Abstract: As legacy software systems have been developed and maintained by a considerable amount of investment over the past decade, the systems became potential assets in an organization. However, the structure and documentation are incomplete and have been deteriorated gradually. To make matters worse, the pressure of adjusting the legacy systems to new technology such as web services and telematics in heterogeneous and distributed environment is on the increase.

In this paper, we introduce a component extraction technique that identifies candidate functionality for reuse in the source codes with the help of candidate criterion for legacy systems written in IBM CICS/COBOL language. The criteria are composed of various metrics with respect to patterns of business logic. And then we design and implement a re-engineering prototype toolset that assists in understanding and reusing legacy systems.

Key-Words: Legacy, Enterprise JavaBean (EJB), CICS, COBOL, Component, Identification

1 Introduction

Legacy system is an important program for corporate strategy, and has been implemented with procedural language such as COBOL or PL/I for the latest several decades. However, due to the continuous amendments and additional requirements its documentation has not been successfully accomplished and maintenance costs continues to increase because of the deficiency of program experts. To meet the changing requirement of computing environments like web services, telematics etc. and minimize the cost of software maintenance, the evolution of legacy systems is inevitable. In order to evolve into a new software paradigm, the legacy system requires some kinds of automated re-engineering toolsets that support identifying a reusable functionality from existing system and packaging it into a new style component.

To find reusable function in business applications seems like to discern crucial business rules in an enterprise. Most business rules contain business policies in the company and have been maintained to adapt to new business environments for a long time. In general, it is not easy to understand and identify the business rules in source codes directly. Ning[5] extracted reusable functional components from legacy system using the program segmentation method which is composed of focusing, selection, and factoring steps. The proposed method supports user in identifying some related parts from source codes, and helps packaging them. Huang[7] divided identification processes into three steps such as domain variable identification, slicing criteria identification, and generalization program slicing step, and applied several heuristic methods on each step. In the domain variable identification step, variables are identified and then classified. In slicing criteria identification step, the criteria are defined with variable sets, program statements, and constraints as parameters for slicing process. And then business rules are extracted using these criteria and various slicing algorithms. In the other hand, Sneed[8] assumed that business rules are much related to output values and introduced an identification method using output value and program stripping.

Generally we make use of data and their flows to find some business rules in source codes. In this paper, we have identified some candidates for domain variables using variable classification in order to find data that have some relations with business rules. Based on these acquired candidate variables we have identified some types of business rules according to the weight
of the elements for each business rules. And then we construct an EJB wrapping component that wraps the identified business rule.

The rest of this paper is organized as follows: Section 2 describes the overview of our research project; Section 3 briefly presents the identification of a reusable functionality; Section 4 concludes with a summary and a discussion on remaining issues.

2 An overview of REFLECT

In order to apply a large legacy system to the state of the art technology, it is necessary to quickly understand and find reusable legacy functions of the system. REFLECT, an acronym of reengineering toolset for legacy COBOL transformation, provides an extensive visualization of analyzed source codes and component wrapping capabilities for the IBM CICS/COBOL program.

The overall structure of REFLECT is illustrated in Figure 1.

REFLECT system consists of three parts; Code Analyzer, Visualization and Modernization. It utilizes some modules of legacy system in a component-based architecture that permits flexibility and allows for scalable systems such as Enterprise JavaBean (EJB). In the first step a commercial parser generator, JavaCC, generates code analyzer which implements code analysis. The main role of Code Analysis is to examine source codes that contains JCL (Job Control Language), COBOL language, embedded CICS statements and embedded SQL statements. The results of the process are then fed into a central repository in a file system. In turn, the repository, which contains some information like abstract syntax tree (AST) as well as relationship information, can be used in subsequent parts. The information contains a set of 5 relations.

Visualization provides various points of view with respect to different levels of granularity such as system-level and program-level. Modernization is comprised of Functionality Mining and Wrapper Generation. Functionality Mining is semi-automatically achieved a series of processes such as variable classification, control flow analysis, candidate criterion and workflow identification. Because the identification of reusable functionality is a difficult process that is largely depends on the creative insight and knowledge of human in the specific domain. In the other hand, Component Wrapper Generation can be processed automatically with the help of component wrapping framework we developed.

