A Knowledge Based Maintenance of Legacy Systems: METASOFT

Byoung Y. Lee*, Wooju Kim** and Jae K. Lee***

* Graduate School of Management
Korea Advanced Institute of Science and Technology
207-43, Cheongryang, Seoul, 130-012, Korea
Tel.: +82-2-958-3682
Fax.: +82-2-958-3604
E-mail: bylee@msd.kaist.ac.kr

** Department of Industrial Engineering
Chonbuk National University
664-14, Chonju, Chonbuk, 561-756, Korea
Tel.: +82-652-70-2332
E-mail: wjkim@moak.chonbuk.ac.kr

*** Graduate School of Management
Korea Advanced Institute of Science and Technology
207-43, Cheongryang, Seoul, 130-012, Korea
Tel.: +82-2-958-3612
Fax.: +82-2-958-3604
E-mail: jklee@msd.kaist.ac.kr

Acknowledgment : We appreciate the support of the Korea Electric Power Corporation for this research, particularly Mr. Deung Yong Cho (Director of Information Processing Center), Mr. Yoo Kwang Won (Senior Manager) and Mr. Sang Kyu Cheon (Manager).
A Knowledge Based Maintenance
of Legacy Systems: METASOFT

Abstract: The maintenance of legacy systems is a continuous problem in the field of software maintenance. To assist in the maintenance of legacy systems, we have represented the legacy systems and the maintenance requirement in a compatible manner so that the maintenance requirement can be a clue for identifying the relevant program clauses and data items in the data base. For this purpose, a maintenance component is represented by the maintenance mode (add, modify, or delete) and property and key words. The corresponding information about the program's clauses is extracted from the source code of the legacy program by the reverse engineering. The maintenance point identification algorithm - MPI algorithm- proposed in this research is theoretically complete and relatively efficient, and is proved so empirically. Using this approach, the system METASOFT has been developed for Korea Electric Power Corporation which uses the COBOL programs and IMS database. It turns out that the system is well accepted by the users.
1. Introduction

Since maintenance of information systems has become one of the most difficult and time consuming phases in the information system development life cycle, a great deal of effort has been concentrated on the enhancement of maintenance. As a result, a number of system development methodologies and CASE (Computer Aided Software Engineering) tools have been developed (Lirov, 1991). However, most of them are devised for the development and maintenance of new information systems. So it is impossible to apply them to the maintenance of existing information systems unless we completely redevelop the entire system, which, in many cases, is a formidable task.

In recent literature, the difficulties of maintaining old systems have been called the Legacy System Problem (Bennett, 1995). The legacy system problem is critical to the operation of many organizations although it was developed in the past with old fashioned technologies like COBOL and hierarchical database. A common characteristic of the legacy system is its vastness, having been changed so many times by different people, and poor documentation. We have studied a daily operational billing system in an electricity utility company. In our case, the system was developed for the first time 27 years ago, and has been revised about 400 times. At the time the system was first developed the best available technology was COBOL for programming, and IMS for database management. Since the system is composed of 900 program units, and comprehension of the current code is not easy, it is extremely expensive and risky to redevelop the system based on the current code. However, from the operational point of view, the system operates fine. So the management's concern is how to support the maintenance process of these systems. Our approach to handling the direction of the legacy system is an alternative to redeveloping the old system using the contemporary CASE tools.
To fulfill the goal of this research, we have to perform the following.

1) Represent the characteristics of legacy systems:
   
   This process requires a reverse engineering process (Sull & Kashyap, 1992; Chikofsky & Cross, 1990; Jacobson, 1991; Markosian, Newcomb, Brand, & Kitzmiier, 1994; Miller, & Strauss, 1987; Ning, Engberts, & Kozanczynski, 1994) along with some semi-automatic human intervention. Knowledge about the legacy system should be technically extractable from the legacy codes, while still useful enough for the identification of relevant program units and database to revise. In this research we have adopted the object oriented representation scheme to represent the characteristics of legacy programs and database.

2) Represent the maintenance requirement:
   
   The maintenance requirement should be simply enough to be expressed by the maintenance people; and relevant enough to be associated with characteristics of legacy systems.

3) Establish the corporate keyword dictionary:
   
   Keywords can be used to link the maintenance requirement with the programs and data items in the legacy systems. Therefore we need to establish a corporate keyword dictionary which includes the variable and data item names along with their associated program units.

4) Identification of relevant programs and data items:
   
   The most time consuming stage in the maintenance process is the identification of relevant programs and data items. Let us call this process *Maintenance Point Identification*. Currently this task is performed by human maintainers, who are apt to miss the revision of some unfamiliar program modules. Considering the characteristics
of maintenance requirements, we need a methodology which can identify the relevant program units and data items. The methodology should be able to guarantee not missing the part to be revised, with a minimum display of irrelevant parts. In this research, we develop an algorithm named MPI (Maintenance Point Identification) Algorithm to achieve the desired result.

