ReSource™ - The Methodology of Recovering Source Code

Overview

Conventional wisdom says it can't be done: recovering source code from object modules. Once a set of source code statements has been compiled and transformed into a machine code equivalent, most individuals would argue that the program knowledge, richness, syntax and semantics of the source language has been lost forever. It's difficult to believe that source code can be coaxed out of the bits and bytes of complex program entities such as load modules. It is as though people view compilers like magicians' top hats into which source code goes and executable programs come out... but never the source code again. It's these beliefs that for years have forced programmers to resort to re-writing programs from scratch should the original source code become lost or mislabeled. Fortunately, there is now an alternative for recovering source code from object code.

This paper describes a formal technique that has been developed by Fred Brandes for the automated recovery of COBOL and Assembler source code statements. Essential Systems Technical Consulting currently markets this technology as a consulting service, either offsite or onsite. The paper focuses on the steps that are used, and the process that ensures the resultant source code is 100 percent functionally equivalent to that of the original.

Note: This white paper is derived from a transcript of a speech originally delivered by Fred Brandes at Share in September, 1998. The technology resulting from this process is called, ReSource™, a patented product marketed by Essential Systems Technical Consulting. For more information, please call 770 479-2250.

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

Recovering Source Code

Methodology

Source code recovery is based on the following concepts or steps:

1. Delink
2. Disassembly
3. Decompile
    - Pattern matching
    - Operand analysis
    - Internals analysis
    - Supporting information
4. Validation

Although these basic steps can be applied to any executable program regardless of the original source language, the examples shown in this paper will refer to programs derived from COBOL source files for the IBM mainframe environment running under the MVS, VM or VSE operating systems. Many of the improvements that have been incorporated into the newer versions of these compilers, provide even more "clues" which facilitate this recovery process.

All of the concepts behind source code recovery are founded on the single assumption that any program presented for recovery is the end result of a *clean compile and link*. In other words, the source code, at the time it was compiled, was syntactically valid and the program can be run on an IBM or compatible mainframe computer utilizing the 360/370/390 instruction set. The program contains everything required by the machine to execute the instructions contained in the original source code and nothing that is superfluous. If the program were viewed as a completed jigsaw puzzle, every piece is required to form a whole and coherent picture and no pieces are left over. And just as a completed jigsaw puzzle may appear seamless, so too does a program. Unless, that is, the puzzle (or program) is inspected closely.

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

Step 1: DeLink

The first step in recovering source code from an executable program is to break down the executable into its core components known as control sections or Csects. This process identifies the individual modules which make up the executable. They include runtime modules provided by the operating system to incorporate system level services such as CICS and/or DB2.

Also “hidden” within an executable are components such as user-written subroutines and BMS maps which also make up critical pieces of the overall application. These smaller components can be recovered individually if necessary.

Each of these Csects are intertwined in the executable so in order to focus on one individual Csect, the executable must first be delinked. From here, the basic “program” can be disassembled.

Step 2. Disassembly

The compiled “program” is really a representation of ones and zeros, called binary. Figure 1 illustrates what the internals of a binary program file might look like if inspected.

```
00000101 00011111 00001000 00000000 00000000 00000000 11010010 00010011
01100001 01100000 01100001 01001000 01011000 01000000 11010010 00110000
01000001 11100000 10111000 01101000 11010010 00000011 01000000 00100000
11000000 01001000 01011000 11110000 01000000 00001100 00000101 00011111
00000100 11000010 00000000 00000000 01000111 11110000 10111000 01110100
01011000 10110000 11000001 00111000 01000111 11110000 10110110 11111100
11010010 00000011 11010010 01100000 11010010 01100100 01000001 00000000
10111000 10000110 01010000 00000000 11010010 01100100 01000111 11110000
10110001 10001000 11010010 00000011 11010010 01100100 11010010 01100000
11010010 01011111 10100000 00000000 10000000 00000000 01001000 00000000
11000001 00001000 01011000 01000000 11010010 00101100 01000001 11100000
10111000 10101000 01011000 11110000 01000000 00001100 00000101 00011111
00001000 00000000 00000000 00000000 01011000 10110000 11000001 00111000
```

