International Journal of Engineering Science Invention

Volume 2 Issue 1 || January. 2013

A Review of Reverse Engineering Theories and Tools

Ramandeep Singh

Asst. Prof., mechanical engineering dept., ctiemt, shahpur, jalandhar, Punjab, India

ABSTRACT : *Reverse Engineering is focused on the challenging task of understanding legacy program code without having suitable documentation. Using a transformational forward engineering perspective, the much of difficulty is caused by design decisions made during system development. Such decisions "hide" the program functionality and performance requirements in the final system by applying repeated refinements through layers of abstraction, and information-spreading optimizations, both of which change representations and force single program entities to serve multiple purposes. The demand by all business sectors to adapt their information systems to the web has created a tremendous need for methods, tools, and infrastructures to evolve and exploit existing applications efficiently and cost-effectively. Reverse engineering has become the most promising technologies to combat this legacy systems problem. Following the transformational approach we can use the transformations of a forward engineering methodology and apply them "backwards" to reverse engineer code to a more abstract specification. This paper presents reverse engineering program comprehension theories and the reverse engineering technology.*

Keywords— *Reverse Engineering, Forward Engineering, Code Reverse Engineering, Data Reverse Engineering*

I. INTRODUCTION

Engineering practice tends to focus on the design and implementation of a product without considering its lifetime. The notion of computers automatically finding useful information is an exciting and promising aspect of just about any application intended to be of practical use [11]. However, the major effort in software engineering organizations is spent after development [3,19] on maintaining the systems to remove existing errors and to adapt them to changed requirements. Unfortunately, mature systems often have incomplete, incorrect or even nonexistent design documentation. This makes it difficult to understand what the system is doing, why it is doing it, how the work is performed, and why it is coded that way. Consequently mature systems are hard to modify and the modifications are difficult to validate. Chikofsky and Cross defined reverse engineering to be "analyzing a subject system to identify its current components and their dependencies, and to extract and create system abstractions and design information" [5]. Current reverse engineering technology focuses on regaining information by using analysis tools [2] and by abstracting programs bottom-up by recognizing plans in the source code [6,8,10,20]. In corporate settings, reverse engineering tools still have a long way to go before becoming an effective and integral part of the standard toolset that a typical software engineer uses day-to-day.

Mainly the reverse engineering is derived for a) legacy code is written using rather tricky encodings to achieve better efficiency, b) all the necessary plan patterns in all the variations must be supplied in advance for a particular application, and c) different abstract concepts map to the same code within one application. Even more important, the main reason for doing reverse engineering is to modify the system; understanding it is only a necessary precondition to do so.

II. FORWARD ENGINEERING

In current forward engineering practice informal requirements are somehow converted into a semi-formal specification using domain notations without underlying precise semantics. The program then is constructed manually from the specification by programmer. Hidden in this creative construction of the program from the specification are a set of obvious as well as non-obvious design decisions about how to encode certain parts of the specification in an efficient way using available implementation mechanisms to achieve performance criteria.

Over time the program code is modified to remove errors and to adapt the system to changed requirements. The requirements may change to allow usage of alphanumeric keys and to be able to handle large amounts of data. Unfortunately, often these changes take place without being reflected correctly in the specification. The gap between the original specification and the program becomes larger and larger. The result

is a program code without a proper specification and with untrustworthy design information. The code becomes difficult to understand and, thus, difficult to maintain. To overcome this deficiency, it is important to change the specification first and then reflect the changes in the program code. A necessary precondition for this is to have reliable information about the relationship between the specification and the program code. There are two known approaches to reduce the gap between the specification and the program.

The first one is the development by stepwise refinement introducing intermediate descriptions of the system between the specification and the final program code. The intermediate descriptions should reflect major design decisions during the construction, which helps to understand the software and its design. However, this approach has some important drawbacks: the development steps are still manual, they are too large not to contain hidden design decisions, and there is no rationale directly associated to the steps [21].

The second approach is the transformational development of system where the fairly large manual development steps are replaced by smaller formal correctness-preserving transformations each of which reflects a small design decision or optimization. A necessary prerequisite for this approach is to have a formal specification which may contain domain notation by giving them a precise underlying semantics. Based on this the program code can then be derived by making small implementation steps using formal correctness-preserving transformations leading from more abstract specifications to more concrete specifications. The final program code then is correct by construction with respect to the formal requirements specification.

III. REVERSE ENGINEERING

If all existing system had been developed using a transformation system recording the design and its rationale there would be no need for reverse engineering. However, we have to make a concession to reality. A huge amount of conventionally developed system exists and these systems have errors and continual demand for the enhancement of their functional and performance requirements. Modification is a necessity. This means that there is a fundamental need to understand them. As they are often badly designed and have an incomplete, nonexistent, or, even worse, wrong documentation without any design information, this is a challenging task.

