Abstract: The Maintainer’s Assistant is an interactive software maintenance tool which helps the user to extract a specification from an existing source code program. It is based on a program transformation system, in which a program is converted to a semantically equivalent form using proven transformations selected from a catalogue. This paper describes an environmental support tool, the COBOL pre-processor, for the Maintainer’s Assistant. The requirements of the tool are stated and the technical methods used in the tool are summarised. The current implementation is then described and results achieved discussed. Finally, both the research into the tool and the experience obtained via implementing the tool are summarised.

1. INTRODUCTION AND BACKGROUND

It is a well-established view that in order to achieve major productivity gains, software maintenance must be performed at a higher abstraction level than the code level i.e. at the design level or specification level. Advantages of maintaining at high level are more compact representation and hence less to understand and the way in which algorithms are expressed is more closely linked to the application domain. In other words, whether it is corrective, perfective, adaptive, or preventive maintenance, the key to effective maintenance is program comprehension [BEN 88], [CHA 88]. Maintainers need to understand what the code does and what it is supposed to do before implementing a change. High level abstractions provide an easy way to gain such understanding. This important step in the software maintenance process of acquiring high level abstract views from existing code is called Reverse Engineering [CHI 90].

Reverse engineering involves the identification or “recovery” of the program requirements and/or design specifications that can aid in understanding and modifying the program. They are then used to discover the underlying features of a software system including the requirements, specification, design and implementation. The purposes of reverse engineering can be separated into the quality issues (e.g. to simplify complex software, to improve the quality of software which contains errors, to remove side effects from software, etc.), management issues (e.g. to enforce a programming standard, to enable better software maintenance management techniques, etc.) and technical issues (e.g. to allow major changes in a software to be implemented, to discover and record the design of the system, and to discover and represent the underlying business model implicit in the software, etc.).

One example of reverse engineering is to extract the specification from the program source code. This is necessary, firstly because the documentation and relevant reference materials are not complete, and
the personnel who may have relevant knowledge have already forgotten about it or left the company. Secondly, although there might be some documents available, the software may not be implemented consistently with the documents. Thirdly, the original documents and reference materials were not written in a modern specification language and they cannot be used in a modern software maintenance environment. This means that the extraction of the specification of old program code is a vital step especially when the program code is the only available documentation or is the only source to rely on.

1.1 The Maintainer's Assistant (MA)

The Maintainer's Assistant is the name given to a tool whose main objective is to develop a formal specification from old code [BEN 92]. It will also reduce the costs of maintenance by the application of new technology and increase quality, so producing improved customer satisfaction. The source code can be a program written in any language such as COBOL. The ultimate specification will be written in a specification language such as Z [SPI 88].

To move from the low-level source code to the high-level specification, different levels of abstraction have to be presented. To express the broad range of the stages (all forms of code) in the maintenance of a program, a Wide Spectrum Language (WSL) is used in the Maintainer's Assistant. This incorporates a variety of constructs, from low-level machine-oriented constructs up to high-level specification ones. It is natural for specification constructs to be mixed freely with programming constructs in the intermediate steps of transformation, by using the wide spectrum language, so that programs are gradually abstracted to high-level specification. The formal definition of WSL can be seen in [WAR 89] and examples of WSL programs will be given in later sections of this paper. The external format of a WSL program looks like a commonly-used programming language such as PASCAL. Furthermore, the system is able to cope with program source code written in any programming language, as long as a pre-processor is built to translate that language into WSL.

The Maintainer’s Assistant is based on a formal system developed by Ward [WAR 88] [WAR 89] in which it is possible to prove that two versions of a program are equivalent. The formal system is independent of any particular programming language and allows the inclusion of arbitrary specifications as statements in a program. Hence it can be used to prove that a program is equivalent to a given specification.

Programs are defined to be equivalent if they have the same denotational semantics. Hence equivalent programs are identical in terms of their input-output behaviour, although they may have different running times and use different internal data structures. A refinement of a program, or specification, is another program which will terminate on each initial state for which the first program terminates, and will terminate in one of the possible final states for the first program. In other words a refinement of a specification is an acceptable implementation of the specification and a refinement of a program is an acceptable substitute for the program.