3 Identification of reusable component

The identification process for reusable functionality is an activity in which human reasoning plays the preponderant role. Therefore, the process cannot be completely automated in the source codes only. Although human intelligence is necessary to identify reusable functionalities, the identification method we proposed minimizes the human interaction. The process is composed of four steps. In the first step, we make some groups of variables to identify functionalities using variable classification scheme as static analysis. And then we analyze control flows existed in a program or between programs, and find executable flow paths as dynamic analysis. After considering the business patterns selected through user interaction, we find the closest parts of source codes by taking advantage of some information acquired in the previous steps.

2.1 Variable Classification

Variable classification means to group program variables according to predefined classes. This process assists a program maintainer in understanding and maintaining COBOL program that has no visibility for variables. Kawabe[1,2] grouped variables through procedural and structural analysis in solving the Y2K problem. Joiner[3] classified variables into 8 types of
variables such as domain variable, program variable, local variable, global variable, input variable, output variable, constant variable, and control variable.

In order to find business rules that contain reusable functionalities it is very important to find domain variables that represent the business rules and the related variables. Business rule is a type of function describing an equation with a set of input variables \( (x) \) and output variables \( (y) \).[5][7]

\[
\sum y_j = f(\sum x_i)
\]

In general, business rules have some relationship with input and output variables, and have a possibility to contain domain variables. We can analyze data division to acquire structural information about variables used in COBOL program. In CICIS COBOL program, input and output variables are defined in the Basic Mapping Support (BMS) MAP files that have information for screen decoration. MAP file is a kind of control program that helps the programmer to code map programs without any consideration as to what physical device will be used as terminal. Therefore, it can provide the application program with some degree of data independence. We also have to consider variables transferred externally from a linkage section in the data division.

In the following subsection, we classified automatically according to the usages of variables with the help of relation information in the central repository.

3.1.1 Input variables
Input variables are related with input events such as a keyboard in a program. They can have two types of properties as follow.

1. **Input data** - those variables are transferred from keyboard events via MAP fields to programs. Those may contain local and global variables. Global variables are usually utilized to communicate between programs and able to be found in the Linkage Section. Those data could be a good candidate for describing reusable functions in the problem domains.

2. **Control data** – those variables can be used to control the flow of program and appear in the branch statements such as IF statements and EVALUATE statements.

Two types of variables can be identified by a field name that has a suffix -I in the symbolic description map or relationship facts from parsed sources.

3.1.2 Output variables
Output variables are related with output events that display result values to screen or store data in the VSAM or database system. They may not exist according to the type of business rules. Output variables can be identified by a field name that has a suffix -O in the symbolic description map or relationship facts from parsed sources.

3.1.3 Temporary variables
Temporary variables are used for holding the result of expression in a memory space if the result will otherwise be lost by the time it needs. They may yield a lot of problems in the variable classification because of those dual-positioning. It means that temporary variables can be assigned from antipodal variable groups. Therefore, when it comes to partitioning variable groups, specifying the depth of variable impact is non-trivial.

3.2 Control flow analysis
Although, as we have stated above, the variable classification may be a good start point in searching reusable functionalities, the scheme is limited as static analysis. Meanwhile, functionality is organized as static as well as dynamic information. In this paper, as a dynamic analysis, we introduce a path identification using backward slicing in the bottom-up fashion. To acquire this information, control flow of program logics at the system level and program level is required.

We categorize control flows into two types such as inter-program control flow and intra-program control flow.

3.2.1 Inter-program control flow
Inter-program control flow searches program flows among programs at the system level. In CICS/COBOL programs we can identify calling-called relations by analyzing JCL (Job Control Language) files or source files. In our case, we consider only program source files to find call relationship by calling statements such as CALL, EXEC LINK and EXEC XCTL. There can be cyclic call relation among programs in order to call back the previous program. Therefore, the relation should be removed through depth first search (DFS) algorithm in the graph theory widely known.

3.2.2 Intra-program control flow
Intra-program control flow describes reachable paths among paragraphs at the program level. The paths can
be classified two types of flows: sequential flows and branch flows. A sequential flow represents a series of flows that execute paragraphs sequentially in a source code regardless of any branch statements. A branch flow of control, on the other hand, is a kind of flows of paragraphs by branch statements and is fulfilled with structured and unstructured control commands. We should analyze these two flows of control together to find reachable paths in a program. In this paper, four steps for intra-program control flow are described.

