PROGRAM COMPREHENSION FOR REVERSE ENGINEERING

Spencer Rugaber

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
spencer@cc.gatech.edu

INTRODUCTION: REVERSE ENGINEERING

This paper motivates and describes a research program in the area of reverse engineering being conducted at the Georgia Institute of Technology. Reverse engineering is an emerging interest area within the software engineering field. Software engineering itself is concerned with improving the productivity of the software development process and the quality of the systems it produces. However, as currently practiced, the majority of the software development effort is spent on maintaining existing systems rather than developing new ones. Estimates of the proportion of resources and time devoted to maintenance range from 50% to 80%. [Boehm, 1981]

The greatest part of the software maintenance process is devoted to understanding the system being maintained. Fjeldstad and Hamlen report that 47% and 62% of time spent on actual enhancement and correction tasks, respectively, are devoted to comprehension activities. These involve reading the documentation, scanning the source code, and understanding the changes to be made. [Fjeldstad and Hamlen, 1979]

The implications are that if we want to improve software development, we should look at maintenance, and if we want to improve maintenance, we should facilitate the process of comprehending existing programs. Reverse engineering provides a direct attack on the program comprehension problem.

Definition

The process of understanding a program involves reverse engineering the source code. Chikofsky and Cross [Chikofsky and Cross II, 1990] give the following definition. "Reverse enginering is the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." The purpose of reverse engineering is to understand a software system in order to facilitate enhancement, correction, documentation, redesign, or reprogramming in a different programming language.

Difficulties

Reverse engineering is difficult. It is difficult because it must bridge different worlds. Of particular importance are bridges over the following five gaps.

- The gap between a problem from some application domain and a solution in some programming language.
- The gap between the concrete world of physical machines and computer programs and the abstract world of high level descriptions.
- The gap between the desired coherent and highly structured description of the system and the actual system whose structure may have disintegrated over time.
- The gap between the hierarchical world of programs and the associational nature of human cognition.
- The gap between the bottom-up analysis of the source code and the top-down synthesis of the description of the application.

The difficulties manifest themselves in three ways: lack of a systematic methodology, lack of an appropriate representation for the information discovered during reverse engineering, and lack of powerful tools to facilitate the reverse engineering process.

METHODOLOGY

Background: Bottom Up versus Top Down

There are two approaches to understanding a program: bottom-up, starting with the source code and generating a description; and top-down, formulating hypotheses and confirming them by examining the program. An example of the former is the approach taken by Soloway and Ehrlich. They propose a bottom-up model of analysis based on the recognition of *plans* in the source code. [Soloway and Ehrlich, 1984] The plans are organized into subgoals and then goals. Experiments have been conducted that support this approach, and Letovsky has built an analysis tool that implements part of the analysis process. [Letovsky, 1988] Other examples of the bottom-up approach are

the program analyzer component of the Programmer's Apprentice project [Rich and Wills, 1990] and the work of Basili and Mills based on control flow analysis and formal documentation. [Basili and Mills, 1982]

The top-down approach is championed by Ruven Brooks. In his approach, the program understander attempts to recreate a series of mappings between the application domain and the program. Exploration is driven by expectations derived from the application description. [Brooks, 1983] There have been some human factors experiments that support Brooks' ideas.

Synchronized Refinement

Rugaber et al. have developed an approach, called Synchronized Refinement, that coordinates the bottom-up analysis of the source code with the top-down synthesis of the application description. [Kamper and Rugaber, 1990; Rugaber *et al.*, 1990] It produces a description of the functioning of the system annotated by references to locations in the program text that implement the various aspects of the application. The description is highly cross-referenced, indicating how programs are built from component pieces that are interleaved to accomplish the total purpose.

Synchronized Refinement has been used to help analyze numerical, data processing, and real-time software systems ranging in size from one hundred to one million lines of code written in Fortran, Cobol, and PL/M (a system programming language). The application of Synchronized Refinement proved to be highly labor intensive, reflecting the need for support tools. Moreover, the descriptions that were constructed and the source code segments that were analyzed were managed using regular text files. This emphasized the need for a comprehensive representation or *data model* based on which a data base can be constructed to hold the information. Finally, it should be pointed out that all program analysis methods are just a part of the overall reverse engineering task that also requires consideration of file structures, organization of runs, user interface, etc..

REPRESENTATION

Requirements for a Representation

A key ingredient for successful reverse engineering is a suitable representation for the understanding obtained when analyzing source code. In order to design such a representation, it is important to understand how it might be used. The following requirements hold for a representation suitable for dealing with reverse engineering information.

