Reverse Engineering and Reusing Software

Dr. James M. Neighbors
System Analysis, Design and Assessment

(excerpts from an unfunded 1989 SBIR proposal)

1 Project Summary

1.1 Technical Abstract

Software Reuse is a key technique for achieving significant increases in programmer productivity. However, the reuse of software will only succeed if the engineering decisions made during the construction of the software are understood. *Reverse Software Engineering* (RSE) creates such an understanding. This project will develop a knowledge representation capable of describing systems from different algorithmic languages (e.g., Ada, CMS, Pascal). The initial phase of the project will focus on extracting system architecture (*"How"*) descriptions of working systems from their source code and supporting documentation. Later phases will integrate system requirement (*"What"*) and system performance models (*"How many"*) into the representation. It will be shown that many of the workproducts of the forward software engineering life cycle may be constructed from the representation. The problems of system validation, program translation, maintenance programming, systems programming and parallel computation of existing system function will be formulated as problems across the system requirement, architecture, and performance representation.

1.2 Anticipated Benefits

The ability to produce a system architecture representation from source code provides: 1) increased maintenance programmer productivity by providing an understanding of a system as it currently exists; 2) increased systems programmer productivity through software reuse; 3) a basis for the translation of systems between algorithmic languages (e.g., CMS to Ada); and 4) a framework for integrating system requirement and performance models to the existing, working system.

1.3 Eight Keywords

Software Reuse, Reverse Engineering, Program Translation, Program Understanding, Software Portability, System Validation, Software Engineering, Parallel Computation

Contents

1 Identification and Significance of the Problem or Opportunity

While computing hardware technology continues to increase on a cost/performance basis by an order of magnitude every 2.4 years the productivity of the labor-intensive process of software construction has increased *only 3 percent per year over the last 30 years* [9]. An increase in software productivity is required to be able to take advantage of the computing hardware of today and tomorrow.

1.1 Reverse Engineering and Software Reuse

Software Reuse has been identified as a key technique for achieving significant increases in programmer productivity by simply avoiding the expensive process of constructing all software from scratch [6, 14, 13, 18]. A problem with software reuse is understanding the structure and function of the software intended for reuse. *Reverse Software Engineering* (RSE) provides this understanding for an existing software system or component. The applicable forms for such information are similar to the workproducts developed during the forward Software Engineering process.

1.2 Large System Context

Our work over the last five years has been to understand the structure of large software systems and the failure modes of their development. The goal was to determine the size and scope of *reusable software components* which are appropriate for large systems development by programmers and semi-automatic agents [11].

Very small software components (such as sorts and list management [4]) are helpful in the construction of large software systems, but do not significantly reduce the development time of a very large system.

> "… packages are not a *sufficient* mechanism for decomposition or reusability. The reason for this is that there are some abstractions that are simply too intellectually large to be conveniently captured in a single package."[4, pg. 556]

The *software architects* of large systems must reason with units much larger than these small software components. The systems are so large that they require architects to insure the system's integrity through sound structural building practices.

The approach we have used to understand large system structure has been to reverse engineer commercial software systems. This provided the developing companies with a better insight on their system's structural design and allowed our access to actively modified, large systems. To date three large systems have been reversed: a voice/data communications switching network (2mega lines in Pascal, C, and assembly), a computer-aided design and manufacturing system (CAD/CAM, 2mega lines in FORTRAN and C, DoD prime contractor), and a computer-aided engineering system (CAE, 4mega lines in FORTRAN, C and assembly, DoD prime contractor). Many smaller systems in Ada, Pascal, and assembler have also been reversed as test experiments.

In this work we experimented with informal tools for scanning the software and saving information about the structure found. Much of the information is similar to the information that an optimizing compiler produces but the information from all of the individual "compilations" is kept. This data is used to extract the architectural design structures of the system.

We found the following structure in large systems:

1. A large "mature" system (greater than 5 years in age) has one programmer for approximately every 10k to 30k lines of source code.

2. A large system is built out of highly cohesive large subsystems which are approximately 10k to 30k lines of source code. These subsystems consist of between 25 to 300 major modules.

3. The mapping of programmers to large subsystems is not 1:1. *Making the mapping of programmers to large subsystems 1:1 can bring a failing development effort back under control.* It provides quality by placing a tightly coupled set of modules with an interface under the control of a single programmer. A pride of ownership and workmanship arises in the support of a natural interface.

