Reverse Engineering of Legacy Code is Intractable

Bruce W. Weide
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

Technical Report OSU-CISRC-10/94-TR55 (October 1994)

Abstract — Reverse engineering of large legacy software systems is widely recognized to be a difficult problem. How bad is it? By an argument that identifies key underlying sources of the difficulty, reverse engineering of legacy code is shown to be intractable in the usual computational complexity sense. This conclusion implies that we should not be too enthusiastic about the ultimate value of reverse engineering as the centerpiece of a cost-effective approach to constructing new generations of systems.

THIS PAGE INTENTIONALLY BLANK

1. Introduction

Many 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 experience dealing with "legacy" code, i.e., programs[1] 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 with legacy code when it needs to be retrofitted into a new execution environment, interconnected as a subsystem of a larger system, changed to exhibit new or enhanced functionality, or tuned for better performance? Clearly, we must try to understand it before we can attempt any modification. According to Waters and Chikovsky in their introduction to a recent issue of *Communications of the ACM* :

> 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. [Waters 94, p. 23]

1.1. Research Question

There seems to be general agreement that, in practice, successful reverse engineering of legacy code is quite laborious. 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. To see how hard reverse engineering actually is — and to understand why — we examine it here from the standpoint of inherent computational complexity.

Our conclusion is that every large legacy system is hard to reverse engineer. 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. This fundamental conclusion and the supporting argument follow up on a suggestion by Hopkins and Sitaraman [Hopkins 93] that the effort required to reverse engineer a system is related to the effort required to formally verify its functional correctness.

---

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

Our approach to the problem is to treat formally one essential subtask in the reverse engineering of legacy code, as a computational problem called EXPLAIN: Given the source code for a system, and a hypothesis stating some behavior the system is purported to have and the underlying origin of that behavior in the code, decide whether the hypothesis is valid. The argument that EXPLAIN is intractable relies on a few basic premises which we introduce and justify prior to the main argument. Although the presentation is more technical than the skeletal conclusion as stated above, we neither require nor use deep formal analysis; an argument with the simplicity of a back-of-the-envelope calculation is sufficient.

1.2. Motivation

Reverse engineering of legacy code has proved to be such a difficult practical problem — experience which lends credence to our thesis that it is intractable — that serious attention ought to be devoted to the subject. This is particularly true because the alternatives to reverse engineering are also costly. But we need to have realistic expectations about the ultimate role of reverse engineering in a comprehensive vision of software engineering. Specifically, we are disturbed that the emphasis has been on building tools to solve problems whose inherent complexity suggests that those tools cannot be expected to scale up to realistically large systems [CSTB 90]. And we are frankly alarmed by the sentiment expressed by Waters and Chikovsky (and apparently shared by many others) that:

> … 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. [Waters 94, p. 23]

On the contrary, we hold that most software hasn't been written yet,[2] and that widely practiced "modern" approaches to the nuts-and-bolts of software engineering do *not* lead to well-designed new systems [Hollingsworth 92]. 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 engineering well-understood systems.

---

2 For example, Emmett Paige, Jr., in his keynote address to *Reuse '94* on August 9, 1994, noted that at least 70% of systems currently coming out of the U.S. Department of Defense are new, as opposed to being re-engineered legacy systems.

1.3. Contributions

At first glance, the conclusion that reverse engineering of legacy code is intractable 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. In light of this, defining terms carefully enough so the question comes to the table for debate is one contribution of the paper. But the technical argument we make also has some interesting implications.

First, reverse engineering is as hard as program verification (see also Section 2.1). 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 computationally so costly as to be effectively impossible.[3] 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 consider this question in Section 2.2.