- **Requirements Related to the Information Content of the Representation:** The representation must be able to contain a variety of types of information. These include informal rationale and annotations, program segments, pointers to other documentation, and application descriptions. Most importantly, it must be able to represent the organization of the program in terms of detected abstractions. In fact, the reverse engineer constructs a complex information structure that describes the organization of the program and the interrelationships of its pieces. There must be a place in the representation to hold observations made by the reverse engineer during this process.
- **Requirements Related to the Relationships Among the Data Being Represented:** The representation is constructed incrementally by the reverse engineer. It must allow an observation concerning a section of code to be associated both with related sections of code and with the overall functional description being constructed. This includes both hierarchical connections among abstractions and heterarchical (cross-reference) associations. Finally, the representation should support instances where a section of code contains several components interleaved together.
- **Requirements Related to How the Representation is Constructed:** The representation needs to be easy to construct incrementally, both computationally and from a user interface point of view. Additionally, it should be language independent in the sense that it can be used during the reverse engineering of programs written in a variety of languages and programming paradigms.
- **Requirements Related to How the Representation is Used:** The representation must be formal enough to support automatic manipulation. For example, after a program has been reverse engineered into the representation, it should be possible to apply tools to adapt segments for reuse. This process is called *transformational programming*, and a variety of such transformations exist. [Feather, 1987; Partsch and Steinbruggen, 1983]
- **Requirements Related to How the Representation is Accessed and Viewed:** A predominant use of the representation will be to facilitate program browsing. That is, a maintenance programmer desiring to fix a bug or make an enhancement needs to be able to peruse the information structure either to answer specific questions (which functions call a given function), obtain an architectural overview (in graphical form), or locate a specific section of the code (where are all of the statements that could affect the final value of a given output variable). The representation must, at the same time, be independent of any particular design method or notation and be capable of generating information in any of a variety of formats.

Design Decisions

When a program is constructed, the original designer makes a series of decisions that break the problem solution into pieces and then indicates how the pieces work

together to solve the problem. It is natural to base a methodology for reverse engineering on the recognition of design decisions in code. Furthermore, the representation for the information detected during reverse engineering is naturally structured to reflect the interrelationships of the code segments used to implement the detected decisions.

Synchronized Refinement is based on the detection of design decisions in code. Moreover, a representation is being designed to satisfy the requirements mentioned above that uses design decisions as a structuring mechanism. Design decisions and how they can be recognized are described in. [Rugaber *et al.*, 1990] The description is summarized here. Design decisions can be divided into several classes based on the type of abstraction they provide. Among the classes that are useful to detect during the reverse engineering process are the following.

- Composition/decomposition - Programs are built up from parts, and problems are broken down into smaller, more easily solvable sub-problems. This type of decision is manifested in the code by such constructs as modules and data structures.
- Encapsulation/interleaving - Subcomponents interact with each other. If the interactions are limited and occur through explicit interfaces, the component is said to be encapsulated. If, usually for reasons of efficiency, two or more plans are realized in the same section of code or by the same data structure, then the components corresponding to those plans are said to be interleaved.
- Generalization/specialization - Often one component is similar to another. It may then be possible to construct a higher level parameterized component capable of realizing both as special cases. In object-oriented programming, the process is often reversed, with the more general component constructed first, and the special cases added later.
- Representation [1] - In translating from the problem domain to the solution domain, decisions are made that result in a program component serving as a model for some application domain entity. If efficiency is a concern, high level programming constructs can be further represented by other constructs closer to the machine, such as using an array to represent a stack. Languages such as Ada are emerging that support explicit representation, but the reverse engineering of programs from older languages requires the detection of these decisions.
- Data/Procedure - Programs are sequences of computations organized by control structures. Variables are ways of saving intermediate results for later use, either to avoid recomputation or to simplify the expression of the computation. The introduction of a variable is an important design decision that is, unfortunately, too easy to make without appropriate thought and annotation.
- Non-determinism removal - In some situations, a designer has a choice of how to express the relationship between input and output parameters. This is particularly true in logic programming languages, such as Prolog. A single relationship, expressing a high level specification, can lead to alternative functions depending on the modes (input/output designations) of the parameters. This decision is usually made at a very early stage of design, if it is made explicitly at all.

[1] This use of the term *representation* should not be confused with its use as a notation for capturing a high-level understanding of a program.

TOOLS

Synchronized Refinement is a labor-intensive process for reverse engineering a program. Many of its component activities are, however, automatable using well-understood techniques. The information that the tools produce needs to be saved in a data base so that it can be later accessed by software maintainers.

- **Analysis Tools:** Recognizing design decisions requires intensive, non-linear access to the source code. When a decision is suspected, it often needs to be confirmed by examining related sections of code. Also, the code needs to be manipulated so that the details of the decision can be hidden and a summary displayed in its place. Many of the decisions are detected by recognizing syntactic patterns of program constructs and variable usage. Their recognition involves much of the same processing as occurs during the early stages of compiling a program. In fact, the artifacts of parsing a program, the abstract syntax tree and the symbol table, can serve as a source of data from which to build an initial representation of the program.
- **Browsers/Hypertext:** If a program is suitably analyzed and stored in a structured fashion, browsing activities are facilitated. In particular, the abstract syntax tree can serve to guide those perusals that are aimed at understanding the hierarchical nature of the program code. Likewise, the symbol table information can serve to support the cross-reference-like queries. The technology being described bears a striking resemblance to that of hypertext systems. [Conklin, 1987] There, high bandwidth displays and direct manipulation interfaces are used to explore non-linear organizations of text. In the case of reverse engineering, the text is source code and the non-linear relationships are provided by the parser.
- **Object Server:** The information structure being assembled from detected design decisions needs to be saved in a repository for use by software maintainers. Although some of its organization is supportable by existing data base systems, these are not en-

tirely adequate. In the same sense that CASE tools are turning to object-oriented data bases and object servers in order to support forward engineering activities, reverse engineering needs to be supported by non-traditional methods.

