TRANSFORMATION FRAMEWORK FOR LEGACY SOFTWARE MIGRATION

N. I. Yusop, K. R. Ku-Mahamud, A. T. Othman

School of Information Technology,
Universiti Utara Malaysia, 06010 UUM Sintok, Kedah Darul Aman, Malaysia
email: noriadah@uum.edu.my, ruhana@uum.edu.my, abutalib@uum.edu.my

ABSTRACT

A legacy system is a system that is built using old technologies, but it is still beneficial to an organization. However, today it is facing a great challenge to meet the demands of current applications. Organizations are coming under great pressure to decide on the fate of the legacy system as they try to cope with the changing scenario. Among the alternatives offered are: discarding the legacy system and replacing it with a totally new system, allowing the system to deteriorate until the organization is out of business, redeveloping the system, or reengineering the system. Reengineering seems to offer the best solution to the challenge. This paper focuses on transformation, which is one of the reengineering technologies for migrating a legacy system toward an evolvable system. A few legacy software transformation approaches are discussed. A framework for legacy software migration that combines the strengths of each of the described approaches is proposed.

Key words : Program transformation, Software migration, Legacy system

1.0 INTRODUCTION

Computers and application software are ubiquitously used in many organizations to help them in their daily business operations. Today's fast changing business and technological environments have become a great challenge to most organizations as they strive to maintain their competitive advantage. Application software in today's environment, such as electronic commerce and groupware, demands better productivity and quality of services delivered by the system. According to Wu et al. (1997), the current software may not be able to deliver the required services expected by the

hosting organizations in today's highly competitive business environment. Businesses still run on obsolete hardware which is not only slow, but also expensive to maintain. This state of affairs will sure lead to lower productivity. In addition, the lack of documentation, and a general lack of understanding of the internal workings of the system make tracing failures costly and time consuming, thus causing maintenance to become more expensive. Furthermore, the absence of clean interfaces hampers efforts to integrate the legacy system with other "latest" systems. The problems posed by the systems, referred to as legacy systems, have become roadblocks to the organization's progress.

Bennet et al. (1999) define a legacy system as "any computerized information system that has been in use for some time, that was built with older technologies (perhaps using a different development approach), and most importantly continues to deliver benefit to the organization". Despite the problems that such systems posed, their role should not be overlooked, for the systems have in the past facilitated the operations of the hosting organization. To cope with new challenges, the organization is offered with alternatives on what to do with the legacy system. Among the alternatives are:

1. discard the legacy system and replace it with a totally new system
2. redevelop the legacy system
3. reengineer the legacy system

The first alternative is a devastating action since the system provides the main functions needed most by the organization, and the organization may have invested so much on the system. In addition, replacing the legacy system also involves the expense of rediscovering the organization's accumulated knowledge about business rules and processes. Furthermore, it is too risky for most organizations (Yourdon, 1989).

The second alternative, redevelopment, rewrites the legacy system from scratch using a new hardware platform and modern architecture, tools, and databases (Bisbal et. al, 1999). Tilley and Smith (1995) have noted that "legacy systems embody substantial corporate knowledge that includes requirements, design decisions, and business rules." Legacy systems also have a considerable number of assets that are viable for reuse. Redevelopment however, impedes the reuse of the legacy assets, especially the source code. Lauder and Kent (2000) state that, "the code is the only repository of domain expertise, and that expertise is scattered throughout the code. This results in the lack of understanding of the underlying system requirements, and a considerable resistance of the code to safely reflect on-going business process change." The domain expertise and the corporate knowledge are difficult to recover after many years of operation, evolution, and personnel change. Hence, if the two alternatives are undesirable,

then we are left with the third option, reengineering. This approach seeks to migrate a legacy system towards an evolvable system (Tilley and Smith, 1995).

