CHAPTER 1

INTRODUCTION

1.1 PREAMBLE

Historically, measuring the evolution ability of the software has been performed rather unsatisfactorily. It is now widely accepted that software systems continue to evolve in response to environmental changes. The ability to evolve such systems is effectively paramount to their remaining usefulness. There are currently no generally accepted measures for the evolution of Information System (IS). Over time, an active software system keeps growing in size and complexity gets increasingly hard to understand and maintain. It is difficult to predict the impact of a process change on the basis of long term evolution of a system.

Legacy systems are old computer systems that are still being used because they function based on the user requirements. A software program in the legacy system is expected to perform for many years and also undergo frequent updates and changes. Constant changes to such legacy software (Harry Sneed 2006) systems are always expected to face some quality issues. Major researchers are still working with systems that were designed and implemented long ago for various organizations. These systems depend upon some older technologies and their intended platforms would not provide easy support to the users. Moreover it is very difficult to add more functionalities in a legacy system and also too complex to recover from
situations like hardware failure, disaster etc. The legacy system always needs continuous maintenance which requires a significant cost.

A legacy system is any application based on older technologies and hardware that continues to provide the core services to an organization. These systems are often hard to maintain, improve and expand. Legacy transformation is about retaining and extending the value of the legacy investments through migration to new platforms. Migrate to new platform also provides an opportunity to align applications with the current and future needs through the addition of functionalities and also through application restructuring. The reasons of reengineering the legacy systems into components are listed as follows:

- For software reusability, customers want the reusable business rules to be shared by other applications so that they do not have to duplicate the same business rules across the entire applications (Jesus Bisbal et al 1999).

- Due to the availability of new technologies such as middleware and web services, customers want to pull their reusable business rules from their legacy code into components so as to integrate them with the business applications seamlessly (Spencer Rugaber and Jim White 1998).

1.2 CHALLENGES IN REENGINEERING

Reengineering transformations involve the reabstraction of an existing implementation. However, reabstraction techniques frequently suffer from the noise induced in the implementation by maintenance reports (Joiner and Tsai 1998). When reengineering the legacy systems, however the
existing Software Performance Engineering (SPE) methods face the following challenges that need to be effectively addressed:

- To obtain realistic inputs to the model and validate assumptions when the system already operates in production. Measuring model parameters in the existing system are essential to achieve accurate predictions. Experience-based estimates of parameter values as assumed in the existing SPE methods cannot meet this requirement.

- To characterize the real system load in a source-sharing production environment. For instance, the target system performance can be adversely affected by the database sharing and hardware usage of applications that are outside the targeted scope of performance analysis.

- To cater for variations of model parameters due to the changes of load conditions. For example, the Lines of Code (LOC) increase in size, the service demands for hardware resources will likely increase due to the large volume of data to be retrieved.

A legacy system is aged when its quality has decayed. In order to verify the aging of a system, the following parameters were identified.

- The system includes many components which are not necessary to carry out the business functions (Neill and Gill 2003).

- The knowledge of the application domain and its evolution is spread over the programs and can no longer be derived from the documentation.
- The names of the components that have little lexical meaning or in any case inconsistent with the meaning of the components they identify.
- The programs and their components are linked by an extensive network of data or control flows.
- The system’s architecture consists of several different solutions that can no longer be distinguished. Even though the software started out with a high quality basic architecture, the superimposition of these other hacked solutions during maintenance has damaged its quality.

1.3 GENESIS OF THE THESIS

The motivational support of this thesis is based on the trends in system reengineering. This thesis also addresses the role of reengineering in the field of software migration especially in IS evolution process. After the proper evolution of the IS, there may be a situation that the existing system should be upgraded or to be migrated to the new environment. In this case, reengineering is playing a vital role in the software system migration.

1.3.1 Trends in System Evolution

The prime focus of the IS evolution process is the aggrandize productivity and quality of the various components of the system. The evolution process is always challenging as it leads to an increase in overall complexity especially when the system changes are mostly confined to part of it. Software evolution most often implies the integration of legacy components with newly developed ones leading to mixed architectures (Philippe Thiran et al 2005). The evolution process focuses on the increased productivity and quality of such IS. It is also considered as an effective
methodology when the functionalities and requirements are satisfied for the legacy systems (Antonini et al 1994). Due to the availability of the new technologies, there is a certain need to reuse the application of their legacy code into components. Hence the components can be integrated into the new applications seamlessly. With newer technologies emerging, conversion of the legacy system to accommodate newer generation hardware and operating systems is essential task in software evolution.