1.2 Reverse Engineering Using Program Transformation

Program transformation is the process of formally changing a program to a different program with the same semantics as the original program. Much work has been focused on the program transformation as one kind of programming paradigm in which the development from specification to implementation is a formal, mechanically supported process. The long range objective of this approach is dramatically to improve the construction, reliability, maintenance, and extensibility of software. There have been several transformational system implemented based on this idea [BAL 81] [BAU 89] [FEA 87] [FIC85]. Generally speaking, these systems have the following features:

- The main goal is to experiment with the mechanically assisted development of a broad range of programs. This includes: general support for program modification, e.g. optimisation of control structures, efficient implementation of data structures, and the adaptation of data structures and given programs to particular styles of programming; program synthesis, e.g. the generation of a program from a formal description of the problem; program adaptation to particular environments and verification of the correctness of programs.

- Their functions are to provide a transformation data base for keeping the information collected by the user; user guidance for offering the user advise on the choice of transformations; history
recording for documenting the development process; assessment of programs for measuring the effect of transformations.

- Their working modes are mainly semi-automatic, working both autonomously and manually, e.g., the CIP (Computer-Aided, Intuition-Guided Programming) System [BAU 89]; while a manual system or a fully automatic system may have certain disadvantages, e.g., the user being responsible for every single transformation step in a manual system, or the system not having enough built-in heuristics in a fully automatic system.

- Basic types of transformation are the catalogue approach and the generative set approach. A catalogue of rules is a linearly or hierarchically structured collection of transformation rules relevant for a particular aspect of the development process. A generative set means a small set of powerful elementary transformations to be used as a basis for constructing new rules.

The transformation system used in reverse engineering should support the particular demands of the maintenance programmer, i.e. helping the maintainer to understand the program transformed. There are a number of benefits to using formal transformations:

- Increased reliability: errors and inconsistencies are easier to spot at high level of abstraction.
- Formal links between the specification and code can be maintained.
- Maintenance can be carried out at the specification level.
- Large restructuring changes can be made to the program with the confidence that the program's functionality is unchanged.
- Programs can be incrementally improved instead of being incrementally degraded.
- Data structures and the implementations of abstract data types can be changed easily.

Distinguishing features of the program transformer of the Maintainer's Assistant include:

- A wide spectrum language is used. The wide spectrum language is an intermediate language, which means that program transformations as well as the tool itself only needs to be built once and programs in any programming language can be dealt with by the tool.
- The applicability of a transformation is tested before it is applied.
- A history/future database is built-in to allow back-tracking and “forward-tracking”.
- The system is interactive and incorporates a browser which provides a graphical mouse-driven front end and pretty-printer.

1.3 Requirements of the Supporting Environment for MA

The success of building systems is dependent both on the application of the method and on the method itself. A method typically consists of a sequence of steps combining management procedures, technical methods, and automated support to produce systems [WAS 83]. When a good technical method is found and a good management procedure is adopted, the automated support or environment becomes a very crucial part of producing systems. The software maintenance environment provides a framework for describing the way that software maintainers make use of a software maintenance methodology. In deciding what are needed in the MA, the following points must be addressed:

1. **Reverse Engineering** — An important requirement of reverse engineering is to understand software that is probably without any available specification, documentation, or, indeed, any helpful information.

2. **Program Transformation** — The key elements of the environment of the Maintainer's Assistant are those which support program transformations.

3. **Rapid Prototyping** — There are several advantages of rapid prototyping which can be taken to develop the Maintainer's Assistant itself. For instance, the system can be developed much faster by rapid prototyping, so that it can speed up the implementation. The user is involved in the process, so that he or she (mainly the builder in our system) can determine if the development is moving in the right direction. Rapid prototyping produces a model of the final system, so that the user, as well as builder, can see the final system in the early development stage.

From the above, the environment of the Maintainer's Assistant should support the following characteristics:

1. It should support the maintenance of large, to very large, software.
2. It should support software independently of its source programming language so that specifications and programs can both be included in the environment in the form of the Wide Spectrum Language.

3. It should provide a set of tools covering all the activities required by, and provided by the current technical methods and make provision for other possible tools in the future. It should provide a set of tools covering all the activities required by, and provided by the current technical methods and make provision for other possible tools in the future.

1.4 Architecture of MA

MA supports the transformation of an existing source code to a specification in three phases (Figure 1). In phase 1, a "Source-to-WSL" translator will take source code in COBOL or other language and translate it into its equivalent WSL form. The maintainer does all his operation through the Browser. Then the Browser checks the program and uses the Program Slicer to chop the program into manageable sized lumps. The maintainer may conduct the process more than once until he realises that he has split the code in such a way that it is ready for transformation. Eventually, this code is saved to the Repository, together with other information, such as the relations among these code modules, in order to assemble specifications of those code modules.