5) Development of a tool that aids in maintenance point identification:

Since the manual detection of maintenance point is a very time consuming, a software tool which can facilitate the above four steps will be very helpful. In this research, we have developed a system - METASOFT - for the environment of COBOL programming and IMS database to be applied to Korea Electric Power Corporation (KEPCO).

The remainder of this paper is organized as follows:

Section 2 overviews the architecture of METASOFT. Section 3 describes the nature of legacy systems, and Section 4 explains the representation and process of extracting knowledge about legacy systems from the currently existing codes. Section 5 describes the structure of the key word dictionary and the acquisition process of key words. Section 6 represents the maintenance requirement and Section 7 proposes the Maintenance Point Identification (MPI) algorithm. The properties of completeness and efficiency of MPI algorithm are discussed and its performance in METASOFT is demonstrated with the real cases in KEPCO in Section 8. Section 9 summarizes the contribution and potential of this research.

2. Architecture of METASOFT
The architecture of METASOFT is depicted in Figure 1. Documentary Knowledge Extractor extracts knowledge about the program units and databases from the Source Codes of Legacy Program Units and Data Items. A Maintenance Requirement Representor receives the maintenance requirement from the maintainer. The Maintenance Point Identifier matches the maintenance requirement with the knowledge base about the program units and databases using the connecting keywords, and detects the Relevant Program Units and Data Items to the specific maintenance requirement. Then the maintainer performs the maintenance task.

3. Nature of Legacy Systems

3.1. An Illustrative Legacy System

A program unit CAM30JOJ written in COBOL is shown in Figure 2. A COBOL program consists of four divisions: identification, environment, data, and procedure divisions. In the procedure division, a program unit consists of sections (e.g. 0000-START-PROCESS, 2000-GET, and 3000-KWAN-CHECK-RTN). Each section consists of a set of sentences. The section names are usually called by PERFORM command to execute the sentences in the section. A sentence is mostly composed of a single clause, except the complex sentences like IF statements. The sentence with IF expression is composed of multiple clauses which are delimited by reserved words like IF, ELSE and COMPUTE.
As the detecting unit of maintenance point for procedure division, we attempt to identify at clause level. So let us formally define the terms.

[Definition] A *sentence* of the COBOL program codes of procedure division is a statement partitioned by a period.

[Definition] A *clause* of the COBOL program codes is a unit statement partitioned by a leading reserved word.

To identify the clauses relevant to a specific maintenance requirement, we need to characterize the *property* of each clause. With this purpose in mind, we can categorize the 8 properties of clauses by checking the existence of COBOL reserved words enlisted in Table 1.

```
Table 1 here
```

A database represented by the database management system (DBMS) IMS is illustrated in Figure 3. A database CAPMLOW1 is composed of a set of segments which correspond to *entities*. A segment is preceded by the reserved word SEGM, and defines its name (e.g. CA00ROOT) and field names which correspond to *attributes*. The structural data attribute hierarchy is depicted in Figure 4.

```
Figure 3, 4 here
```

The variables in the program units and attributes of databases are potential key words, which can match with the ones in the maintenance requirements. By using the property and key words in the program clauses and database definition, we can detect relevant ones with the
maintenance requirement.

4. Representation of Legacy Systems

To represent the characteristics of legacy program units and data items, we have adopted the object oriented representation approach. What we mean by object here is the same as the notion of frame (Minsky, 1985; Lee & Song, 1994; Sull & Kashyap, 1992). As mentioned earlier, the representation of legacy systems should be detailed enough to match with the maintenance requirement, and at the same time be extractable from the source code without sufficient documents by reverse engineering (Biggerstaff & Mitbander, 1994; Burson, Kotik & Markosian; 1990; Cleveland, 1989; Gopal & Schach, 1988; Housier & Pieszkoch, 1990; Kozaczynski, Letovski, & Ning, 1991; Markosian, Newcomb, Brand, Burson, & Kitzmiller, 1994; Biggerstaff, Mitbander, & Webster, 1994; Ning, Engbert, & Kozaczynski, 1994; Premerlani & Blaha, 1994; Aiken, Muntz & Richards, 1994). METASOFT is designed to support the automatic construction of documentary knowledge about the COBOL program units and IMS databases. Therefore, although the approach that we propose here is applicable to other programming languages and databases, the tool should be tailored to the specific language and DBMS.

4.1 Representation of Legacy Programs

The typical documentational knowledge concerning COBOL program units, section and sentences is represented as follows in METASOFT.

