From COBOL to SQL through program transformation and XML

Meziane, F and Aliwa, J

From COBOL to SQL Through Program Transformation and XML

Farid Meziane
School of Computing, Science and Engineering
University of Salford
Salford M5 4WT, UK
+40 161 295 3699
f.meziane@salford.ac.uk

Jamila Aliwa
School of Computing, Science and Engineering
University of Salford
Salford M5 4WT, UK
j_aliwa@hotmail.com

ABSTRACT
The cost of maintaining legacy software systems has spiralled and their maintenance became a burden for many organisations. In this paper we present the first prototype of the LOBS-COQ system that attempts to transform COBOL legacy systems into relational database schema and produce SQL statements for data querying. The approach first transforms the COBOL source code into XML files, which are then processed to create tables and develop SQL statements. Frames are used to develop the SQL statements. COBOL statements are parsed and the information extracted is used to fill the frames' slots.

Keywords
Legacy Systems, Program Transformation, COBOL, SQL, XML.

1. INTRODUCTION
Legacy software systems were developed years ago using old programming language such as COBOL and FORTRAN. These systems are still critical to the day-to-day activities of many organisations and considered to be irreplaceable for many of them. For these reasons, and many others, these systems were maintained for years by many programmers. In the last few years the cost of maintaining these systems has spiraled as there is a shortage of skilled programmers in these old programming languages and the documentation associated with most of these systems became obsolete as it does not reflect any more their implementation after years of modifications and updating. Maintaining these systems has become a considerable burden for these organisations. In this paper we concentrate on legacy systems implemented in the COBOL (COmmon Business-Oriented Language) programming language for two main reasons. First it has been reported there are still hundreds of billions of COBOL lines code still in use today [5, 1] and second this project is developed for an organization that wishes to transform its COBOL based systems into more modern technologies.

There have been many attempts to convert legacy systems to new environments using new programming languages or new design tools and techniques. However, if done manually, these conversions are just as expensive as maintaining the old systems. In this paper, we present the first prototype of the LOBS-COQ (an anagram of COBOL and SQL omitting one L!) system that attempts to transform COBOL programs into an implementation based on relational databases. The data contained in the files used by the COBOL programs are transformed into tables and the programs transformed into SQL statements. The remaining of this paper is organised as follows: In section 2 we give an overview of some approaches and systems that attempted to reverse engineer COBOL systems. Section 3 gives a general view of our approach. In section 4 we describe how COBOL programs are transformed into XML documents and in sections 5, 6 and 7 we give a flavor of the data representation and SQL statement of the transformed COBOL programs. Finally section 8 summarises the current prototype development and the future developments of the system.

2. RELATED WORK
Reverse Engineering is the discipline that attempts to improve and maintain legacy systems. It uses many techniques that range from simple control restructuring to design and specification recovery in preparation for new forward engineering. Edward and Munro [6] developed the RECAST system that takes COBOL source code and produces specifications and system documentation in the Structured Systems Analysis and Design Method (SSADM). The aim of their approach was to recover the design of the COBOL programs, which may help in the redesign, and reimplementation of the system using new design methods and new programming languages. Another promising approach is to freeze and encapsulate the legacy system as a component in a new implementation. The functions provided by the legacy system can then progressively be taken over by the new software until the legacy software becomes redundant [9]. Other approaches have adopted data reverse engineering [4] techniques in their attempt to reengineer legacy systems. Nagaoka et al. [8] developed the DORE system (Data Oriented Re-Engineering) to produces reusable business specification in the form of entity relationship models from COBOL data description. More recent approaches have attempted to reengineer legacy systems through object oriented models. Millham [7] for example used UML to reverse engineer COBOL programs.