4. The subsystem is the unit of reusable software component for which we were looking. Thus, the "architectural parts" with which the architects of a large software system reason are between 10K to 30k lines. A megaline system in not thought of as 1 million lines or 3500 modules, but rather as between 30 and 100 interacting subsystems.

5. The simple informal tools we constructed to scan and analyze the huge amounts of source code quickly became part of the development toolset since they were capable of answering "what if I make this change" questions for the programmers. Many of the questions were module interconnection questions such as "do only my subsystem's modules call this module". The individual programmers use the tools to defend the interfaces of their subsystems as a module interconnection language (MIL) [16] processor might do.

6. The informal tools (even though we rewrote them over and over at each commercial site) turn out to be very similar even for strongly typed algorithmic languages and assembly language. Only the amount of information which can be extracted directly from the source code varies.

The primary goal of phase I of this project is to turn the experience with these informal tools into a formal tool and demonstrate its usefulness on an existing system.

1.3 Knowledge Representation

Three basic kinds of information are involved in the description of a software intensive system. Most work in system understanding deals with system requirements rather than system architecture or performance.

1.3.1 System Requirements Knowledge

System requirements state *"what"* a particular system is to do and the constraints on the physical environment in which it is embedded. The difficulty with such knowledge is its inherent open-endedness since each system operates in a different problem domain. The interactions between the system specifier and the system analyst provides both the requirements of the system and a model of the problem domain on which the system will operate.

The process of *problem domain analysis* inherent in every system requirements specification is particularly error-prone even for small systems. The specifier and analyst must agree on the objects and operations of the problem domain for the system. Since requirements can be a problem even for small systems (<50k lines) it was one of the first problems encountered and historically more work has been done on it than large architectural models.

1.3.2 System Architecture Knowledge

The system architecture representation primitives are concerned with the programming domain rather than the problem domain. The *"how"* explanations speak the language of the designer and programmer. Structural objects such as data types, instances of types (variables), definition scopes, interval analysis, data flow analysis (live, dead and uninitialized variable analysis), coupling of modules, cohesion of modules, module groupings (packages), package groupings (subsystems), and connection languages (MILs) are architectural information. Notice that these concepts are all common to algorithmic languages developed in the last 30 years.

Compilers construct some of this information for each compilation unit in an individual language, use it for code optimization, code generation, and then promptly throw the information away. This information will be constructed and saved for system structural analysis. We have found with our informal tools that the gross architectural structure of a system may be constructed from the code. The addition of actual design documents and system designer interaction can the help disambiguate and refine the structure which exists in the executing software.

Historically, more work has been done on *programming-in-the-small* system structures such as data types, control constructs, and procedural encapsulation than has been done on *programming-in-the-large* structures such as packages, subsystems, and connection languages. The reason may be that the need was not apparent. If you are building a simple structure (small system or dog house) then specification is most important while the lack of a good architecture will seldom cause a failure. If you are building a complex structure (large system or skyscraper) the architecture is just as important as the requirements. A minor architectural flaw can destroy the entire project. *Many large software intensive system failures are architectural failures not requirements failures.*

1.3.3 System Performance Knowledge

There are three basic means of obtaining system performance data:

1. *Analytic solutions* provide a mathematical formula for the time (number of operations) or space required to provide a specific function. The O(n) complexity notation is an example of this kind of information.

2. *Simulation statistics* are statistical measurements taken on a computational model which purports to simulate the architecture of the system under study. Activation models such as finite state machines (FSM), Petri-Net models, and simulation languages are examples of this kind of information.

3. *Execution monitoring* is statistical measurement of the actual system (or actual parts of the system) in execution under a test load.

The performance goals of the system are given in the system requirements while the structures to be measured are given in the system architecture. The performance data comes from sources outside the reverse engineering tool such as execution monitors and simulation languages. The system architecture representation must be rich enough to provide descriptive "hooks" on which to hang the performance data. As an example execution monitors usually instrument each source code statement for execution counts. The system architecture representation must be able to identify the computation associated with such a unit.

System architecture knowledge will be of primary concern during phase I of this project. However, allowance must be made for the integration of system requirements and performance information in later phases.

1.4 System Validation

System validation attempts the difficult job of interrelating the representations of system requirements, system architecture, and system performance to provide accountability for each in terms of the others. Obviously it is impossible to do this without all three representations. The weakness of architectural and performance representations has made it difficult to provide existing system rationalizations for system requirements.