Figure 5 here

For instance, we can generate the following objects from the COBOL program in Figure
1. The program unit CAM30JOJ is represented by the object CAM30JOJ, and three illustrative clauses CAM30JOJ_2000-GET_021800, CAM30JOJ_2000-GET_021900 and CAM30JOJ_2000-GET_022000 are demonstrated. The title of clauses is synthesized by the system by concatenating the program unit title, section name, and line number.

```plaintext
{{ CAM30JOJ
  IS-A : PROGRAM
  RELATED-SYSTEM : CHARGING-SYSTEM
  TITLE : Charging Main Program
  LANGUAGE : COBOL
  TYPE : BATCH
  MAIN-SUB : MAIN
  VARIABLES : CA00-MST-CD CAJOJ-ED-KWH-FARE
               CA-USAGE    MST-CNT
  INPUT-DB : CAPMLLOW1
  OUTPUT-DB : CAPMLLOW1
  INPUT-FILE : MAST-F   DATE-F
  OUTPUT-FILE : DATE-F   MAST-F
  CALL : CAJ92201 CAJ94101 CAS30EXT
  FIRST-SECTION-NAME : CAM30JOJ_0000-START-PROCESS }}

{{ CAM30JOJ_2000-GET_021800
  IS-A : IF
  IN-SECTION : CAM30JOJ_2000-GET
  CONDITIONAL-VARIABLES : CA00-MST-CD
  THEN-CLauses : CAM30JOJ_2000-GET_021900
                 CAM30JOJ_2000-GET_022000
  ELSE-CLauses : }}

{{ CAM30JOJ_2000-GET_021900
  IS-A : ASSIGNMENT
  IN-SECTION : CAM30JOJ_2000-GET
  LHS-VARIABLE : CAJOJ-ED-KWH-FARE
  RHS-VARIABLES : CA-USAGE
}}

{{ CAM30JOJ_2000-GET_022000
  IS-A : ASSIGNMENT
}}

8
The structural relationship between clauses in the graphically display will also be very useful. So METASOFT automatically generates the clause structure chart as illustrated in Figure 6. Refer to the clauses 2000-GET_021800, 2000-GET_021900 and 2000-GET-022000. This means that the clause 2000-GET_021800 is an IF-clause object, and the two clauses 2000-GET_021900 and 2000-GET-022000 are THEN-clauses.

4.2 Representation of Legacy Database

The typical documentational knowledge about the database IMS is represented as follows in METASOFT.

```plaintext
{{{ DATABASE
    RELATED-SYSTEM :
    HAS-ENTITIES :
    DB-STRUCTURE :
    ACCESS-METHOD :
    USED-PROGRAM :
}}}

