Reverse Engineering and its Application in Modern Software Scenario and Implications

Anupam Singh
3rd year Computer Science and Engineering
IEEE Student Member
Vellore Institute of Technology

Anupam Das
3rd year Computer Science and Engineering
IEEE Student Member
Vellore Institute of Technology

***Abstract:* This paper dwells into the concept of Software Reengineering and its scope, demand and effectiveness in the contemporary professional world. The authors give an introduction and analyses of the present scenario of software trends followed in the industry and highlight various suggestions and improvements to cut down costs and fasten the process of Software development. The comparative study of various Reengineering tools and their evolvement to suit the changing needs have also been highlighted. The paper concludes with a prophetic look at the emergent technologies in this field and paves path for the scope of further research**.

Keywords: Re-Engineering, CASE, DBMS, Legacy Systems

An Introduction to the Science of Looking Backwards

"Living backwards!" Alice repeated in great astonishment.
"I never heard of such a thing!"
— Lewis Carroll

Reverse-engineering is the process of taking a piece of software or hardware, analyzing its functions and information flow and then translating those processes into a human-readable format. The goal is often to duplicate or improve upon the original item's functionality. In the software field Reverse Engineering is the process of analyzing the software with the objective of recovering its design and specification. Reverse engineering is an important part of Software re-engineering process.

Need for Re-Engineering Process

Since the mid 1960s, investment in software by business, government and other organizations has grown incredibly quickly. Software systems are used in almost all organizational activities. These systems must be maintained and evolve as new requirements emerge and new hardware is introduced into the organization.

In some business, it has been estimated that 80% of all software expenditure is consumed by system maintenance and evolution. The number of systems to be maintained is still increasing. There is a huge backlog of maintenance requests. This means that it is sometimes impossible for organizations to invest in new systems to improve organizational efficiency.

Old systems that must still be maintained are called as *legacy* systems. These old systems normally use immense amount of code and are generally written in COBOL or FORTRAN which are no longer in use. Moreover these languages have the limitations of providing limited program structuring facilities and support for data structuring. So the legacy systems may be poorly structured and their documentation may be out of date or non existent.

But in spite of having all these limitations many legacy systems are critical to the operation of the organization which uses them. They embed business knowledge and procedures which have emerged over the lifetime of the system. This knowledge may not be documented elsewhere. The risk of scrapping and rewriting these systems is very high. Much of this knowledge would have to be rediscovered by trial and error.
Consequently, organizations cannot afford to make their legacy system obsolete. They must somehow keep them in operation and continue to adapt them to new requirements and here only

comes the role of software re-engineering.

Software re-engineering is concerned with taking existing legacy systems and reimplementing them to make them more maintainable. As part of this reengineering process, the system may be redocumented or restructured. It may be translated to a modern programming language, implemented on a distributed platform rather than a mainframe or its data may be migrated to a different database management system. These all can only be done if a better understanding of the subject system is developed and such understanding is provided by reverse engineering which enables the re-engineering process to derive the design and specification of the system from its source code.

Objectives of Reverse Engineering

What are we trying to accomplish with reverse engineering? The primary purpose of reverse engineering a software system is to increase the overall comprehensibility of the system for both maintenance and new development.
There are six key objectives that will guide its direction as the technology matures:

1. *Cope with complexity*:

We must develop methods to better deal with the shear volume and complexity of systems. A key to controlling these attributes is automated support. Reverse-engineering
methods and tools, combined with CASE (Computer-aided software engineering) environments, will provide a way to extract relevant information so decision makers can control the process and the product in systems evolution. .

2. *Generate alternate views*:

Graphical representations have long been accepted as comprehension aids.
However, creating and maintaining them continues to be a bottleneck in the process. Reverse-engineering tools facilitate the generation or regeneration of graphical representations from other forms. While many designers work from a single, primary
perspective (like dataflow diagrams), reverse-engineering tools can generate additional views from other perspectives (like control-flow diagrams, structure charts, and entity-relationship diagrams) to aid the review and verification process. We can also create alternate forms of non graphical representations with reverse-engineering tools to form

an important part of system documentation.

3. *Recover lost information*:

The continuing evolution of large, long-lived systems leads to lost information about the system design. Modifications are frequently not reflected in documentation, particularly
at a higher level than the code itself. While it is no substitute for preserving design history in the first place, reverse engineering - particularly design recovery – is our way to salvage whatever we can from the existing systems. It lets us get a handle on systems when we don’t understand what they do or how their individual programs interact as a system.