3. SYSTEM PRESENTATION
COBOL is a business oriented programming language and was developed in the late fifties. It has been adopted by many organisations and government agencies and became a standard in the development of business-oriented applications. COBOL was not intended to operate at the hardware level and was not meant for the development of scientific applications, as at that time, FORTRAN was the de facto language for scientific applications. With the advent of distributed applications and the Internet other problems have appeared and added to an already long list as COBOL programs are difficult to share. Given the nature of COBOL systems, business oriented and mainly concerned with data processing, we believe that the appropriate target transformation of COBOL systems is relational databases and SQL statements. There were many attempts to transform COBOL program to other programming languages. However, we believe

that COBOL programs do not have and do not require such functionality. Three basic types of programs in COBOL have been identified [9]: online transaction programs, batch processing programs and general subprograms. The current LOBS-COQ prototype is limited by the following constraints:

- It deals only with batch processing program although it can handle the program part of the online transaction programs (i.e. omitting the parts dealing with the interfaces).
- It does not attempt to migrate the existing data stored in the files used by old COBOL programs but attempts only to construct the tables that will ultimately be used to store the data at a later stage.

Based on these assumptions, the prototype presented in this paper can be summarised as shown in Figure 1. COBOL source files are used as inputs to LOBS-COQ. The COBOL-Transform module transforms these files into XML files as described in section 4. The XML files are then used as an input to the XML-Transform module. Using the knowledge of COBOL programs structures and its programming standards, tables' schema are extracted and SQL statements developed.

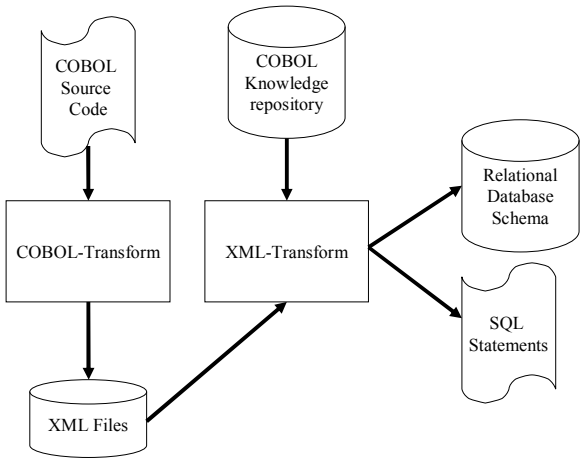

Figure 1: LOBS-COQ Architecture

4. COBOL to XML

COBOL programs are presented as a hierarchical structure. The levels of the hierarchy are Divisions, Sections, Paragraphs, Sentences and Statements respectively. There are four divisions and each division provides part of the information needed by the compiler. The divisions are defined in the following order:

- **Identification Division:** identifies the program
- **Environment Division:** describes the environment in which the program will run
- **Data Division:** describes the input/output files and data to be used by the program
- **Procedure Division:** Describes the tasks performed by the program

Each division begins with the name of the division followed by the reserved word DIVISION and a period. Divisions are divided into sections. For example the environment division is composed of the CONFIGURATION section and the INPUT-OUTPUT section. Each section begins with the name of the section and the reserved word SECTION followed by a period. Sections are divided into paragraphs and paragraphs into sentences. In the procedure division, a sentence describes an operation or series of operations the computer is to perform. Each sentence is composed of one or more statements, which are the basic instructions that are used to describe the processing that the computer is to perform. There are also standards when writing COBOL programs that include indentation at the beginning of each structure and the use of the period (.) to end statements. The rigid structure of COBOL programs and the standards used when writing the programs, made it easy to transform COBOL programs into XML documents. The process consists at tokenizing the source file and then looks for the COBOL reserved words that delimit each grouping. The reserved words are then changed to XML tags and a closing tag is added at the end of the grouping that can be identified by a period and/or the start of another grouping. An example of an XML document is given in Figure 2 which represents a transformation of a complete but very simple COBOL program. This program will be used for illustration in this paper.

5. DATA REPRESENTATION

The first step in the transformation process is the definition of the database tables and their structures. This information is provided by the Data Division of the COBOL programs. However, the Environment Division is also used to complete some details when constructing the tables. The Environment division will tell us which files are used as input and which ones are output files. When building tables, output files are not considered as these will be produced as reports by SQL statements. However, minimum information about them is kept; particularly their names, attributes and types (output file). In the program given in Figure 2, the statement **"SALES-PERSON-FILE ASSIGN TO INPUT-DEVICE"** indicates that the **"SALES-PERSON-FILE"** is an input file and the statement: **"REPORT-FILE ASSIGN TO OUTPUT-DEVICE"** states that the file **"REPORT-FILE"** is an output file. At this stage a table named **"SALES-PERSON"** is identified. Files can have two organisations, sequential or indexed. This is stated by the statements **"ORGANIZATION IS INDEXED"** or **"ORGANIZATION IS SEQUENTIAL"**.