In the second phase, the maintainer will take one piece of code out from the Repository to work with. The Browser allows the maintainer to look at and alter the code under strict conditions and the maintainer can also select transformations to apply to the code. The program transformer works in an interactive mode. It presents WSL on screen in Human Computer Interface (HCI) format and searches a catalogue of proven transformations to find applicable transformations for any selected piece of code. These are displayed in the user interface's window system. When the Program Transformer is working, it also depends on the supporting tools such as General Simplifier, the Program Structure Database and the Knowledge Base System by sending them requests. The maintainer can apply these transformations or get help from the Knowledge Base as to which transformation are applicable. Once a transformation is selected it is automatically applied. These transformations can be used to simplify code and expose errors. Finally, the code is transformed to a certain high level of abstraction and the code is saved back to the Repository.

Figure 1. Components of MA
The third phase comes when all the source code in the Repository has been transformed. The Program Integrator is called to assemble the code into a single program in high-level WSL. A WSL to Z translator will translate this highly abstracted specification in WSL into specification in Z.

1.5 Components of MA

From the outset of the project, attention has been paid to the particular aspects of the environment for supporting the program transformation of reverse engineering. The Maintainer's Assistant system has been implemented on a SUN workstation system and the Program Transformer itself is written in Common LISP. Other tools are discussed below.

- **COBOL-to-WSL Pre-processor** --- In this study, the COBOL Pre-processor is addressed. Pre-processors for other languages such as an IBM-Assembler to WSL Pre-processor have been implemented.
- **Program Slicer** --- With the maintainer's assistance, the Program Slicer pre-process the program code split before saving the code in the Repository. Its main function is to help in naming program modules to be transformed.
- **Program Integrator** --- Using the information stored in the Repository when the program was split, the Program Integrator will reassemble those pieces of WSL code after transformation.
- **Specification Output and Reuse/Redevelopment Interface Tool** --- This mainly consists of two parts (1) WSL to Z Translator, dealing with both the symbolic translation and the output in Z notation to both the screen and the printer; and (2) Reuse/Redevelopment Interface, dealing with output obtained high-level WSL programs for the purposes of reuse and redevelopment.
- **Transformer Supporting tools** --- These include: (1) A History/Future Database, which allows the maintainer to go back to an older version of the program he has transformed. It is usual for the maintainer to move forwards and backwards several times through a sequence of transformations in order to reach an optimal version of the program. Two commands "Undo" and "Redo" are provided. (2) A Program Structure Database, which is a dynamic database mainly serving the Program Transformer. The Program Transformer accesses the Database via the Database Manager. When the Program Transformer is transforming a section of program code, queries about the program are sent to the Database Manager. When a query is made for the first time, the Database Manager will go through the program structure and calculate the answer to that query. The result will both be sent to the Program Transformer and saved in the database. When the question is asked again, the database manager will check the database and simply return the result. (3) A General Simplifier, which carries out symbolic calculations in mathematics and logic for helps the Program Transformer. Mathematical and logical operations defined in the system are: +, -, *, /, **, Min, Max, Div, Mod, =, $>$, $<$, $<>$, Not, And, Or, etc.
- **Browser and Interface** --- The Browser and Interface is implemented together as a graphical user interface to the other subsystems of the Maintainer's Assistant using the X-Windows System. It provides all the commands necessary to use other Maintainer's Assistant programs via buttons and pop-up menus and uses several windows to display the output from the system and to receive text input from the user. In particular it provides a browser to display the program being transformed by the Transformer, and this has facilities not provided by the transformer such as pretty printing the program and a mechanism to fold or unfold sections of code. It runs displaying a frame made up of three windows. The first window is a box containing several buttons and labels. By clicking these buttons the user can invoke various commands, change options, and pop-up other windows. The second window (the application window) is an interface to the transformer command driven user interface and the third window (the display window) is used to display the program being transformed by the user. A manual page for the front end is available for the novice user.

In the remaining of the paper, Section 2 presents an overview and the design of the COBOL Pre-processor for MA; Section 3 explains how to implement the COBOL Pre-processor and Section 4 concludes our study.

