WHITE PAPER ON REVERSE ENGINEERING

Spencer Rugaber

College of Computing
and
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450

March 9, 1994

WHITE PAPER ON REVERSE ENGINEERING

Spencer Rugaber

College of Computing
and
Software Engineering Research Center
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450

1. 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%[6].

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[12].

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.

1.1. Definition

The process of understanding a program involves reverse engineering the source code. Chikofsky and Cross[8] 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.

1.2. 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.

1.2.1. The Application Domain and the Program Domain

Programs are models or representations of problem situations from some application domain. There may or may not be hints in the program about the particular problem. Hints can take the form of mnemonic variable names and in-line comments. The hints are inherently informal and tend to be out-of-date with respect to the program. Because of this, totally automatic reverse engineering tools are restricted to working with the formal program text. It is the job of the reverse engineer, then, to reconstruct the mappings from the application domain to the program domain[7]. This of course requires knowledge, not only of programming, but also of the application domain. It is no surprise, therefore, to find that most automatic tools are restricted to analyzing the program text and do not address the application domain. An exception to this is the DESIRE system being built at MCC[4].

1.2.2. The Concrete and the Abstract

Computer programs are incredibly detailed. In essence they control the values of million of bits of memory inside of the computer. One of the jobs of the reverse engineer is to decide, from all this detail, which are the important concepts. This process is called abstraction; the reverse engineer must create an abstract representation of the program from the mass of concrete details.

The abstraction process is not linear. That is, a given section of a program may be a part of several abstractions. The abstractions are said to be *interleaved* or *delocalized* in the section[20]. Typically there is no explicit indication in the source code of the interleaving.

Abstraction involves pattern recognition, and the larger the vocabulary of patterns, the more likely that an appropriate one can be recognized. Unfortunately, some of the abstractions reside in the application domain where they are less accessible to the reverse engineer.

1.2.3. Structure and Chaos: Finding a Plan Where None Exists

When a program is originally constructed, there is a coherent structuring of details. The process that creates the structuring is called design. A large variety of design methods and representation techniques have been developed to aid this process[3, 22]. Although programming languages have some features intended to facilitate abstraction and structuring, the higher-level design representations may have been lost or allowed to become out-of-date by the time reverse engineering is required. More importantly, through maintenance activities such as porting, bug fixing, and enhancement, the original structure of the program may have deteriorated[2]. That is, it is the job of the reverse engineer to detect the purpose and high level structure of a program when the original purpose of the program may have changed and where, in fact, there may be no such single purpose left in the program.

1.2.4. The Formal/Cognitive Distinction

Computer programs are highly formal. They have strict rules that limit the expression of ideas and that control how those ideas effect the computer when they run. The two types of rules are called, respectively, syntax and semantics. In the formal world, the meaning of a syntactically correct program determines the output that is produced when a specific input is presented.

Human cognition, to the extent that it is understood, is not at all formal. Comprehension works associatively. Raw data are perceived, patterns are detected, and higher level chunks are constructed from lower level concepts[19]. The process is controlled by expectations derived from the application domain of the program and the large body of programming knowledge held by the reverse engineer: knowledge of the programming language, typical programming practices, and the application domain. A program is understood to the extent that the reverse engineer can build up correct high level chunks from the low level details available in the program.

1.2.5. Top-Down and Bottom-Up

When a reverse engineer looks at a program, he or she is detecting patterns that indicate the intent of some section of code. Low level patterns are part of higher level constructs intended to accomplish larger purposes. In this way, the process of analyzing a program proceeds bottom-up[21].

At the same time, the programmer has some idea of the overall purpose of the program and how it might be accomplished. As the program is perused, the overall concept is refined into more lower level details[7]. This synthesis process proceeds top-down. The difficulty is that both of these activities need to proceed at the same time, in a synchronized fashion[17].

2. HOW ARE THESE DIFFICULTIES MANIFESTED?

These then are the difficulties that reverse engineers face: the distance between the application domain and the programming language used; the need to construct an abstract description from a concrete artifact; the loss of structure inherent in software under maintenance; the need to understand a program using human cognition where only formal methods and tools exist; and the need to synchronize top-down and bottom-up activities.

