only if the decision is positive, the reengineering effort continues. In this case, the essential features of the system are extracted into the intermediate representation format IR1. Important components of IR1 are the symbol table (SymTab) and the statement table (StmtTab [10, 24]. They form the basis of several graphs representing dependencies and interactions between components of the system; these graphs are described in more detail in Section 5. Since the information extracted here tends to be overwhelming for human consumption, essential aspects are summarized in metrics[19].

3 The Reengineering Toolset: An Overview

Based on the reengineering and reverse engineering methods outlined above, a collection of software tools has been developed in collaboration between NJIT's Software Engineering Lab, the Navy's NSWC, and the University of Texas at Arlington. We currently have two versions of the toolset, one using C/C++ with Motif as front end and the second in Java; in this paper, the emphasis is on the Motif version. The main components of the toolset are

- parsers for translating legacy code,
- an intermediate representation format,
- extraction of the intermediate representation from the translated legacy code,
- various metrics to measure important system aspects,

- integration of graphical and textual information via hypertext,
- tools for parallelizing and distributing the new system on parallel machines or networks of workstations, and
- a graphical user interface to view relevant information.

3.1 The Main Window

The main window (see Figure 3.1) has nine buttons: Call Graph, Task Rendezvous Graph, Call Rendezvous Graph, Call Data Rendezvous Graph, Distribution Specification, Dynamic Scheduling, Load Application, Quit Tool, Help. All these button-options are views of an application that is to analyzed and therefore will work with respect to that particular application, and only after the application has been loaded. When the tool is first invoked the first six buttons are grayed-out, only the last three buttons are highlighted and accessible initially.

Clicking on the Help button will bring up the overall system on-line help; each screen of the tool has its own help-button that gives a detailed description of the view. The Quit button will exit the application. A click on the Load button will load an application. In the pop-up window, the application path to locate the code is entered. Now the various views can be examined to analyze the given application. The first, Call Graph is the application call graph representing the call relations. The second is the Task Rendezvous graph showing the task rendezvous. The third is the introduction of tasks in the call graph, which is the Call Rendezvous Graph. And the fourth is the summary of the first three graphs and also contains additional data objects and access information. The fifth button displays the Distribution Specification screen. This screen is the graphical representation of the application processes able to run on a distributed platform. This representation is also in the ASCII file in an internal format, DADS. The partitions are generated by parsing the application source code. The graphical interface allows the user to change the partitions and examine the resulting communication and concurrency costs. If the user arrives with a better partition, the interface provides the facility to save the changes to the DADS specification (ASCII) file, and DADS can execute the application on a distributed platform with the better partition. The sixth button is the Dynamic Scheduling of the execution process of the application. By clicking on this button, a pop-up screen will appear with the ability to load an application or run an already loaded demo application. This screen has a grid with nodes as the application processes. As the application is executing, it shows the communication between the processes.

4 Intermediate Representation

Comparing the quality of programs written in different languages based on their source codes is unpractical at the best; this should be done on the basis of a representation which is as independent as possible of the particular language used. We use an intermediate representation that captures the essential statical and dynamical aspects of a program or large software system, and represents them in an appropriate way, independent from a particular programming language. Its main parts are

- a symbol table,
- an extended statement table,
- various relations between program components, and
- an analysis and evaluation of the program.

For each statement, the statement table contains the relevant information [20]. A short overview of the different graphs is given in the following sections; detailed information can be found in other publications [20, 19].