Second, even if the intractability of reverse engineering seems "obvious" it is worth trying to formalize the argument for it. For example, saying that large traveling salesperson problems were very difficult to solve must have been equally obvious to those who tried to do it before 1972. But this in no way detracts from the value of the explanation for this phenomenon which was provided by the theory of NP-completeness.[4]

Third, there is a close relationship between reverse engineering and re-engineering. Given the clear understanding of legacy code obtained by successful reverse engineering, we can try to restructure or redesign the system, i.e., we can try to "re-engineer" it. The focus of this paper is on reverse engineering: the subtask of understanding enough about a legacy system to make whatever changes are needed. If the ultimate purpose of reverse engineering is to support re-engineering, then significant additional work is required. Re-engineering is based largely on the

---

[3] We couch the argument in terms of reverse engineering because it is more relevant to our motivation for considering the problem, as discussed below, and probably more significant to the software engineering community at large.

[4] Of course, we do not claim that our formalization is as deep or general as this one.

notion that by achieving an adequate understanding of a legacy system, we can determine how it *should have been designed*. Implicit in this approach is that we know how to engineer new systems "the right way". This assumption is dubious [Weide 93], but it is not the subject of interest here.

Finally, we offer some help for reverse engineering advocates in Section 4.2. By identifying threats to modularity 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 who must do reverse engineering.

No less important than these technical considerations is drawing the implication for software engineering practice and policy. Of course we do not expect this paper to be the final word on the subject. But we hope it can serve as a wake-up call to those who dream of engineering software systems by transforming defective raw materials into quality products. The message is not that it is impossible, but that we need to be realistic about expectations and claims for such an approach.

2. The Nature of the Reverse Engineering Task

By the putative definition given in Section 1, reverse engineering involves achieving an "understanding" 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?

2.1. Generating and Checking Hypotheses

Based on its avowed purposes, we argue that successful reverse engineering of a legacy system entails at least the following:

- Identifying the functional components of the system and the roles they play in producing the behavior of the higher-level systems (or subsystems) that employ them.
- Creating valid explanations 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 such hypotheses, which are in fact correct, and to establish why they are correct.

Rephrasing slightly to reflect the connection with related work on program understanding [Littman 86, Chandrasekaran 93], a reverse engineering project can be considered to include many instances of two subtasks:

- Making a hypothesis about the existence of particular functional components and their roles, especially causal relationships, in producing particular higher-level run-time behavior of the system.

- Deciding whether the hypothesis is valid and justifying that decision.

Specifically, a hypothesis has four parts:

(H1) *Statement of system behavior* — higher-level (than code) description of some system behavior;

(H2) *Identification of functional components* — description of functional components purported to lead to that system behavior;

(H3) *Statement of component behaviors* — descriptions of the behaviors of these functional components; and

(H4) *Causal story* — explanation of how and why the system behavior arises from the individual behaviors of the components as combined in the system’s source code.

For example, we might formulate a hypothesis about how a certain procedure’s use of other procedures results in meeting a putative specification of its externally visible behavior; or about why only certain variables’ values affect a module’s externally visible behavior, as claimed in its documentation; or about why some particular subset of all of an avionics system’s code constitutes the attitude control subsystem that is described in a reference architecture for re-engineering the system.

2.2. Hypothesizing vs. Reverse Engineering

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, in the above terms, what people really do all the time is to make plausible hypotheses. They do not seem to check the validity of those hypotheses in any decisive way. For example, one of us (JEH) was once involved in a reverse engineering project that seems typical in this respect. Painstaking perusal of source code resulted in a hypothesis about what it did and how it worked. This was followed by testing of one particular consequence of accepting the hypothesis: making some changes to the code that should not have caused problems according to the hypothesis, then testing to see whether those changes actually did cause 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* "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 subtasks of the form 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 mere approximation is insufficient to achieve the ultimate objective of reverse engineering (see Section 2.3). And in any event there should 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.

2.3. Useful Hypotheses

A useful 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 reverse engineering 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. [Biggerstaff 94, p. 72]

We insist that at least one hypothesis involved in successful reverse engineering should be *useful* in this sense. The hypothesis must be:

- *Effective* — its formal symbolic representation must provide the ability to predict relevant system behavior and to answer questions about what-if situations that the hypothesis claims to address.