- **Task-Oriented Tools/Debugging:** Once a comprehensive information structure is populated with information about a program, tools specific to a particular software maintenance task can be applied. The data model and the information structure are the prerequisites for an integrated collection of tools. As an example consider the following debugging tool. The software maintainer begins with a trouble report that indicates that a program is producing unexpected output on a given run. The maintainer desires to quickly localize the problem to a small segment of the code. He uses a tool that indicates for a given set of correct and incorrect output values, which statements are potentially responsible for the problem. The tool examines the dependency relationships among the program statements and the execution history of the program to determine the appropriate statements. The tool has a mode where only relevant statements are displayed in a given situation. In this way the maintainer can concentrate on the appropriate code sections. The tool makes it determination from the information contained in the data base.

CONCLUSIONS

Five Gaps

The introduction to this paper describes the reasons why reverse engineering is difficult. The remainder of the paper presents an integrated approach to solving these problem based upon a methodology called Synchronized Refinement. This approach is based upon the detection of design decisions in the source code and the organization of the information into an information structure suitable for browsing by software maintainers. This approach addresses the five gaps discussed in Section 2 in the following ways.

- Application domain/program domain - Synchronized Refinement involves the parallel exploration of the source code and construction of a functional description of the application domain. The process itself constructs the bridge between them.
- Concrete/abstract - Synchronized Refinement constructs an information structure that organizes low level details into more abstract constructs. The process continues until a concise high level description of the program's main purpose is expressed.
- Coherency/disintegration - It is only by looking at the overall structure of a program that organizational difficulties can be appreciated. Synchronized Refinement constructs a representation of the actual structure of the program and allows the reverse engineer to annotate the representation with questions and suggestions about improvements. Moreover, the data model enables the construction of transformation tools useful for improving the structural aspects of the program.
- Hierarchical/associational - The data model supports a variety of relationships including both hierarchical and cross-reference information. Moreover, the model is extensible to new relationships deemed appropriate by the reverse engineer.
- Bottom-up/top-down - Synchronized Refinement coordinates both of these activities.

Productivity and Quality

The introduction also discusses how reverse engineering relates to other activities in the software engineering field. Software development productivity and quality are improved if programs can be enhanced instead of being rebuilt. They are also improved if major pieces of existing systems can be reused with reduced effort. These activities require that software engineers know in detail what existing programs do.

The purpose of this research program is to support the comprehension process. The support comes in the form of a methodology called Synchronized Refinement. The methodology produces an information structure suitable for use by software maintainers in understanding programs and adapting them for alternative uses. Moreover, it enables the construction of tools, such as debuggers and program transformers, useful in maintaining and improving software quality.

References

Basili, V. R. and Mills, H. D. 1982. Understanding and documenting programs. *IEEE Transactions on Software Engineering* SE–8(3):270–283.

Boehm, Barry W. 1981. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ.

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

Chikofsky, Elliot J. and Cross II, James H. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 7(1):13–17.

Conklin, J. 1987. Hypertext: An introduction and survey. *IEEE Computer* 20(9):17–41.

Feather, Martin S. 1987. A survey and classification of some program transformation approaches and techniques. In Meertens, L. G. L. T., editor 1987, *Program Specification and Transformation*. Elsevier North Holland. 165–195.

Fjeldstad, R. K. and Hamlen, W. T. 1979. Application program maintenance study: Report to our respondents. In *Proceedings GUIDE 48*, Philadelphia, PA. Also in *Tutorial on Software Maintenance,* G. Parikh and N. Zvegintozov, editors, IEEE Computer Society, IEEE Order No. EM453.

Kamper, Kit and Rugaber, Spencer 1990. A reverse engineering methodology for data processing applications. Technical Report GIT–SERC–90/02, Software Engineering Research Center, Georgia Institute of Technology.

Letovsky, Stanley 1988. *Plan Analysis of Programs.* Ph.D. Dissertation, Yale University.

Partsch, H. and Steinbruggen, R. 1983. Program transformation systems. *ACM Computing Surveys* 15(3):189–226.

Rich, Charles and Wills, Linda M. 1990. Recognizing a program's design: A graph-parsing approach. *IEEE Software* 7(1):82–89.

Rugaber, Spencer; Ornburn, Stephen B.; and Jr., Richard J. LeBlanc 1990. Recognizing design decisions in programs. *IEEE Software* 7(1):46–54.

Soloway, E. and Ehrlich, K. 1984. Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering* SE–10(5):595–609.