Reverse Engineering of Legacy Code Exposed[1]

Bruce W. Weide and Wayne D. Heym
Department of Computer and Information Science
The Ohio State University, Columbus, OH 43210
{weide,heym}@cis.ohio-state.edu

Joseph E. Hollingsworth
Department of Computer Science
Indiana University Southeast, New Albany, IN 47150
jholly@ius.indiana.edu

Abstract — Reverse engineering of large legacy software systems generally cannot meet its objectives because it cannot be cost-effective. There are two main reasons for this. First, it is very costly to "understand" legacy code sufficiently well to permit changes to be made safely, because reverse engineering of legacy code is intractable in the usual computational complexity sense. Second, *even if* legacy code could be cost-effectively reverse engineered, the ultimate objective — re-engineering code to create a system that will not need to be reverse engineered again in the future — is presently unattainable. Not just crusty old systems, but even ones engineered today, from scratch, cannot escape the clutches of intractability until software engineers learn to design systems that support modular reasoning about their behavior. We hope these observations serve as a wake-up call to those who dream of developing high-quality software systems by transforming them from defective raw materials.

1. Introduction

Most large software systems, even if apparently well-engineered on a component-by-component basis, have proved to be incoherent as a whole due to unanticipated long-range "weird interactions" among supposedly independent parts. The best anecdotal evidence for this conclusion comes from reported experience dealing with *legacy* code, i.e., programs[2] in which too much has been invested just to throw away but which have proved to be obscure, mysterious, and brittle in the face of maintenance.

What should we do when we require a new system whose behavior is intended to be similar to that of an old system we already have? One option is to build the new one from scratch, relying perhaps on experience obtained through design or use of the old one, but not relying substantially on the old code. Another option is to try to understand the old code well enough to keep much of it, modifying it to meet the new needs. The latter approach — *re-engineering* — necessarily involves *reverse engineering*:

> Reverse engineering encompasses a wide array of tasks related to understanding and modifying software systems. Central to these tasks is identifying the components of an existing software system and the relationships among them. Also central is creating high-level descriptions of various aspects of existing systems. [15, p. 23]

We consider reverse engineering in its role as an integral part of the re-engineering approach to new system development. The objective of reverse engineering is not (just) to create documents that chronicle a path from the original requirements to the present legacy system as, say, a substitute for the documentation that probably was not created while that journey was in progress. The goal is to achieve a sufficient understanding of the whats, hows, and whys of the legacy system as a whole that its code can be re-engineered to meet new requirements on behavior, performance, structure, system dependencies, etc.

1.1. Reverse Engineering of Legacy Code is Intractable

There seems to be general agreement that, in practice, reverse engineering of legacy code is at least quite laborious [14]. Even if many aspects of large systems are easy to understand, inevitably there is important behavior whose explanation is latent in the code yet which resolutely resists discovery. The basic reason is that software engineers seek modularity — and they generally achieve it well enough create a very compact representation of system behavior in the source code, but not well enough to support modular reasoning about that behavior. In Sections 3-4 we summarize how this implies that reverse engineering of legacy code is intractable in the usual computational complexity sense [19]. This fundamental conclusion and the supporting argument follow up on a suggestion by Hopkins and Sitaraman [9] that the effort required to reverse engineer a system is related to the effort required to formally verify its functional correctness. In fact, if we argued that program verification of legacy code is intractable, there probably would be little debate (at least with those from whom the current position is likely to draw fire). Yet these are technically equivalent.

1 This position paper is adapted from [0,19].

2 We do not consider legacy systems that consist primarily of data (e.g., databases).

1.2. Forward Engineering is Not a Solved Problem

Of course intractable does not mean impossible. One even hears occasional stories about "successful" reverse and re-engineering projects [1]. These should be taken with a grain of salt, if only because the real problem — successful re-engineering — cannot be known to have been solved for years, after there is a long history of maintenance tasks to sort through or the system has to be re-engineered again. Even then, without a controlled study, there is no way to know that building from scratch would not have been more cost-effective. And even if the reverse engineering battle can be won once, the re-engineering war ultimately will be lost without subsequent use of forward engineering techniques that effectively prevent software "rot".