1.5 Maintenance Programming requires Reverse Engineering

A maintenance programmer approaching an existing system with the intent of causing a structural or functional change cannot reasonably expect existing design documentation to answer specific questions about the change in any detail. If the design documentation attempted to consider all possible changes it would increase to a very large and unmanageable size. Instead maintenance programmers become expert at very quickly examining large sections of programming language code to determine the impact of a given change. This examination is carried out on the structure of the existing system. The examination is a *reverse engineering* of the architectural and detailed design of the system from a code level. *An opportunity exists to improve maintenance programming productivity if we can provide support for this reverse engineering activity.*

1.6 Systems Programming requires Reverse Engineering

A systems programmer constructing a new system or subsystem while practicing software reuse from *reusable software components* or from similar components in existing systems has the same problem as the maintenance programmer. The interconnections and dependencies between the candidate software component and its development environment must be determined quickly. In many cases, large functions of the original component must be removed without damaging the remaining functions [17, 8]. Keep in mind that our studies show that the size of the code considered for reuse in large systems should be between 10k and 30k lines.

Languages such as Ada which foster the reuse of software through the package concept will increase the need to be able to examine a large suite of modules, remove unwanted functions safely, restructure the subsystem, and then incorporate the reused software into a new system. *An opportunity exists to improve systems programming productivity if we can provide support for this reverse engineering and reuse activity.*

1.7 Program Translation

The architectural knowledge representation provides a very strong basis for *program translation.* Translation from one algorithmic language to another requires a "deep structure" of how the program is providing its function. Initial Artificial Intelligence (AI) attempts at natural language translation (e.g., French to English) were syntactic (word for word, phrase for phrase) translations similar to today's program translation. Significant gains in automatic natural language translation were only made after a "deep structure" common to both languages was used as an intermediate form. As with a human translator, what is being said in the natural language (the problem domain) is not fully understood by the translator, but the expressive schemes of the individual languages (the architectures) are understood by the translator. The expressive schemes of algorithmic languages (types, variables, control constructs, packages, subsystems) are much more restricted than those of natural languages.

1.8 Parallel Computation and Architectural Descriptions

Converting an *existing* software base for parallel computation will first require knowing “how” the software performs its function. For the initial transformation “what” the system does is relatively unimportant as long as the system function is preserved by the translation.

Many multicomputer architecture schemes are called “dataflow computers” because of the central role that the flow of the system’s data plays in allocating parts of the process to different processors. This data is basic to system architecture but is seldom available for an existing system. Some parallel computation speed may be obtained by exploiting parallelism inherent in a system implementation by examining the flow of data between different parts of the system. However, extreme care must be taken that the communications overhead of data passing between processors doesn’t overwhelm the computation improvement from using multiple processors. To effectively partition an existing system into a cooperating set of processors we must know the static data flow connections in the particular system (system architecture information) and the degree to which those connections are used in the system during a representative execution (system performance information).

Ultimately, the largest gains in parallel computation will only be obtained by reworking the theories of the problem area (system requirements information). Our work in *domain analysis* [13, 14] has shown that the optimizations which may be gained from optimizations in the problem domain are much more powerful optimizations than those available on a particular implementation of the theory. As an example consider the computer-aided engineering (CAE) finite-element method (FEM) which analyzes physical structures, the last 25 years of theory work has made the method efficient on vectorizing supercomputers where the partitioning of data is not as important as the order of arithmetic operations. A massively parallel computer such as a “connection machine” will place the partitioning of data in importance before arithmetic operations. This will require FEM theory to know what partitionings of data are useful.

1.9 Pragmatics

The project goal is a reverse engineering system which may be used in many organizational environments and is accessible to the first line programmers. Improving the abilities of the individual programmers will have the highest impact on software productivity [3]. This requires some pragmatics to be considered.

1.9.1 Data Collection Computation Cost

Scanning the complete source code of a megaline system takes about the same time as compiling all of the compilation units of the system (a 1 MIP CPU takes between 30 and 40 CPU hours). The process can seriously degrade the performance of a large timesharing mainframe. It may easily produce 50 to 60 megabytes of information about how the system is interconnected. It is difficult to get this degree of CPU and disk service from a timesharing mainframe especially if the mainframe is also used for system development.