This paper concentrates on how, one of the reengineering technology, that is the transformation of legacy software, is used to migrate the legacy system towards an evolvable system. Section 2 discusses reengineering from the software perspective. Software transformation approaches and their strengths are illustrated in Section 3. Combining the strengths of the approaches described in Section 3, a framework for legacy software transformation is proposed in Section 4. Concluding remarks follow in Section 5.

2.0 REENGINEERING

There are many definitions of software reengineering (hereafter, reengineering). Among the most commonly cited is the definition by Chikofsky and Cross (1990), which says "reengineering is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of a new form." This definition entails that the original functionality of the subject system remains.

Arnold (1993), defines software reengineering to be "any activity that (1) improves one's understanding of software, and (2) prepares or improves the software itself usually for increased maintainability, reusability, or evolvability." This is very well supported by Tilley and Smith (1995), that states "reengineering is the systematic transformation of an existing system into a new form to realize quality improvements in operation, system capability, functionality, performance, or evolvability at a lower cost, schedule, or risk to the customer."

The need to reengineer may be motivated by the desire to utilize more cost effective hardware or software platforms, to reduce the cost of software maintenance, to add significant new functionality, or several other plausible reasons. To achieve these, Arnold (1993) highlights reengineering technologies that revolve around three themes. These themes are improving software, understanding software, and capturing, preserving, and extending knowledge about software. Table 1, an extension of Arnold's table on reengineering technology, summarizes the technologies associated with each theme, as well as the goal that each technology seeks to achieve for the respective themes. Although there are a number of technologies associated with each theme, they are interdependent.

Table 1: Reengineering technologies and their goals (summarized from Arnold (1993))

Reengineering themes	Associated technologies	Goals of associated technology
Improving software	• Restructuring • Redocumenting, annotating, updating documentation • Reuse engineering • Remodularization • Data reengineering • Business process reengineering • Maintainability analysis, portfolio analysis, economic analysis	• The modification of software to make it easier to understand or easier to maintain • The creation of updated, correct information about software from code or other documentation • The modification of software to make it more reusable • The changing of the module structure of a system • The improvement of system data • To make software into business • The discovering parts of a system should be reengineered
Understanding software	• Browsing • Analysis, measuring • Reverse engineering, design recovery	• To make the connection between related parts and multiple view system • A technology for understanding program properties such as complexity • The generation of new information about software, usually in different views
Capturing, preserving, extending knowledge about software	• Decomposition • Reverse engineering • Object recovery • Program understanding • Knowledge bases & transformation	• The creation of objects and their relationships out of a program • The generation of new information about software, usually in different views • To obtain objects form source code • To gain better understanding of software (manual/automated)

3.0 SOFTWARE TRANSFORMATION

In general, the transformation approaches discussed here have assumed the source code of the legacy software system as the main input of the transformation process. Most of them apply the reverse engineering, design recovery, restructuring, and re-documentation techniques. Following is the discussion on three of the approaches. These are: (i) structural documentation (Wong et. al., 1995); (ii) program transformation process (Gall and Klosch, 1994); and (iii) interface-reengineering process (Merlo et al., 1995). The first two seek to transform the whole system, while the third concentrates on the transformation of the user interface of the system.

(i) Structural re-documentation (Wong et al., 1995)

Legacy software requires a different approach to software documentation than has traditionally been the practice. The Structural re-documentation approach proposed by Wong et al., (1995) at the University of Victoria in Canada called the Rigi methodology, has been successfully applied to a number of softwares with about 120,000 lines of codes previously written in Cobol and C, and run on a Unix environment. This methodology aims at producing updated documentation from the legacy source code. It also applies the reverse engineering technique in extracting required information from the source code. The Rigi re-documentation process consists of:

(a) Rigireverse - an automatic parser that parse through source codes and store the extracted artifacts in the repository.
(b) Rigiserver - a repository that contains the extracted artifacts in graphical format.
(c) Rigiedit - an interactive, window-oriented graph editor to manipulate program representations.

