Reverse Engineering of Legacy Systems: A Path Toward Success *

Alex Quilici
University of Hawaii at Manoa
Department of Electrical Engineering
2540 Dole St, Holmes 483
Honolulu, HI, 96822

Abstract

This paper addresses the question of whether the reverse engineering of legacy systems is doomed to failure. Our position is that the answer is highly dependent on the specific goals of the reverse engineering process. We argue that while most reverse engineering efforts may well fail to achieve the traditional goal of automatically extracted complete specifications suitable for forward engineering, they are likely to succeed on the more modest goal of automatically extracting partial specifications that can augmented by system-assisted human understanders.

1 Introduction

Is reverse engineering of legacy systems doomed to failure? Answering this question requires a definition of reverse engineering and its goals. A paraphrase of the consensus definition of reverse engineering is [1]:

> The process of deriving abstract formal specifications from the source code of a legacy system, where these specifications can be used to forward engineer a new implementation of that system.

There are several important assumptions underlying this definition. One is that the process of deriving abstract formal specifications is completely automatic. Another is that these specifications are at a sufficiently abstract level so that the system can be implemented in a new language, executed on a new architecture, or recoded in a substantially more maintainable way—not merely restructured. The last is that the time and effort required to derive these specifications is less than forming the specifications without reference to the existing code. The result is that, under this definition, whether reverse engineering can ever be successful boils down to a simple question: Is it possible to automatically derive the necessary abstract specifications in a cost-effective way that operates within a reasonable amount of time?

The current state of the art in real-world reverse engineering is far from this goal. Perhaps the closest efforts are those using Reasoning System's REFINE [11, 12] to generate an augmented abstract syntax tree (essentially a parse tree with additional data and control flow information) and to apply transformation rules that operate on this AST to generate a new version of the original system. One such effort, for example, used this approach to automatically divide a large COBOL program into a number of individual modules that grouped together related functionality [8]. The AST for the original COBOL program can be thought of as a slightly more abstract specification of the program's behavior than either the original or the resulting code (since numerous implementations can result in the same AST), and the application of transformation rules to create new modules can be thought of as a portion of the forward engineering of a new system.

Unfortunately, this state of the art effort is far from the goal of reverse engineering, which gives rise to the question: How much of this gap can be closed? In this paper we argue that, regardless of whether the standard goal of reverse engineering can be achieved, a modified goal of reverse engineering is both attainable and worthwhile. In particular, the goal of reverse engineering should be to extract a knowledge base describing a legacy system that can be used to

* This work has been partially funded by US Air Force Contract #F30602-93-C-0257 as part of the Rome Labs KBSA project.

ICSE '95, Seattle, Washington USA
© 1995 ACM 0-89791-708-1/95/0004...$3.50

improve forward engineering—and not specifically to abstract a complete set of formal specifications. In addition, the process of extracting this information should be assumed to combine both automation and assistance—and not be assumed to be completely automated. Essentially, we are arguing for this revised definition of successful reverse engineering:

> The automated or assisted process of deriving a knowledge base describing a legacy system from its source code, where this knowledge base lessens the effort required to forward engineer a new implementation of that system.

Thus, under this definition, reverse engineering is successful if the cost of extracting information from a legacy system is less than the cost savings that arise from having that information available to support forward engineering. The result is that there are a variety of ways reverse engineering can be successful: by automatically forming a partial set of formal specifications, by automatically extracting a subset of design elements implemented in a particular program, by assisting in the formation of a complete set of specifications, and so on.

Rather than focusing on whether it is theoretically or practically possible to automatically extract complete specifications from code, the real issue of interest is in what information can be automatically extracted from source code and how this information can contribute to lessening the cost of forward engineering.

2 What Can Be Automatically Extracted

Whether it is possible to automatically extract formal specifications from legacy systems is an open question. However, numerous current research efforts suggest that it is tractable to extract the equivalent of partial specifications from code automatically, and that this information can be cost-effectively used in forward engineering.