2. OVERVIEW AND DESIGN OF COBOL PRE-PROCESSOR

2.1 COBOL Pre-processor
It is believed that there are 800 billion lines of COBOL programs existing in the world [LAY 94] today and they are a big portion of legacy systems still used in industry. Therefore building a COBOL Pre-processor for MA is an inevitable task.

The COBOL language was first developed in 1959. The CODASYL committee (Conference on Data Systems Languages) produced the initial specification of COBOL in 1960, and a revised version appeared in 1961. The first ANSI (American National Standard Institute) specification of the COBOL language was published in 1968. Later standards were the ANSI 1974 and the ANSI 1985 Standard. COBOL offers the following advantages within the standard language [ING 89], which are related to the research:

- Uniform treatment of all data as records.
- Extensive capabilities for defining and handling files.
- Incorporation of many functions which in other contexts would be regarded as the province of system utilities.
- The ability to construct large programs from independently compiled modules which communicate with each other by passing parameters or by using common files.

The COBOL language used in this research not only is unrestricted to any dialect of COBOL but also covers features written in ANSI COBOL Standard 1985. More importantly, this research will be not only of benefit to COBOL programs but also to other data intensive programs written in other languages.

2.2 Compiler Writing Techniques and Available Tools (Flex and Bison)

After studying different techniques of implementing a pre-processor of this kind, we have chosen a technique that, in our opinion, is the easiest in respect of available tools. The BISON and FLEX are today belonging to the most popular tools, and the technique based on using them is simple and clear. Briefly, it is working like this: firstly descriptions of the syntax and semantics of COBOL are written; then FLEX and BISON will convert the descriptions into a code of the scanner and the parser respectively written in C language; finally that a code generator is needed to develop data structures and a set of procedures for creating WSL code.

Flex, Fast Lexical Analyser generator, is a tool for generating scanner which recognise lexical patterns in text. Flex reads the given input files, or its standard input if no file names are given, for a description of a scanner to generate. The description is in the form of pairs of regular expressions and C code, called rules. Flex generates as output a C language source file, which is a scanner. The Flex is the newest version of Lex, a Unix’s systems tool.

Bison is a parser generator in the style of YACC. It should be upwardly compatible with input files designed for YACC. Bison converts a grammar description for an LALR(1) context-free grammar into a C program to parse that grammar. The most common formal systems for presenting such rules for humans to read is Backus-Naur-Form (BNF), which was developed in order to specify the language Algol 60. Any grammar expressed in BNF is a context-free grammar. The input to Bison is essentially machine-readable BNF.

2.3 COBOL Program Structure

A COBOL program consists of four parts (divisions):

1. Identification division --- An identification of the program. This division identifies both the source program and the resultant output listing. In addition, the user may include the date when program was written, the date of the source program compilation and some other information as desired under the paragraphs in this section [MIC 88].

2. Environment division --- A description of the equipment to be used to compile and run program. This division specifies standard method of expressing those aspects of a data processing problem that are dependent upon the physical characteristics of a specific computer. The Environment Division allows specification of the configuration of the source computer and the object computer. In addition, information relating to input-output control, special hardware characteristics and control techniques can be given.
3. **Data division** --- A description of the data to be processed. This division describes the data that the object program is to accept as input, to manipulate, to create, or to produce as output.

4. **Procedure division** --- A set of procedures to specify the operations to be performed on the data. This Division may contain declarative and non-declarative procedures [MIC 88].

Each division is divided into sections which are further divided into paragraphs, which in turn are made up of sentences.

### 2.4 Rules of Pre-Processing COBOL into WSL

The key problem of pre-processing a COBOL program into a WSL programs for MA is to retain the semantics of the original COBOL program. This has been consistently considered throughout the design of the COBOL Pre-processor.

The COBOL specification is composed of four divisions and therefore all rules of COBOL into WSL conversion will be divided using these divisions.

**Identification division** --- All the information from this division will be converted into a standard comment statement in WSL, e.g.:

```
COBOL                                          WSL
PROGRAM-ID. program-name.                      COMMENT: "program-id: program-name,
AUTHOR. comment-entry.                        author: comment-entry:";
```

**Environment division** --- Almost all constructions of this division will be converted into comment statements (e.g. description of source and object computers) with the exception of SPECIAL-NAMES in CONFIGURATION SECTION and FILE CONTROL in INPUT-OUTPUT SECTION, which are used to describe data in COBOL language. These two statements will not be converted into comment statement but names of these statements will be put on the table of names with the following prefixes:

- `spe`, i.e. special name,
- `sfc`, i.e. sequential files,
- `rfc`, i.e. relative files,
- `ifc`, i.e. indexed files,
- `srt`, i.e. sorted files.