- *Concise* — it must be at worst not much bigger than the source code for the system. (It probably is smaller, but one can imagine a somewhat longer explanation that is easier to understand than the source code because it is so regular or uniform.)

- *Independent* — at least the description of system behavior (part H1) must not be just a paraphrase of source code.

We doubt that the first property is quite what Biggerstaff, *et al*., have in mind, however, as they subsequently call for hypotheses to be "intentionally ambiguous (i.e., informal)". The problem with this is that the validity of any such hypothesis is debatable precisely because it is ambiguous. We only fool ourselves into believing we understand a system when we have guessed plausible informal hypotheses about it. We literally have *no way to know* whether such hypotheses are correct because — like the words quoted above — we do not even know whether they mean the same thing to different people.

There might be good reasons for making *some* hypotheses that are informal and ambiguous. But if the understanding of the system achieved by reverse engineering is not to be entirely superficial — say, we are supposed to use it to decide how to modify the program — then it is necessary to have a "strong mental model" of the system [Littman 86]. So *some* other hypotheses simply must get to the heart of the matter and deal with a level of detail where ambiguity (intentional or not) is inappropriate.

2.4. Substantive Hypotheses

To argue that our main technical claim (see Section 3.4) implies the intractability of reverse engineering, we rely on:

> **Substantive Hypothesis Premise** — Every reverse engineering task involves deciding the validity of at least one *substantive* hypothesis.

In fact, every serious reverse engineering project involves deciding the validity of many such hypotheses, but we only need to assume there is at least one. What makes a hypothesis "substantive"? Hypothesis H about system S is *substantive* whenever it is useful (has the three properties above) and has the following two additional properties:

- H is *systemic* — The validity of H depends on essentially all the code for S.

This condition excludes trivial hypotheses[5] which are essentially independent of S. Fortunately, these are just the hypotheses that contribute nothing to the understanding of S. In fact, it turns out that 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 for S, and there is simply no way to know whether it is there without looking for it. Furthermore, as a technical matter we must have this restriction because there is no basis for claiming that it is hard to decide the validity of H if H is independent of most or all of S.

- H is *comprehensive* — The validity of H cannot be decided by a small set of test cases.

This restriction is reasonable because it is possible to make a hypothesis H whose validity might be decided and explained simply by creating a record of the execution of S on a small set of particular inputs or conditions. Such a limited hypothesis cannot lead to substantially improved understanding of S. And as above, we wish to rule it out on technical grounds because there is no basis for claiming that it is hard to decide the validity of such a hypothesis — it might explain just a few arbitrarily short execution histories.

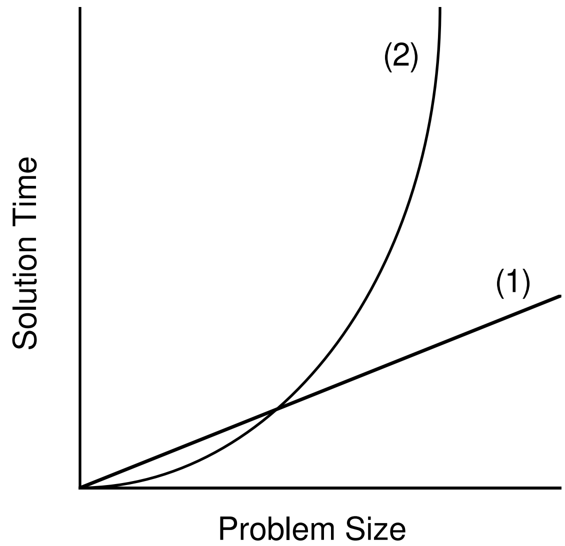
3. The Computational Problem Model

In this section we outline a framework for making the intractability argument. It is based on Figure 1, which depicts a typical situation for complexity analysis. The functions labeled (1) and (2) illustrate two possible lower bounds on complexity of the problem at hand. An argument establishing any such bound implies that there is no method for solving the problem whose running time falls below the corresponding function. If the strongest (fastest-growing) lower bound grows, say, linearly with problem size, like (1), then the problem *might be easy*; an upper bound still requires exhibiting a method that has good execution time. But if there is a lower bound that grows exponentially with problem size, like (2), then the problem is inherently *intractable*. Practically speaking, the latter conclusion — which we demonstrate for the problem of reverse engineering of legacy systems — means that the minimum solution time grows so fast that even for fairly small problem sizes it is doomed to be unacceptably large.