One example is recent program understanding research into how to extract design information at various levels of abstraction [6, 10, 7, 14, 15, 5, 4] These approaches generally match code patterns (plans) against the code to recognize higher-level abstractions, such as implementations of domain-independent data structures and algorithms [14, 3]. These abstractions are clearly useful for forward engineering in that they are largely language-independent, can often be replaced with prepackaged verified or validated code, and so on. Similar efforts have examined how to recognize implementations of domain-dependent abstractions [10]. One example is the abstraction of validating an input transaction. Recognizing abstractions such as this one helps forward-engineering by determining the conditions the existing system validates, which can serve as the starting point when forming specifications for the new system or can be used to see how well the existing code corresponds to the new specifications.

All of these approaches to program understanding face two key questions. Can they be used to extract a complete hierarchical design from existing real-world legacy systems? And do they scale?

The Completeness Question

Unfortunately, the answer to the first question is "No", at least for the standard paradigm of matching entries in a library of plans against the source code. The problem is that code is understood only when it successfully matched against an entry in the plan library. This implies that the library must contain all possible plans in advance; however, there is always some code that is idiosyncratic in nature and unlikely to be present in the plan library [13]. That means the pattern-matching approach is always doomed to incomplete understanding of legacy systems.

Fortunately, this failing does not mean that reverse engineering is therefore doomed as well. In particular, given a domain-independent plan library, existing program understanding algorithms are still potentially useful for extracting domain-independent design information, such as which data structures are being used. In addition, given a domain-specific plan library (such as would be constructed to try to understand a family of related transaction processing applications), these algorithms are also potentially useful for extracting domain-dependent design information from stereotypical domain-dependent code. The extracted design information can form a portion of the initial design of the new version of the system to be constructed.

Whether program understanding algorithms can produce a useful, cost-effective partial understanding of an existing legacy system is likely to depend on how much of the code within the system is stereotypical rather than idiosyncratic—a question that can only be answered empirically. On the bright side, the cost of developing the portion of the the plan library that may be necessary to recognize domain-independent design elements in existing systems can be amortized over an extremely large set of programs. Similarly, the cost of developing a domain-dependent

plan library can be amortized over the collection of applications in that domain. This suggests that while complete abstract specifications are out of reach of library-based program understanding algorithms, extracting partial specifications of significant portions of existing systems may well be within reach.

The Tractability Question

The other question deals with tractability. At first glance, the news appears to be bad. Most existing program understanding algorithms, such as those that rely on flowgraph-matching, are NP-complete in the worst case. The good news, however, is that empirically they appear to have much better average case performance, at least on the relatively small programs and libraries to which they have been applied (around 1000 lines of code or so) [14]. In addition, there have been recent attempts to explicitly augment the plan library with search control information to reduce the amount of matching that must take place to recognize programming plans, even in the presence of a large plan library [10]. While this work is relatively new, the initial performance results with this approach have also been promising, although again, only on textbook-sized COBOL programs.

The problem is that a successful program understanding algorithm must scale both with the size of the program and the size of the library. There is obviously a large difference in scale between a 1000-line program and a 100,000-line program, and between a 100-plan library and a 10000-plan library. But surprisingly, the success current program understanding algorithms have recently shown with 1000-line programs and 100-plan libraries may well be close to what's necessary to understand much larger programs.

The scaling problems inherent in trying to extract design information from large programs may be addressable by applying automatic modularization techniques. In particular, if the programs to be understood can be automatically modularized, the understanding algorithms can simply be run on the modules and then applied to understanding the connections between these modules. This greatly shrinks the search space the understanding algorithm is required to examine to determine whether items in a particular plan library are present.

Furthermore, some recent research into efficiently determining whether high-level program elements (such as a process table for a debugger) are likely to be present, suggests a way out of the scaling problems that arise with large plan libraries [2]. These approaches use indexing techniques, such as connectionist networks, to suggest what high-level design elements may be present in a given program—without actually attempting to verify their presence. This allows other program understanding algorithms to try to verify their presence using only the portion of the library that is likely to be relevant. Doing so has the potential to greatly shrink the search space that the program understander must traverse.

3 Cooperative Extraction

Given that pattern-based program understanding algorithms are likely to be able to only partially understand legacy systems, we are at best likely to obtain a partial specification out of the automated program understanding process (such as recognition of a subset of the design elements present in the system). But what can be done when the extracted information is insufficient for forward engineering?