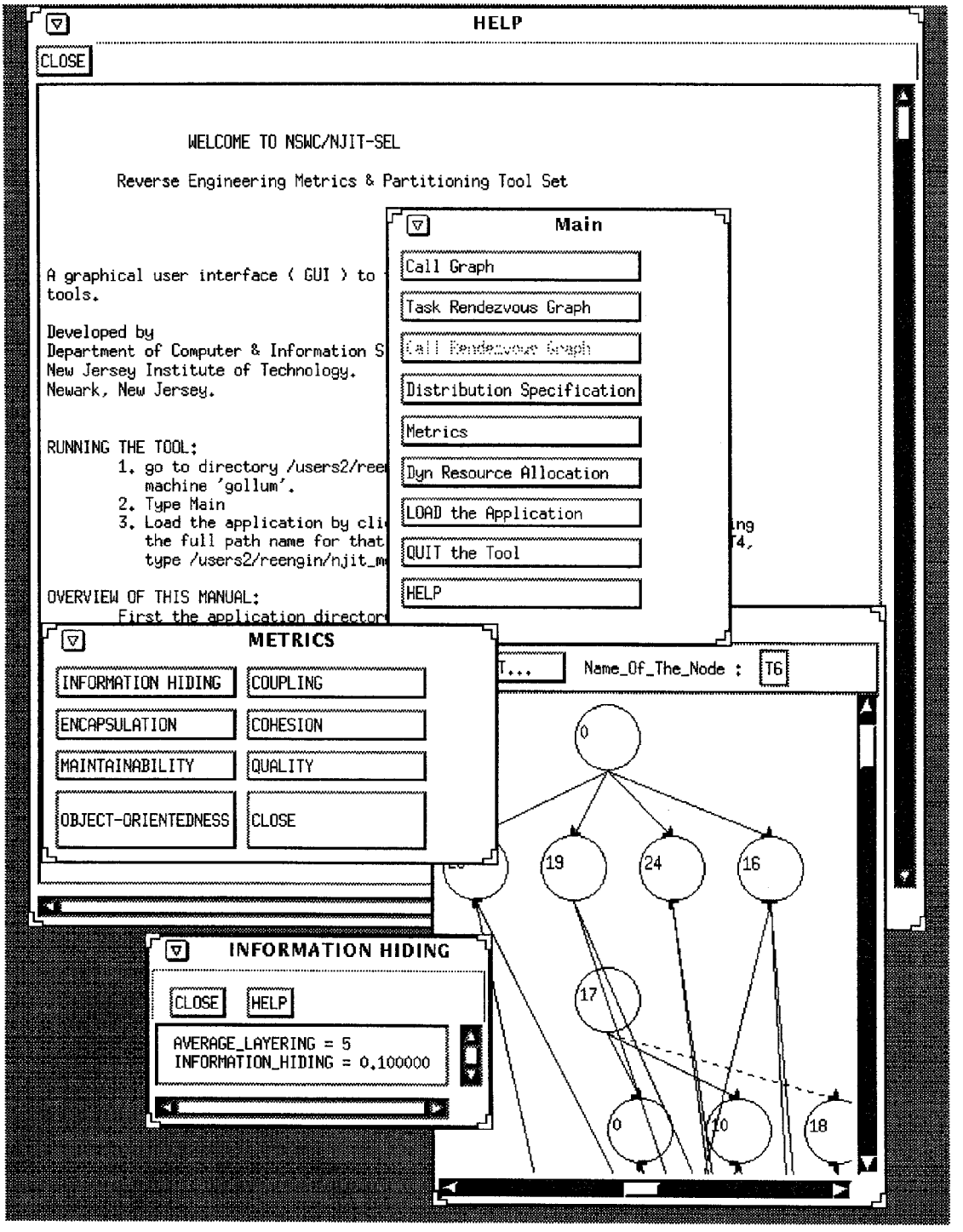

Figure 1: Main Window

5 Dependence Graphs

In general, dependence graphs are constructed on the basis of the statement table, which usually is defined for a unit of the program at a certain level, e.g. subprograms. Dependence graphs represent program statements as nodes and use di-

rected edges to denote statement ordering implied by the dependences in a source program.