---

5 For example, "S executes at least zero instructions before halting."

Figure 1 — Two Lower-Bound Curves for Some Problem



3.1. Problem to be Solved

The particular computational problem whose complexity we consider is just the second phase of each 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 hypotheses to be explained, or for the fact that generally there are many such claims involved in obtaining a useful understanding of a system, because these just add to the time required to reverse engineer a system. There is every reason to suspect that generating substantive hypotheses is hard, too, but we do not try to demonstrate this.

3.2. Problem Size

Using "problem size" as a surrogate for "problem" is a common technique for presenting complexity as though it were a function of one simple independent variable. The actual time to solve an instance of EXPLAIN is a function of the entire input (source code S and hypothesis H), not just the "size". Nonetheless, it suffices for our argument to follow standard practice and use a traditional measure of problem size: the total number of characters of source code in all the files constituting the system (denoted $|S|$), plus the total number of symbols needed to represent the hypothesis (denoted |H|). In fact, since our argument is based entirely on |S|, we do not need to worry about |H|.

We could as well talk about lines of code in defining |S| , since the number of lines and the number of characters presumably differ by at most a constant factor. But this would just change the scale of the horizontal axis of Figure 1, not the shape of any lower bound curve.

One key property of |S| is that it is not artificially inflated. In traditional complexity analyses, for example, considering an integer to be represented in unary rather than radix notation can make an intractable problem *appear* tractable. But this is only because unary notation is not a compact way to encode an integer. When considering the size of a program, it is important to count each character of source code only once — even if it is, say, "included" in other files in many places and even if it influences the execution of the system at many places by, say, being part of the code for a procedure that is called from many places.

The documentation for a system might be helpful or misleading in reverse engineering, but it has no impact on whether or why a hypothesis holds for that system. It might be useful in generating hypotheses, but we do not include this as part of the problem to be solved, nor do we count the time to do it as part of the solution time. Therefore, we do not include documentation in measuring size.

3.3. Execution Time

The vertical axis of Figure 1 might be considered to depict, e.g., wall-clock time for a human or CPU time for a tool; or maybe some parts of EXPLAIN are handled by tools and other parts by humans. However, there is no magic — a tool can only do something that a human can do, at best merely faster by a constant factor. It does not matter which interpretation of time we choose because the decision simply affects the scale, or units, of the vertical axis of Figure 1, not the shape of the lower bound curve.

3.4. Lower Bounds

It is easy to see a trivial lower bound, a curve like (1) in Figure 1, simply because any algorithm for EXPLAIN must at least read its input:

Trivial Lower Bound — EXPLAIN(S,H) requires time at least |S|.

We aim to demonstrate a stronger lower bound, like (2) in Figure 1, which implies that reverse engineering of legacy code (as defined in Section 2) is intractable:

Exponential Lower Bound — There is a constant $c > 1$ such that, for every legacy system S and every substantive hypothesis H, EXPLAIN(S,H) requires time at least $c^{|S|}$.

4. Additional Premises

We need two more premises before building a case for the stronger lower bound.

4.1. Source Code is a Compact Representation of System 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 smaller units that are meant to be independent except at their interfaces — an objective that we call *modularity*. One intended result of modularity is the ability to reason modularly about program behavior. Liskov and Guttag nicely summarize this objective in their description of how we should like to reason about total correctness, but the statement 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. [Liskov 86, 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 often 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 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 legacy 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 holds in the typical case where there is no *a priori* bound on the longest execution sequence.