A popular solution of the legacy system evolution problems is moving the applications and the databases to new platforms and technologies. It is normally an expensive and complex process, but it greatly increases the control and evolution to meet the future requirements. The incremental strategy allows the migration projects to be more controllable and predictable in terms of deadline. The difficulty lies in the documentation of the migration steps or procedures.

Modern system evolution can make the legacy system easier and improve its reusability. Incorporation of new design and implementation techniques can create a more modular and compassable system, accommodating the modifications more effectively. The strategy of the system evolution is to build the domain architecture and reusable software components that reduce the dependence on the domain expert. The challenge is to take the existing system and instill good software development methods and properties, generating a new target system that maintains the required functionality while applying the new technologies. The purpose of the IS evolution is to provide change, propagation and impact analysis in the older systems.

Today most Object Oriented (OO) software systems are developed using an evolutionary process model. Therefore, understanding the phases
that the system’s logical design has gone through and the style of their evolution can provide valuable insights in support of consistently maintaining and evolving the system without compromising the integrity and stability of its architecture. In many cases, these systems are the only repository of business rules, historical data and other valuable information. Accessing this information is of vital importance to new open environments and to system integration in general.

A wrapper is often used to extend the list of components of the existing data systems by facilitating their integration into modern distributed systems (Phillipe Thiran and Jean-Luc 2006). System evolution is partly the result of inadequate design capture in the software development and partly the result of inexorable advance of software science. An inadequate design capture result in code is always difficult to support, maintain and reuse. The continuous advancement in software science can result in even today’s best code seeming archaic tomorrow. Both situations demand the tools and techniques of Reverse Engineering. Few systems have the same maintainers as they had Engineers and few designers possess the ability to remember for a long time about the multitude of design choices made during a software development effort.

In the IS evolution life cycle, the identification of errors or bugs play an important role as the final product should be bug free when it is delivered to the end users (Cohen et al 2003). To achieve this, program slicing is used as one of the method to identify these kinds of errors. Component mining is the extraction of executable slice from source code, which satisfies a specific use case or a set of use cases and provides a standard component interface for its use. Program slicing has a wider range of applications that
include debugging, testing, maintenance, code understanding, complex measurement, security etc., (Manoj Parameswaran et al 2007).

1.3.2 Trends in System Reengineering

The basic reengineering tasks are the preparation for functional enhancement, improve maintainability, migration, improve reliability etc. Reengineering makes the software easier to change and improve its reusability. The ultimate aim of reengineering is to keep the old system as it is and add the new things to it that leads to the target system. The prices of the new system are fallen so much recently and the things are feasible because of wide availability of components.

Analyzing and representing the software in terms of its internal dependencies is important for a variety of Software Engineering applications. This includes the operations such as slicing and the computation of program metrics of the specific application software (Georgios Lajios et al 2008). Since OO programming can be more complex than imperative programming, some well established metrics may not be directly relevant to OO software (Bixin Li and Xiaobing Sun 2011).

OO design and programming are popular nowadays and much is being written about the benefits of using OO techniques in software development process, including the benefits for software maintenance. New software development can proceed from the ground up using the new techniques, legacy systems that were developed using an old style and more difficult to incorporate the new software development ideas. As a result, at some point in time the system design-evolution decisions, a fact that should be taken into account in all subsequent development and maintenance decisions.
Some development tasks require that the developer understands the current trend of the design. For example, when a new behavioral feature is being implemented; the developer has to understand the types of objects that need to collaborate to deliver the desired behavior in order to decide in which class the new method should be implemented. Other tasks require that the developer understands the evolution trajectory through which the system design has reached its current state. During reengineering of a legacy system component, all the requests for change that have an impact on this component must be put on hold until the component has been reengineered.

1.3.3 Reengineering in Programming Languages

Aspect-Oriented programming is an improved programming paradigm based on the other module mechanisms such as procedural programming and Object-Oriented Programming (OOP) (Bounour et al 2011). In OOP, an object implements a part of the system’s functionality. Objects interact with each other via messages to achieve the system goal. While the hierarchical modularity of OO languages is extremely useful, they are inherently unable to modularize the crosscutting concerns such as logging and synchronization. Since such concerns are implemented as an interaction of related objects. In general, the scattered code causes the following problems.

