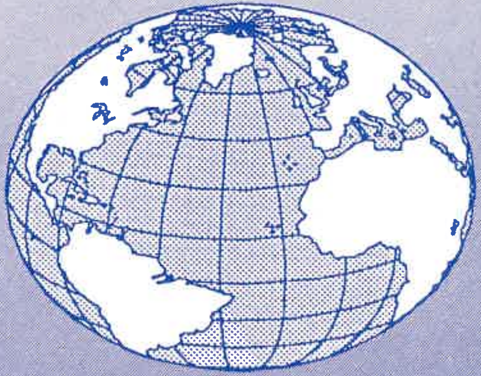
JOURNAL OF COMPUTER INFORMATION SYSTEMS

official journal of

INTERNATIONAL ASSOCIATION FOR COMPUTER INFORMATION SYSTEMS



VOLUME XXXIII
NUMBER 1
FALL 1992

JOURNAL OF COMPUTER INFORMATION SYSTEMS

Volume XXXIII, Number 1 Fall 1992

CONTENTS

ON CHALLENGES OF REVERSE ENGINEERING FOR LARGE SOFTWARE SYSTEMS

HOSSEIN SAIEDIAN and MANSOUR ZAND
University of Nebraska
Omaha, Nebraska

JAMES WELBORN
General Dynamics
Bellevue, Nebraska

INTRODUCTION

When faced with new system requirements, information systems managers have three general options.

1. modify the existing system to include the new requirements;
2. develop a new system, to satisfy the new requirements, and merge the new system with the existing system; or
3. develop a new system to completely replace the existing system.

Due to the vast amount of existing systems already in production, it is a rare case in today's software industry that new systems are built from scratch. Instead, they are built by enhancements of systems already in production. Unfortunately, current software design methodologies are focused on new systems development and few discuss the prospect of merging existing systems, or portions of an existing system, into a new system design. This process becomes even more difficult, if not impossible, when the original requirements of the existing system are unknown due to a lack of system level documentation. If this is the case, system level requirements need to be *reverse engineered* from the existing system before a decision can be made as to the extent an existing system can be re-used.

REVERSE ENGINEERING – DEFINITION

The traditional software development process is most commonly represented by a *software development life cycle*. There is a fair amount of agreement that the early stages of the software development life cycle consist of *requirements analysis, functional specification, design,* followed by *implementation* (Berzins & Gray 1985). This traditional approach is sometimes referred to as *forward engineering* (Biggerstaff 1989) and essentially involves moving from high-level abstractions and implementation-independent representations to low-level and implementation-dependent constructs.

Reverse engineering, on the other hand, is defined as the process of analyzing a subject system to identify its components and their interrelationships, and to create representations of the system in another form or at a higher level of abstraction (Chikofsky & Cross 1990). In a rare technical paper on reverse engineering, Rekoff (1985) more specifically defines reverse engineering as "the process of developing a set of specifications for a complex hardware system by an orderly examination of specimens of that system . . . without the benefit of any of the original drawings . . . for the purpose of making a clone of the original hardware system. . ." When applied to the software systems, the concept of reverse engineering would help in gaining a basic knowledge of system and its structure. Furthermore, as Chikofsky & Cross (1990) observe, while the hardware objective is to duplicate the system, the software objective would be to gain a sufficient understanding of the system to aid maintenance or support replacement and/or enhancement. The distinction between reverse engineering and re-engineering should be made clear here. Although re-engineering may involve some aspects of reverse engineering, it is defined as the examination and alteration of a subject system to reconstitute it in a new form (Chikofsky & Cross 1990).

Chikofsky and Cross divide reverse engineering into to major sub-areas, namely, *redocumentation* and *design recovery*. They define *redocumentation* as 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 intended for a human audience. During *design recovery*, the designers develop abstractions from a combination of code, existing design documentation (if available), personal experience, and general knowledge about problem and application domains. According to Biggerstaff (1989), 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. Thus it deals with a far wider range of information than found in conventional software engineering representations or code.

ISSUES IN REVERSE ENGINEERING

While these definitions are clear and concise and widely accepted, they fail to define the level of abstraction that needs to be achieved when performing reverse engineering. What information needs to be captured from the existing design, and how much detail is necessary to fully describe the requirements of the existing system? The answers to these questions are causing some controversy in the computer industry.