Different kinds of ordering requirements are represented in different dependence graphs. In the data dependence graph (DDG) a directed edge denotes a data dependence (which means that destination and source nodes need the same variable). The instance dependence graph (IDG) uses undirected edges to denote instance dependences (which occur when two nodes use operations exported by the same instance). The subprogram dependence graph (SDG) uses an undirected edge to denote when two statements use the same subprogram. A directed edge in the control dependence graph (CDG) denotes that execution of the destination statement depends on a decision made by the source statement. In addition to the dependence graphs, the control flow graph (CFG) is extracted at the statement level, indicating the sequential flow of control dictated by the order of the statements in the source code. The analysis of dependences between system components is also used as the basis for distributing the components of a system among different processing elements.

5.1 Statement Dependence Graphs

Relationships between program units are visualized by statement dependence graphs. Various types of dependences on the statement level can be of interest, and a graphical representation frequently is easier to inspect than the table-based one.

5.2 Control Dependence Graphs

The flow of control, and the corresponding relationships between program components, are shown in control dependence graphs. A very important control aspect is the call relationship between procedures or other program units.

5.3 Data Dependence Graphs

Relationships between data structures are visualized in the data dependence graphs. Changes in the value of one data item can have consequences for other items, e.g. if their values are computed on the basis of the former.

5.4 General Dependence Graph

In many cases, it is necessary or more convenient to inspect various kinds of dependences simultaneously. The integration of control, data and instance dependences in one graph is also referred to as a general dependence graph. The obvious potential drawback is the complexity of the resulting graph: it can easily become confusing to be faced with a large number of different lines connecting the nodes of the graph.

5.5 Call Graph

The call graph of an application is built by parsing the application source code. It represents the call relationships among the modules of the application. The toolset parses the source code and builds an ASCII file, which represents the call graph. The ASCII file is represented in X-Windows using sophisticated graph layout algorithms. Packages are represented by blue circles and subprograms are represented by purple triangles. It is a directed graph, the nodes represent the modules of the application and the edges represent the call relationships among the modules. The initial parser was built for the Ada language, hence the modules/nodes in this case are packages and sub-programs. When a node is double-clicked, a window will pop up listing all the methods of that node/package (if that package has methods in it). Clicking on the method another window will show three labels - Source code, Metrics and Dependence Graphs. Clicking on the source code label will show the source code of the method selected. Clicking on the Metrics label a window will result in the list of method-level metrics (McCabe and Halstead). The Dependence Graph button will produce a window displaying all the method-level graphs. All graphs are built similar to the call-graph by parsing the source code and generating an ASCII file, which is read and displayed in the window.

5.6 Task Rendezvous Graph

The Task Rendezvous Graph represents the rendezvous among tasks of an application. The tool parses the source code and generates an ASCII file which represents the task relations in the application. This ASCII file is then used to display the graph. This graph is a directed graph, and the nodes represent the tasks of the application. The edges indicate task rendezvous, and the direction of the graph determines the caller and the callee. Tasks are represented by red squares, and the task rendezvous are pink-colored edges. The downward arrows are solid-lines and the upward edges are two-colored (dashed) lines.

5.7 Call Rendezvous Graph

The Call Rendezvous Graph represents the call and rendezvous relations of the application; it is a combination of the application call-graph and tasks rendezvous graph. The nodes of the graph are packages and tasks. This is a directed graph where edges represent the call and rendezvous relationships of the application, and the direction of the edge determines the caller and the callee. The upward edges are two-colored (dashed) lines and the downward edges are solid blue lines. If a node is double-clicked a window will pop up displaying all the methods of that node (a package or task).

5.8 Call Data Rendezvous Graph

The call data rendezvous graph represents the call relations, the rendezvous relations and the data-object access in the application. The source code of the application is parsed to generate the ASCII file representing the graph. This is a directed graph, the nodes represent packages, tasks and data-objects. The edges represent the calls, the rendezvous and the data accesses of the application. The package nodes are blue circles, the task nodes are red squares and the data-object nodes are shown as green rectangles. The downward edges are drawn as solid blue lines and the upward edges are drawn using two-colored dashed lines. The names of the nodes (packages/tasks/data-objects) are written on the nodes and with a single click on the node can also be displayed on a button on the menu bar. There is help button on the top right hand side of the screen, which describes the graph generation method. When a node is double-clicked, a window will pop-up displaying the methods of that node.