If the file's organisation is indexed, the **"RECORD KEY IS..."** statement provides the key and the equivalent attribute extracted from the Data Division is used as the key for the database table. If the file's organisation is sequential, no key attribute is provided and the system will create a new numerical attribute which will be used as a key, in the same way as many relational database systems do when you omit to define a key field. However, the user will be alerted about this and he/she will be given the opportunity to define a key and remove the one created by the system. The table attributes are extracted exclusively from the Data Division. Records may have different levels in COBOL and the top levels, those not containing a PIC description, are ignored.

```
<?xml version="1.0" ?>
 <COBOLPROGRAM>
   <IDENTIFICATIONDIVISION>PROGRAM-ID.  TEST3.  REMARKS.  SALES
       QUOTA PROGRAM (INDEXED FILE)</IDENTIFICATIONDIVISION>
   <ENVIRONMENTDIVISION>
     <CONFIGURATIONSECTION>
       <SOURCE-COMPUTER>XYZ-1</SOURCE-COMPUTER>
       <OBJECT-COMPUTER>XYZ-1</OBJECT-COMPUTER>
     </CONFIGURATIONSECTION>
     <INPUT-OUTPUTSECTION>
       <FILE-CONTROL>
         <SELECT>SALES-PERSON-FILE ASSIGN TO INPUT-DEVICE </SELECT>
         <ORGANIZATION>ORGANIZATION IS INDEXED</ORGANIZATION>
         <ACCESS>ACCESS MODE IS SEQUENTIAL</ACCESS>
         <KEY>RECORD KEY IS SP-NUMBER</KEY>
         <SELECT>REPORT-FILE ASSIGN TO OUTPUT-DEVICE</SELECT>
       </FILE-CONTROL>
     </INPUT-OUTPUTSECTION>
   </ENVIRONMENTDIVISION>
   <DATADIVISION>
     <FILESECTION>
       <FD>FD SALES-PERSON-FILE LABEL RECORDS ARE OMITTED</FD>
       <R01>
         <R-NAME>01 SALES-PERSON-RECORD</R-NAME>
         <ATTRIBUTE>05 SP-NUMBER PIC 9999</ATTRIBUTE>
         <ATTRIBUTE>05 SP-NAME PIC X(20)</ATTRIBUTE>
         <ATTRIBUTE>05 SP-AMOUNT PIC 9(6)V99</ATTRIBUTE>
       </R01>
       <FD>FD REPORT-FILE LABEL RECORDS ARE OMITTED</FD>
       <R01>
         <R-NAME>01 REPORT-RECORD</R-NAME>
         <ATTRIBUTE>05 RT-NUMBER PIC 9999</ATTRIBUTE>
         <ATTRIBUTE>05 RT-NAME PIC X(20)</ATTRIBUTE>
         <ATTRIBUTE>05 RT-AMOUNT PIC 9999</ATTRIBUTE>
       </R01>
     </FILESECTION>
     <WORKING-STORAGSECTION>
       <W01>WS-EOF-FLAGE PIC X VALUE "N"</W01>
     </WORKING-STORAGSECTION>
   </DATADIVISION>
   <PROCEDUREDIVISION>
     <PARAGRAPH> MAIN-ROUTINE
       <STM>OPEN INPUT SALES-PERSON-FILE OUTPUT REPORT-FILE</STM>
       <STM>MOVE "N" TO WS-EOF-FLAGE</STM>
       <STM>READ SALES-PERSON-FILE AT END MOVE "Y" TO WS-EOF-FLAG</STM>
       <STM>PERFORM MAIN-LOOP UNTIL WS-EOF-FLAG IS EQUAL TO "Y"</STM>
       <STM>CLOSE SALES-PERSON-FILE</STM>
       <STM>STOP RUN</STM>
     </PARAGRAPH>
     <PARAGRAPH> MAIN-LOOP
       <STM>IF SP-AMOUNT GREATER THAN 500</STM>
       <STM>MOVE SPACES TO REPORT-RECORD</STM>
       <STM>MOVE SP-NUMBER TO RT-NUMBER</STM>
       <STM>MOVE SP-NAME TO RT-NAME</STM>
       <STM>MOVE SP-AMOUNT TO RT-AMOUNT</STM>
       <STM>WRITE REPORT-RECORD</STM>
       <STM>READ SALES-PERSON-FILE AT END MOVE "Y" TO WS-EOF-FLAG</STM>
     </PARAGRAPH>
   </PROCEDUREDIVISION>
 </COBOLPROGRAM>
```