The general approach of the re-documentation process used in Rigi methodology consists of two phases: structural re-documentation and subsystem composition. Structural re-documentation is the process of parsing through the source codes to extract artifacts and store them in the repository. In this phase a flat resource-flow graph of the software is produced. On the other hand, subsystem composition is a recursive process of grouping building blocks such as data types, procedures, and other components into composite subsystems, so as to generate multiple, layered hierarchies for higher level abstractions of software structure. This phase involves human pattern-recognition skill and features of language-independent subsystem-composition techniques.

(ii) Program transformation process (Gall and Klosch, 1994)

This approach transforms procedural system to objects. The primary input to the process is the procedural source code. In general, the transformation approach tries to identify potential objects in the procedural source code. It applies reverse engineering techniques for transforming procedural to object-oriented systems. The approach also makes use of intermediate system representations on different levels of abstraction such as structure charts, dataflow diagrams, entity-relationship diagram. There is also a need to split the procedural program into different kinds of objects. In this respect, interface interpretation between procedural and object-oriented parts is not necessary. During the transformation process, semantic information is acquired. This information is utilized during the syntactic source-to-source translation.

(iii) Interface-reengineering process (Merlo et al., 1995).
Before the reengineering process of user interface takes place, one must understand how the old interface was conceived and implemented, and what constraints the new interface must respect to remain compatible with the rest of the system.

Merlo et al. (1995) introduces an approach that transforms a character-based paradigm to one which is graphical object-based. The approach adopts a restructuring technique in the process. Other than obtaining the new interface, the process also produces interface specification in *a*bstract *u*ser *i*nterface *d*escription *l*anguage (AUIDL) that describes the interface behavior. Interface representations that formally describe complex interactions as direct, time-sensitive manipulation is also produced. The new object-based interface will eventually be integrated with the original legacy system to produce a system with new interface.

Table 2 illustrates the strengths of each of the discussed transformation approaches. The strengths of the approaches may be combined in such a way that a model for transforming a legacy system can be proposed.

Table 2: Strengths of transformation approaches

Transformation approach	Strength(s)
Program transformation process	1. has intermediate outcomes/assets that may be reuse in future development 2. transforms the whole software system including the database 3. produces intermediate outcomes which are OO language compatible
Structural redocumentation	1. has repositories for intermediate documentation produced, thus can be reused 2. transforms the whole system 3. has intermediate documentations that are language- and paradigm-independent 4. has outcomes which may be produced from different levels of abstraction
Interface reengineering	1. transforms only the interface part of the system 2. produces outcomes which are language- and platform-independent 3. uses the original system for its functional parts

4.0 A FRAMEWORK FOR LEGACY SOFTWARE TRANSFORMATION

Program transformation process and interface-reengineering process are targeted in producing OO language compatible outcomes, and they concentrate more on the functional parts of the legacy software. In addition, the program transformation process also transforms the data part which is originally stored as a flat file. On the other hand, structural re-documentation technique produces outcomes that are (language- or paradigm independent), and it concentrates on the interface part of the software. Combining the three techniques seems to be a promising approach towards getting a more comprehensive framework for

transforming legacy software, a framework that reengineers the functional and the interface parts separately, and eventually integrates them to create the system. Figure 1 shows such a framework.

The proposed framework, which takes legacy code as the main input, is composed of three main parts: the functional part, the user interface part, and the integration part. The functional part is made up of two processes, namely the structural re-documentation and the application modeling. This part is actually a combination of Wong's et al. (1995) structural re-documentation approach, and part of Gall and Klosch's program transformation process. As mentioned earlier, the structural re-documentation aims at obtaining updated documents from the legacy source code, which among others is the logical software structure. Application modeling, on the other hand, consumes the outcomes of the structural re-documentation to produce objects from the code called the object-oriented application model (OOAM). These objects are obtained through the reverse object-oriented application modeling (ROOAM) process as described by Gall and Klosch (1994).