Figure 1: Example of Binary Code

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

Programs represented in this manner, while accurate, are extremely difficult to inspect and interpret. At a somewhat higher level, the same program can be viewed from a byte oriented or hexadecimal level of granularity as illustrated in Figure 2.

```
05 1F 08 00 00 00 D2 13
61 60 61 48 58 40 D2 30
41 E0 B8 68 D2 03 40 20
C0 48 58 F0 40 0C 05 1F
04 C2 00 00 47 F0 B8 74
58 B0 C1 38 47 F0 B6 FC
D2 03 D2 60 D2 64 41 00
B8 86 50 00 D2 64 47 F0
B1 88 D2 03 D2 64 D2 60
D2 5F A0 00 80 00 48 00
C1 08 58 40 D2 2C 41 E0
B8 A8 58 F0 40 0C 05 1F
08 00 00 00 58 B0 C1 38
```

Figure 2: Example of Hexadecimal Code

While somewhat easier to decipher, hexadecimal representation of a program is still at too low a level for most programmers. The same code can be further represented at a higher level: Assembler. Figure 3 illustrates the previous examples as a set of Assembler language statements and operands. Disassembly of the code, no matter what the original source language, is the first concept in source code recovery:

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

```
         BALR  R1,R15                          05 1F
         BALR  R1,R15                          05 1F
         DC    XL4'08000000'                   08000000
         MVC   WS100160(20),WS100148-G20       D2 13 6 160 6 148
         L     R4,TGT00230                     58 40 D 230
         LA    R14,PGM06BFC                    41 E0 B 868
         MVC   020(4,R4),EODAD4                D2 03 4 020 C 048
         L     R15,00C(,R4)                    58 F0 4 00C
         BALR  R1,R15                          05 1F
         DC    XL4'04C20000'                   04C20000
PGM06BFC DS    0H
         B     PGM06C08                        47 F0 B 874
         L     R11,PBL1                        58 B0 C 138
         B     PGM06A90                        47 F0 B 6FC
PGM06C08 DS    0H
         MVC   TGT00260(4),TGT00264            D2 03 D 260 D 264
         LA    R0,PGM06C1A                     41 00 B 886
         ST    R0,TGT00264                     50 00 D 264
         B     PGM0651C                        47 F0 B 188
PGM06C1A DS    0H
         MVC   TGT00264(4),TGT00260            D2 03 D 264 D 260
         MVC   000(96,R10),000(R8)             D2 5F A 000 8 000
         LH    R0,H_96                         48 00 C 108
         L     R4,TGT0022C                     58 40 D 22C
         LA    R14,PGM06C3C                    41 E0 B 8A8
         L     R15,00C(,R4)                    58 F0 4 00C
         BALR  R1,R15                          05 1F
         DC    XL4'08000000'                   08000000
PGM06C3C DS    0H
         L     R11,PBL1                        58 B0 C 138
```

Figure 3: Example of a Disassembled Program

The code illustrated in the above three examples is identical. The differences are solely the result of the format in which the information is presented.