Unfortunately, almost without exception software engineers do not know how to design and build truly modular systems when starting from scratch, let alone when starting from legacy code [17, 18]. Except for egregiously poor design practices, they cannot distinguish fair-to-good software designs from excellent ones. The reason is that beyond "structured programming" aphorisms there are almost no accepted community standards for what software systems should be like at the detail level. By the intractability argument, some key quality criteria would seem to be understandability in general, and susceptibility to modular reasoning about behavior in particular. Yet this degree of modularity is almost universally *not* achieved by designs in computer science textbooks, technical papers, and commercial software.

2. Observations and Implications

Reverse engineering of legacy code has proved to be such a difficult practical problem — experience which lends credence to the thesis that it is intractable — that serious attention ought to be devoted to the subject. This is particularly true because the alternative is also costly. But we need to have realistic expectations about the ultimate role of reverse engineering in a comprehensive vision of software engineering. We are disturbed with the emphasis on building tools to solve problems whose inherent complexity suggests that those tools cannot be expected to scale up to realistically large systems. And we are frankly alarmed by the following sentiments, which seem typical among reverse engineering advocates:

> ... while many of us may dream that the central business of software engineering is creating clearly understood new systems, the central business is really upgrading poorly understood old systems. [15, p. 23]

> [The problem of having to deal with] legacy software is basically the result of management inaction rather than technology deficiency... [1, p. 23]

Quite the contrary! Most software hasn't been written yet, and widely taught and practiced "modern" approaches to the nuts-and-bolts of software engineering still do *not* lead to well-designed modular systems. So, if we as a community act as though we believe that "the central business [of software engineering] is really upgrading poorly understood old systems," then we will squander a fortune yet continue to face the Sisyphean task of upgrading poorly understood old systems into slightly less poorly understood new systems. We might have spent our efforts developing and exploring truly productive techniques for forward engineering of well-understood modular systems — and this progress would help even those who insist that re-engineering is ultimately where the action will be.

We mentioned above that reverse engineering is as hard as program verification, and this leads to a common misunderstanding about the claim of intractability. General program verification is in principle unsolvable, because the verification conditions generated from code and specifications might include arbitrary mathematical assertions. The practical consequences of this observation, however, are minimal. It can be used to show that *there exist* esoteric systems for which program verification (hence reverse engineering) is *impossible*; but it does not mean that program verification/reverse engineering cannot succeed on code that arises in practical situations. Our claim is about such practical situations. Specifically, *for all* large legacy systems, program verification/reverse engineering is *prohibitively expensive*; not impossible in principle but manifestly not cost-effective — and this bears directly on the business decision regarding whether to re-engineer or to build anew. The obvious rejoinder to this claim from reverse engineering advocates is, "People do reverse engineering all the time; how can it be prohibitively expensive?" We address this question in Section 3.1.

The news is not all bad; we offer some assistance to reverse engineering advocates. Specifically, by identifying threats to modular reasoning from common design and coding practices as a key technical factor that thwarts cost-effective reverse engineering, we implicitly suggest an area where new reverse engineering tools might be helpful — namely, finding such trouble spots. The ability to do this will not change the underlying intractability but might incrementally help those stuck with reverse engineering.

3. The Nature of the Reverse Engineering Task

At first glance, the conclusion that reverse engineering of legacy code is doomed to fail strikes most people as either ridiculous and wrong (the "reverse engineering advocates" camp), or obvious and trivial (the "reverse engineering skeptics" camp). Some hedge, claiming it could be either depending on the definition of reverse engineering.

By the putative definition quoted in Section 1, reverse engineering involves achieving an "understanding" [3, 12, 14] of a system, including "identifying the components of an existing software system and the relationships among them" and "creating high-level descriptions". What does this mean? We argue that successful reverse engineering of a legacy system entails at least the following two subtasks:

(1) Identifying the functional components of the system and the roles they play in producing the behavior of the higher-level system that employs them.

(2) Creating a valid explanation of *how* and *why* the behavior of the higher-level system arises from these functional components and their roles.

We use "functional" here in the sense of contributing to functional run-time behavior. This means that the relevant components of a system, from the standpoint of understanding system behavior, are not necessarily the structural components of its source code (e.g., modules, subroutines, loop bodies, statements). Some functional components might correspond to easily-identified structural components, but others might span several of them — especially where interesting behavior arises from poor design or from unanticipated interactions between structural components.