6 Dependence Graph Displays

The primary purpose of the various graphs is the visual display of relationships between different program units. Displaying this information visually, however, is only useful if the arrangement of the items displayed is easy to understand by the user of the tool. The initial version of the tool set used an ad hoc graph display algorithm which did not try to optimize the appearance of a displayed graph. In a second version, a more sophisticated version of a graph display algorithm is used, aiming at a systematic arrangement of the nodes and links, with relatively few crossings of links.

The usage of large graphs is supported by additional features like zooming into areas of particular interest, automatic adaptation to the current window size, and focusing on a particular node or region of the graph.

7 Related Work

The authors have been involved in efforts [15] to reengineer portions of the AEGIS Weapon System from CMS-2 to Ada, and to migrate from militarized AN/UYK-43s to commercial workstations. These projects were performed for two primary reasons: to aid in the refinement of a process for reengineering control systems, and to provide proven algorithms for an experimental open system hardware and software environment (HiPer-D) directed at defining the future architecture and functionality of Navy ship computer systems.

Related work has also been performed within other projects. In [2], an approach is presented for capturing abstractions inherent in software

systems and for transforming those abstractions into an object-oriented paradigm; the focus was not on concurrency, but large-scale systems were considered. The consideration of concurrency is proposed in [9], by considering the translation of operating system calls into Ada constructs. Techniques and tools have been developed for source-to-source translation of program code [14, 1]; these tools are pragmatic, allowing a reengineered system to become operational quickly, but they do not attempt significant transformation. Additionally, several techniques and tools have been developed to perform basic dependence analysis, including the Xinotech program composer [22], a tool and language independent IR developed by MITRE [13], and Refine [11], which performs reverse engineering of code written in Fortran, Cobol, C and Ada. However, none of these tools attempts to perform the analysis required for enhancement of concurrency and object-orientedness, or for partitioning and mapping. Other techniques and tools for dependence analysis are presented in [4, 12, 3]. A hierarchical approach to reverse engineering was taken in [5], but the levels of the hierarchy were not based on granularity, as in our model, but consisted of implementation, structure, function and domain levels.

8 Future Work

Work in progress and planned for the near future includes other front ends (C++, Java, Pascal, Fortran, COBOL), and the realization of a full Web version implemented in Java. Metrics are refined, and new ones investigated based on the integration of Artificial Intelligence techniques, in particular neural networks [7]. Distribution and parallelization tools are extended towards a limited consideration of dynamical aspects of program execution. Finally, the toolset is tested for its real usefulness through its application to real-world reengineering problems.

9 Conclusions

This article describes a tool set for reverse engineering and reengineering of complex computer-based systems. The tool set is based on intermediate information extracted from the legacy system, and is used to make important features of the system explicit to the system engineer. The graphical display portion visualizes some of these aspects, e.g. the dependence relations between various program units, and provides additional help with the understanding of the system to be reengineered. The methods described as well as initial versions of the tool set have been successfully applied to components of the Navy's AEGIS Weapon System [20]. In order to increase usability of the system, a Web version is currently under development, allowing remote access to the tools without the need to install them locally.

References

[1] G. Arango et al., "Maintenance and Porting of Software by Design Recovery," *Proceedings of The Conference on Software Maintenance*, pages 42-49, IEEE CS Press, 1985.

[2] T. J. Biggerstaff, "Design Recovery for Maintenance and Reuse," *IEEE Computer*, volume 22, number 7, July, 1989.

[3] C. Castells-Schofield, "Engineering a Language-Independent Approach to Parsing for Analysis and Testing," *Vitro Tech. Journal*, volume 8, number 1, 1990.

[4] S. Dietrich and F. Calliss, "A Conceptual Design for a Code Analysis Knowledge Base," *Software Maintenance: Research and Practice*, volume 4, 1992.

[5] M. Harandi and J. Ning, "Knowledge-Based Program Analysis," *IEEE Software*, volume 7, number 1, 1990.