4. *Detect side effects*:

Both haphazard initial design and successive modifications can lead to unintended ramifications and side effects that impede a system’s performance in subtle ways. As
Figure 1 shows, reverse engineering can provide observations beyond those we can
obtain with a forward-engineering perspective, and it can help detect anomalies and problems before users report them as bugs.

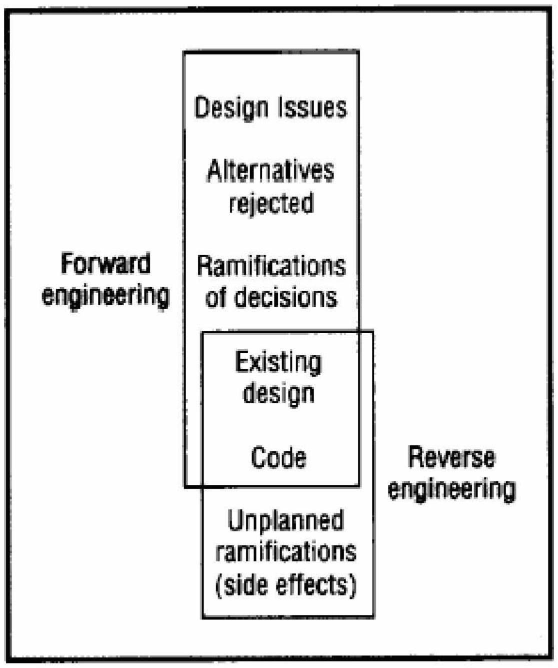


FIGURE 1.

5. *Synthesize higher abstractions:*

Reverse engineering requires methods and techniques for creating alternate views that transcend to higher abstraction levels. There is debate in the software community as to how completely the process can be automated. Clearly, expert system technology will play a major role in achieving the full potential of generating high level abstractions.

6. *Facilitate reuse*:

A significant issue in the movement toward software reusability is the large body of existing software assets. Reverse engineering can help detect candidates for reusable software and are components from present systems.

Reverse Engineering Process

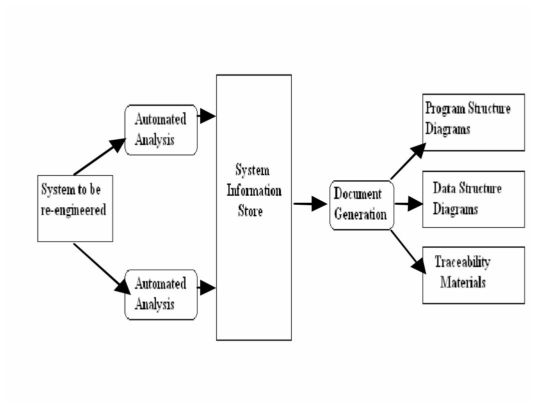


FIGURE 2

The reverse engineering process is illustrated in fig2.The process starts with an analysis phase. During this phase the system is analyzed using automated tools to discover its structure. In itself, this is not enough to re-create the system design. Engineers then work with the system source code and its structural model. They add information to this which they have collected by understanding the system. All of this information is maintained in some information store, usually in the form of directed graph. The program code is also stored.

Information store browsers may be available to compare the graph structure and the code. These may be used to add further information that has been inferred about the design. Documents of various types may be generated from this information. These might include program and data structure programs and traceability matrices. Traceability matrices show where entities in the system are defined and referenced. The process of document generation is an iterative one as the design information is used to further refine the information held in the system repository.

As part of the reverse engineering process, various tools for program understanding may be used. These usually present different kinds of system view and allow easy navigation through the source code. After the system design documentation has been generated, further information may be added to the information store to help re-create the system specifications. This usually involves further manual annotations to the system structure. The system specifications cannot be deduced automatically from the system model.

Sub areas of Reverse Engineering

Reverse engineering is the process of analyzing a subject system to:-

1. Identify the system’s components and their interrelationships and

2. Create representations of the system in another form or at a higher level of abstraction.

Reverse engineering in and of itself does not involve changing the subject system or creating a new system based on the reverse-engineered subject system. It is a process of examination, not a process of change or replication.

There are many sub areas of reverse engineering. Two sub areas that are widely referred to are redocumentation and design recovery.