1.9.2 Size and Structure of Data Collected

Managing the information about a large system requires a large database where the data changes slowly over time as the modules are modified by the programmers. There are many more queries than updates. It is difficult to achieve database program portability between mainframes used by different organizations.

1.9.3 Graphical Analysis Needs

Graphical browsing of connectivity data is a basic software engineering workproduct. The graphical rendering of the cooperating modules of a subsystem are invaluable. The two largest systems which we have reversed (CAD/CAM system and CAE system) both had high resolution graphic displays available. The large communications system had no graphics capability and we found that each programmer had drawn structure charts of different parts of the system by hand. These messy hand-drawn charts where seldom shared.

1.9.4 Small Computer Solution

The simplest way we have found to provide 80 megabytes of disk storage, medium resolution graphics, and the ability to run CPU and I/O intensive analysis programs for individual first-line programmers is to use a stand-alone small computer such as the IBM PC or Apple MacIntosh. Small stand-alone computers are inexpensive and available in most organizations. They can provide a stable and portable platform for the reverse engineering software. Typically the source code control system on a mainframe used for development can be directed to send all updated source codes to the small computer for incremental analysis.

2 Phase I Technical Objectives

2.1 Getting the Data

1. *Source Code Scanners:* A user- modifiable scheme for describing the source syntax for source code scanning for algorithmic languages needs to be developed. Augmented Backus-Naur Form (BNF) is usually used to produce LR or LL parsers.

2. *Type Algebra:* Most algorithmic languages are strongly typed and the type information is used to disambiguate parses of the language. The type algebra needs to be able to augment the source language syntax to provide type information. The BNF is augmented with the type algebra to produce parsers which produce trees augmented with type information.

2.2 Annotating the Data

Assuming the supporting documentation for a system has the content and form of DoD-STD 2167A (Defense System Software Development) and is available in electronic form, it needs to be determined how this documentation may be used to annotate the information gathered from the source code. Either the source code or the documentation should be able to be used as an index for the other.

2.3 Storing the Data

1. *Data Schema:* A flexible data schema needs to be defined to hold the basic scanned information about the algorithmic language, its supporting documentation, and the results of analysis of the representation. Functional Dependencies (FDs) formed into Entity-Relationship (ER) diagrams which are synthesized into third normal forms (3NF) are the usual descriptive schemes for such information.

2. *Database:* The queries required to be answered by the database should determine the expressive power needed in the database. If the analyzers and examination tools require relational queries then a relational database mechanism is required. Efficient access and large storage are the criteria for selection.

2.4 Examining the Data

1. *Tabular Reports:* Many reports on the system are the usual textual printouts such as cross-reference.

2. *Graphical Reports:* Many workproducts of the software engineering life cycle are graphical such as structure charts, type dependency diagrams, and data flow diagrams. For a large system the complexity of the underlying graph theory must be considered and an appropriate device independent graphical interface must be determined.

3. *Browsing:* The large amount of information requires some interaction to simply reduce the amount of information which must be considered. If a system has 3500 modules then a structure chart of the entire system would be inappropriate to contemplate a local change.

2.5 Analyzing the Data

1. *Interval and Dataflow Analysis:* Interval Analysis which is the first phase of Data Flow Analysis can provide valuable information about the system. These algorithms need to be implemented for individual system modules (as in compilers) and across modules in the system (as is very rarely done).

2. *Coupling and Cohesion Analysis:* Coupling (how tightly coupled are two modules) and cohesion (how cohesive are statements in this module) are two *quality measures* which are derivable from the architecture representation.

2.6 Restructuring the Data

1. *System Restructuring:* Restructuring the system into cohesive subsystems is dependent on the control and dataflow in the system.

2. *Module Interconnection Specifications:* Once a system has been restructured module interconnections languages (MIL) [16] can be used as a defense of the newly created subsystem boundaries.

3 Phase I Work Plan

3.1 Phase I Strategy

In the previous section we identified the reverse engineering phases of 1) getting the data; 2) annotating the data; 3) storing the data; 4) examining the data; 5) analyzing the data; and 6) restructuring the data. *The project's strategy for phase I is to produce a prototype reverse engineering system which integrates at least some function from each of the reverse engineering phases.* The SBIR phase I timeframe will significantly shorten the depth of each phase; but the feasibility of the approach may only be shown by an integration of the phases.