{{{ ENTITY
    IN-DATABASE :
    HAS-ATTRIBUTES :
    RELATED-WITH :
    KEY-FIELD :
    USED-PROGRAM :
}}}

{{{ ATTRIBUTE
    IN-ENTITY :
}}}
```
HAS-ATTRIBUTES:
KEYWORD:
LENGTH:
TYPE:
USED-PROGRAM: }}

From the illustration in Figure 3 and 4, we can extract the objects as follows.

{{ CAPMLOW1
  IS-A: DATABASE
  RELATED-SYSTEM: CHARGING-SYSTEM
  HAS-ENTITIES: CA00ROOT CA01YOKM CA02SUKM
  CA03JNG CA04YOKS CA05JNGY
  CA06SEND CA07SPCL CA08DAYS
  CA09BRNC CA0APOWR CA0BGARO
  USED-PROGRAM: CA68XMRR CAM30JOJ CAM30KWH
  CAM30MKE CAM45CH1 CAU11JOJ
  CAJ94101 CAM52U04
  DB-STRUCTURE: HIDAM
  ACCESS-METHOD: VSAM
}}

{{ CA00ROOT
  IS-A: ENTITY
  IN-DATABASE: CAPMLOW1
  RELATED-WITH: (ONE-TO-MANY CA01YOKM)
  KEY-FIELD: CA0KROOT
  USED-PROGRAM: CA68XMRR CAM30JOJ CAM30KWH
  CAU11JOJ CAJ94101
  HAS-ATTRIBUTES: CA00-ED-KWH-FARE
}}

{{ CA00-ED-KWH-FARE
  IS-A: ATTRIBUTE
  IN-ENTITY: CA00ROOT
  KEYWORD: Monthly-Fare
  LENGTH: 10
  TYPE: N
  USED-PROGRAM: CAM30JOJ CAM30KWH
  CAU11JOJ CAJ94101
}}
Note the attribute CA00-ED-KWH-FARE belongs to the entity CA00ROOT, and has the key word Monthly-Fare. All values, except the key word, can be automatically generated from the source code.

5. Key Words Dictionary

Key words that we would like to keep in the dictionary are the variables in the programs and the attribute's names in the entity of database. The attribute names can be directly extracted from the database description file. However, automatic extraction of key words from the variables in legacy programs is almost impossible because the local variables are arbitrarily named. Therefore all we can prepare in advance is a potential list of key words by consulting the corporate regulations (Moulin & Rousseau, 1992), reports and screens generated from the system. In this circumstance, reverse engineering can generate the variable names from the source code so that the key word manager, who is familiar with the programs, can easily link the variables with the key words in the dictionary. This linkage information can provide a link to the key words in a specific maintenance requirement.

The key words in the dictionary may be used for naming the variables of new programs so that the variables can be easily understood by other programmers, and the linkage with the dictionary can be automatically established. The clause names may also be stored in the dictionary, but this will take a vast amount of preparation. So it is a matter of revision frequency whether or not we would generate the linkage between the key words and clauses in advance. Since the maintenance tasks do not occur on a daily basis, the strategy we have taken is to search the relevant clauses at the time of use, starting with the identified program units. In a case where the dictionary does not possess the variable names generated from the program, such variable names should be inserted into the dictionary.
The useful relationships of key words that we need to keep are synonyms, parent and child relationships along with the program units and data attributes in which those words have appeared. For example, a key word *Monthly-Fare* has a parent *Fare*, and a child *Monthly-Over-Fare*. The word *Monthly-Fare* exists in the entity CA00ROOT of database CAPMLOW1 with the data attribute name of CA00-ED-KWH-FARE (see the line 3000 in Figure 4), and also exists in the program unit CAM30JOJ with the local variable name CAJOJ-ED-KWH-FARE (see the line 21900 in Figure 2).

{{
  Monthly-Fare
  IS-A : KEYWORD
  EXISTING-DATABASE : (CAPMLOW1 CA00ROOT CA00-ED-KWH-FARE)
  EXISTING-PROGRAM : (CAM30JOJ CAJOJ-ED-KWH-FARE)
  PARENT : Fare
  CHILD : Monthly-Over-Fare
  SYNONYM : }}

The current representation of the legacy system and key word dictionary has the following implications.

1) If the key word in the maintenance requirement exists in the dictionary, we can directly retrieve the program unit and data attributes.

2) However, to detect the clauses to be maintained, the system METASOFT has to search around the clause objects that belongs to the program unit.

3) If the key word in the maintenance requirement is a parent (or above) of the variables in the program, the scope of program units that will be retrieved via the dictionary will become wider than necessity. So it is important to request the variables at the right level.

4) Even though the maintenance requirement is the revision of a value about a variable,
METASOFT will retrieve out all clauses that include the variable with any values. This can be a reason of potential over-retrieval.

6. Representation of Maintenance Requirement

Now let us represent the maintenance requirement to be natural enough but compatible with the representation of legacy systems. This implies that the maintenance requirement should include the key words and property of clauses.

Although there are 8 properties of clauses in the COBOL programs, the properties of maintenance requirement can be classified into 4 categories because the maintainer cannot tell the difference of the following properties in advance.

1) A maintainer may know that a requirement is about the CONDITIONAL statement, but cannot tell whether it is programmed by IF statement or LOOP (this means to use a PERFORM command in COBOL program) statement.

2) A maintainer may know a requirement is about the DATA-INTERFACE, but cannot tell whether the data is stored in DATABASE or FILE.

3) A maintainer cannot tell if a requirement is calling another program.

4) A maintainer cannot know the section name in advance.

Since we need an additional property - its relevance to database, we have 5 maintenance properties in total. Another feature of maintenance is whether the maintenance requires the addition of new sentences, or deletion or modification of existing sentences. Let us call this feature maintenance mode. Naturally, if the maintenance mode is an addition of clauses, we will not hustle to find nonexistent clauses to modify. Instead, we may need to show the structural position of clauses that include the required key words.

Considering the above factors, we have designed the syntax of maintenance tasks as