One approach to this problem that we have been exploring is *cooperative extraction* of the remaining specifications [9]. The idea is to provide intelligent assistance to programmers who are attempting to understand the parts of the legacy system that could not be automatically understood. The vision is that understanding becomes a task shared by automated extraction tools and a set of programmers who are examining the legacy system's code.

In our version of this approach, we provide a single knowledge base that captures all design information extracted about the program—whether by an automated program understander or by users examining the code. This knowledge base is formally represented so that it can be used to support forward engineering, but is visually presented to users as a graph, whose nodes represent design elements and are linked to the various pieces of code that implement those design elements. Users can visually examine this graph to determine where various design elements are implemented and to see the underlying relationships between various code segments. Users can also visually edit this knowledge base to add new design elements that they recognize in the code, and to link these elements to the code fragments that implement them.

The key is to have the knowledge base containing what has been extracted about a given legacy system accessible to those who are trying to extract additional information about that system. In our approach, we provide users with the ability to ask a variety of queries about the extracted design and its relationship to the program. These queries include questions about the conceptual function of any arbitrary code segment, about which portions of the code

are related in various ways to a particular design element, and about which parts of the system are not currently understood.

There may well be other approaches and techniques that better aid programmers in extracting design information and specifications from a legacy system. And while we have used an initial prototype cooperative design extraction environment to extract the conceptual objects and operations underlying a set of textbook COBOL programs, it is an open question how well this approach will scale up in practice to real-world legacy systems and to extracting more complete or complex information about the design of the system. However, the notion of combining automated and assisted understanding does suggest one way to potentially overcome the likely incompleteness of automatically extracted specifications.

4 Conclusions

The old notion of reverse engineering as a process of automatically deriving complete specifications of the behavior of a legacy system may well be doomed to failure. But we are not nearly as pessismistic about reverse engineering if we relax the demands on automation to a partial understanding of the legacy system and then provide tools that assist programmers in completing this understanding. As a result, our position is that reverse engineering of many legacy systems is eventually doomed to success, not failure.

References

[1] R. Arnold, *Software Reengineering*, IEEE Press, 1992.

[2] T.J. Biggerstaff, B.G. Mitbander, and D.E. Webster, 1994. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5) (May 1994), 72–82.

[3] R. Dekker and F. Ververs, Abstract Data Structure Recognition, In *Proceedings of the Ninth Knowledge-Based Software Engineering Conference.* (September 1994), Monterey CA, 133–140.

[4] W.L. Johnson, *Intention Based Diagnosis of Novice Programming Errors.* Morgan Kaufman, Los Altos CA, 1986.

[5] S. Letovsky, *Plan Analysis of Programs.* Ph.D. Thesis, Yale University, New Haven CO, 1988.

[6] W. Kozaczynski and J.Q. Ning, Automated Program Understanding By Concept Recognition, *Automated Software Engineering*, 1(1) (March 1994), 61–78.

[7] W. Kozaczynski, J.Q. Ning, and A. Engberts, Program concept recognition and transformation. *Transactions on Software Engineering*, 18(12), (December 1992), 1065–1075.

[8] P. Newcomb and L. Markosian, Automating the Modularization of Large COBOL Programs: Application of an Enabling Technology for Reengineering. In *Proceedings of the Working Conference on Reverse Engineering*, Baltimore, MD, (May 1993), 222-230.

[9] A. Quilici and D.N. Chin, A cooperative program understanding environment. In *Proceedings of the Ninth Knowledge-Based Software Engineering Conference.* (September 1994), Monterey CA, 125–132.

[10] A. Quilici, A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5) (May 1994), 84–93.

[11] Reasoning Systems, *REFINE/COBOL User's Guide*, 1992.

[12] Reasoning Systems, *REFINE/COBOL Programmer's Guide*, 1992.

[13] E. Soloway and K. Erdlich, Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, 10(5) (1984), 595–609.

[14] L.M. Wills, *Automated Program Recognition by Graph Parsing.* Ph.D. Thesis, Technical Report 1358, MIT Artificial Intelligence Lab, Cambridge MA, (September 1992).

[15] L.M. Wills, Automated program recognition: a feasibility demonstration. *Artificial Intelligence* 45(1-2), (September 1990), 113–172.