Mitsubishi Electric Research Laboratories
Cambridge Research Center

Technical Report 93-02 **February 10, 1993**

Challenges to the Field of Reverse Engineering

by

Peter G. Selfridge[1]
Richard C. Waters[2]
Elliot J. Chikofsky[3]

Abstract

Driven by the economic importance of maintaining and improving the enormous base of existing software systems, the reverse engineering of software has been of rapidly growing interest over the past decade. More and more commercial software tools support aspects of reverse engineering, and more and more researchers in academic and industrial organizations are addressing themselves to the fundamental problems of reverse engineering.

In the best of all worlds, we researchers on reverse engineering would be working together toward clear goals of great economic importance. Unfortunately, it appears that we are mostly just groping around in a swamp, each looking for a bit of dry ground (whether or not it actually leads out of the swamp), and running into each other only occasionally. If we are to make rapid and effective joint progress, a number of improvements need to be made in the way we are pursuing research.

This is a deliberately controversial paper presented in a confrontational manner in the hope that it will trigger lively discussion. The views expressed are those of the authors and not necessarily those of their employers or funders.

Submitted to *Working Conference on Reverse Engineering*, Baltimore MD May 1993.

201 Broadway
Cambridge Massachusetts 02139

[1]AT&T Bell Laboratories; Room 2B-425; Murray Hill, NJ 07974
[2]Mitsubishi Electric Research Laboratories; 201 Broadway; Cambridge, MA 02139
[3]Northeastern Univ.; 360 Huntington Ave.; Boston, MA 02115

Publication History:-

1. First printing, TR 93-02, February 1993

1 Introduction

Research on reverse engineering of software is clearly in its infancy. Inasmuch as this is the case, it is not surprising that much reverse engineering research is of an early exploratory nature. In particular, there is a natural tendency to gravitate toward contrived problems that are designed to be soluble by the particular technologies being explored. This permits progress to be made on the technologies in question. However, it inherently limits the economic impact of the research, because such contrived problems typically have little to do with reality. In addition, it limits the impact of the research on other researchers, because such contrived problems are typically so different from the problems being attacked by other researchers that the results don't transfer.

If reverse engineering research is to grow from infancy to robust and vibrant maturity, we must graduate from contrived problems to real problems with real economic impact. In addition, we must focus on the effective interchange and cross fertilization of ideas between researchers.

This paper challenges reverse engineering researchers with nine concrete suggestions for improving the state of reverse engineering research. There is no doubt that people can argue at length about the details of these suggestions. (We hope they will.) However, we believe that taken together, these suggestions point in an important direction for improving the maturity of our field.

2 Avoid Artificially Contrived Data

As a general matter, research must proceed in steps, first considering simplified data that are capable of easy attack and then graduating to more realistic data as technology improves. However, there are significant risks to this approach. If simplification is taken too far, one can end up working on data that has nothing to do with reality. Also, if one continues working on simplified data too long, good initial research can loose its momentum. As much as possible, we must strive to do research based on real data.

Use Real Programs As Examples, Not Toy Programs

A significant amount of reverse engineering research has used toy programs as input data. These programs tend to be toys in two dimensions. First, to reduce computational requirements, the programs tend to be very short (under one hundred lines). (This is particularly true of research aimed at deep program understanding where the computational requirements are extreme.) Second, and perhaps worse, the programs tend to be written by the researchers themselves or by students, rather than being real legacy programs.

One can rightly wonder whether a twenty line student program has anything to do with real legacy programs. Therefore one can rightly wonder whether research applied to such programs has anything to do with solving real reverse engineering problems. To counter these questions one must aggressively move towards using real programs as data.

In experiments, we should use preexisting programs of realistic size (i.e., thousands of lines). If small programs have to be used, we should nevertheless insist on using preexisting programs.

Use Systems As Examples, Not Programs

A lot of reverse engineering research focuses on programs (pieces of code that take a few inputs and compute a few outputs). The problem with this is that the real legacy software problems we face are not programs, but systems (large conglomerations of programs that together take copious quantities of input data, often interactively, and create copious quantities of output data, often interacting with complex state objects such as databases).