By "valid explanation" we mean, effectively, a proof that the claimed higher-level behavior results from the identified functional components and roles. The challenges in achieving understanding of a poorly understood system are to generate a hypothesis, which is in fact correct, and to establish why it is correct.

3.1. Testing vs. Proving

We now consider the claim, "People do reverse engineering all the time; how can it be prohibitively expensive?" Certainly one can define reverse engineering so this is true. But what people really do all the time is to make plausible hypotheses. They do not check the validity of those hypotheses in any decisive way. They might, for instance, make some changes to the code that should not cause problems according to the hypothesis, then test to see whether those changes cause obvious problems.

Such an approach can only hope to show that a hypothesis is invalid, not that it is valid — a conclusion similar to the well-known aphorism that program testing can only hope to demonstrate the presence of bugs, not their absence. We do not trick ourselves into believing we have built correct software by *defining the problem* of building correct software in such a way that testing alone is sufficient to decide whether we have succeeded. Yet defining reverse engineering to consist of hypothesize-and-*test*, not hypothesize-and-*prove*, amounts to the same thing.

Advocates of the weaker definition might contend that all they are hoping for is to obtain "approximate" understanding of a system. But an approximation is *not* sufficient to achieve the ultimate objective of reverse engineering. Furthermore, we can find no reasonable technical definition of "approximate" reverse engineering. In any event there must be an absolute standard by which to judge the quality of an approximation. We therefore define successful reverse engineering to entail decisive checking of the validity of hypotheses, not merely guessing.

3.2. Substantive Hypotheses

The reverse engineering hypothesis should contribute enough to the understanding of a system to suggest and/or rule out potential modifications that are intended to achieve the objective of the project. Biggerstaff, *et al.*, seem to summarize nicely:

> A person understands a program when able to explain the program, its structure, its behavior, its effects on its operational context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code for the program. [2, p. 72]

We therefore stipulate that reverse engineering hypothesis H for system S should be *substantive* in that it is:

- *Effective* — it provides the ability to predict relevant behavior of S (e.g., relevant input-output behavior) and to answer questions about what-if situations (e.g., the effects of various changes to the source code of S).
- *Comprehensive* — its validity cannot be decided by a small set of test cases.
- *Concise* — it is at worst not much bigger than the source code of S.
- *Independent* — it is not a paraphrase of the code of S.
- *Systemic* — judging its validity requires examining essentially all the code of S.

The first property is basic to utility. All the others are technically necessary to rule out trivial hypotheses that might otherwise be seen as counterexamples to intractability, but which in practice contribute nothing to the understanding of S. These conditions are not really very strong. For example, nearly every non-trivial hypothesis is systemic because there are many ways to get S to exhibit unhypothesized behavior via long-range weird interactions among its components. Whether a particular system actually has such interactions does not even matter; they *might* exist because they are not ruled out by static (e.g., programming language) constraints. An instruction that influences whether and why H holds might be lurking anywhere in the code of S, and there is simply no way to know whether it is there without looking for it.

4. The Intractability Result

The particular computational problem that we claim to be intractable is the second reverse engineering subtask:

> **EXPLAIN** — Given as input (S, H) — source code for a system S and hypothesis H about that system's behavior — decide whether, and explain why, H does or does not hold for S.

We do not need to account for the extra time it takes to generate a hypothesis to be explained. There is every reason to suspect that generating substantive hypotheses is hard, too, but we do not need to or try to demonstrate this.

We claim there is a lower bound for EXPLAIN for valid hypotheses which implies that reverse engineering of legacy code (as defined in Section 3) is intractable:

EXPLAIN is Intractable — There is a constant c>1 such that, for every legacy system S and every valid substantive hypothesis H, EXPLAIN(S,H) takes time at least $c^{|S|}$, where |S| is the size of S's source code.

This result follows from two premises, which we outline here because they are empirical statements which, in principle, are falsifiable and therefore debatable. The main argument is completed elsewhere [19].