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.

2.1. Methodology

A methodology is a comprehensive procedure with a specific production goal and a mechanism for determining whether the goal has been reached. A methodology adds predictability and repeatability to an activity. This provides valuable information to management that can be used to adjust staffing and schedules. Builtin feedback mechanisms can provide valuable quality checks. Once a methodology has been defined, tools can be built to support it. Furthermore, training materials, standards, and courseware can be developed to support it.

Methodologies abound in the area of initial software development. Methodologies for reverse engineering are in their infancy. Until systematic methodologies are developed and validated, the benefits of management feedback, quality control, tool support, and training materials will be limited.

2.2. Representation

There is no agreed upon form for expressing the results of reverse engineering a software system. Such a descriptive mechanism might take the form of a graphical representation, a formal notation, a data model, or an informal description. For a description of two approaches to the representation of this information, see[5] and[9].

Without such a mechanism, understanding cannot be communicated among maintainers, management cannot appreciate the extent and quality of the understanding, and tools cannot be integrated.

2.3. Tools

Reverse engineering tools are currently limited in power. Among the features they provide are the following.

- Restructurers detect poorly structured code fragments and replace them by equivalent *structured* code.
- Cross referencers list the places where each variable is defined and used.

- Static analyzers detect anomalous constructs such as uninitialized variables and dead code.
- Text editors and other simple tools support the browsing of source code.

Together, these tools provide necessary but not sufficient functions for performing reverse engineering. In order to provide a full suite of tools, an appropriate methodology and a comprehensive representation are required for the derived reverse engineering information.

3. METHODOLOGY

3.1. 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[21]. 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[14].

Another bottom-up approach is that taken by the Programmer's Apprentice project. This work, although primarily concerned with initial program construction, does feature an analyzer that attempts to recognize plans and represent them in a *plan calculus*[16].

A completely different bottom-up approach is described by Basili and Mills. They describe an analysis procedure based on control flow analysis and formal documentation[1]. Hausler and his colleagues have described tools they are building to support this approach[13].

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[7]. An expectation is confirmed by locating a *beacon* in the code. This is a stereotypical programming construct, similar in concept to the *plans* mentioned above. There have been some human factors experiments that support Brooks' ideas.

3.2. Synchronized Refinement

Kamper and Rugaber 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[17]. 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 the fact that programs are built from component pieces that are interleaved to accomplish the total purpose.

Synchronized Refinement has been used to help analyze an operational software system comprising ten thousand lines of Cobol code. 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..

4. REPRESENTATION

4.1. 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.

4.1.1. 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 is constructing 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.

4.1.2. 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.

4.1.3. 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.

4.1.4. 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[11, 15].

4.1.5. Requirements Related to How the Representation is Accessed and Viewed

The predominant use of the representation will be to explore 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.

4.2. 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 decision 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[18]. 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* - 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.
- Modality - 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. For example, imagine the relationship between two arrays, both of which contain the same elements, one of which is ordered. If the ordered version is offered as input and unordered versions are output on subsequent calls, then the program is a permutation generator. In the situation where the unordered array is offered as input and the ordered version is output, then the program is a sorter. 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.

4.3. Data Modelling

The characterization of design decisions described above is suitable for guiding an experienced programmer through the Synchronized Refinement process. However, it is not precise enough to build tools for automating the process. Moreover, as noted above, the detected decisions and annotated code need to be stored in a data base suitably organized to support software maintenance.

---

* 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.

In order to define such a data base, a data model is required. A data model describes the various data items to be stored and the relationships among them. It is formal in the sense that the data items must all be representable using standard data types and structures and that all of the relationships must be representable in terms of those provided by the underlying data base management system.

5. 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. However, the information that the tools produce needs to be saved in a data base so that it can be later accessed by software maintainers.

5.1. 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.

5.2. 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[10]. There, high bandwidth displays and direct manipulation interfaces are used to explore non-linear organizations of text. In this case, the text is source code and the non-linear relationships are provided by the parser.

5.3. 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 entirely 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.

5.4. 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 tools. 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.