The reverse engineering allows addition of further plans, realizing patterns, and possibly further intertwining rules to enable a plan recognizer to detect the actual realization that can be found in the code. Thus, the reverse engineer not only has to provide the plan pattern for the actual realization of a plan but also the transformations and the justifications for applying them. As an alternative approach, the same effect can be achieved by using small transformations backwards and abstract the code step by step. So that the ends result of which is an abstract plan corresponding to the given code. In essence, the transformations applied backwards “jitter” the code into a form in which canonical plans are more easily found. Such an abstract plan may already exist in the system or may be new domain knowledge developed during the reverse engineering process. Doing so then avoids the need for repeatedly reverse engineering the system over its lifetime, caused by the continuous modification of its code.

IV. CODE REVERSE ENGINEERING

In current era of research, 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 [23]. During the evolution of system, 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. However, the code does not contain all the information that is needed [24]. Over time, memories fade, people leave, documents decay, and complexity increases [1]. Consequently, an understanding gap arises between known, useful information and the required information needed to enable software change. At some point, the gap may become too wide to be easily spanned by the syntactic, semantic, and dynamic analyses provided by traditional programming tools. Thus when focused only at the low levels of abstraction, the big picture behind the evolution of a system get overlooked [17].

Reverse engineering has typically been performed in an ad hoc manner. To address the technical issues effectively, the process must become more mature and repeatable, and more of its elements need to be supported by automated tools. The subsystem view to present this information should not require tedious manual manipulation. Instead, the mapping between responsibility and components should be consulted and a script would then generate the required view, with the option for minor, personal customization by the user. Such a script is an instance of a reverse engineering pattern [18], a commonly used task or solution to produce understanding in a particular situation. By cataloging such patterns and automating them through tool support, the maturity of the reverse engineering process can be improved.

V. DATA REVERSE ENGINEERING

Most systems for business and industry are information systems, i.e. 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 under-represented 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 attention in the reverse engineering arena. Researchers now recognize that the quality of a legacy system‟s recovered data documentation can make or break strategic information technology goals. The increased use of data warehouses and data mining techniques for strategic decision support systems [23] have also motivated an interest in data reverse engineering technology. Incorporating data from various legacy systems in data warehouses requires a consistent mapping of legacy data structures on a common business object model.

VI. 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 [26]. Unaided browsing is essentially “humanware”: the 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 engineer may be able to keep track.

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. 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 engineer extract high-level information from low-level artifacts, such as source code. This frees engineers from tedious, manual, and error-prone tasks such as code reading, searching, and pattern matching by inspection.

VII. EFFECTIVENESS OF REVERSE ENGINEERING TOOLS

Reverse engineering tools seem to be a key to aiding program understanding. In both academic and corporate settings, reverse engineering tools have a long way to go before becoming an effective and integral part of the standard toolset a for day-to-day usage [13]. Perhaps the biggest challenge to increased effectiveness of reverse engineering tools is wider adoption. While there is a relatively healthy market for unit-testing tools, code debugging utilities, and integrated development environments, the market for reverse engineering tools remains quite limited.

In addition to awareness, adoption represents a critical barrier. Most people lack the necessary skills needed to make proper use of reverse engineering tools. The art of program understanding requires knowledge of program analysis techniques that are essentially tool-independent. Since most programmers lack this type of foundational knowledge, even the best of tools won‟t be of much help. From an integration perspective, most reverse engineering tools attempt to create a completely integrated environment in which the reverse engineering tool assumes it has overall control. However, such an approach precludes the easy integration of reverse engineering tools into toolsets commonly used in both academic research and in industry.

VIII. EVALUATING REVERSE ENGINEERING TOOLS

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 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 [14], 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 the programs that are being maintained. Coarse-grained analyses of these types of results can be attempted. There are several investigative techniques and empirical studies that may be appropriate for studying the benefits of reverse engineering tools. The main

techniques applied to the evaluation of reverse engineering tools are expert reviews, user studies, field observations, case studies, surveys etc.

IX. CONCLUSIONS

Adoption of reverse engineering technology in industry has been very slow. So the reverse engineering is performed to maintain legacy code. Therefore, it should not be focused on program understanding but on system maintenance instead. This should be done in a way that frees us from reverse engineering a system again and again because of modifications made to its code over time. Researchers will continue to develop technology and tools for generic reverse engineering tasks, particularly for data reverse engineering but future research ought to focus on ways to make the process of reverse engineering more repeatable, defined, managed, and optimized. For large evolving systems, forward and reverse engineering processes have to be integrated and achieve the same appreciation for product and process improvement for long-term evolution as for the initial development phases. It is tough to predict all needs of the reverse engineers and, therefore, tools must be developed that are end-user programmable. Pervasive scripting is one successful strategy to allow the user to codify, customize, and automate continuous understanding activities and, at the same time, integrate the reverse engineering tools into personal development process and environment. Infrastructures for tool integration have evolved dramatically where as it is expected that control, data, and presentation integration technology will continue to advance at amazing rates. Even if we perfect reverse engineering technology, there are inherent high costs and risks in evolving legacy systems. Developing strategies to control these costs and risks is a key research direction for the future.