4.2. Maintainability 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, engineering and implementing code that supports effective reasoning about behavior is more subtle than it appears at first [Neumann 93, Wilde 93]. Problems arise from coupling through side-effects and aliased variables [Cook 78, Harms 91, Weide 91], arrays, pointers, and dynamic storage management [Hollingsworth 92, Ernst 94], generics [Ernst 91], inheritance [Leavens 90, Weber 92], 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.

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

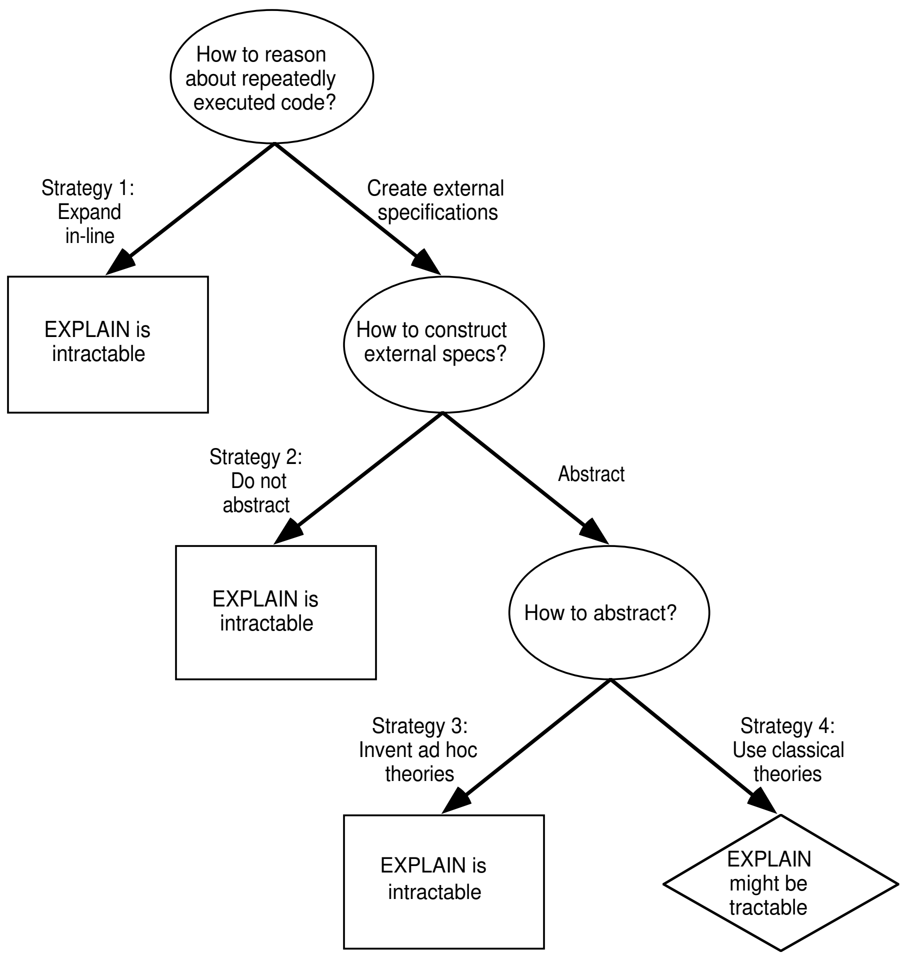
> **Non-Modularity Premise** — Every legacy system is hard to maintain because, in some crucial places, it has been designed or coded so that modularity is not achieved.

We 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; this is one major reason why.

5. Why EXPLAIN is Intractable

Our argument that EXPLAIN is intractable is based on a case analysis of strategies that might be used for reverse engineering; see Figure 2. The situation is this: We are given source code for system S and an substantive hypothesis H, and we must decide whether H is valid, and why or why not. How might this be approached?