6. CONCLUSIONS

6.1. 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 the 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 (program nesting) 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.

6.2. 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 tools, such as debuggers and program transformers, useful in maintaining and improving software quality.

REFERENCES

References

1. V. R. Basili and H. D. Mills, ‘‘Understanding and Documenting Programs,’’ *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, pp. 270-283, May 1982.
2. L. A. Belady and M. M. Lehman, ‘‘Programming System Dynamics or the Meta-Dynamics of System in Maintenance and Growth,’’ RC 3546, International Business Machine Corp., September 17, 1971.
3. G. D. Bergland, ‘‘A Guided Tour of Program Design Methodologies,’’ *IEEE Computer*, October 1981.
4. Ted J. Biggerstaff, ‘‘Design Recovery for Maintenance and Reuse,’’ IEEE Computer, July 1989.
5. Mark R. Blackburn, ‘‘Toward a Theory of Software Reuse Based on Formal Methods,’’ SPC-TR-88-010, Version 1.0, Software Productivity Consortium, April 1988.
6. Barry W. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.
7. Ruven Brooks, ‘‘Towards a Theory of the Comprehension of Computer Programs,’’ *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.
8. Elliot J. Chikofsky and James H. Cross II, ‘‘Reverse Engineering and Design Recovery: A Taxonomy,’’ *IEEE Software*, vol. 7, no. 1, January 1990.
9. Derek Coleman and Robin M. Gallimore, ‘‘A Framework for Program Development,’’ *Hewlett-Packard Journal*, vol. 38, pp. 37-40, October 1987.
10. J. Conklin, ‘‘Hypertext: An Introduction and Survey,’’ *IEEE Computer*, pp. 17-41, September 1987.
11. Martin S. Feather, ‘‘A Survey and Classification of some Program Transformation Approaches and Techniques,’’ in *Program Specification and Transformation*, ed. L. G. L. T. Meertens, pp. 165-195, Elsevier North Holland, 1987.
12. R. K. Fjeldstad and W. T. Hamlen., ‘‘Application Program Maintenance Study: Report to Our Respondents,’’ *Proceedings GUIDE 48*, Philadelphia, PA, 1979. Tutorial on Software Maintenance, G. Parikh and N. Zvegintozov, editors, IEEE Computer Society, April 1983, IEEE Order No. EM453.
13. Philip A. Hausler, Mark G. Pleszkoch, Richard C. Linger, and Alan R. Hevner, ‘‘Using Function Abstraction to Understand Program Behavior,’’ *IEEE Software*, vol. 7, no. 1, pp. 55-63, January 1990.
14. Stanley Letovsky, ‘‘A Program Anti-Compiler,’’ Draft Technical Report, Department of Computer Science, Yale University, July 8, 1988.
15. H. Partsch and R. Steinbruggen, ‘‘Program Transformation Systems,’’ *ACM Computing Surveys*, vol. 15, no. 3, pp. 189-226, September 1983.
16. Charles Rich and Linda M. Willis, ‘‘Recognizing a Program’s Design: A Graph-Parsing Approach,’’ *IEEE Software*, vol. 7, no. 1, January 1990.
17. Spencer Rugaber and Kit Kamper, ‘‘Design Decision Analysis Research Project,’’ GIT-SERC-90/01, Software Engineering Research Center, Georgia Institute of Technology, January 28, 1990.
18. Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr., ‘‘Recognizing Design Decisions in Programs,’’ *IEEE Software*, vol. 7, no. 1, January 1990.
19. Ben Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*, Little Brown and Co., Boston, Massachusetts, 1980.
20. Eliot Soloway, Jeannine Pinto, Stan Letovsky, David Littman, and Robin Lampert, ‘‘Designing Documentation to Compensate for Delocalized Plans,’’ *Communications of the ACM*, vol. 31, no. 11, November 1988.

21. E. Soloway and K. Ehrlich, ‘‘Empirical Studies of Programming Knowledge,’’ *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, September, 1984.

22. Dallas E. Webster, ‘‘Mapping the Design Representation Terrain: A Survey,’’ MCC STP-093-87, Microelectronics & Computer Technology Corporation, July 1987.