depicted in Figure 7. A maintenance task can be partitioned into a set of maintenance requirements, and a maintenance requirement is composed of requirement components using **IF**, **AND**, and **THEN** operators. The requirement component which corresponds to the clause in a program consists of maintenance mode, property of clause, and relevant key words.

For example, suppose the maintenance task MR190.05-2754.

If the contract category is for industrial usage, and the consumed KiloWatt Hours for the month is greater than 450, then charge an extra 50% on the amount used over 450.

For this example, the maintenance task has one requirement. The requirement is composed of three requirement components: "if contract category is for industrial usage," "if consumed KWH for the month is greater than 450KW," and "charge an extra 50% on the amount used over 450." This maintenance task can be formally represented as follows.

\[
\text{MR190.05-2754} := \text{"Extra Charge on Over-Usage"} \\
\text{"Extra Charge on Over-Usage"} := \\
\text{(IF } \text{(AND } \text{"if contract category is for industrial usage"} \\
\text{"if consumed KWH for the month is greater than 450KW")}) \\
\text{(THEN } \text{"charge an extra 50% on the amount used over 450")}
\]

where

"if contract category is for industrial usage" :=

\text{REQ (Modification, Conditional, Contract-Category)}

"if consumed KWH for the month is greater than 450KW" :=

\text{REQ (Modification, Conditional, Monthly-Consumption)}
"charge an extra 50% on the amount used over 450" :=
   REQ (Modification, Assignment, Monthly-Fare)

This requirement description can be represented in objects as follows:

{{ MR190.05-2754
   IS-A : MAINTENANCE-TASK
   HAS-REQUIREMENTS : MR190.05-2754-1
   ACCEPTED-DEPT : IS-DEPT
   MAINTENANCE-DUE-DATE : 96/5/1
   REFERENCE-DEPT :
   TITLE : Extra Charge on Over-Usage
   REQUEST-DEPT : OP-DEPT }}

{{ MR190.05-2754-1
   IS-A : REQUIREMENT
   IN-TASK : MR190.05-2754
   RELATED-SYSTEM : CHARGING-SYSTEM
   HAS-REQUIREMENT-COMPONENTS :
     (IF (AND MR190.05-2754-1-1
            MR190.05-2754-1-2) )
       (THEN MR190.05-2754-1-3)
   REQUIREMENT-DESCRIPTION :
     If the contract category is for industrial usage, and the consumed KiloWatt Hours for the month is greater than 450, then charge an extra 50% on the amount used over 450.
   RESULT : }}

{{ MR190.05-2754-1-1
   IS-A : REQUIREMENT-COMPONENT
   IN-REQUIREMENT : MR190.05-2754-1
   MAINTENANCE-MODE : Modification
   MAINTENANCE-PROPERTY : Conditional
   KEYWORD : Contract-Category }}

{{ MR190.05-2754-1-2
   IS-A : REQUIREMENT-COMPONENT
   IN-REQUIREMENT : MR190.05-2754-1
   MAINTENANCE-MODE : Modification

15
7. A Maintenance Point Identification Algorithm : MPI Algorithm

To identify the program clauses and data items that are relevant to the required maintenance, we devised an algorithm named Maintenance Point Identification (MPI) algorithm.

**MPI Algorithm**

1. Decompose the requirement task to a set of requirement components.
2. Identify the key words for each of the requirement components.
3. Identify the program units and data items that conjunctively satisfy the key words in required components.
4. For each program module identified in step 3, search the clauses which include the required key words in the clauses. Add the derived variables from the key variables while performing this step.
5. Filter out the clauses that are irrelevant to the required maintenance properties.
6. In case the maintenance requirement includes two or more components in a syntactic relationship, filter out the identified clauses that do not satisfy such a syntactic relationships.
For the example in section 6, which has 3 requirement components, we can illustrate the MPI algorithm as follows.

[Step 1] The maintenance task MR190.05-2754 is decomposed into 3 requirement components:

MR(1): *Contract category is for industrial usage*

MR(2): *Used KWH for the month is greater than 450*

MR(3): *50% additional charge on the used amount over 450*

[Step 2] The key word identified from each requirement component are

Key_Word(MR(1)) = Contract-Category

Key_Word(MR(2)) = Monthly-Consumption

Key_Word(MR(3)) = Monthly-Fare

[Step 3] The program units that are associated with the key words are the following:

Program(Contract-Category) = {CA0IWR99 CA123LST ...
CAU11RWX CAU99P48}

Program(Monthly-Consumption) = {CA31CHEK CA31SYGP ...
CAU11JOJ CAU99P48}

Program(Monthly-Fare) = {CA0KWA02 CA35XMRP ...
CAU11JOJ CAU99P48}

The number of program units for each set is 109, 59, and 27 respectively. The number of program units that conjunctively exist in all of the three cases is 23, which are {CA35XMRP CA64MYRP ... CAU11JOJ CAU99P48}.

[Step 4] As an illustrative program unit of CAM30JOJ, there are 910 sentences and 1,353 clauses. For this program, the clauses that include one of the above key words are 170. In this program unit, there are 5 derived variables: CATEG-OVER-X, OVER-CHECK-SKIP, CAJOJ-OVER-SIGN, CAJOJ-SAVE-KWH and CAJOJ-SAVE-
KWH-FARE. So the clauses relevant to this program unit is expanded to 221. This is a sharp contrast to the number of clauses relevant only to the key variables. Figure 8 shows some clauses which are related to the key variables, and derived variables of the program CAM30JOJ. In total, out of 13,649 clauses in the 23 programs, we have identified 1,121 relevant clauses in this maintenance task.

**[Step 5]** For 1,121 relevant clauses (to key and derived variables) in the program unit CAM30JOJ, we need to check the property of clauses for each requirement component. By filtering out the clauses that do not include the property in each requirement component, the number of relevant clauses in the program unit CAM30JOJ is reduced from 221 to 38. In total, the number of relevant clauses for the task MR190.05-2754 is reduced from 1,121 to 113.

**[Step 6]** By filtering out the clauses that do not satisfy the IF-THEN relationship, the final number of relevant clauses in the program CAM30JOJ is reduced to 3. Since all the clauses in the other 22 program units do not satisfy the IF-THEN relationship, they are all filtered out. So the final number of relevant clauses for the maintenance task MR190.05-2754 is 3. In this example, the clauses that need modification are exactly identified.

8. **Properties and Performance of MPI Algorithm**