Figure 2 — Structure of the Intractability Argument



5.1. Strategy 1: Do Not Specify Functional Component Behavior

The most obvious way to attack EXPLAIN is to rely on the source code for the components to check the claims of H, e.g., by some sort of tracing, symbolic execution, or non-modular proof system. Only the description of system behavior in H is qualitatively different from source code. For example, H (stated very concisely) might be something like "P computes function f_P by calling Q1 followed by Q2".

It is easy to decide whether P calls Q1 followed by Q2. But it is not easy to decide whether the net effect is to compute f_P. With only source code to work from, we must expand the calls to Q1 and Q2 in-line, and similarly for the calls they make to other routines, and so on down to the lowest level of the system. But since H is substantive, the Compact Source Code Premise says that it explains at least one execution sequence E whose length is exponential in |S|. Therefore, reasoning about whether H is a valid explanation just for E, by considering the dynamic effects of procedure calls and loop bodies without using specifications for them, involves examining at least exponentially more code than S contains statically.

5.2. Strategy 2: Specify Functional Component Behavior Without Abstracting

The most obvious approach to overcoming the above effect of the Compact Source Code Premise is to use not just the source code for the hypothesized functional components as in Strategy 1, but previously hypothesized external behavior specifications. Specifications of Q1 and Q2 in the example above might come from previous (separate) hypotheses about their own functional components. In a bottom-up fashion, we might hope to impose a modular reasoning structure on the entire system so that every instance of EXPLAIN is easy to solve. H might now be something like "P computes function f_P by calling Q1 (which, as we know, computes f_{Q1}) followed by Q2 (which, as we know, computes f_{Q2})".

A brute force attack in this direction involves making *ad hoc* definitions of new symbols with which to write component specifications. In the running example, then, the symbols f_{Q1} and f_{Q2} are nothing but short-hand notation for mathematical functions of the program state actually computed by Q1 and Q2. (We must deal similarly with loop bodies.) Down a level, suppose Q1 consists of a call to R1 followed by a call to R2, so $f_{Q1} = f_{R1} \circ f_{R2}$ *by definition*, and so on through the call hierarchy.

If f_P is defined in the same fashion as f_{Q1}, then $f_P = f_{Q1} \circ f_{Q2}$ is easy to check; it is a tautology. But in this case, H is not substantive. So suppose f_P is an independent statement of what P does, making $f_P = f_{Q1} \circ f_{Q2}$ an assertion that actually needs to be checked. Then f_{Q1} and f_{Q2} — when their definitions are expanded — are not materially different expressions of behavior than the source code for Q1 and Q2 and the procedures they call. This means that the exponential blow-up predicted by the Compact Source Code Premise is not avoided by this strategy. H is concise, but the alleged "specifications" involved in it give no leverage in reasoning about whether H is valid because they have to expanded just like the source code in Strategy 1.

5.3. Strategy 3: Specify Functional Component Behavior Using ad hoc Theories

It is necessary, then, for functional component specifications to do more than merely parrot the source code. The most obvious next approach is a refinement of Strategy 2 in which hypotheses have the same basic form, but in which the symbols used in specifications (e.g., f_P, f_{Q1}, and f_{Q2} in the running example) have calculational properties that are independent of the source code for P, Q1, and Q2. This approach — in which the hypotheses are generated by methods known variously [Ning 94, Biggerstaff 94] as concept recognition, plan identification, concept assignment, etc. — seems to characterize most recent work on reverse engineering of legacy code.

As above, we rule out the trivial special case that $f_P = f_{Q1} \circ f_{Q2}$ by definition, and consider only the situation where $f_P = f_{Q1} \circ f_{Q2}$ actually must be checked. Unfortunately, there is no reason to believe that any of the *ad hoc* "domain-specific" theories used to specify P, Q1, and Q2 should have any useful calculi individually, let alone in combination. This is troublesome because, if we hope to decide whether $f_P = f_{Q1} \circ f_{Q2}$, then there must be some theorems that allow us to combine symbols from the three theories. Mathematicians know well from history that few theories can withstand such a test of utility, i.e., that few symbolic systems actually result in much simplification of the phenomena they might be used to describe. It is totally implausible that *ad hoc* theories invented expressly for the purpose of understanding poorly understood software should be any different in this regard — especially that they should be useful in combination.