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

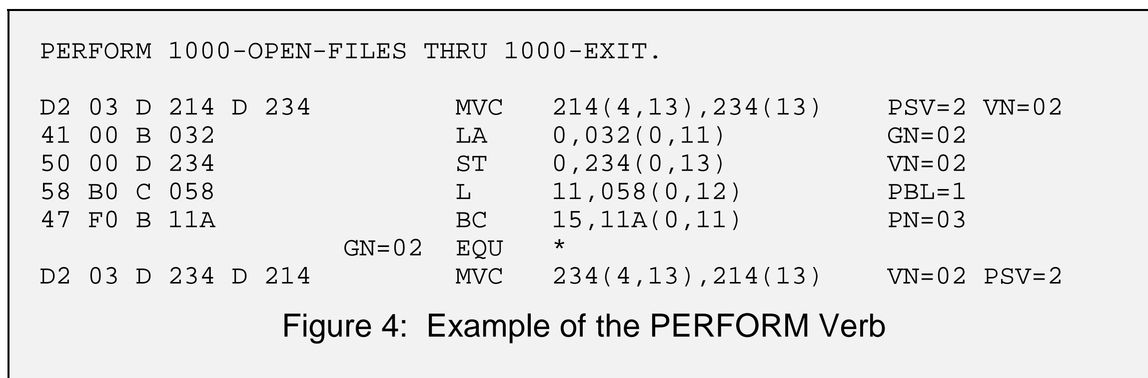
Step 3. Decompile

Pattern Matching

The next phase of the concept in the recovery of COBOL is that *every COBOL verb will generate a distinct pattern of machine instructions* that can be analyzed and then programmatically stored, retrieved and applied to the programs to be recovered. These patterns may be as simple as a single machine instruction or they may encompass many dozens of machine instructions. Just as olving a jigsaw puzzle requires a sharp eye for subtle differences in the shapes of the puzzle's pieces, source code recovery requires an intimate knowledge of COBOL verbs and the patterns of machine code that they produce.

For source code recovery, COBOL compiler research begins by creating source code files that provide examples of all the various programming elements in all their flavors, particularly Procedure Division verbs. Next, these research files are compiled with the PMAP option, or in the case of COBOL II the LIST option. Finally, the compile listings are analyzed to determine what verbs generate what particular patterns of machine instructions.

The PERFORM verb provides a good example of this type of research. Figure 4 illustrates the source code version of a simple PERFORM verb followed by the PMAP portion of the compile listing for that same PERFORM:



```
PERFORM 1000-OPEN-FILES THRU 1000-EXIT.

D2 03 D 214 D 234          MVC   214(4,13),234(13)    PSV=2 VN=02
41 00 B 032                LA    0,032(0,11)          GN=02
50 00 D 234                ST    0,234(0,13)          VN=02
58 B0 C 058                L     11,058(0,12)         PBL=1
47 F0 B 11A                BC    15,11A(0,11)         PN=03
                    GN=02  EQU   *
D2 03 D 234 D 214          MVC   234(4,13),214(13)    VN=02 PSV=2
```

Figure 4: Example of the PERFORM Verb

Analyzing this pattern, and others like it, leads to the following conclusions: The pattern begins and ends with MVC instructions and the operands of these instructions are reversed. In addition, both operands are full word fields in the TGT (i.e., they both employ R13 as their base and the machine length in the MVC instructions is 3). Finally, two instructions prior to the last statement in the pattern is an unconditional branch to the paragraph to be PERFORMed. These instructions are distinctive enough for us to conclude that any identical

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

pattern found in a COBOL program we are recovering must be generated by a simple PERFORM.

Having done this type of analysis for many different verbs, and indeed for many different COBOL compilers and compiler options, we want a means to describe this information so that it can be stored and readily retrieved and applied during decompiles. We developed a proprietary *pattern description language* (PDL), a *pattern description compiler* (PDC) and a *decompiler* (DECOMP). The decompiler is the expert system engine, and the pattern description source files and compiled pattern programs are the knowledge base at the heart of the expert system.

Figure 5 illustrates the pattern description source file for the simple PERFORM verb:

```
pattern PERFORM
        find  stmt1 inst   eq MVC
        test  stmt1 op1(3) eq TGT
        test  stmt1 op2(3) eq TGT
        test  stmt1 l      eq 03

        find  stmt3 op1    eq stmt1 op2 after stmt1
        test  stmt3 op2    eq stmt1 op1
        test  stmt3 inst   eq MVC
        test  stmt3 l      eq 03

        stmt2 2 before stmt3
        test  stmt2 class  eq unconditional

        src1  perform &stmt2.op2 thru &stmt1.op2-exit
pend
```