Figure 2: COBOL to XML Document

5.1 COBOL Data Types

Data is composed of symbols and/or characters. The three basic types of characters are: numeric characters or digits (0, 1... 9), alphabetic characters or letters (a, b... z, A, B... Z), and special characters (comma, decimal point etc.). Each category has a special picture representation in COBOL, where the description of an elementary item in the Data division which is its PICTURE (PIC) clause determines its category. An alphanumeric input field is described in the Data division with a picture clause containing Xs and for numeric item COBOL uses 9s.

A numeric item can include a V (to represent a decimal point) and S to represent a sign, see Figure 2 for examples. The signed input field's picture does not count as a character position. A signed field can be either positive or negative if the S is not used the field can only be positive. The VALUE clause is used to initialise the value of working field in the WORKING-STORAGE section. This clause appears in the entry of the field's description following the PICTURE clause.

5.2 SQL Data Types

The basic data types supported by SQL are: Character String, Numeric String, Date and Time. Characters can either be of fixed or variable length. Numeric values which are defined as some types of numerical values are typically referred to as NUMBER, INTEGER, REAL, DECIMAL and FLOAT. Standard SQL supports DATETIME data types, which can be DATE, TIME, INTERVAL, and TIMESTAMP. Date and Time data type can contain a date and time portion in the format: DD-MM-YY HH:MI:SS. Note that these are ORACLE types as it is our target Database Management System.

5.3 Mapping COBOL Data Types to ORACLE Data Types

As show in the previous subsections, although we have more ways of representing data in ORACLE, the basic types are available in both and Table 1 summarises the rules used for the transformation of data types from COBOL to ORACLE.

Table 1. Data Types Transformation

Type	COBOL	ORACLE
Character String	PIC X…X	CHAR(n)
n-Character String	PIC X(n)	CHAR(n)
Variable Length String	PIC X(n) VARYING	VAR CHAR2(n)
Numeric	PIC 9…9 PIC 9(n)	NUMBER(n)
Decimal Point	PIC 9…9V9…9 PIC S9…9V9…9	DECIMAL(p,s)
Floating Point Number	COMP-1 COMP-2	FLOAT(n)
Date	PIC DD-MM-YY PIC X(n)	DD-MM-YY

At this level, once the table names and attributes are identified, we use these rules to associate types to the attributes and the system is ready to create the tables. The SQL Statement to create the table representing the file described in the Environment division and data division of the program given in Figure 2 is:

```
CREATE TABLE SALES_PERSON_FILE
(SP_NUMBER NUMBER (4),
SP_NAME VARCHAR2 (20),
SP_AMOUNT NUMBER (6,2),
CONSTRAINT SP_NUMBER_KEY
PRIMARY KEY (SP_NUMBER));
```

6. AN OVERVIEW OF COBOL AND SQL STATEMENTS

This section summarises the most common statements used by COBOL and SQL. For a complete list of statements and operators the reader is advised to consult specialised literature [3, 2]. COBOL statements can be divided into the following three categories:

- **Files and data manipulation statements:** They include statements to open and close files (OPEN, CLOSE), read data from files (READ), write data to files (WRITE) and move data in the computer's internal storage (MOVE). To delete records from a file COBOL uses DELETE and to update a record it uses REWRITE.
- **Arithmetic statements:** COBOL arithmetic statements are ADD, SUBTRACT, MULTIPLY, DIVID, and COMPUTE. Each statement has several forms that depend on the data used in the computation and the outcome of the output result. Sometimes these statements are followed by the reserved word GIVING (not included in the COMPUTE statement) that states that the result is stored in the variable that follows GIVING.
- **Other statements:** Like other programming languages, COBOL uses an IF statement. An IF statement in COBOL can take the simple one sided decision form "IF Condition THEN Statements", a two sided decision form "IF Condition THEN Statements1 ELSE Statements2" or the nested decision form. These are the most diffcult statements to translate into SQL. Other statements in COBOL are SEARCH to search for records in a file and SORT to sort the records of a file using a key.