Systems are quantitatively much more complex than programs, consisting of hundreds of thousands if not millions of lines of code. More importantly, systems are qualitatively much more complex than programs. In particular, many programs have relatively concise and precise specifications that are more or less domain-independent, while systems typically have enormous imprecise specifications that implicitly rely on large quantities of domain-dependent information. Further, while there is often a relatively simple relationship between the algorithms used by a program and its specification, this is almost never the case for a system.

Even though systems consist of programs, there is a fundamental qualitative difference between the two, because the relationship between the specification for a system and the specifications of the programs it consists of is typically very complex. As a result, an ability to reverse engineer programs does not directly imply an ability to reverse engineer systems. This calls reverse engineering research that focuses solely on programs seriously into question.

We should use complete systems as examples. If small systems have to be used, they should still be real systems that perform coherent user-oriented tasks, rather than isolated programs. As above, they should be real preexisting systems, not contrived or simplified ones. Above all, remember that if code isn't ugly, it's not real.

Utilize the Full Array of Potential Inputs

Much reverse engineering research looks at facts about computation (i.e., what operations are applied to what data items) as the sole source of information. This bias probably stems from the bias toward operating on programs, which is noted above. When looking at an isolated program (particularly a small one) it is not unreasonable to think that everything worthy of note about the program is evident in the computation it performs. However, when looking at a system (or a program in the context of a system) this is not the case.

Suppose two inputs are added together. When looking at an isolated mathematical program, it may well be sufficient to merely record the fact that a sum has been computed and assume that the computational context within the program itself will specify any additional information that is relevant. However, in the context of some business data processing system, this is unlikely to be sufficient. One has to figure out what the inputs mean and what the sum means. Perhaps the computation is adding regular hours to overtime hours to determine total hours. Alternatively, perhaps the computation is adding the charitable contributions for the current month to the total contributions for the year so far, to determine a new total.

To determine what is going on in a system, one has to look at sources of information beyond the mere computation being performed. These sources include the names of programs, variables and files, comments in code, separate documentation such as design rationale and user manuals, program state information such as the contents of databases, domain models, etc. Each of these sources of information must be treated differently, and each has its inherent problems. Therefore, a reverse engineering tool that utilizes multiple sources of information is bound to be more complex than one that relies on only a single source. However, there is no reason to believe that reverse engineering is possible based solely on any single source of information.

To date, reverse engineering researchers have looked at many of these sources of information. However, only a few research efforts have looked at integrating diverse sources of information in a single reverse engineering tool. For the health of the field, we should adopt a general bias toward using multiple sources of information, rather than the current bias of only using one. Beyond this, we should strive to develop reverse engineering tools that simultaneously use EVERY available source of information.

3 Focus On Concrete Economic Impact

This section addresses the same fundamental problem as in the last section, but from a different perspective. Research, by its very nature, seeks to abstract problems from the real world so they are tractable to the scientific method and scientific modes of understanding. However, such an abstraction process is always in danger of throwing the baby out with the bath water—abstracting out the real essentials.

Researchers must counter this danger by choosing what to attack very carefully. The last section argues that researchers must ensure that they use real data. This section argues that researchers must ensure that the tools they develop attack real problems.

Strive For Pragmatic Utility Above All Else

Researchers often complain that the people developing commercial tools ignore research results no matter how promising they are. To a considerable extent, this is undoubtedly due to narrow perspective and over conservatism on the part of commercial tool builders, and it would certainly be wise for commercial developers to follow research more closely and be more adventuresome. However, it must be admitted that much of the lack of communication stems from bad choices made by researchers. Often, it just isn't easy for anyone to tell whether a research solution to an artificial research problem can be generalized to real problems.

We should strive to develop tools that have direct pragmatic utility—i.e., tools that attack real problems of economic importance and deliver real benefits. Doing this typically brings in a lot of messiness that it would be nice to avoid (such as investigating what user's needs really are, providing good user interfaces, and dealing robustly with exceptional cases), however, it is the only way to produce results that are guaranteed to impress end users and commercial tool developers.

A key part of this is that we should be problem directed in our research, not technology directed. Much current research is technology directed in the sense that it is aimed at proving the utility of a technique, rather than on solving a problem. This can lead to the construction of a problem that the technique solves without much regard for whether the problem is a real problem that needs to be solved.