The user interface part adopts the approach proposed by Merlo et al. (1995). It consists of a process called the interface reengineering. Interface reengineering involves the understanding of interface code of the original system and the extraction of the code fragments. These fragments are analyzed further to obtain interface specification. In this case, abstraction and inference processes are performed. The interface specification is represented using abstract user interface description language (AUIDL). The abstraction and inference processes also produce graphical specifications of the interface. The interface is further improved through restructuring process, and finally produced the new object-based interface. The products of the functional and user interface parts are integrated to create the system.

We hope that the proposed framework for total transformation of the legacy software is able to maintain the preference of the software engineering community, that is to keep the functional and the interface part separated. This feature will ease the subsequent maintenance activities. In addition, the language-independent and paradigm-independent intermediate outcomes of the transformation processes allows the resulting software to be developed using any programming language on any paradigm. However, this framework is only conceptual and its practicality is yet to be proven.

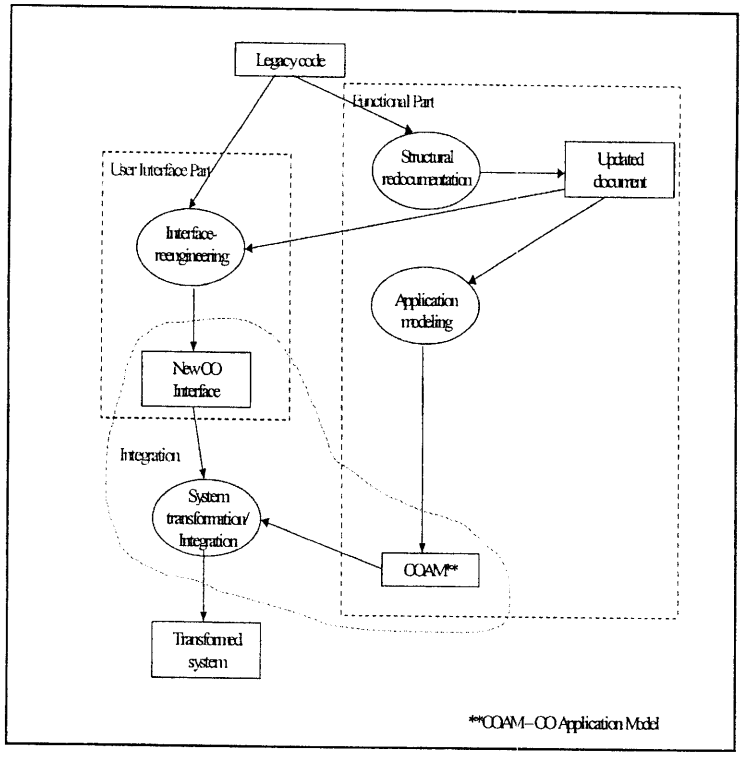

Fig. 1: A legacy software transformation framework

5.0 CONCLUSION

Software reengineering offers an approach to migrate a legacy system towards an evolvable software. A framework of legacy software transformation developed by combining the strengths of previous approaches has been proposed. The proposed framework for legacy software transformation is meant to be more comprehensive in the sense that it takes care not only of the functional, interface and the data part of the system, but also the integration of these parts to create a "new" system. In addition, the framework allows the "transformer" to keep the functional part and the interface part separated to ease the subsequent maintenance activities.

REFERENCES

Arnold, R.S. (1993). A road map guide to software reengineering technology. In R.S. Arnold (Ed.). *Software reengineering*. Tokyo: IEEE Computer Society, 3-22.

Bennet, S., McRobb, S. and Farmer, R. (1999). *Object-Oriented System Analysis and Design Using UML*. London: McGraw Hill.

Bisbal. J., Lawless, D., Wu, B. and Grimson, J. (1999) Legacy Information Systems: Issues and Directions. *IEEE Software*, 16 (5), 103-111.