The ever-growing computer aided software engineering (CASE) market is becoming an integral part of the software design process. CASE vendors are rushing to automate the reverse engineering process. Many boast that their products are performing automated reverse engineering today. It is true that there are many tools on the market that adequately provide automated redocumentation and restructuring features. In the area of design recovery, however, no products fully meet Chikofsky and Cross' definition. Several approaches can abstract the structure and design of a system, but they fall short in continuously refining and adding the human dimension of fuzzy knowledge to the computer-generated model (Hanna 1990). Some vendors are using expert-system technology which utilize user-defined knowledge-based systems, but these may not adequately abstract the information either.

The problem in discussing automated reverse engineering tools is the lack of definition of an approach one would follow when performing reverse engineering. A thorough understanding of how a process is performed manually is fundamental in understanding how the same process can be performed automatically. A detailed description follows of a manual reverse engineering effort currently being performed for the United States Air Force.

A Case Study: Reverse Engineering at a U.S. Air Force Base

One of the authors has experience in working on a manual reverse engineering effort that has spanned the last two years. A software system that plays a critical role in mission planning for the United States Air Force was deemed to be in need of major repairs and enhancements. It consists of over a million lines of COBOL, PL/I, FORTRAN, and Assembler source code, some of which are fifteen years old. The system is coupled very tightly with one of the largest network databases in the world. A majority of the nearly 1,200 modules comprising the system have been modified dozens of times and the original designers have been long since replaced. Very little system level documentation exists, with no system level requirement specifications. The system is continuously subjected to a tremendous amount of change, is very difficult to maintain, and provides only a rudimentary mission planning capability, but at the same time is very accurate and highly regarded by the mission planners who use it daily.

An effort was initiated to develop a replacement system using state of the art algorithms that could greatly enhance the mission planning capabilities using operations research technology. The new system would utilize a relational database, and follow strict top-down design principles according to DoD-MIL-STD-2167A. (Department of Defense Military Standard 2167 is the Air Force standard on software design and development.) Since the new system was to be a total replacement, the old system would continue to be maintained, but no effort would be expended to further document it at the system level.

The new system took several years to develop. During this time period, the old system underwent major changes to keep in line with national policy, modernization of the force, etc. When the new system was delivered, it was unable to replace the old system. It provided some enhanced phases of mission planning, but was unable to plan an end-to-end mission.

A decision was now necessary regarding how to incorporate the new system. The following options were considered.

- incorporate the missing requirements into the new system;
- add the enhanced planning feature to the old system; or
- build a third system which included the best characteristics of both systems.

It was soon clear that none of the options could be accurately assessed, because no one knew the actual requirements of the old system. It was decided to reverse engineer the old system to construct a system requirements specification. Once this was accomplished, then a rational decision as to which option to consider could be made objectively.

Approach: Determination of Module Interfaces

Many factors went into deciding how to reverse engineer a system this large, including time and manpower constraints. It was obvious that there would not be enough time to start at the lowest level of abstraction. According to Chikofsky and Cross, reverse engineering can be performed starting from any level of abstraction or at any stage of the life cycle, but sheer volume of code dictated that the effort would begin at least one level of abstraction above the actual code.

It was determined to begin by describing how the individual modules interact with each other, with the database, and with the end-user. This approach allows a description of what the system is performing, without going into detail on how it is performing it. The "how" can be ignored for the time being since the system is currently in production and assumed to work correctly. This approach allows the individual modules to be treated as black-boxes. The actual requirements of each module can be directly related to the output that it is producing. In other words, the purpose each module serves is to produce an output. The input to the module then becomes a derived requirement, or, the data the module requires to produce the required output. (The input and output would become easier to identify if module contained I/O predicates or pre- and post assertions, but unfortunately there are no such specifications or other forms of documentations in of the existing codes.) Describing where an output is going, in the case it is going to other modules, forces a definition of the calling structure of each module. As it was mentioned earlier, these tasks are currently done manually. As it can be observed, this process is tedious and time consuming due to the lack of any formal approach or automated tools. Based on our experience, two things need to be done:

- A formal method needs to be developed to link systems requirements and reverse engineering requirements. In particular, *interconnection* or interface languages/models need to be examined to study their applicability in determining the interface between modules in a large system. We elaborate on this issue in next section.
- Automated tools that can extract information from multiple source languages need to be developed. A CASE environment that could collect information about module interactions and display it graphically would be of particular interest.