In contrast, we should attack problems of known value without enormous regard for whether our favorite technologies will work. In particular, while it is certainly laudable to seek deep solutions, we should not overlook shallow approaches. It is much better to come up with a shallow solution to an economically important problem, than a deep solution to a problem nobody cares about.

Develop Semi-Automatic Systems

There is a natural desire to create totally automatic reverse engineering systems. After all, it seems obvious that if a totally automatic solution to a problem is possible, it is going to be better than a semi-automatic solution. However, reverse engineering is complex enough that, at least in the short term, it does not appear that very many real problems are susceptible to totally automatic solutions. In fact, human intervention may be essential even in the long term.

Due to the inherent complexity, the continued search for totally automatic solutions has forced a lot of reverse engineering research into unrealistic corners attacking artificial problems with unreasonable constraints placed on them. It is high time that we place more emphasis on figuring out how to constructively put people in the loop. This is probably essential if we are to attack full scale reverse engineering problems. In addition, it forces us to pay very close attention to exactly what people do on these tasks. This has manifold advantages, both focusing us on real problems and yielding insight on how they can be solved.

Do Empirical Studies

The above complaints notwithstanding, there is reverse engineering research that uses real data and attacks real problems. However, much of this research has relied on gee whiz scenarios and anecdotal evidence to argue that the experimental tools developed are valuable. This is not sufficient to convince skeptical users and commercial tool developers. (It is sometimes sufficient to convince other researchers, however it shouldn't be. One cannot tell what is really going on without measuring it in detail.)

What is needed is empirical studies on real software systems in real situations that show quantifiable results. This is difficult to achieve, but we must try.

The whole field of software engineering is hobbled by a lack of empirical studies. The state of affairs in reverse engineering research is even worse.

4 Facilitate Inter-Researcher Communication

Friendly competition and the free interchange of ideas is the life blood of scientific research. For this to happen, we must be able to communicate with each other and we must be able to easily compare the tools we develop with the tools developed by others. We cannot gain from the work of others if we cannot figure out how to apply their work to our problem, or whether their approach has advantages over our own approach.

Establish Standard Terminology

Probably due to the youth of reverse engineering as a field of research, a standard set of terminology has yet to emerge. This impedes the free flow of information by introducing non-trivial difficulties when reading each other's papers. It also leads to significant confusion on the part of new people coming into the field.

One might say that this is not a fundamental problem, and perhaps it isn't. However, there is no excuse for the problem to exist at all. We would all be better off with a standard set of terminology, almost no matter what the definitions are. We should all be willing to give up some of our favorite definitions for the greater goal of easy common understanding.

A start toward standardization was made by [2], which proposes standard definitions for six key terms (forward engineering, reverse engineering, redocumentation, design recovery, restructuring, and reengineering). However, much more needs to be done, both in encouraging adoption of standard definitions by the research community and in defining more terms. In particular, there are probably several score terms that could profitably be standardized.

Further work on standardized terminology is being carried out by the IEEE Computer Society, Technical Committee on Software Engineering, Subcommittee on Reverse Engineering. They are working toward publishing an enlarged set of definitions, however, this is not yet complete [3].

As a specific example of the terminological problems we face, consider the Workshop on Artificial Intelligence and Automated Program Understanding that was held in conjunction with AAAI-92 in July of 1992. This workshop did not have a formal proceedings, however, it did have a set of informal notes written by the participants that was handed out at the workshop. Perusal of these notes reveals that essentially every researching on automated program understanding uses their own idiosyncratic terminology.

For example, almost every program understanding researcher uses some kind of abstract representation for programs and attempts to locate standard patterns of computation in programs. However, there is no agreement on what these two concepts should be called. Worse, terms used by some researchers for one concept are used by other researchers for the other concept. We propose the following definitions.