3.2 Final Products to be Delivered

The final product to be delivered will contain:

1. A prototype reverse engineering tool for the IBM PC environment with user documentation.

2. The source code of a large, public-domain system which will serve as the target for this feasibility experiment. The Ada Software Repository (ASR) on the network host at White Sands Missile Range (WSMR-SIMTEL20.ARMY.MIL) contains large public-domain systems.

3. The reverse engineering database resulting from the application of the prototype tool to the demonstration target system.

4. A demonstration of the information available from the tool about the target system.

5. A report on the experience of using the tool on the target system and an estimation of the technology contained in the tool.

The prototype system will be executable on an IBM PC in the funding agency's environment to provide a "hands on" evaluation of the technology for phase II support.

3.3 Detailed Plan

These tasks would appear to take much more than the 6 month timeframe of SBIR phase I development. However, our experience with unintegrated, informal tools which performed some of these functions places the development effort ahead on the learning curve.

3.3.1 Phase I Task List

The tasks needing to be performed to create and demonstrate the reverse engineering tool prototype are:

1. BNF Parser descriptions (2 weeks).

2. Develop type algebra (1 week).

3. Augment BNF parser descriptions with type algebra (1 week).

4. Develop at least two parser descriptions. (Ada and Pascal, 3 weeks)

5. Develop database schema capable of representing the data from scanning, annotation, and analysis. Describe relations as functional dependencies, produce Entity-Relationship diagrams and synthesize as 3NF (3 weeks).

6. Select a database technology capable of supporting the prototype schema (1 week).

7. Implement interval and dataflow analysis on database schema results from scanning (2 weeks).

8. Implement tabular cross reference reports on the database schema (1 week).

9. Implement graphical reports on the database schema (e.g., structure charts and data dependencies) (2 weeks).

10. Implement subsystem clustering analysis on the database schema (2 weeks).

11. Implement simplistic program translation between the two algorithmic languages (2 weeks).

12. Select a large, public-domain target system for reverse engineering demonstration (2 days).

13. Import source code of demonstration target system (1 day).

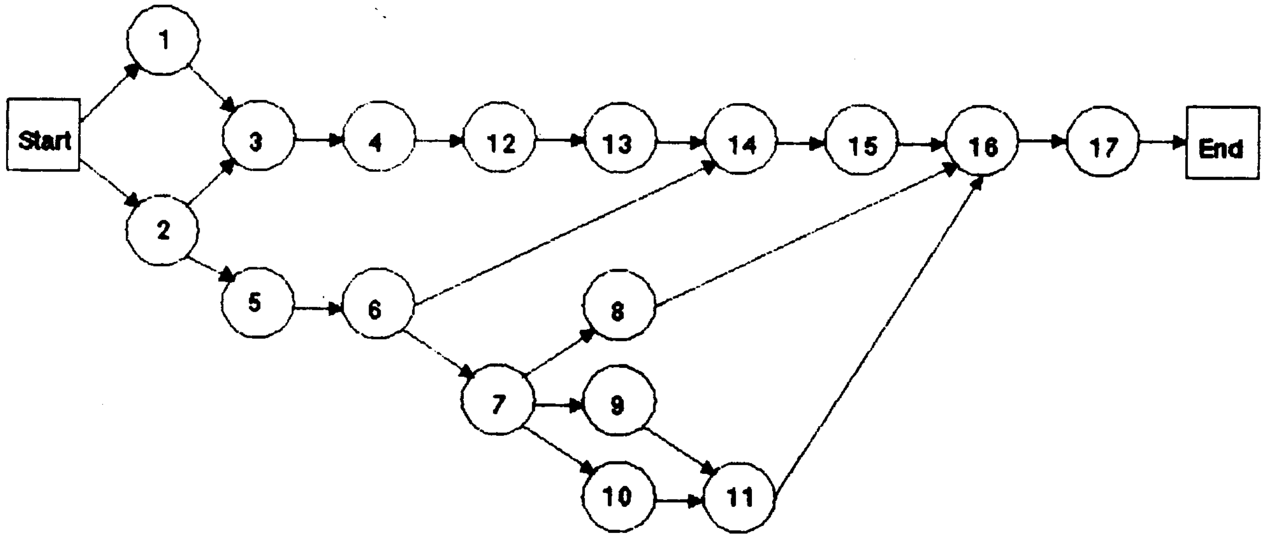
14. Scan demonstration target system (4 days).