### 3.2.2.1 Structure control relation
Structured control relation means the calling-called relation by using PERFORM or other calling statements. Unlike unstructured statement such as GOTO, paragraphs called by the structured statements are modularized easily. Paragraph call graph, as we have mentioned above, helps to view visually this structure. In Figure 2, paragraph names are represented in the vertexes and PERFORM relations are described in the edges with straight line. For example, we can see that P2 paragraph is called by P1 paragraph. From the graph we can consider some modules more than the level of paragraph using the structured control relation.

**Figure 2. Paragraph Call graph**

### 3.2.2.2 Unstructured control relation
Unstructured control relation represents the branch of paragraph by GOTO statements or others. Paragraph call graph helps to view visually this structure, too. The relation also describes in the edges with solid line. For example, P1 paragraph is branched off P3 paragraph by GOTO statements. When it comes to the statements, a cyclic flow, so called callback function, which P4 recurs to the P1 in Figure 2 should be considered as a module because of tight coupling.

### 3.2.2.3 Program control flow

In this step, flows of paragraphs can be identified through calling-called relations acquired from the former two steps at the program level. The flows are described as a call tree. The call tree depicts structural flows and branching information by IF statements or call statements using a tree structure. Figure 3 represents call tree for a sample program in an inventory management we applied.

A paragraph name using <> is shown in the topmost node. A variable name in [ ] describes control variable used in IF statement. A % means GOTO statement. If 1000-SEND-SECTION paragraph is branched by GOTO statement it would be represented with %1000-SEND-SECTION. Otherwise, it is branched by PERFORM statement. For example, 0000-SECTION-CONTROL is branched off 1000-SEND-SECTION and 2000-RECEIVE-SECTION by CA-SECT-CODE control variable using GOTO statement. In the other hand, 2410-INSERT-CHEORI is called by 2400-CHEORI-CONTROL using PERFORM statement. The graph can help user searching and understanding reachable paths in a program. The preliminary modulation according to the control relations is enabled to reduce search space for the identification of reachable paths and provides more than program understanding at the high viewpoint.

**Figure 3. Call tree graph**

### 3.3 Candidature criteria for reusable function

Candidature criteria proposed is to identify reusable function, namely business rule, which represents one step in a business transaction process such as interest computation. In general, its function tends to be formed from dynamic paths with I/O variables and performs assignment, computation, database handling, and so forth. Our criterion is for rating the codes
describing business rules. The resulting ratings represent the degree to which the reachable paths match the pattern of desired business rule. The fitness function is the following:

\[ M(P) = \sum_{i \in S} \sum_{j \in W} S_i \cdot W_j \]

Where, \( W \) – a set of code rating parameters for business rules
\( S \) – a set of program statements with the unit of a reachable path.

\( W \) represents the sensitivity of various code rating parameters adjusting to which types of business rule user want. They are classified as screen handlings, mathematics, file handling, and database handlings. The weights of mathematics consist of assignment, control, and computation function related with the results of variable classification mentioned above. On the other hand, the weights of file comprise of read, write, delete and update operations. In the case of database handling select and insert operations instead of read and write can be considered. For example, if we want to find a function related with inquiry in the bank account, we should give high values to the code rating parameters such as assignment and select operations. This is shown in Figure 4.

3.4 Workflow identification

A candidate path, which was selected through interactive candidature criterion as stated above, represents the business rule and workflow necessarily. Therefore, we should identify some flows in one program and then among programs to find some workflows that can execute it. Call tree graph acquired from intra-program control flow provides some flows in one program. Also flows among programs are easily identified using program-to-program relations we have mentioned before. Dependencies of some variables among programs can be identified by impact analysis based on external reference variables in the linkage section between programs.

4 Conclusion

In reusing functionalities in legacy system, program understanding and mining functionality, a kind of the process of identifying reusable functionalities from an existing functionality-rich software base, are an essential condition. Moreover, a candidate out of the elicited functionalities encapsulated by component wrapping can be composed to create more powerful components and integrated with a component-based system to provide more specialized services.

The development of REFLECT can be considered as an attempt to rehabilitate a piece of an old system to serve in a new system environments such as Enterprise JavaBeans or web services. REFLECT also can generates a component wrapper through visualization, mining a reusable functionality as well as component wrapping.

The method of mining reusable functionality provides a series of the process to identify a specific function through the bottom-up fashion and the sensitivity of various code-rating parameters systematically and more effectively.

References

[6] H. M. Sneed, Extracting business logic from existing COBOL programs as a basis for