> **Plan** - An abstract representation for a program (or anything else) is a plan. This is in analogy with the plan for a building. The key intuition is that a plan highlights some pieces of information while discarding others. For example, the floor plan of a building shows you where they walls are while omitting information about elevations and other aspects of a building. The wiring plan shows you where the electrical wires and devices are without saying anything about the plumbing (which is indicated on the plumbing plan).
>
> This should not be confused with the sense of a plan as an intended action. (However, even in this sense, plans are still typically abstract in nature specifying only some of the details of the intended action.) It is proposed here that there is no reason to overly differentiate between existing programs, intended programs, and compendiums of knowledge about programming. In each case, if the information is represented abstractly (as opposed to in full detail as program text), then it is a plan.

> **Cliche** - A cliche is a common pattern that is used over and over again. This is in analogy with the standard English meaning of cliche—a sequence of words that is used over and over again. In English writing, cliches are considered bad form because they lack creativity. However, in engineering they are good form, because they embody standard, well understood, practice.
>
> The essence of the meaning of the term cliche is repeated use. The word plan has no such bias. One can have a plan for a computation that appears in only one program in the whole world. A computation (whether represented as a plan or not) is a cliche only if it occurs many times. A compendium of knowledge about programming inevitably contains many cliches.

Automated program understanding typically involves representing both an input program and a compendium of programming knowledge in terms of some plan representation. A searching process then locates the cliched plans in the compendium in the plan representing the input program.

> **Domain model** - A somewhat related term is that of a domain model. A domain model is a compendium of information about a particular application domain, rather than about programming. For example, a domain model of accounting would contain the cliches of accounting—the standard concepts accountants use, the standard kinds of information accountants collect, and the standard procedures accountants follow. These domain cliches are just as important for understanding an accounting program as programming cliches are.

While we believe the definitions above are good ones, we believe much more strongly that there should be standard definitions, even if they differ from the ones above. Over a span of decades it is very likely that standard definitions would naturally appear of their own accord as new researchers copy the terminology of older researchers. However, if we wait this long we will pay an unfortunate price in reduced communication in the interim. It would be much better for the researchers in the field of reverse engineering to get together as a group and take proactive action to develop a set of agreed upon definitions.

Make Goals Explicit

A factor that can make it hard to compare different research approaches is that the goals are often not clearly stated. Reverse engineering is a large field and researchers have very different opinions about what goals should be pursued. It is impossible to judge a piece of research without having a clear idea of exactly what its goals are.

When describing an experimental tool, a researcher should make several points very clear. What is the purpose of the tool—system/program understanding, redocumentation, restructuring, design recovery, test planning, extracting some specific piece of global information, or something else? What information is being recovered—specifications, designs, control structure, business rules, interface descriptions, data architecture, algorithms, or something else? How exactly is the tool supposed to interact with the user?

Select Standard Data Sets.

Another factor that can make it hard to compare research prototypes is that everyone uses their own test data. As long as this is the case, it is impossible to get any hard quantitative comparison between tools. What we need are a set of benchmark examples that can be used as a basis for comparison.

The importance of this can be seen in other fields. For example, the ongoing series of International Workshops on Software Specification and Design (proceedings available form IEEE Press) has made very good use of standard specification problems (the university library database problem and the elevator control problem) [1] as a means of comparing different specification languages and specification tools.

This is a good example of the value of standard problems, however, the exact problems used are not a good analogy with the needs of reverse engineering research, because the library and elevator problems are both toy examples stated in less than a page of text.

A more relevant example of the power of standard problems comes from the DARPA spoken language research community. Over the past several years they have compiled a huge store of data about the task of providing an airline travel information system [4] and are using this as the basis for comparing research systems that understand spoken utterances. The data includes spoken utterances, the sentences they correspond to, and what the sentences mean. A number of different research groups are constructing systems that can respond to spoken input, retrieving flight information and making airline reservations by connecting to the airlines' computerized flight reservation systems.

Rather than trying to compare speech recognition and natural language understanding in some abstract way, the various systems produced are compared based on how easy it is for a user to actual determine what reservation they want to make and then make it. Doing comparisons at a system-wide level based on identical data leads to much clearer results.

Note that the standard DARPA spoken language problem includes both copious quantities of specific utterances and a standard task to be performed that requires the processing of these utterances. In the field of reverse engineering we need the same thing.