SQL statements can also be divided into three categories:

- **Table and data manipulation:** This includes SELECT to query data in tables, INSERT to insert data into a table, UPDATE to update data in a table and DELETE to delete data from a table.
- **SQL Arithmetic statements and aggregate functions:** Aggregate functions operate on a single column and return a single value. These functions are: COUNT, SUM, AVG, MIN and MAX. The usual arithmetic operators (*, +, - and /) are used for arithmetic statements.
- **Multi-table queries:** When using more than one table, SQL queries can become complex. Subqueries can be used to further filter the selected data. Statements using the operators JOIN, UNION, INTERSECT and EXCEPT are also used.

The SELECT statement in SQL is the most important and used one. Usually the SELECT statement contains other clauses that specify extra conditions on the data selected. The general format of a SELECT statement is [3]:

SELECT	Specifies which columns to appear in the output
FROM	Specifies the table or tables to be used
WHERE	Filters the rows according to some condition
GROUP BY	Forms groups of rows with the same column value
HAVING	Filters the groups subject to some conditions
ORDER BY	Specifies the order of the output

This statement is going to be used as the basis for developing the SQL statements.

7. DEVELOPING SQL STATEMENTS

From the transformation point of view, COBOL statements can be divided into three categories. The first category contains the statements that are discarded by the system as they are not used in the generation of SQL statements and do not have their equivalent in SQL. A list containing these statements is created and named StopWords. StopWords contains the following statements:

StopWords = [OPEN, READ, WRITE, CLOSE, STOP,
ACCEPT, DISPLAY]

The second category of statements contains the statements that have their equivalent in SQL and their transformation is straight forward. This list is called StraightWords and is initialised as follows:

StraightWords = [ADD, SUBTRACT, MULTIPLY, DIVIDE,
SEARCH, SORT, DELETE]

The third category contains statement that can be (and should be) translated to SQL but do not have their direct equivalent. We have to use transformations to achieve the same result. This list is called ComplexWords and is initialised as follows:

ComplexWords = [IF, IF...THEN, IF...THEN...ELSE, MOVE,
REWRITE, PERFORM/UNTIL]

The LOBS-COQ approach to transform COBOL statements into SQL is based on the PROCEDURE DIVISION node of the XML document produced from the COBOL program. The PROCEDURE DIVISION node is the parent of one or more PARAGRAPH nodes. Each PARAGRAPH node in turn is composed of one or more statements. Each statement is identified and parsed using the COBOL syntax rules as defined in [2]. The type of the statement is identified and relevant information extracted if the statement belongs to the StraightWords list or ComplexWords list. The statement is ignored if it belongs to the StopWords list. We then use frames as the basis for the development of the SQL statements. The information extracted from the COBOL statements is used to fill the different frame's slots. Usually, all the statements of a Paragraph will contribute to the filling of the frame's slots and all the paragraphs of the PROCEDURE DIVISION have to contribute too.

We used an incremental approach for the development of the frames. We first started with some easy COBOL programs and we used the frame defined for the SELECT statement described in section 6 and shown in Figure 3 as the starting point. The aim of the transformation process is to fill all the required slots of the frame at the end of the procedure division analysis and then merge the different parts to produce the SQL statements. We illustrate our approach through some examples.

The simplest COBOL program would be the opening of a file and the printing of its records to an output file as illustrated by the program in Figure 2.