Let us elaborate the desirable properties of MPI algorithm. For this, we need to define *completeness* and *efficiency*. 
[Definition] A maintenance point identification algorithm is complete if and only if the algorithm identifies all relevant clauses in a program unit without missing any of them.

[Definition] Completeness Ratio = "number of correctly identified clauses" /"number of clauses to be revised"

[Definition] A complete maintenance point identification algorithm is more efficient if and only if the algorithm identifies less irrelevant clauses in a program unit.

[Definition] Efficiency Ratio = “number of identified clauses by an algorithm” / "actual number of clauses to be revised"

[Theorem 1] The MPI algorithm is a complete algorithm.

The MPI algorithm is complete because the clauses are identified by the properties and key words of the maintenance requirement including the derived variables.

[Theorem 2] The MPI algorithm is relatively efficient.

This issue is discussed in (3)-(4) of the last paragraph of section 5. The efficiency level of the MPI algorithm may be diminished due to the maintainer's high level representation of a maintenance task requirement and the existence of key variables that possess more than the value to be changed. This sacrifice of efficiency is intentionally devised in MPI algorithm to reduce the effort of preparing too large a knowledge base.

To show the empirical efficiency of the MPI algorithm, we have experimented with 10 real maintenance tasks collected in KEPCO as summarized in Table 2. According to this experiment, we can confirm that the completeness ratio in column (7-3) is all 1’s; which means the algorithm is complete not only theoretically but also empirically. Note that the average completeness ratio in column (5), before considering the derived variables, is 83.3%; this implies that maintainers without the capability of MPI algorithm in METASOFT may fail in
finding the relevant program units and data items by 16.7%. The average efficiency ratio of
1.6 in column (8-3) implies that the maintainers waste only 37.5% of their time in checking the
relevance. The worst efficiency in the MR-01 case is 2.44. However, this efficiency is not bad
at all if we consider the fact that human maintainers without META-SOFT currently spend 2
or 3 months testing the completeness.

==========================================

Table 2 here

==========================================

9. Conclusions

We have represented the legacy systems and maintenance requirement in a compatible
manner so that the maintenance requirement can be a clue for identifying the relevant program
clauses and data items in the data base. For this purpose, a maintenance component is
represented by the maintenance mode (add, modify, or delete) and property and key word.
The corresponding information for the program's clauses is extracted from the source code of
legacy program by reverse engineering. The maintenance point identification algorithm - MPI
algorithm- proposed in this research is a complete and relatively efficient algorithm, both
theoretically and empirically. The system METASOFT, developed for COBOL programs and
IMS databases in KEPCO, is well accepted by the users, and is now under comprehensive
testing.

The implication of this research is that this approach, along with METASOFT, can be
extended to the integrated maintenance of programs built by heterogeneous programming
languages. To use METASOFT for the development of a new system which utilizes a CASE
tool, the documentary knowledge has to be generated from the CASE tool.

We have also observed that it is essential to build a corporate key word dictionary, and
let the programmers use the standard words in naming the variables and data items so that they can be linked to the dictionary automatically. Finally, to reduce the burden of representing recurrent maintenance requirement, the case based reasoning can be applied supplementarily (Lee & Lee, 1995).
References


Laboratory in Korea Advanced Institute of Science and Technology.


Figure 1. Architecture of METASOFT
Figure 2. Source Code of Program Unit CAM30JOJ

002200 IDENTIFICATION DIVISION.
002300 PROGRAM-ID. CAM30JOJ.
002400 AUTHOR. KEPCO.
002500
002600 ENVIRONMENT DIVISION.
002700 CONFIGURATION SECTION.
002800 SOURCE-COMPUTER. IBM-4381.
002900 OBJECT-COMPUTER. IBM-4381.
003100 INPUT-OUTPUT SECTION.
003200 FILE-CONTROL.
003300 SELECT DATE-F ASSIGN TO DATEFILE.
003400 SELECT MAST-F ASSIGN TO MASTFILE
003500 ACCESS MODE IS SEQUENTIAL
003600 ORGANIZATION IS INDEXED
003700 RECORD KEY IS M-KEY.
003800
003900 DATA DIVISION.
004000 FILE SECTION.
004200 FD DATE-F
004300 LABEL RECORD STANDARD
004400 BLOCK CONTAINS 1 RECORDS
004500 RECORD CONTAINS 180 CHARACTERS.
004700 DATE-R PIC X(180).

018800
018900 PROCEDURE DIVISION.
019000 0000-START-PROCESS SECTION.
019300 OPEN I-O MAST-F.
019500 PERFORM 1000-DATE-READ.
019600 PERFORM 2000-GET.
019800 PERFORM 3000-KWAN-CHECK-RTN
019900 UNTIL END-SIGN = 'END'.
020100 PERFORM 4000-DISPLAY.
020300 CLOSE MAST-F.
020400 STOP RUN.

021800 IF CA00-MST-CD = SPACE
021900 COMPUTE CAJOJ-ED-KWH-FARE = CA-USAGE * 10
022000 ADD 1 TO MST-CNT.

037800 3000-KWAN-CHECK-RTN SECTION.

Figure 2. Source Code of Program Unit CAM30JOJ
<table>
<thead>
<tr>
<th>Property of Clause</th>
<th>COBOL Reserved word</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECTION-NAME</td>
<td>SECTION</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>MOVE, SET, ADD, DIVIDE, SUBTRACT, MULTIPLY, COMPUTE</td>
</tr>
<tr>
<td>IF</td>
<td>IF, WHEN, SEARCH</td>
</tr>
<tr>
<td>LOOP</td>
<td>PERFORM - UNTIL - BY</td>
</tr>
<tr>
<td>I/O</td>
<td>ENTRY (All options excepting DB-INTERFACE's options), ACCEPT, DISPLAY</td>
</tr>
<tr>
<td>DB-INTERFACE</td>
<td>ENTRY (Option : GU, GN, GHN, GHU, GNP, GHNP, ISRT, DLET, REPL, CHNG, PURG, GHUP, GUP)</td>
</tr>
<tr>
<td>FILE-INTERFACE</td>
<td>OPEN, READ, WRITE, REWRITE, CLOSE</td>
</tr>
<tr>
<td>CALL</td>
<td>CALL</td>
</tr>
</tbody>
</table>

Table 1. Reserved Words Associated with the Property of Clause
Figure 3. Source Code of Database CAPMLOW1
Figure 4. Source Code of Attribute Hierarchy of the Entity CA00ROOT