1. *Redocumentation:*

Redocumentation is the creation or revision of a semantically equivalent representation within the same relative abstraction level. The resulting forms of representation are usually considered alternate views (for example, dataflow, data structure, and control flow) intended for a human audience. Redocumentation is the simplest and oldest form of reverse engineering, and many consider it to be an unintrusive, weak form of restructuring. The "re-" prefix implies that the intent is to recover documentation about the subject system that existed or should have existed.

Some common tools used to perform redocumentation are pretty printers (which display a code listing in an improved form), diagram generators (which create diagrams directly from code, reflecting control flow or code structure), and cross-reference listing generators. A key goal of these tools is to provide easier ways to visualize relationships among program components so that we can recognize and follow paths clearly.

2. *Design recovery*:

Design recovery is a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself. Design recovery is distinguished by the sources and span of information it should handle. Design recovery recreates design abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains. Design recovery must reproduce all of the information required for a person to fully understand what a program does, how it does it, why it does it, and so forth.

Reverse Engineering Tools

Techniques used to aid program understanding can be grouped into three categories: *unaided browsing, leveraging corporate knowledge and experience,* and *computer-aided techniques like reverse engineering.*

Unaided browsing is essentially “humanware”: the software engineer manually flips through source code in printed form or browses it online, perhaps using the file system as a navigation aid. This approach has inherent limitations based on the amount of information that a software engineer may be able to keep track of in his or her head.

Leveraging corporate knowledge and experience can be accomplished through mentoring or by conducting informal interviews with personnel knowledgeable about the subject system. This approach can be very valuable if there are people available who have been associated with the system as it has evolved over time. They carry important information in their heads about design decisions, major changes over time, and troublesome subsystems. For example, corporate memory may be able to provide guidance on where to look when carrying out a new maintenance activity if it is similar to another change that took place in the past. This approach is useful both for gaining a big- picture understanding of the system and for learning about selected subsystems in detail.

However, leveraging corporate knowledge and experience is not always possible. The original designers may have left the company. The software system may have been acquired from another company. Or the system may have had its maintenance out-sourced. In these situations, computer-aided reverse engineering is necessary.

A reverse-engineering environment can manage the complexities of program understanding by helping the software engineer extract high-level information from low-level artifacts, such as source code. This frees software engineers from tedious, manual, and error-prone tasks such as code reading, searching, and pattern matching by inspection.

Adequate Reverse Engineering

Unfortunately, the definition of reverse engineering is not helpful to software engineers and their managers in planning and managing reverse engineering efforts. In particular, it does not give any guidance to determining the quality of the efforts and the resulting representations. That is, it is hard to know when the understanding gained adequately represents the original program. The question we explore is *how we know if a reverse engineering effort has produced an adequate representation of a program*. Our answer is that a program representation produced by reverse engineering is adequate if it is sufficiently complete and accurate that an automated tool is capable of reconstructing a program equivalent to the original from it.

We apply two key insights to explore adequate reverse-engineering representations. Our first insight comes from examining the idea of *reversing* the reverse engineering process; that is, in taking the representation that results from reverse engineering and using it to reconstruct a program. In order to reduce variation and uncertainty in the process, we want the reconstruction to be done automatically. If we are able to do these two tasks, representation and generation, then we have an operational means for determining the completeness of our effort. The second insight concerns the quality of the representation; that is, how do we ensure that our representation provides useful insight into the program? Here we use a model of the program's application domain as an external standard against which we compare the representation as it is built up during reverse engineering. A domain model provides a set of expectations for constructs in the program and how they relate to each other. Comparing understanding gained while reverse engineering a program to expectations provided by the domain model encourages an accurate program representation.

Research Trends in Reverse Engineering

In summarizing the major research trends, accomplishments, and unanswered needs, we can divide our discussions into three major parts. First part concentrates on code reverse engineering, which has been the main focus of attention in this field over the past decade. In contrast, data reverse engineering, the topic of second part. Third and the last part explores the evaluation of reverse engineering tools.

1. *Code reverse engineering:*

In current research and practice, the focus of both forward and reverse engineering is at the code level. Forward engineering processes are geared toward producing quality code.

The importance of the code level is underscored in legacy systems where important business rules are actually buried in the code. During the evolution of software, change is applied to the source code, to add function, fix defects, and enhance quality. In systems with poor documentation, the code is the only reliable source of information about the system. As a result, the process of reverse engineering has focused on understanding the code.

Over the past ten years, reverse engineering research has produced a number of capabilities for analyzing code, including subsystem decomposition, concept synthesis, design, program and change pattern matching, program slicing and dicing, analysis of static and dynamic dependencies, object-oriented metrics, and software exploration and visualization. In general, these analyses have been successful at treating the software at the syntactic level to address specific information needs and to span relatively narrow information gaps.