RESEARCH DIRECTIONS

There are several important research questions that need to be answered with respect to reverse engineering for large systems. The foremost question concerns level of abstraction that needs to be captured during the process. Another issue is the definition and formalization of an *interconnection* model or language that can be used to determine and specify the interface between the components of a software system. Yet another issue is the development of automated tools for reverse engineering. We elaborate on these issues in the following sections.

Determining the Level of Abstraction

According to Pressman (1987), when a modular solution to a problem is considered during forward engineering, many levels of abstraction are posed. At the highest level of abstraction, a solution is stated in broad terms using the language of the problem environment. At lower ends of abstraction, a more procedural orientation is taken. Problem-oriented terminology is then coupled with implementation-oriented terminology in an effort to state a solution. Finally, at the lowest level of abstraction, the solution is stated in a manner that can be directly implemented. Moving from one level of abstraction to a lower level of abstraction has been the classic, the oldest and the most widely used approach for software engineering and hence its generic phases, e.g., *analysis, specification, design, coding* and so forth, are well-known and widely accepted (although there is a lack of agreement as to where the exact boundaries between these phases are).

We believe that a similar generic approach to reverse engineering has to be established. Based on our experience, the main issue in reverse engineering is the unavailability of an established paradigm to determine the functionality of an existing system that has no documentation. In other words, a paradigm is needed to determine how the system and its subsets work and how they interact. We know that the goal of reverse engineering is to identify the building blocks of a software system and their interrelationship and to create a representation of that system at a higher level of abstraction. However, we are faced with quite a few questions that need precise answers:

- Where do the reverse engineers start from?
- What information needs to be captured from the system?
- How much detail is necessary?
- What "stages" should the reverse engineers go through?
- What level of abstraction needs to be achieved?
- Who are the final audiences of the captured information?

We provide some short answers for the above questions below.

It is obvious that it would not be efficient to start reverse engineering at the lowest level of abstraction, i.e., the code level. Theoretically speaking, reverse engineering can be performed starting from any level of abstraction, however sheer volume of code dictate that the effort should begin at least one level of abstraction above the actual code. We believe the process should begin by describing why the individual modules interact with each other, with the databases and with the end-users. This would allow a description of what the system is performing without going into details of how it does it. The actual requirements of each module can be directly related to the output that it produces. The input to the module then becomes a derived requirement.

The above approach would also eliminate the need to describe why the system performs what it is performing. Trying to guess why a module outputs a data item is not necessary. Someone during the life time of this module decided it needed to produce this output. It is also not necessary at this time to improve the software, or to change the software as a result of reverse engineering. Reverse engineering is a process of examination, not change or replication, and thus in and of itself does not involve changing the system.

Once the above has been accomplished for each module, coupled modules (e.g., those belonging to similar tasks) are grouped into components. This will move up one level of abstraction, allowing the component to be treated as a black box and focusing on the inputs and outputs of the component. Components can be grouped into higher-level components, and so on until the system level is achieved. Once the highest level of abstraction is obtained, the system requirements should be described in enough detail that a decision can be made as to which of the options described earlier should be chosen. The advantage of this method is that *only the modules that are identified as potential constituent parts* of the new system will become candidates for redocumentation.

Application of Interconnection Models

The purpose of interconnection models and languages is to provide a means for defining the interface between components of a large software system. The interface definition is most commonly given as structured graphs where the components of the system and their relationship are represented. The exact definition of relationships depends on the context. Most often, however, the relationships depict a *call graph* – that is, they indicate which component calls which other component(s) and what is exchanged between them. The interconnection models provide a good starting tool for reverse engineering where one can use them to formally identify and define the relationship among the modules.

Perry (Perry 1987b) defines an interconnection model (IM) as a two-tuple structure consisting of a set of *objects* and a set of *relations* which represent the interconnections that exist among the objects:

IM = ({objects},{relations})

An IM may be employed to construct a graphical representation of a system where objects are depicted as nodes of the graph while the relations are shown as edges (Perry 1987a, Yau & Tsai 1987). Interconnection languages are based on four different models:

- Unit Interconnection Model (Perry 1987a),
- Syntactic Interconnection Model (Perry 1987a),
- Semantic Interconnection Model (Perry 1987a), and
- Module/Template Interconnection Model (M/TIM) (Zand 1990).