Figure 5: Example of PERFORM Pattern

Figure 6 illustrates a portion of the disassembly example shown earlier:

```
         MVC   TGT00260(4),TGT00264(4)       D2 03 D 260 D 264
         LA    R0,PGM06C1A                   41 00 B 886
         ST    R0,TGT00264                   50 00 D 264
         B     PGM0651C                      47 F0 B 188
PGM06C1A DS    0H
         MVC   TGT00264(4),TGT00260(4)       D2 03 D 264 D 260
```

Figure 6: PERFORM Disassembly

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

When the decompiler is asked to FIND PERFORM it retrieves the compiled version of the PERFORM pattern description and executes each instruction in turn until a find or test statement fails or the entire pattern is found in the program being recovered. In this example, the decompiler will attempt to locate (or find) a MVC instruction and then test that instruction's first and second operands for a prefix of TGT. A final test for a length of 3 must succeed for the decompiler to designate the instruction as stmt1 within the current pattern matching structure. The decompiler then proceeds to find and test stmt3 which is the MVC with reversed operands. Finally, the decompiler checks that the instruction that is two instructions prior to stmt3 is an unconditional branch. When the decompiler has successfully found code that matches all of the statements described in the pattern it assigns the PERFORM pattern to these statements. The src1 statement in the pattern description supplies information about how the final COBOL source statement is to be constructed when the decompiler is told to generate source code for the recovered program.

Operand Analysis

The pattern description language does a good job of recovering COBOL PROCEDURE DIVISION verbs but does not provide much information about other elements in a COBOL program, particularly data items in file descriptions, WORKING STORAGE and the LINKAGE SECTION. These items are recovered by applying the next concept in source code recovery which is referred to as operand analysis.

Operand analysis is the detailed examination and eventual analysis of the operands of the machine instructions making up the PROCEDURE DIVISION. This examination and analysis allows us to determine the relative placement of a data item and the PICTURE clause that is most appropriate for the data item. Data item recovery relies on analysis of research programs in much the same manner as the PROCEDURE DIVISION verb recovery. Once again, source code files are prepared that encompass operations employing data items with various PICTURE clauses. The compile listings are analyzed and permit the determination of the types of PICTURE clauses that generate specific machine instructions.

The following example illustrates type of research and involves elementary data item that is defined as a PIC 9. Figure 7 shows the source code for the PIC 9 data item definition and a simple verb employing the data item followed by the PMAP portion of the compile listing for that same verb:

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

```
01  WS-COUNT    PIC 9(5).

    ADD 1 TO WS-COUNT.

F2 74 D 208 6 048   PACK  208(8,13),048(5,6)   TS=01 DNM=1-106
FA 30 D 20C C 020   AP    20C(4,13),020(1,12)  TS=05 LIT+0
F3 43 6 048 D 20C   UNPK  048(5,6),20C(4,13)   DNM=1-106 TS=05
96 F0 6 04C         OI    04C(6),X'F0'         DNM=1-106+4
```

Figure 7: Example of PIC 9 Syntax

Our research has determined that whenever a PIC 9 item is involved in an arithmetic operation the final machine instruction generated will be determined as:

```
OI item+itemlength-1,X'F0'.
```

This machine instruction ensures that the resulting value in the data item is in zoned decimal format. Finding this type of machine instruction during source code recovery allows us to conclude that the data item being operated on is an elementary item of type PIC 9. In addition the PACK and UNPK instructions indicate the length in bytes of the item which can then be expressed as the number of 9s in the PICTURE clause. The actual operand analysis occurs during both the disassembly and the decompilation of the program being recovered. As each operand is encountered, the probable PICTURE clause value based on the machine instruction acting on the operand is determined and stored for later use during the source code generation phase of the decompiler.

Operand analysis allows YOU to recover accurately any data item that is referenced within the program.