```c
000400 *--------------------------------------------------------------------------------------------------*
000500 *   C U S T O M E R - D B SEG - LENGTH = 160 *
000600 *   CA00ROOT (SEQ,U,19) FIELD NAME ---- CA00KROOT *
000700 *--------------------------------------------------------------------------------------------------*
000800 *
000900 05 CA00-KEY.
001100  07 CA00-UPMU-KEY.
001200   09 CA00-CYC       PIC X(002).
001300   09 CA00-OFFC      PIC X(004).
001400   09 CA00-UPMU      PIC X(003).
001500    11 CA00-BUN       PIC X(002).
001600    11 CA00-KA       PIC X(002).
001700    11 CA00-JI       PIC X(001).
001800     09 CA00-CHK-DGT PIC X(001).
001900     09 CA00-KEY-FILL PIC X(004).
002800  05 CA00-GUMCHM-DD-X.
002900    07 CA00-GUMCHM-DD PIC 9(002).
003000  05 CA00-ED-KWH-FARE PIC 9(010).
003100 *
003200  05 CA00-DONG-CODE.
003300    07 CA00-DONG-PROV.
003400     09 CA00-DONG-SIDO PIC X(002).
003500     09 CA00-DONG-GUGUN PIC X(002).
003600     07 CA00-DONG-DONGRI PIC X(002).
003700     07 CA00-DONG-LAW PIC X(003).
003800     07 CA00-DONG-CHK-DGT PIC X(001).
      ......
Figure 5. Object Oriented Representation of Program and Eight Properties of Clauses
Figure 6. Automatically Generated Structure Chart of CAM30JOJ Program Unit
<table>
<thead>
<tr>
<th><strong>&lt;Maintenance-Task&gt;</strong></th>
<th>::= &lt;Requirement&gt; {&lt;Requirement&gt;}</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;Requirement&gt;</strong></td>
<td>::= ( &lt;operator&gt; &lt;Requirement&gt; {&lt;Requirement&gt;} )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>&lt;operator&gt;</strong></td>
<td>::= IF</td>
</tr>
<tr>
<td><strong>&lt;Requirement-Component&gt;</strong></td>
<td>::= REQ ( &lt;Maintenance-Mode&gt; , &lt;Maintenance-Property&gt; , &lt;Keyword&gt; {&lt;Keyword&gt;} )</td>
</tr>
<tr>
<td><strong>&lt;Maintenance-Mode&gt;</strong></td>
<td>::= Addition</td>
</tr>
<tr>
<td><strong>&lt;Maintenance-Property&gt;</strong></td>
<td>::= IO</td>
</tr>
</tbody>
</table>