We are currently studying and examining the above models to evaluate their potential applications in reverse engineering.

Need for Automated Tools

For the reverse engineering process to contribute most effectively in capturing and recording information about an existing software system, automated support is desirable. However, our experience in reverse engineering has taught us that every old system that is to be reverse engineered will have a set of unique problems associated with it which makes that system distinguished from others. This implies that the paradigm of using off-the-shelf generic products is no longer appropriate when attempting to reverse engineer a system that is unique. Instead, specialized tools are needed via which the operations of the system in question are analyzed and its functionality assessed. Currently, the solutions continue to fall back on a system in which the user is allowed to enter his/her own *rules* for the system regarding what information is to be extracted. Specialized expert systems may play a central role in this regard. There are a number of CASE tools (e.g., *Excellerator, KnowledgeWare*, and IBM's new product *AD/Cycle*), however, none are able to extract information from multiple source languages, nor are they able to capture information and display it in terms of dataflow diagrams or structure charts.

SUMMARY

The work reported here is only a start on a large set of issues. We discussed many of the problems that one may face while reverse engineering a large software system. One adage that can be applied is that while simple systems are easier to understand, complex systems are much harder to understand and thus imply a greater cost. However, one has to consider the benefit of reverse engineering. In addition to increasing the overall comprehensibility of a system for maintenance and new development, it can assist in detecting and removing side effects, it will facilitate reuse, and it will synthesize higher abstractions.

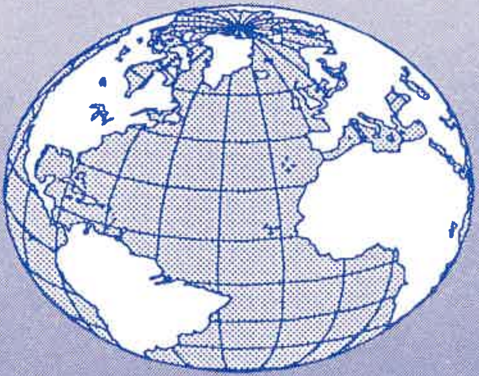
REFERENCES

1. Berzins, V. and M. Gray. "Analysis and Design in MSG.84: Formalizing Functional Specifications," **IEEE Transactions on Software Engineering SE-11,** 8, 1985, pp. 657-670.
2. Biggerstaff, T. "Design Recovery for Maintenance and Reuse," **IEEE Computer,** 22:7, 1989, pp. 34-49.
3. Chikofsky, E. and J. Cross. "Reverse Engineering and Design Recovery: A Taxonomy," **IEEE Software,** 23:1, 1990, pp. 13-17.
4. Hanna, M. "Defining the 'R' Words for Automated Maintenance: Reverse Engineering Restructuring, Reusability Packages Help MIS bring Code Under Control," **Software Magazine,** 1990, pp. 41-46.
5. Perry, D. "Software Interconnection Models," **Proc. 9th International Conf. on Software Engineering,** IEEE, 1987a, pp. 61-69.
6. Perry, D. "Version Control in the Inscape Environment," **Proc. 9th International Conf. on Software Engineering,** IEEE, 1987b, pp. 142-149.
7. Pressman, R. **Software Engineering: A Practitioner's Approach,** 2nd ed., McGraw-Hill, 1987.
8. Rekoff, M. "On Reverse Engineering," **IEEE Transactions on Systems, Man, and Cybernetics,** March-April 1985, pp. 244-252.
9. Yau, S. and J. Tsai. "Knowledge Representation of Software Component Interconnection for Large-Scale Software Modifications," **IEEE Transactions on Software Engineering SE-13,** 3, 1987, pp. 355-361.
10. Zand, M. "ROPCO: An Environment for Micro-incremental Reuse," PhD thesis, Oklahoma State University, Stillwater, Oklahoma, 1990.

JOURNAL OF COMPUTER INFORMATION SYSTEMS

official journal of

INTERNATIONAL ASSOCIATION FOR COMPUTER INFORMATION SYSTEMS



VOLUME XXXIII
NUMBER 1
FALL 1992