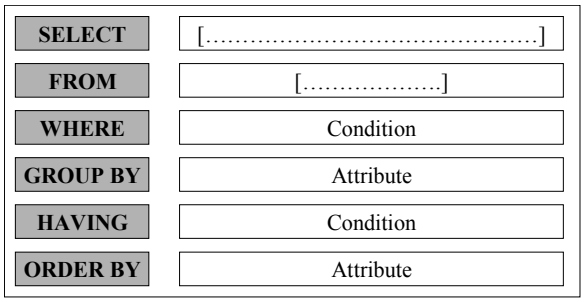

Figure 3: The Frame for the SELECT statement

At this stage the tables and their attributes are identified and information about the COBOL output files is recorded. Let us consider the paragraph MAIN-LOOP and as a start we ignore the first statement

"<STM>IF SP-AMOUNT GREATER THAN 500</STM>"

of this paragraph. The remaining statements is a set of "MOVEs" from the Input File to the Output file. The name of the table SALES_PERSON is used to fill the FROM slot of the SELECT frame.

COBOL MOVE statements are usually transformed into SQL SELECT statements. However, the first Statement containing the MOVE SPACES will be ignored as SPACES is not an attribute of the input file (SALES_PERSON_FILE). The processing of the second statement would initialise the slot SELECT to SP-NUMBER, the third will add SP-NAME and the fourth SP-AMOUNT. The fifth and sixth statements will be ignored as WRITE and READ belong to the StopWords list identified earlier. At the end of processing this simple procedure division, the following SQL statement is produced:

"SELECT SP_NUMBER, SP_NAME, SP_AMOUNT FROM SALES_PERSON_FILE;"

Now, let us bring back the IF statement and consider the full program given in Figure 2. Now, we are only printing those records which SP-AMOUNT is greater than 500. Note that the condition is given first in COBOL and its parsing in this case is very simple and the condition (the expression following the IF statement until the period) is just copied to the WHERE slot of the select frame. The remaining statements will be processed in the same way and this would produce the following SQL statement:

"SELECT SP_NUMBER, SP_NAME, SP_AMOUNT
FROM SALES_PERSON_FILE
WHERE SP_AMOUNT GREATER THAN 500; "

A case where MOVE is processed in a different way is when we are dealing with two files and an attribute in one file is the record key in another file. Let's assume we have two Files: the Stock File and the Supplier File. The Stock file records have the following attributes:

<ATTRIBUTE> 05 STOCK-NUMBER PIC 9999</ATTRIBUTE>

<ATTRIBUTE> 05 STOCK-DESC PIC X(20)</ATTRIBUTE>

<ATTRIBUTE> 05 SUP-CODE PIC 9999</ATTRIBUTE>

<ATTRIBUTE> 05 STOCK-PRICE PIC 9(4)V99</ATTRIBUTE>

<ATTRIBUTE> 05 STOCK-QUANTITY PIC 9(6)</ATTRIBUTE>

and the Supplier File records have the following attributes:

<ATTRIBUTE>05 SP-CODE PIC 9999.</ATTRIBUTE>

<ATTRIBUTE>05 SP-NAME PIC X(20).</ATTRIBUTE>

<ATTRIBUTE>05 SP-ADDRES PIC X(20).</ATTRIBUTE>

We note that the third attribute in the Stock File record is the supplier code, the same as the first attribute (key) of the supplier file. The following COBOL code outputs all items of stock which quantity is less then 100 together with the details of their supplier.

<PARAGRAPH>

MAIN-LOOP.

<STM>IF STOCK-QUANTITY LESS THAN 100</STM>

<STM>MOVE SUP-CODE TO SP-CODE</STM>

<STM>PERFORM READ-SUPP.</STM>

<STM>READ STOCK-FILE AT END MOVE "Y" TO WS-EOF-FLAG</STM>

</PARAGRAPH>

<PARAGRAPH>

READ-SUPP.

<STM>READ SUPPLIER-FILE INTO SUPPLIER-RECORD

INVALID KEY DISPLAY SP-CODE MOVE SPACES TO SP-NAME</STM>

<STM>MOVE SPACES TO REPORT-RECORD</STM>

<STM>MOVE SP-NAME TO RT-SUP-NAME</STM>

<STM>MOVE STOCK-NUMBER TO RT-NUMBER</STM>

<STM>MOVE STOCK-DESC TO RT-NAME</STM>

<STM>MOVE STOCK-QUANTITY TO RT-AMOUNT</STM>

<STM>WRITE REPORT-RECORD.</STM>

</PARAGRAPH>

In the MAIN-LOOP Paragraph, IF the quantity is less than 100, we move SUP-CODE to SP-CODE. However, SP-CODE is not an attribute of the output file but an attribute of another input file. Therefore our system deduce that it is not a "SELECTION" but we are looking for the Supplier record which attribute SP-CODE equals the attribute of the item of stock SUP-CODE. This is processed as a condition (equality of the two attributes) and added to the condition already obtained from the IF statement. This would result in the production of the following SQL statement:

SELECT SP_NAME, STOCK_NUMBER, STOCK_DESC, STOCK_QUANTITY

FROM STOCK_FILE, SUPPLIER_FILE

WHERE SP_COD_KEY = SUP_CODE AND STOCK_QUANTITY LESS THAN 100;

Note that both input file names are added to the FROM slot. As shown through these simple examples, the development of SQL statements is based on the process described in Figure 4.

As mentioned earlier, we took an incremental approach to the development of LOBS-COQ. Although the basic syntax used to transform COBOL statements into SQL statement is developed, It is the combination of several COBOL statement that makes it sometimes difficult to produce the full SQL statements. For this purpose, new rules are introduced with every new example. At this stage the prototype is dealing with programs using up to two files and around 15 lines of code and up to three paragraphs. All the statements in the StraightWords are easily translated to SQL and some of the ComplexWords list has been successfully transformed.

Figure 4: The Transformation Process

```
Identify the Procedure Division
  For each Paragraph do
    For each statement in the Paragraph do
      IF statement belongs to StopWords List
         ignore statement;
      ELSE
       Parse the COBOL statement;
       Identify the COBOL statement;
       Extract variables, conditions,etc. From COBOL
         Statement;
       Fill appropriate Frame slot;
    end
  end
Merge the frame's slots
Format the SQL statement
Produce the SQL Statement
```

8. CONCLUSION AND FUTURE WORK

In this paper we presented the first prototype of the LOBS-COQ system that attempts to transform COBOL legacy systems into Relational database schema and produce SQL statements for data querying. We gave a general overview of the approach and illustrated it using some examples. At this early stage the prototype does not deal with some complex situations such as nested IF statements. The system has not attempted yet to transform the data stored in the files used by COBOL programs to populate the tables that have been created. To test our programs, we had to manually enter the data into the tables and run the queries. We do not expect this to be a major problem as this has been already done by some systems. The results obtained so far are very encouraging and we still believe that relational databases are a good and cheap way of transforming COBOL legacy systems. The case studies used so far are academics and in the future developments more complex, industry-oriented programs will be used.

9. REFERENCES

[1] Douglas Bell. Software Engineering A Practical Approach. Addison-Wesley, 2000.

[2] ACM Mo Budlong. Teach Yourself COBOL in 21 days. Macmillan Computer Publishing, 2nd edition, 1997.

[3] Thomas Connolly and Carolyn Begg. Database Systems A Practical Approach to Design, Implementation and Management. Addison Wesley, 3rd edition, 2002.

[4] Kathi Hogshead Davis. Lessons learned in data reverse engineering. In Proceedings of the 8th Working Conference On Reverse Engineering, pages 323_327, 2002.

[5] Brown Gary DeWard. COBOL the failure that wasn't. http://cobolreport.com/columnists/gary/05152000.htm, 1999.

[6] Helen M. Edwards and Malcolm Munro. RECAST: Reverse engineering from COBOL to SSADM specification. In Proceedings of the 15th international conference on Software Engineering, pages 499_508, Baltimore, Maryland, United States, 1993.

[7] Richard Millham. Investigation: Reengineering sequential procedure-driven software into object-oriented event-driven software through UML diagrams. In Proceedings of the 26th Annual International Computer Software and Applications Conference, pages 731--733, Oxford, England, 2002.

[8] Ikuyo Nagaoka, Katsuaki Sanou, Daisuke Ikeo, Michio Tsuda, and Shin'ichi Akiba, A reverse engineering method and experiences for industrial COBOL system, In Proceedings of the 4th Asia-Pacific Software Engineering and International Computer Science Conference, pages 220_228, Clear Water Bay, Hong Kong, 1997.

[9] Harry M. Sneed. Wrapping legacy COBOL programs behind an XML-interface. In Proceedings of the 8th Working Conference on Reverse Engineering, pages 189--197, Suttgart, Germany, 2001