Both in the **Environment division** and **Data division** there are sections not included in the standards of COBOL. For such sections a syntax element called “unknown section” has been created. All “unknown sections” will be dealt with as follows:

1. <Name of section> will be converted into comment state with warning:
   
   ```
   COMMENT: "WARNING !!! Unknown section: name."
   ```

2. <Body of section> will be ignored.

**Data division** --- Data division is made up of elements which describes the structure of data in COBOL programs. It is similar to `var` statement in Pascal. This division is composed of six sections:

- FILE SECTION which will be converted into `file` - the new extension of WSL,
- WORKING-STOREAGE SECTION which will be converted into `record`,
- LINKAGE SECTION which will be converted into `record`,
- COMMUNICATION SECTION which will be converted into `record`,
- REPORT SECTION which will be ignored,
- SCREEN SECTION which will be converted into `record`.

These sections are based on the statement called `record description entry`. Record description entry is a basic statement record of WSL - a standard type WSL describing data structure - with the following extensions, which identifies structural data:

- keyword `asc-key` - used before the field name in the record statement for marking fields which are an ascending key,
- keyword `dsc-key` - used before the field name in the record statement for marking fields which are an descending key,
- keyword **index** - used before the field name in the record statement for marking fields which are index,
- keyword **redefine** - used after the name of the record for redefining one record into another.

The following example presents the idea of the conversion:

```cobol
WSL

01 test
   02 field-three
      field-one;
   02 field-four REDEFINES field-three
   02 field-five
01 test INDEXED BY field-one
01 test ASCENDING KEY IS field-two
01 test INDEXED BY field-one
01 test ASCENDING KEY IS field-two

record field-five;
end;
```

Special numbers in COBOL record description entry will be converted into “normal” names (variable) with prefixes:

- COBOL level 77 will be converted into a name with the prefix `nsb` (non subdivided name),
- COBOL level 78 will be converted into a name with the prefix `cnt` (constant name),
- COBOL level 88 will be converted into a name with the prefix `cnd` (conditional name).

Procedure division --- Rules for this division are as follows:

- Main program procedure is:
  . name of program == body of program.
- All sections from Procedure division are converted into separate procedures. The names of these procedures are named by names with the prefix: section.
- All paragraphs from Procedure division are converted into separate procedures. The names of these procedures are named by names with the prefix: paragraph.
- COBOL statement: Declaratives is ignored.
- Functions present in Procedure division are converted into their counterparts in WSL.

**2.5 Structure of the Pre-processor**

The Pre-processor consist of three modules: a parser, a scanner and a code generator. Figure 2 presents the way in which it has been created.

![Figure 2. Generation of COBOL Pre-processor](image)

On the basis of the description of the COBOL grammar the BISON creates the parser’s source code in C language (y.tab.c). The parser’s task is to analyse the semantics of the COBOL grammar. The analyser divides the COBOL program into syntax elements and checks whether or not it can apply to them the rules described in the analyser. If the application of a rule is possible, a suitable procedure of the code generator is called. If not, an error is indicated. While generating the parser, the BISON creates an additional file (y.tab.h) containing a declaration of all the tokens required by the parser.

The second module of the Pre-processor is the scanner, which is generated by the FLEX (the generated file: yy.lex.c). The scanner is created on the basis of the lexical description of COBOL, i.e. the description of the COBOL syntax. The task performed by the scanner is the division of the input file
of the COBOL program into a set of tokens which will then be put into the parser’s input. The set of the tokens required by the parser is declared in the file (y.tab.h). The tokens of the input file are identified according to their type and allocated in the names table of the code generator. They will be used in the procedures of the generator called by the parser.

The last module of the Pre-processor is the code generator (gen.cc, gen.h) controlled by the parser and provided with data by the scanner, it generates the WSL code. It consists of the set of procedures called by the parser, the names table, and the object data structure that organizes information from the scanner.

2.6 Description of Code Generator Structure Using Object Oriented Methodology

The internal structure of the code generator consists of two parts. One part is the set of the procedures called by the parser, the other part is the data structure that organizes information from the scanner. The structure has been implemented by means of object oriented methodology (Coad-Yourdon) and presented in Figure 3.