15. Annotate demonstration target system (1 week).

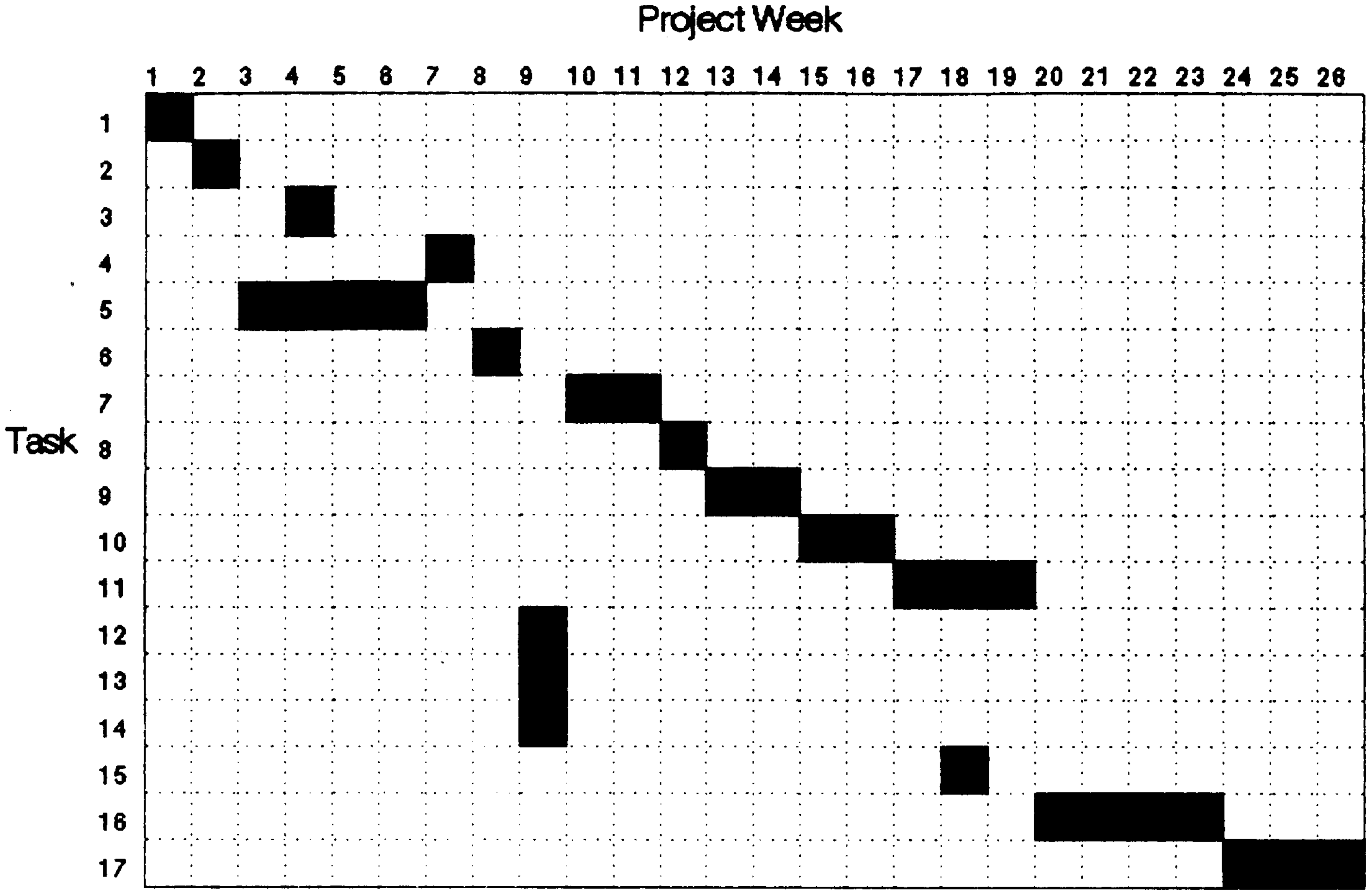
16. Demonstrate prototype tool, produce reports on the target system, and suggest subsystem restructuring (4 weeks).