To start with, we need real large systems with all there attendant sources of information. At the very least, this would amount to thousands of pages (if not much more). Given the necessary size, it would not be practical to just make up these examples. This is all to the good, because as argued above, we should not be making up test data in any case. It would be much better to seek out real example systems from actual commercial settings. We also need some standard tasks to be performed on the example data.

In addition to getting examples of actual legacy systems, it is also interesting to consider developing standard problems in areas where automatic forward engineering is possible. For example, consider the problem of taking the output produced for the compiler of a fourth generation language and using reverse engineering to recover the original high level input. This task has the advantage that large quantities of test data can be constructed with ease, and the exact desired result of reverse engineering is very precisely specified.

5 Recommendations For the Next Conference On Reverse Engineering

One might say that some of the suggestions above (like using real data and attacking real problems) are mom-and-apply-pie statements—general (and somewhat vague) statements that everybody agrees with, but nobody knows how to do. This may be true, however, it is important that we agree publicly on the importance of these goals and strive toward them, even if only slowly. Further, we should judge each other's work on how closely we approach these goals.

The above notwithstanding, several of the suggestions above are quite detailed and there is no excuse for not taking immediate action. One possible mechanism for doing this is this conference on reverse engineering. If the conference grows into a series of reverse engineering conferences, it should be possible for them to make a significant contribution to the goals annunciated above. In particular, five steps seem eminently practical.

Make Application To Real Data an Acceptance Criteria

Starting with the next conference on reverse engineering, the application of an experimental tool to real data instead of contrived data should be an explicit evaluation criteria. This would not be a hard and fast requirement, but reporting experiments using real data would definitely weigh in favor of a paper and using contrived data would weigh against it.

Make Application To a Problem of Economic Importance an Acceptance Criteria

Similarly, starting with the next conference on reverse engineering, the extent to which an experimental tool attacks a problem of real economic importance should be made an explicit evaluation criteria. As above, this would not be a hard a fast requirement but would be significantly considered when comparing papers.

Require Papers To State Their Goals In Detail

Again starting with the next conference on reverse engineering, stating the goals of an experimental system in explicit detail should be made a minimum requirement for acceptance. There is no reason for anyone to overlook doing this, and no reason for any paper that fails to state goals clearly to be accepted.

Make Forming a Terminological Consensus an Explicit Goal

At this conference on reverse engineering, we should explicitly adopt the goal of establishing standard terminology and recruit people to assist the IEEE Subcommittee on Reverse Engineering in this task. In particular, we should adopt the specific goal of publishing a much expanded set of proposed definitions at or before the next conference on reverse engineering. The goal would be to discuss these definitions at the next conference and attempt to come to a consensus.

Once a consensus has been reached and a final set of definitions published, using the standard terminology should be made a minimum requirement for acceptance at subsequent conferences on reverse engineering.

Make Developing a Collection Of Standard Data Sets an Explicit Goal

At this conference on reverse engineering, we should formally adopt the goal of developing a set of standard reverse engineering data sets. In particular, we should solicit papers for the next conference on reverse engineering whose sole content is describing reverse engineering data sets that are publicly available.

For example, someone working at a commercial concern might be able to persuade the company to make one of their actual legacy systems publicly available, along with all associated information. (It might be easiest to do with a system the company had just decided to abandon.) It should be recognized that setting up the easy availability of such a data set would be a very worthwhile research contribution.

Once a couple of standard data sets are available, using them as test cases, should be made an explicit evaluation criteria for subsequent conferences on reverse engineering. It should not be required, because it may be unreasonable in some situations, however, it should be greatly encouraged.

References

[1] R. Babb, *et al.*, “Workshop on Models and Languages for Software Specification and Design”, *IEEE Computer*, 18(3):103–108, March 1985.

[2] E.J. Chikofsky & J.H. Cross, “Reverse Engineering and Design Recovery: a Taxonomy”, *IEEE Software*, 13–17, January 1990.

[3] J.H. Cross II, “Message from the Chair”, *Reverse Engineering Newsletter*, (4):1, January 1993.

[4] C.T. Hemphill, J.J. Godfrey, & G.R. Doddington, “The ATIS Spoken Language Systems Pilot Corpus”, *DARPA Speech and Natural Language Workshop*, Hidden Valley PA, June 1990.