4.1. Source Code is a Compact Representation of Behavior

It has long been accepted by software and other engineers that the key to dealing with large systems is to design and construct them by composing some smaller units that are independent except at their interfaces — the objective of *modularity*. One intended result of modularity is the ability to reason modularly about program behavior. Liskov and Guttag clearly state this objective in their description of how we should like to reason about total correctness, but the conclusion applies equally to reasoning about any substantive hypothesis about system behavior:

> We reason separately about the correctness of a procedure's implementation and about parts of the program that call the procedure. To prove the correctness of a procedure definition, we show that the procedure's body satisfies its specification. When reasoning about invocations of a procedure, we use only the specification. [11, p. 227-228]

This observation is based on something routinely taught to first-year programmers: It is hopeless to reason about execution of non-trivial programs by tracing instruction execution sequences, either for particular values or by symbolic execution, because even a small program can describe arbitrarily long execution sequences through recursive calls and looping. (Effective reasoning about program behavior also requires loop bodies to be replaced by specifications, e.g., loop invariants or loop functions.) In short, it must be possible to reason about the effect of any repeatedly-executed piece of code by using a specification of that piece, *without tracing the code* for each dynamically-occurring use of it. We take as a premise that software engineers strive to achieve, and succeed in achieving, part of what they have been taught — to encode long execution sequences in a concise way by identifying commonalities in source code and by factoring them out into separate pieces that are used repeatedly.

Consider any instruction execution sequence E of system S, and define |E| as the length of a record of the steps (say, instructions) taken in E. We claim:

Compact Source Code Premise — There is a constant c > 1 such that, for every legacy system S and for every substantive hypothesis H, there is some instruction execution sequence E which H purports to explain and for which $|E| > c^{|S|}$.

This premise is really quite a weak statement about legacy systems because most real code describes potential execution sequences that are not bounded *a priori* by any function of |S|, but only by the inputs to S. Consider that if E were achieved by straight-line code, for example, then we would need to have $|S| \geq |E|$. How could this hold for any realistic system? Rephrased in these terms, the premise says the source code for a real legacy system is substantially smaller than the length of the longest behavior history it can effect, i.e., its size is at most $\log_c |E|$. Clearly this always holds where there is no *a priori* bound on the longest execution sequence of S.

4.2. Problems Result From Failed Attempts at Modularity

We should hope that software engineers always succeed in separating specification from implementation in a way that achieves modularity. However, designing and implementing code that supports modular reasoning about behavior is more subtle than it appears at first [13, 20]. Problems arise from coupling through side-effects and aliased variables [4, 7], arrays, pointers, and dynamic storage management [6, 8], generics [5], inheritance [10, 16], and from many other sources. Potentially troublesome techniques are permitted by the programming languages used for real legacy systems because, in the interest of performance and other essential considerations, these techniques can be useful when applied carefully.

However, history gives no evidence that software engineers in practice do — or that they even know how to — exercise adequate care in the use of such powerful language constructs. We therefore claim:

Non-Modularity Premise — Every legacy system is hard to maintain because, in some crucial places, it has been designed or coded so that modularity is not achieved.

We need make no assumption about how the legacy system got into this state. Perhaps the system was poorly understood from day one, or perhaps became poorly understood through the cumulative toll of patches, upgrades, and adaptations. Whatever the cause, when an "existing" system graduates to the status of "legacy" system it has already been observed to be difficult to maintain. Non-modularity of reasoning about its behavior is a major reason for this.

5. Conclusion

Reverse engineering of large legacy systems is intractable in the following sense: Given real legacy code, the time required to show the validity of a proposed explanation for why it exhibits any significant system-level behavior is at least exponential in the size of the source code. This does not mean that the task is impossible. It means that it is prohibitively costly for large legacy systems.

One lesson from this should be that we need to put more emphasis, not less, on careful engineering of new systems [13]; and that this emphasis needs to focus (at least) on creating systems that admit modular reasoning. There are many good reasons to continue to work on reverse engineering of legacy code — it is an exciting intellectual challenge and a problem that sometimes has to be faced in practice. But at the same time we need to be realistic about what outcomes to expect. Researchers and developers, and especially their sponsors and the customers buying their wares, should not be disappointed that nothing seems to work very well for large legacy systems.

Acknowledgment

Dean Allemang, B. Chandrasekaran, Steve Edwards, John Hartman, Doug Kerr, Tim Long, Bill Ogden, Murali Sitaraman, Neelam Soundararajan, Michael Stovsky, Sergey Zhupanov, and Stu Zweben have provided helpful insights and/or feedback on the ideas presented here. We also gratefully acknowledge the support of the National Science Foundation through grant CCR-9311702; and the Advanced Research Projects Agency under ARPA contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714.