- When a specification of a cross cutting concern is changed, programmers must modify all related objects.
- Programmers cannot reuse an object independently of other objects since objects are to each other with a crosscutting concern.
Programmers are not able to reuse the implementation of a concern independently of objects. If another set of objects interacts in the same way, programmers must reimplement the concern objects.

Due to its portability to different hardware platforms, Sun’s Java programming language gains more and more significance. An important reason for the fast distribution is its usage as a development platform for the applications that are integrated into Web Pages (Rakesh Agarwal et al 2000). The growing demand for Java applications forces the programmers to implement standard algorithms and data structures for universal use. It is very useful for identifying and implementing source code changes and that its extensibility is a very important factor especially to adapt and convert various features of C++ (CPP) code to its equivalent Java code.

1.3.4 Testing in Reengineering

The objective of software testing is to uncover as many errors as possible at a minimum cost. Testing is not confined only to the detection of bugs; it also assists in the evaluation of the functional properties of the software. A piece of software can be tested to increase the confidence by exposing potential flaws in or deviations from the user’s requirements. Software testing ensures quality and the other objectives of testing are to authenticate incorrectness and succeed when an error is detected.

Fagan defines a defect as, “Any condition which causes malfunctioning or which prevents the attainment of expected or previously specified results”. Testing never proves the absence of defects; it only possibly reveals the presence of faults in the legacy system. A piece or an incorrect piece of the code is a fault and it remains undetected until an event
activates it. Logical errors occur when the code does not perform the way it is intended to perform (Xing and Stroulia 2005). The detection and elimination of the logical bug are some of the aims of testing. These errors are very difficult to track since the compiler does not provide any proper assistance for debugging.

A class is the basic unit of testing in OO software. The class is a complete unit that can be isolated and tested independently. The data structure and the behavior of the class are represented by the data and the methods defined in the class respectively. An object is an instance of the class and it has a state, defined behavior and a unique identity. The state of the object is represented by the data defined in the object.

The OO approach to software development does not rule out the need to test the software as it always needs to be tested. As a result of the different architecture of the unique features of OO software, some of the testing issues of OO software are different from the testing issues of the conventional software (Dikel et al 1997). The unique testing issues of OO software necessitate improvisation of the conventional software testing techniques and development of new testing techniques (Sue Kelly et al 1999). OO software is implemented by invoking the methods in an order that implement its functionality. This method of the class tends to be less complex and small in size, though large methods may also exist.

In reengineering, it is a tough task to make the changes so that the changes made will not introduce any bugs in the system. In addition, writing tests in one pattern to support understanding, encode the assumption in a test and check whether the test fails or succeeds. Testing language should query for the objects that are not in the scope of each other i.e. Objects that are not necessarily in direct relation but that can be found in the execution stack. Here
is a non-exhaustive list of actions that a tester would like to be able to use to express the tests.

- Identification of a message based on its same, its sender, its receiver or its arguments
- Identification of object creation
- Identification of specific message sequences within complex interactions
- Identification of messages inclusions and certain messages are not sent for not received by an object
- Access to the state of an object at a given point in time such as before and after an invocation
- Detection of stage changes
- Observation of the history of an object as it is created and passed around as an argument or serves as sender or receiver
- Access the state of an object in the recursive state of another object
- Access whether a reference between two objects exists is established or detached.

Black box or functional testing ignores the code literally. Test cases are designed based on the specification document and rely on all the functional activities of the program. The functionality of each module is tested with regard to its specifications like requirements and its context or events. In general, every combination of input and output would require an inordinate number of test cases. Therefore exhaustive black box testing is
impossible in a situation like this. The art of testing is to design a small, manageable set of test cases so as to maximize the chances of detecting a fault while minimizing the redundancy amongst the test cases. Various inputs are exercised and the outputs are compared against the specifications to validate the correctness. All the test cases are derived from the specification document of the software.

1.4 LIMITATIONS OF CURRENT REENGINEERING APPROACHES

The reengineering approaches include user interface compatibility, transition support, system interface compatibility and training etc. The main limitations of current approaches include and may not be limited to the following.

- Various third-party tools are available that translate the source program from one programming language to another programming language. This language translation tool does not provide the design models of the application.

- Programs produced by the language translator are poorly structured, contain cryptic variable names and use non-optimal data structures. Maintainability of such programs is a quite difficult task.

- When a change is made to software program most of the time the entire program is reengineered. This problem can be solved by identifying the core of the system or parts of the system that tend to remain stable.