2. *Data reverse engineering*:

Most software systems for business and industry are information systems, that is, they maintain and process vast amounts of persistent business data. While the main focus of code reverse engineering is on improving human understanding about how this information is processed, data reverse engineering tackles the question of what information is stored and how this information can be used in a different context. Research in data reverse engineering has been underrepresented in the software reverse engineering arena for two main reasons. First, there is a traditional partition between the database systems and software engineering communities. Second, code reverse engineering appears at first sight to be more challenging and interesting than data reverse engineering for academic researchers. Recently, data reverse engineering concepts and techniques have gained increasing attention in the reverse engineering arena.

3. *Evaluating reverse engineering tools*:

Previously we have made a discussion about the reverse engineering tools. But a question arises as to how to measure the success of the tools or theories that may be selected. Many reverse engineering tools concentrate on extracting the structure or architecture of a legacy system with the goal of transferring this information into the minds of the software engineers trying to maintain or reuse it. That is, the tool’s purpose is to increase the understanding that software engineers or/and managers have of the system being reverse engineered. But, since there is no agreed-upon definition or test of understanding, it is difficult to claim that program comprehension has been improved when program comprehension itself cannot be measured. Despite such difficulty, it is generally agreed that more effective tools could reduce the amount of time that maintainers need to spend understanding software or that these tools could improve the quality of the programs that are being maintained. Coarse-grained analyses of these types of results can be attempted.

Conclusion

There will always be old software that needs to be understood. It is critical for the software industry to deal effectively with the problems of software evolution and the understanding of legacy software systems. Since the primary focus of the industry is changing from completely new software construction to software maintenance and evolution, software engineering research and education must make some major adjustments. In particular, more resources should be devoted to software analysis in balance with software construction. Program understanding tools and methodologies address the problems of software evolution by helping software engineers to understanding large and complex software systems. Effective reverse engineering technologies can have a significant impact on the maintenance and evolution of these systems.

Even if we perfect reverse engineering technology, there are inherent high costs and risks in evolving legacy software systems. Developing strategies to control these costs and risks is a key research direction for the future. Practitioners need a reengineering economics book, which would serve as a guide to determine reengineering costs and to use economic analyses for making improved reengineering decisions.

Moreover the most critical issue for the next decade is to teach students about software evolution. Computer science, Computer Engineering, and Software

Engineering curricula, by and large, teach software construction from scratch and neglect to teach software maintenance and evolution. Topics such as software evolution, reverse engineering, program understanding, software reengineering, and last but not the least software reverse engineering must be stressed upon because reverse engineering, used with evolving software development technologies, promises to provide significant incremental enhancements to our productivity which is our final aim.

References

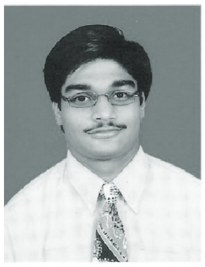
1. M.G. RekoffJr., "On Reverse Engineering,"
LEtX Trans. Systems, Man, and Cybernetics,
March-April 1985, pp. 244-252.
2. T.J. Biggerstaff, "Design Recovery for Maintenance and Reuse," Complter, July 1989,
pp. 3649.
3.*"Adequate reverse engineering" by* Spencer Rugaber, Terry Shikano, R. E. Kurt Stirewalt
4. Victor R. Basili and Harlan D. Mills. Understanding and
Documenting Programs. *IEEE Transactions on Software*
Engineering, SE-8(3):270-283, (May 1982).
5."Software Engineering" Ian Sommerville

6."Reverse Engineering and Design Recovery: A Taxonomy"

By Elliot j.Chikofsky and James H.Cross N

About the authors:

Anupam Singh is a Bachelor's degree student of Computer Science and Engineering at Vellore Institute of Technology, India. He is a student member of the IEEE, ISTE and IEI. His research interests include Human Computer Interfaces, Intelligent Systems, Reverse Engineering and Applied Software Engineering. He can be contacted regarding any queries at anupam.vit@gmail.com.



Anupam Das is a Bachelor's degree student of Computer Science and Engineering at Vellore Institute of Technology, India. He is a student member of the IEEE and ISTE. His areas of interest include Artificial Intelligence, Software Re-Engineering and Theory of Computation. He can be contacted regarding queries at anu_einstein2003@yahoo.co.in.