The premise that reverse engineering needs to be applied continuously throughout the lifetime of the software and that it is important to understand and potentially reconstruct the earliest design and architectural decisions has major tool design implications.

Reverse engineering has to be focused on design recovery. As we need sophisticated tool support to perform system maintenance efficiently, the recovered design has to be based on formal methods.

REFERENCES

[1]. M. Lehman., Programs, life cycles and laws of software evolution, *Proceedings of IEEE Special Issue on Software Engineering*, 68(9), 1980, 1060–1076.

[2]. V.R. Basili and H.D. Mills., Understanding and Documenting Programs. *IEEE Transactions on Software Engineering,* 8(3), 1982, 270-283.

[3]. T. Guimaraes., Managing Application Program Maintenance Expenditures. *Communications of the ACM,* 26(10), 1983, 739-746.

[4]. J. Ning., *A Knowledge-based Approach to Automatic Program Analysis*. PhD thesis, Department of Computer Science, University of Illinois at Urbana, Champaign, 1989.

[5]. E. Chikofsky and J. Cross., Reverse engineering and design recovery: A taxonomy, *IEEE Software*, 7(1), 1990, 13–17.

[6]. C. Rich and L. Wills., Recognizing a Program"s Design: A Graph Parsing Approach, *IEEE Software* 7(1), 1990, 82-89.

[7]. I.D. Baxter., *Transformational Maintenance by Reuse of Design Histories*. PhD thesis, University of California, Irvine, 1990.

[8]. V. Kozaczynski, J.Q. Ning, and A. Engberts., Program Concept Recognition and Transformation, *IEEE Transactions on Software Engineering,* 18(12), 1992, 1065-1075.

[9]. I.D. Baxter., Design Maintenance Systems. *Communications of the ACM*, 35(4), 1992, 73-89.

[10]. P. Tonella and R. Fiutem and G. Antoniol and E. Merlo., Augmenting Pattern-Based Architectural Recovery with Flow Analysis: Mosaic - A Case Study, *Working Conference on Reverse Engineering,* 1996, 198-207.

[11]. T. Munakata., Knowledge discovery. *Communications of the ACM,* 42(11), 1999, 26–29.

[12]. M. Blaha., Reverse Engineering of Vendor Databases, *Working Conference on Reverse Engineering (WCRE-98),* Honolulu, Hawaii, USA, 1998, 183–190.

[13]. S. R. Tilley., Coming attractions in program understanding II: Highlights of 1997 and opportunities for 1998. Technical Report CMU/SEI-98-TR-001, *Carnegie Mellon Software Engineering Institute*, 1998.

[14]. R. Clayton, S. Rugaber, and L. Wills., The knowledge required to understand a program, *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE-98),* Honolulu, Hawaii, USA, 1998, 69–78.

[15]. J. Singer and T. Lethbridge., Studying work practices to assist tool design in software engineering, *Proceedings of the 6th International Workshop on Program Comprehension (WPC-98),* Ischia, Italy, 1998, 173– 179.

[16]. A. Blackwell., Questionable practices: The use of questionnaire in psychology of programming research, *The Psychology of Programming Interest Group Newsletter*, 1998.

[17]. R. Kazman and S. Carrie,re., Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering*, 6(2), 1999, 107–138.

[18]. K. Wong., *Reverse Engineering Notebook,* PhD thesis, Department of Computer Science, University of Victoria, 1999.

[19]. B. Boehm., *Software Engineering Economics*. (Prentice-Hall, New York,1981).

[20]. C. Rich and R.C. Waters., *The Programmer's Apprentice*. (ACM Press,1990).

[21]. H.A. Partsch., *Specification and Transformation of Programs: A Formal Approach to Software Development*. (Springe, 1990).

[22]. J. Nielsen., *Usability Engineering*. (Academic Press, New York, 1994).

[23]. A. Umar., *Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies*. (Prentice Hall, New York, 1997).

[24]. L. Bass, P. Clements, and R. Kazman., *Software Architecture in Practice*. (Addison-Wesley, 1997).

[25]. B. Shneiderman.,: *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. (Third Edition, Addison-Wesley, 1998).

[26]. S. R. Tilley., *The Canonical Activities of Reverse Engineering*. (Baltzer Science Publishers, The Netherlands. 2000).