17. Produce report on the experience of use (3 weeks).

3.3.2 Phase I Pert Chart



3.3.3 Phase I Gantt Chart



4 Related Work

4.1 Software Engineering and Software Reuse

All of the forward software engineering workproducts are candidate reverse engineering workproducts. Software engineering workproducts include: requirement data (DoD-Std 2167A and ANSI/IEEE Std 830-1984), dataflow diagrams (DFDs as used in SADT, IDEF, SA, SSA), structure charts (as used in SD, JSD), activation models (Petri-Nets and finite state machines), policy description tools (decision tables, decision trees, program description language(PDL)), data description tools (entity-relationship diagrams, functional dependencies, normal forms) and many others [5]. These representations are useful to system developers and as such are a good candidates to provide system architecture understanding.

Software Reuse requires information to evaluate the potential reuse and modification of a software component. The information required is similar to that produced by reverse engineering a software system. The concepts important to software reusability are surveyed in [6, 18, 1]. Our earlier work in software reuse [14, 13] was successful only in building smaller systems (less than 20k source lines) because of the problem of software system architecture. The component parts with which we were reasoning were too small to produce larger systems. The Draco synthesis system became overwhelmed with small details. This prompted the study of large system structure in an attempt to understand the granularity with which the human architects of large systems reason.

4.2 Reverse Engineering

Brachman Engineering's Data Analyst (1988) is probably the best known commercial reverse engineering product. This tool specializes in reverse engineering IBM mainframe COBOL database systems. Most commercial "reverse engineering" systems are aimed at restructuring COBOL EDP applications.

A presentation of our initial results in reversing large systems [12] prompted the Microelectronics and Computer Technology Corporation (MCC, a research consortium in Austin, Texas) to start a research project on Design Recovery for Maintenance and Reuse. Their approach is to use hypertext systems to provide different views of the abstractions of programs and structures [2].

4.3 Development Environments

Many commercial systems are able to provide information about a systems source code. DEC's VaxMate package of source code tools is representative of the best tools available. VaxMate includes: 1) Module Management System (MMS) which is similar to the Unix "make" facility; 2) Code Management System (CMS) which is similar to Unix's source code control facility; 3) Source Code Analyzer (SCA) which provides information about the variables in the source code; and 4) Performance Coverage Analyzer (PCA) which can provide execution monitoring data from test runs of the system. The are two basic problems with tools like the VaxMate package: 1) there is no common database structure so the tools cannot share information and 2) there is no analysis of the data by the tools. The tools simply provide textual reports of the given data. No analysis, graphical descriptions, or restructuring of the data takes place. Also the failure to use a database causes the tools not to scale up. DEC does not recommend the use of these tools for systems with over 400 modules.

Research on development environments has focussed on the *process* of creating software systems. Current work in the area is surveyed in a recent IEEE Transactions on Software Engineering [15]. The management of the hundreds of programmers required to build a large system is a laudable goal, but it only obliquely address the problem of the architectural structure of the system being developed. Some issues like the use of a common database for information and the extraction of interdependencies between parts of the system have been addressed. A few environments [see Hudson in 15] provide some "fine grain" information resulting from the application of *data flow analysis* to the source code. However, as with the commercially available tools, the development environment research does not emphasize the *capturing and restructuring* of the system architecture. This is still viewed as a manual task. For a large system this cannot be a manual task.

4.4 Program Understanding

Most of the work in system understanding attempts to understand the system requirements ("what the system does") rather than the system architecture ("how it performs its function"). This causes most of this research not to scale up to large system sizes since "what a system does" involves *problem domain* knowledge. We cannot expect a general program understanding or reverse engineering system to know much about the problem domains to which computers will be applied. Indeed, even small systems use knowledge from many problem domains.

Typically research in this area is based around program understanding during program synthesis where problem domain knowledge is present (see Kant or Barstow or Waters in [10]) or general program understanding in a very small problem domain [7]. These works do provide some general structural program analysis rules (programming cliches) independent of the problem domain which can be used to recognize tightly coupled regions of the system.

4.5 References

[1] Biggerstaff, T., and Perlis, A., Eds., *Reusable Software Engineering*, Addison-Wesley, 1989.

[2] Biggerstaff, T., "Design Recovery for Maintenance and Reuse", MCC Technical Report STP-378-88, November 1988.

[3] Boehm, B., *Software Engineering Economics*, Prentice-Hall, 1981.

[4] Booch, G., *Software Components with Ada*, Benjamin/Cumminings Publishers, 1987.