- Execution effectiveness and artifacts consistency is not repeatable as it depends on the development team’s skill set and the application knowledge the team possesses.
- Mapping platform services from the source to the target environment has always been a manual task as this is difficult to automate.

- As new technology arrives at a faster rate, older technologies do not disappear, but are concentrated in the legacy software. As a consequence, the evolution problem is augmented by a heterogeneous problem.

- As part of system integration, middleware technologies are aimed to hide the heterogeneity of hardware systems. The proliferation of middleware implementations has however created another level of heterogeneity, since none of them may dominate the market. There is no dominant middleware implementation but some standards which are adopted by all those implementations.

A way to eradicate the above problems in the legacy system is reengineering. The reengineering is a method of analysis and gaining comprehensive knowledge from the existing system so as to rebuild the source code according to the new system requirements. In order to capture the effect of the existing system for further changes to it, the concept of inertia is evolved. It is an indirect measure of evolve ability which has two dimensions: change in the size of the system as it is evolved and changed to the structure and code of the system and growth in system size over time may make the system correspondingly more difficult to maintain. However, size alone does not capture the full richness of inertia as a concept, since the two systems of equal size may not be equally evolvable.

During reengineering, it is crucial to write tests to make sure that the changes made do not introduce bugs in the system. Furthermore, writing
tests in one pattern to support software understanding encode the assumption in a test and check whether the test fails or succeeds. But even though it is known that tests are important during reengineering, it is quite difficult to use them in practice. OO legacy system behavior is distributed over many interacting objects, making it necessary to test for complex collaboration scenarios. In summary, writing tests for legacy systems is complicated because

1. The system needs to be brought in a certain state
2. Complicated scenarios need to be expressed

Over time, an active software system keeps growing in size and complexity and gets increasingly hard to understand and maintain. Software Engineers use code metrics as one of the mechanisms for quantitatively measuring the software qualities thereby helping them for better understanding and maintenance of the software systems. Another potentially more precise source of evolutionary information can be documented, either at source-code level or at the change-log level of the version-management system and for the development of the software system. In general, such documentation is always sparse and inconsistent as time progresses.

1.5 OBJECTIVES OF THE THESIS

The fundamental objectives of the present study are threefold.

- This thesis attempts to address a few of the above indicated problems in reengineering and legacy systems. In particular, the implementation of high performance process of migration with simplicity in functionalities.
With strong emphasis on legacy system characteristics, this thesis investigates the important attributes and characteristics of legacy system.

Develops high performance migration methods and provides code conversion selection guidelines.

1.6 THESIS CONTRIBUTION

Based on the issues discussed above, the contribution for this work formulated as follows:

i. To analyze the legacy system and to detect the flaws in the legacy source code that leads to inaccurate conversion using SRASG.

ii. To evaluate the legacy system for the purpose of improving the quality of the code during reengineering and to detect the defects such as interferences automatically which go unnoticed by the compiler through ASE Model.

iii. To migrate the legacy system to the modern system without the loss in the legacy system constraints and user requirements by RL2M system.

iv. To reduce the defect rate in the converted code and to succeed when an error is detected by incorporating OTC along with RL2M.

v. In general, to relieve the users from the burden of challenging tasks in reengineering and migration.
1.7 ORGANIZATION OF THE THESIS

Based on the objectives stated, the work has been carried out and presented as given below:

The thesis contains six chapters.

Chapter one discusses the prelude and the genesis of the thesis with detailed presentation of reengineering and legacy systems technology

Chapter two elaborates the detailed survey of the literature and about the reengineering technology.

The third chapter, discusses about the implementation of a novel slicing method to break down the given legacy system into smaller modules by proposing a migration method. To overcome the limitations in automated source code conversion.

The fourth chapter begins with the evaluation of legacy system. Automatic Specification Evaluator (ASE) method evaluates the legacy system specification in the lower level by introducing some granules in the new system and gives some breakpoints in the new system.

The fifth chapter makes the conversion of legacy system to new modern system. To improve the efficiency and decrease the complexity, a Reengineering model has been proposed which is known as Reengineering Legacy to Modern (RL2M) system. To overcome the drawbacks on every reengineering approach, a new method has been introduced namely One Time Checker (OTC) where the bug detection and rectification is done on the target system.
The fifth chapter also integrates the different modules analyzed rigorously in the previous chapters for the complete legacy IS evolution and migration to the new system. The results thus obtained are summarized for all the individual modules.

The sixth chapter discusses about the conclusion of the present research work and also throws light on the scope of further extension of this work and provides an eye opener for the future scope.