[6] F.J. Kurfeß, X. Pandolfi, Z. Belmesk, W. Ertel, R. Letz, and J. Schumann. PARTHEO and FP2: Design of a parallel inference machine. In P.C. Treleaven, editor, *Parallel Computers: Object-Oriented, Functional and Logical*, chapter 9, pages 259–297. Wiley, 1989.

[7] F.J. Kurfess and L.R. Welch. Categorization of programs using neural networks. In *International IEEE Symposium and Workshop on Engineering of Computer Based Systems (ECBS '96)*, pages 420–426, 1996.

[8] T. McCabe, "A Complexity Measure", *IEEE Transactions on Software Engineering*, Vol. SE-2, December, 1976.

[9] N. Prywes, G. Ingargiola, I. Lee and M. Lee, "Reengineering Concurrent Software into Ada," *Proceedings of The Fourth Systems Reengineering Technology Workshop*, pages 157-177, Naval Surface Warfare Center, February 1994.

[10] B. Ravindran,"Extracting parallelism at compile-time through dependence analysis and cloning techniques in an object-based paradigm," M.S. Thesis, New Jersey Institute of Technology, May 1994.

[11] Reasoning Systems, Palo Alto, CA, "Refine Language Tools," 1993.

[12] C. Rich and R. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, volume 7, number 1, 1990.

[13] H. Rubenstein, R. Piazza, and S. Roberts, "Separating Parsing and Analysis in Reverse Engineering Tools," *Proceedings of the Working Conference on Reverse Engineering*, May, 1993.

[14] C. H. Sampson, "Translating CMS-2 to Ada," *Proceedings of The Fourth Systems Reengineering Technology Workshop*, pages 143-156, Naval Surface Warfare Center, February 1994.

[15] A. L. Samuel, E. Sam, J. A. Haney, L. R. Welch, J. Lynch, T. Moffit, and W. Wright, "Application of a Reengineering Methodology to Two AEGIS Weapon System Modules: A Case Study in Progress," *Proceedings of The Fifth Systems Reengineering Technology Workshop*, Naval Surface Warfare Center, February 1995.

[16] R. A. Steigerwald and L. R. Welch, "Reusable Component Retrieval for Real-Time Applications," *Proceedings of the First IEEE Workshop on Real-Time Applications*, May 1993.

[17] J. P. C. Verhoosel, L. R. Welch, D. K. Hammer, and A. D. Stoyenko, "Assignment and Pre-Runtime Scheduling of Object-Oriented, Hard Real-Time Parallel Processes Using Bead Partitioning," New Jersey Institute of Technology Technical Report CIS-93-16, December, 1993.

[18] L. R. Welch, A. D. Stoyenko and S. Chen, "Assignment of ADT Modules with Random Neural Networks," *The Hawaii International Conference on System Sciences*, IEEE, Jan. 1993.

[19] L. R. Welch, M. Lankala, W. Farr and D. Hammer, "Metrics for Quality and Concurrency in Object-Based Systems."

[20] L.R. Welch, G. Yu, B. Ravindran, F.J. Kurfess, J. Henriques, and M. Wilson. Reverse engineering of complex Navy systems. *International Journal of Software Engineering and Knowledge Engineering*, 6(4), December 1996.

[21] L. R. Welch, G. Yu, J. Verhoosel, J. A. Haney, A. L. Samuel, and P. Ng, "Metrics for Evaluating Concurrency in Reengineered Complex Systems," *Annals of Software Engineering*, **1(1)**, Spring 1995.

[22] Xinotech Research Inc., Minneapolis, MN, "The Xinotech Program Composer 2.0," 1992.

[23] G. Yu and L. R. Welch. Program Dependence Analysis for Concurrency Exploitation in Programs Composed of Abstract Data Type Modules. In *Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 66-73, October 1994.

[24] G. Yu, "Identifying and Exploiting Concurrency in Abstract Data Type-based Systems," PhD Thesis, New Jersey Institute of Technology, Sept. 1995.