Position Paper
Workshop on Program Comprehension

REVERSE ENGINEERING BY SIMULTANEOUS PROGRAM ANALYSIS AND DOMAIN SYNTHESIS

Spencer Rugaber

College of Computing
Georgia Institute of Technology
Atlanta, Georgia 30332-0280
(404) 894-8450

1. INTRODUCTION

There is several issues concerning appropriate strategies for analyzing a program. One such issue concerns whether the program analysis should proceed top-down or bottom-up. For example, Soloway and his colleagues propose a bottom-up approach involving the detection of patterns or *plans* in the program text[9]. Brooks, on the other hand, proposes a scheme whereby analysts are guided by their expections of a program's content and search for confirming *beacons* in a top-down fashion[2].

Another issue arises when considering the relative importance of a program's formal content, the executable statements and declarations in the program's text, and its informal aspects, such as comments and mnemonic variable names. Biggerstaff *et al.*[1] is among the few that concentrate on the informal information to construct a conceptual hierarchy describing the application domain with which a program in concerned.

Synchronized Refinement is a reverse engineering methodology that addresses these issues. It coordinates the bottom-up analysis of the program text with the top-down construction of a description of the specific application that the program accomplishes.

2. SYNCHRONIZED REFINEMENT

Chikofsky and Cross[3] define reverse engineering as "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". Typically, this includes both a description of *what* the system does, but also *how* it goes about doing it.

Synchronized Refinement consists of the parallel analysis of the source code and synthesis of a functional description. The process is driven by the detection of design decisions in the source code. Each decision is annotated in the functional description. The annotation states how the decision contributes to the function accomplished by the corresponding code. After each decision is detected and annotated, the corresponding part of the source code is replaced by an abbreviated description. In this way, the source code continually grows shorter while the functional description grows more complete.

Design decisions are structural decisions made by the original designer or programmer. Typical design decisions include the decomposition of a function into its subfunctions, the handling of special cases, and the use of one data structure to represent another that is not directly provided by the programming language. Other decisions include the encapsulation of a set of procedures into a module

and the introduction of a data item to save the result of a computation for later use. A more detailed description of design decisions is given in[8].

The synthesis process begins with a high-level description of the overall program as obtained from a review of the documentation, possibly augmented with comments from the source code. This description leads to certain expectations in the reverse engineer's mind. For example, if a sorted report is to be produced, it is expected that part of the code will be responsible for the sort (or preparing data for an external sort), and there is likely to be a section controlling the pagination of the report. Note that the expectations need not be complete nor even entirely accurate at this stage.

A dynamic list of expectations is constructed. As the analysis process proceeds, decisions are detected that relate to an expectation. For example, in the case of the pagination expectation, a section of code is found that keeps a counter that resets after reaching the length of a page. The annotation for the detected decision is placed together with the relevant expectation, specifying the type of the decision and the corresponding sections of the program. During the process, certain expectations will be confirmed and others may be refuted. A confirmed expectation engenders others. Gradually, a hierarchical description of the structure of the program emerges. As the program source code shrinks, the functional description expands. An alternative procedure, based on control flow analysis and program slicing, is described in a paper by Hausler *et al*[4].

3. EXPERIENCE

Synchronized Refinement has now been used successfully on several projects. Besides the exercize described in[8] that involved its use on a short numerical program written in FORTRAN, it has been used on two other substantial systems. Papers in this year's Software Maintenance Conference[6, 7] describe its use in understanding a large (> 100KLOC) real-time, embedded system written in a systems programming language. The reverse engineering resulted in both the detection of several bugs and in enabling a redesign that significantly improved the maintainability of the system. Synchronized Refinement has also been used to reverse engineer a moderate-sized (~ 10KLOC) COBOL information system. The reverse engineering enabled an object-oriented re-engineering into Ada[5].

4. ISSUES RAISED

Synchronized Refinement is a labor-intensive process. In principle, it is capable of guiding a reverse engineering activity to any degree of resolution. However, in doing this, the point may be reached where the cost of the reverse engineering approaches the cost of complete redevelopment. Thus the challenge becomes to guide the reverse engineering process to those aspects of the program that will yield the highest payback, for example, the capture of reusable components or the codification of business rules.

An issue also arises of how to represent the information derived from the reverse engineering process. The use of an integrated representation makes possible internal consistency checks, supports the production of documentation in a variety of formats, and may be useful in a later redesign effort.

References

1. Ted J. Biggerstaff, Josiah Hoskins, and Dallas Webster, "DESIRE: A System for Design Recovery," MCC STP-081-89, April 1989.
2. Ruven Brooks, "Towards a Theory of the Comprehension of Computer Programs," *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.
3. Elliot J. Chikofsky and James H. Cross II, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, no. 1, pp. 13-17, January 1990.
4. 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.

5. Reginald L. Hobbs, John R. Mitchell, Glenn E. Racine, and Richard Wassmath, "Re-engineering Old Production Systems: A Case Study of Systems Re-development and Evaluation of Success," *Emerging Information Technologies for Competitive Advantage and Economic Development: Proceedings of the 1992 Information Resources Management Association International Conference*, pp. 29-37, Harrisburg, Pennsylvania, May 1992.

6. Bret Johnson, Steve Ornburn, and Spencer Rugaber, "A Quick Tools Strategy for Program Analysis and Software Maintenance," *Proceedings of the Conference on Software Maintenance*, Orlando, Florida, November 1992.

7. Stephen B. Ornburn and Spencer Rugaber, "Reverse Engineering: Resolving Conflicts between Expected and Actual Software Designs," *Proceedings of the Conference on Software Maintenance*, Orlando, Florida, November 1992.

8. Spencer Rugaber, Stephen B. Ornburn, and Richard J. LeBlanc, Jr., "Recognizing Design Decisions in Programs," *IEEE Software*, vol. 7, no. 1, pp. 46-54, January 1990.

9. E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 595-609, September, 1984.