Chikofsky, E. and Cross, J.H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*. 13 – 17.

Gall, H. and Klosch, R. (1994). Program Transformation to Enhance the Reuse Potential of Procedural Software. *Proceedings of ACM Symposium on Applied Computing*. pp. 99-104.

Lauder, A. and Kent, S. (2000). Legacy system anti-patterns and a pattern-oriented migration response. In P. Henderson (Ed.). *System engineering for business process change*. Kent: Springer-Verlag.

Merlo, E., Gagne, P.Y., Girard, J.F., Kontogiannis, K. and Panangaden, P. (1995). Reengineering User Interface. *IEEE Software*. 65 – 73.

Tilley, S.R. and Smith, D. (1995). *Perspectives on Legacy System Reengineering*. (Draft version 3.0). Available from: http:// www. Sei.cmu.edu/ Isysree.pdf. Accessed: 15 Sept 2001. Software Engineering Institute, Carnegie Mellon University.

Wong, K., Tilley, S.R., Muller, H.A. and Storey, M.A.D. (1995). Structural Redocumentation: A Case Study. *IEEE Software*. 47 – 54.

Wu, B., Lawless, D., Bisbal, J. and Richardson, R. (1997). The Butterfly Methodology: A Gateway-Free Approach for Migrating Legacy Information Systems. In *Proceedings of the 3rd IEEE Conference on Engineering of Complex Computer Systems (ICECCS97),*: IEEE Computer Society. pp. 200 – 205.

Yourdon, E. (1989). RE-3: Reeingineering, Restructuring, Reverse Engineering, Part 2. American Programmer, 2 (10), 3-10.

Information for Authors

A. Submission of Manuscript

1. This journal will be published biannually. Authors must submit four copies of the full paper (one original and three duplicate sets), to the Chief Editor, Journal of ICT, School of Information Technology, Universiti Utara Malaysia, 06010 Sintok, Kedah, Malaysia, for consideration.
2. All contributions will be evaluated by one or more independent referee(s) on accuracy, originality, quality and relevance. A paper may be accepted, returned for revision or declined.
3. Upon acceptance of the full paper, the author is required to submit a copy of the manuscript in digital format to the Chief Editor which must be created using Microsoft Word.
4. Papers should be original and must not be or have been submitted simultaneously to any other journals. Authors are solely responsible for factual accuracy of their papers.
5. A complimentary copy of the journal will be distributed to the author whose paper(s) are published.

B. Style of Manuscript

1. Manuscripts must be in English and prepared on A4 size white paper with 3 cm, 2.5 cm, 2 cm, 1.5 cm margins from top, bottom, left and right respectively.
2. Centred at top of the first page should be the complete title of the manuscript, author(s), affiliation(s), mailing and email address(es). This is followed by abstracts under the heading **ABSTRACT** (centred and bold), not exceeding 15 lines, keywords under the heading **Key words**: (not more than eight keywords) and followed by the text. The text should be typed in double space, using a font similar to the one used in this text (Times, 10 points). Paragraphs should be separated by double spacing.
3. Figures and photos should be labelled with "Fig." and tables with "Table" and should be numbered sequentially, for example, Fig. 1, Fig. 2 and so on. The figure numbers and titles should be placed below the figures, and the table numbers and titles should be placed on top of the tables. The first letter of the title should be placed in the middle of the page between the left and right margins. Tables, illustrations and the corresponding text should be placed on the same page as far as possible. Otherwise they may be placed on the immediate following page. Its size should be smaller than the type area.
4. Each manuscript should not exceed 20 pages including illustrations and tables.
5. Sections and subsections should be numbered and titled as 1.0, 2.0, etc. and 1.1, 1.2, 2.1, 2.2, 2.2.1, etc. Capital letters should be used for the section titles. For subsections, the first letter of each word should be in capital letter and followed by small letters.
6. The Editors reserve the right to edit/format the manuscript to maintain a consistent style.