In the absence of calculi that support combination and simplification within and across theories, any attempt to check hypotheses must fall back to peeking under the covers of the specifications, and be subject to the explosion of symbols predicted by the Compact Source Code Premise.

5.4. Strategy 4: Specify Functional Component Behavior Using Classical Theories

There is still hope: Perhaps the domain-specific theories really *do* turn out to have useful calculi, either because they are wonderful new mathematical theories or (more likely) because they are isomorphic to instances of classical theories such as set theory or number theory. Then nothing seems to stand in the way of making this a tractable strategy for EXPLAIN.

In principle, there can be systems for which *every* substantive hypothesis can be EXPLAINed in less than exponential time by this approach. But consistent achievement of modularity — to the point where modular reasoning about behavior is possible — cannot happen by accident; this degree of modularity has to be designed in and kept there through disciplined design and coding practices. According to the Non-Modularity Premise, a system that achieves complete modularity throughout is not a legacy system.

6. Conclusion

Reverse engineering of large legacy systems is intractable in the following sense: Given real legacy code, the time required to decide the validity of a proposed explanation for why it exhibits any significant system-level behavior is 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.

The underlying reason is that designers and implementers of real legacy code have only partly achieved the objective of making the system modular. They have succeeded well enough to make the source code a very compact representation of dynamic system behavior. But they have not succeeded in the ultimate objective, which is to provide a basis for completely modular reasoning about behavior. One lesson from this should be that we need to put more emphasis, not less, on careful engineering of new systems [Neumann 93]; and that this emphasis needs to focus (at least) on overcoming the remaining barriers to creating systems that admit modular reasoning.

There are many good reasons 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 especially their sponsors and tool customers, 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, 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

[Biggerstaff 94] 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.

[Cook 78] Cook, S.A. Soundness and completeness of an axiom system for program verification. *SIAM J. Comp. 7*, 1 (Feb. 1978), 70-90.

[Chandrasekaran 93] Chandrasekaran, B., Goel, A.K., and Iwasaki, Y. Functional representation as design rationale. *Computer 26*, 1 (Jan. 1993), 48-56.

[CSTB 90] Computer Science and Technology Board. Scaling up: a research agenda for software engineering. *Comm. ACM 33*, 3 (Mar. 1990), 281-293.

[Ernst 91] 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.

[Ernst 94] 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.

[Harms 91] 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.

[Hollingsworth 92] 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 by anonymous FTP from host "ftp.cis.ohio-state.edu" in "pub/tech-report/1993/TR01-DIR/*".

[Hopkins 93] 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.

[Leavens 90] 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.

[Littman 86] 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.

[Liskov 86] Liskov, B., and Guttag, J. *Abstraction and Specification in Program Development*. McGraw-Hill, New York, 1986.

[Ning 94] Ning, J.Q., Engberts, A., and Kozaczynski, W. Automated support for legacy code understanding. *Comm. ACM 37*, 5 (May 1994), 50-57.

[Neumann 93] Neumann, P.G. Are dependable systems feasible? *Comm. ACM 36*, 2 (Feb. 1993), 146.

[Waters 94] Waters, R. C., and Chikovsky, E. Reverse engineering progress along many dimensions. *Comm. ACM 37*, 5 (May 1994), 23-24.

[Weber 92] 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.

[Weide 91] Weide, B.W., Ogden, W.F., and Zweben, S.H. Reusable software components. In *Advances in Computers, vol. 33*, M.C. Yovits, ed., Academic Press,1991, 1-65.

[Weide 93] 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.

[Wilde 93] Wilde, N., Matthews, P., and Huitt, R. Maintaining object-oriented software. *IEEE Software 10*, 1 (Jan 1993), 75-80.