::= consists of

{x} : zero or more occurrence of x

| OR relationship |

Figure 7. BNF of Maintenance Requirement Representation Syntax
......
015600 MOVE CAJOJ-ED-CATEG (I) TO CATEG-OVER-X.
015610* CATEG-OVER-OK is a redefined variable of CATEG-OVER-OK
015700 IF NOT CATEG-OVER-OK
015800 MOVE OVER-CHK TO OVER-CHK-SKIP(I)
015900 GO TO 3210-BUHA-CHNG-EXIT.
......
015610* OVER-CHK-SKIP-ITEM is a redefined variable of OVER-CHK-SKIP
018500 IF TEMP-FARE-ILSU-TOT = W-NOW-YEOK-ILSU AND
018600 OVER-CHK-SKIP-ITEM = SPACE AND
018700 CA01-MON-KW > 3
018800 MOVE OVER-SIGN TO CAJOJ-OVER-SIGN.
......
028500 IF CAJOJ-OVER-SIGN = "*"
028600 PERFORM 3210-OVER-FARE-COMPUTE.
......

Figure 8. Clauses Related with the Component MR(1) in the Program CAM30JOJ
<table>
<thead>
<tr>
<th>(1)</th>
<th>(2)</th>
<th>(3)</th>
<th>(4)</th>
<th>(5)</th>
<th>(6)</th>
<th>(7)</th>
<th>(8)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(6-1)</td>
<td>(6-2)</td>
<td>(7-1)</td>
</tr>
<tr>
<td>MR-01</td>
<td>59</td>
<td>28,246</td>
<td>3,422</td>
<td>0.88</td>
<td>578</td>
<td>23</td>
<td>22</td>
</tr>
<tr>
<td>MR-02</td>
<td>41</td>
<td>23,004</td>
<td>2,901</td>
<td>0.81</td>
<td>586</td>
<td>17</td>
<td>36</td>
</tr>
<tr>
<td>MR-03</td>
<td>12</td>
<td>4,808</td>
<td>743</td>
<td>0.71</td>
<td>42</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>MR-04</td>
<td>23</td>
<td>13,649</td>
<td>1,121</td>
<td>0.33</td>
<td>113</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>MR-05</td>
<td>9</td>
<td>3,560</td>
<td>611</td>
<td>1.00</td>
<td>120</td>
<td>9</td>
<td>18</td>
</tr>
<tr>
<td>MR-06</td>
<td>48</td>
<td>33,086</td>
<td>7,821</td>
<td>0.79</td>
<td>748</td>
<td>30</td>
<td>164</td>
</tr>
<tr>
<td>MR-07</td>
<td>17</td>
<td>9,709</td>
<td>1,523</td>
<td>0.98</td>
<td>531</td>
<td>15</td>
<td>321</td>
</tr>
<tr>
<td>MR-08</td>
<td>27</td>
<td>14,294</td>
<td>3,112</td>
<td>1.00</td>
<td>773</td>
<td>18</td>
<td>25</td>
</tr>
<tr>
<td>MR-09</td>
<td>30</td>
<td>7,711</td>
<td>935</td>
<td>0.92</td>
<td>465</td>
<td>29</td>
<td>318</td>
</tr>
<tr>
<td>MR-10</td>
<td>35</td>
<td>23,296</td>
<td>3,441</td>
<td>0.91</td>
<td>217</td>
<td>25</td>
<td>181</td>
</tr>
<tr>
<td>Average</td>
<td></td>
<td></td>
<td></td>
<td>0.83</td>
<td></td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

(1) Requirement Task  
(2) Relevant Program Units  
(3) Total Clauses in Relevant Program Units  
(4) Relevant Clauses from the Key Words Point of View  
(5) Completeness Ratio  
(6) Number of Relevant Clauses (6-1) and Program Units (6-2) considering the Derived Variables and Filtering by the Property of Clauses.  
(7) Number of Relevant Clauses (7-1) that Satisfies the Syntactic Relationship among the Maintenance Requirement Components, Number of Program Units (7-2) that Include the Clauses, the Completeness Ratio (7-3).  
(8) Number of Clauses (8-1) and Program Units (8-2) to be Revised, and Efficiency Ratio (8-3).  

Table 2. Experimental Performance of MPI Algorithm