[5] Freeman, P., Ed., *Software Design Techniques* 4/e, IEEE Press, 1987.

[6] Freeman, P., Ed., *Software Reusability*, IEEE Press, 1987.

[7] Johnson, W., and Soloway, E., "PROUST: Knowledge-Based Program Understanding", *IEEE Transactions on Software Engineering*, Vol SE-11, No 3, March 1985.

[8] Kibler, D., Neighbors, J., and Standish, T., "Program Manipulation via an Efficient Production System", *Proceeding of Conference on Pattern Directed Inference Systems*, SIGPLAN Notices 12(8):163-173, August 1977

[9] Morrissey, J., and Wu, L., "Software Engineering: An Economic Perspective", *4th International Conference of Software Engineering*, IEEE Press, September 1979.

[10] Mostow, J., Special Issue on Artificial Intelligence and Software Engineering, *IEEE Transactions on Software Engineering*, Vol SE-11, No 11, November 1985.

[11] Neighbors, J., "Draco: A Method for Engineering Reusable Software Systems", in Biggerstaff, T., and Perlis, A., Eds., *Reusable Software Engineering*, Addison-Wesley, 1989.

[12] Neighbors, J., "The Structure of Large Systems", MCC talk, August 1987.

[13] Neighbors, J., "The Draco Approach to Constructing Software from Reusable Components", *IEEE Transactions on Software Engineering*, Vol SE-10, No 5, Sept 1984.

[14] Neighbors, J., "Software Construction using Components", Ph.D. Dissertation, Information and Computer Science Department, University of California, Irvine, January 1980.

[15] Penedo, M., and Riddle, W., Special Section on Software Engineering Environment Architectures, *IEEE Transactions on Software Engineering*, Vol SE-14, No 6, June 1988.

[16] Prieto-Diaz, R., and Neighbors, J., "Module Interconnection Languages", *The Journal of Systems and Software*, No 6, pp. 307-334, Elsevier Science Publishing Co., Inc., 1986.

[17] Standish, T., Harriman, D., Kibler, D., and Neighbors, J., “Improving and Refining Programs by Program Manipulation” *Proceedings of the ACM National Conference 1976*, pp 509-516

[18] Tracz, W., *Software Reuse: Emerging Technology*, IEEE Press, 1988.

5 Relationship with Future Research or Research and Development

5.1 Anticipated Results

We anticipate that the prototype reverse engineering tool developed will be able to make a large body of souce code intelligible to a programmer in a short period of time.

5.2 Phase I foundations for Phase II work

The system architecture and performance representations developed in phase I lay a strong foundation for phase II work in the following areas:

1. *System understanding* based on classical forward software engineering workproducts derived from the existing working system, its documentation and interactions with its maintenance personnel.

2. *System quality measures* based on coupling and cohesion at the control flow, module, package, and subsystem levels.

3. *Component specialization* is the process of reusing a software component with some of its functionality removed. The specialization mechanism should be able to safely remove sections of the software no longer needed.

4. *Program translation* is the recasting of an algorithmic language automatically into a language of similar expressive power (e.g., CMS to Ada).

5. *System restructuring for quality control* identifies and goups definitions & source lines into modules, definitions & modules into packages, and definitions & packages into subsystems to maximize cohesion and minimize coupling.

6. *System restructuring for parallel processing* uses system architecture knowledge (e.g., dataflow analysis) and system performance knowledge (e.g., execution monitoring) to restructure the system to minimize data communication overhead (minimize data coupling with respect to bandwidth) and maximize concurrent program segment sizes (maximize control flow cohesion).

7. *System validation* links system requirements, system architecture, and system performance representations to be able to explain one in terms of the other two. Naturally all three types of information are required.

6 Potential Post Applications

6.1 Potential use by the Federal Government

Even software-intensive systems which go through *perfect* software engineering development life cycles are enhanced and maintained over their useful life. For many reasons the actual system structure tends to drift away from that captured in the delivered system documentation. Ultimately, we are left with an enhanced system which works but has no accurate system architecture description. A programmer wishing to fix, enhance, replace or reuse part of the system must revert to the source code to determine the system architecture. Reverse software engineering provides automated support for this process.

6.2 Potential Commercial Application

To date most of our work in reverse software engineering has been done on commercial systems which go through constant change in a attempt to provide releases every 6 months. The commercial need to reverse engineer working systems is as great as the Federal Government's need for the same reasons.