Bibliography

[0] Andersen, H.C. *The Emperor's New Clothes: A Fairy Tale*, Addison-Wesley, Reading, MA, 1973.

[1] Bennett, K. Legacy systems: coping with success. *IEEE Software 12*, 1 (Jan. 1995), 19-23.

[2] Biggerstaff, T.J., Mitbander, B.G., and Webster, D.E. Program understanding and the concept assignment problem. *Comm. ACM 37*, 5 (May 1994), 72-83.

[3] Chandrasekaran, B., Goel, A.K., and Iwasaki, Y. Functional representation as design rationale. *Computer 26*, 1 (Jan. 1993), 48-56.

[4] Cook, S.A. Soundness and completeness of an axiom system for program verification. *SIAM J. Comp. 7*, 1 (Feb. 1978), 70-90.

[5] Ernst, G.W., Hookway, R.J., Menegay, J.A., and Ogden, W.F. Modular verification of Ada generics. *Comp. Lang. 16*, 3/4 (1991), 259-280.

[6] Ernst, G.W., Hookway, R.J., and Ogden, W.F. Modular verification of data abstractions with shared realizations. *IEEE Trans. on Software Eng. 20*, 4 (Apr. 1991), 288-307.

[7] Harms, D.E., and Weide, B.W. Copying and swapping: influences on the design of reusable software components. *IEEE Trans. on Software Eng. 17*, 5 (May 1991), 424-435.

[8] Hollingsworth, J.E. *Software Component Design-for-Reuse: A Language-Independent Discipline Applied to Ada.* Ph.D. dissertation, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH, Aug. 1992; available from "ftp.cis.ohio-state.edu" in "/pub/tech-report/1993/TR01-DIR/*".

[9] Hopkins, J.E., and Sitaraman, M. Software quality is inversely proportional to potential local verification effort. *Proc. 6th Ann. Workshop on Software Reuse*, Owego, NY, Nov. 1993.

[10] Leavens, G.T., and Weihl, W.E. Reasoning about object-oriented programs that use subtypes. *Proc. OOPSLA '90/SIGPLAN Notices 25*, 10 (Oct. 1990), 212-223.

[11] Liskov, B., and Guttag, J. *Abstraction and Specification in Program Development.* McGraw-Hill, New York, 1986.

[12] Littman, D.C., Pinto, J., Letovsky, S., and Soloway, E. Mental models and software maintenance. In *Empirical Studies of Programmers*, E. Soloway and S. Iyengar, eds., Ablex, 1986, 80-98.

[13] Neumann, P.G. Are dependable systems feasible? *Comm. ACM 36*, 2 (Feb. 1993), 146.

[14] Parnas, D.L., Madey, J., and Iglewski, M. Precise documentation of well-structured programs. *IEEE Trans. on Software Eng. 20*, 12 (Dec. 1994), 948-976.

[15] Waters, R. C., and Chikovsky, E. Reverse engineering progress along many dimensions. *Comm. ACM 37*, 5 (May 1994), 23-24.

[16] Weber, F. Getting class correctness and system correctness equivalent: how to get covariance right. In *Proc. TOOLS USA '92*, R. Ege, M. Singh, and B. Meyer, eds., Prentice-Hall, 1992.

[17] Weide, B.W., Heym, W.D., and Ogden, W.F. Procedure calls and local certifiability of component correctness. *Proc. 6th Ann. Workshop on Software Reuse*, Owego, NY, Nov. 1993.

[18] Weide, B.W., and Hollingsworth, J.E., *On Local Certfiability of Software Components*, tech. report OSU-CISRC-1/94-TR04, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH, Jan. 1994; available from "ftp.cis.ohio-state.edu" in "/pub/tech-report/1994/TR01.ps.gz".

[19] Weide, B.W., Heym, W.D., and Hollingsworth, J.E., *Reverse Engineering of Legacy Code is Intractable*, tech. report OSU-CISRC-10/94-TR55, Dept. of Comp. and Inf. Sci., Ohio State Univ., Columbus, OH, Oct. 1994; available from "ftp.cis.ohio-state.edu" in "/pub/tech-report/1994/TR55.ps.gz".

[20] Wilde, N., Matthews, P., and Huitt, R. Maintaining object-oriented software. *IEEE Software 10*, 1 (Jan. 1993), 75-80.