Internals Analysis

Unfortunately, data items that are not referenced within the program are usually recovered as FILLER PIC X items since the program contains no information about them. However, this is not always the case. If a data item is given an initial value via a VALUE clause then it is possible to discern the PICTURE clause based on analysis of the constant in WORKING STORAGE generated due to the presence of the VALUE clause. Admittedly, the accurate recovery of unreferenced WORKING STORAGE items is of minor value in so far as the correct functioning of the program is concerned, but there are occasions when this type of recovery serves a purpose. One such occasion is the determination of the presence of

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

COPYBOOKs that are standard for the application that the program forms a part of. When these are found, possibly large amounts of FILLER code can be replaced by COPYBOOKs that the application programmers are familiar with thus making the recovered source code much easier to maintain.

Aside from WORKING STORAGE values; we can examine and analyze DCBs and VSAM file control blocks to assist in recovering ASSIGN and FD clauses; the TGT SWITCH to determine some of the compile options in effect during the original compile; as well as program and DISPLAY literals.

Figure 8 illustrates an example utilizing the constants generated in WORKING STORAGE to recover data items that are not referenced and therefore not amenable to operand analysis. Had the data items been referenced the example would be even more germane since the VALUE clauses are recovered by this type of examination.

```
         DC    CL32'START OF ABEND CODES'
WS010220 DC    XL4'00'
         DC    XL4'000003E8'
         DC    XL4'000003E9'
         DC    XL4'000003EA'
         DC    XL4'000003EB'
         DC    XL4'000003EC'
         DC    XL4'000003ED'
         DC    XL4'000003EE'
         DC    XL4'000003EF'
         DC    XL4'000003F0'
         DC    XL4'000003F1'
         DC    XL4'000003F2'
WS010250 DC    XL4'000003F3'
WS010254 DC    XL4'000003F4'
WS010258 DC    XL4'000003F5'
WS01025C DC    XL4'000003F6'
         DC    XL4'000003F7'
WS010264 DC    XL4'000003F8'
```

Figure 8: Example of Disassembled WORKING STORAGE

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

If the original programmer of the source code happened to insert helpful hints of comments, such as "START OF ..." embedded in WORKING STORAGE the recovery becomes somewhat easier. But even without this obvious clue an experienced recovery technician would have deduced the PIC S9(09) COMP VALUE structure of the subsequent data items given their ascending values and the impossibility of the values being generated for PIC 9 or PIC 9 COMP-3 data items.

Figure 9 illustrates the recovered source code for the same WORKING STORAGE shown above.

```
01  WS-ALIGN-0200.
    05  FILLER    PIC  X(32)      VALUE 'START OF ABEND CODES'.
    05  WS010220  PIC S9(09) COMP VALUE    +0.
    05  FILLER    PIC S9(09) COMP VALUE +1000.
    05  FILLER    PIC S9(09) COMP VALUE +1001.
    05  FILLER    PIC S9(09) COMP VALUE +1002.
    05  FILLER    PIC S9(09) COMP VALUE +1003.
    05  FILLER    PIC S9(09) COMP VALUE +1004.
    05  FILLER    PIC S9(09) COMP VALUE +1005.
    05  FILLER    PIC S9(09) COMP VALUE +1006.
    05  FILLER    PIC S9(09) COMP VALUE +1007.
    05  FILLER    PIC S9(09) COMP VALUE +1008.
    05  FILLER    PIC S9(09) COMP VALUE +1009.
    05  FILLER    PIC S9(09) COMP VALUE +1010.
    05  WS010250  PIC S9(09) COMP VALUE +1011.
    05  WS010254  PIC S9(09) COMP VALUE +1012.
    05  WS010258  PIC S9(09) COMP VALUE +1013.
    05  WS01025C  PIC S9(09) COMP VALUE +1014.
    05  FILLER    PIC S9(09) COMP VALUE +1015.
    05  WS010264  PIC S9(09) COMP VALUE +1016.
```

Figure 9: Example of Recovered Source for WORKING STORAGE Definition

Supporting Information

The previous example leads directly into the discussion of the final concept of the decompile process, namely the use of supporting information.

Supporting information consists mainly of the source code for COPYBOOKs and related programs along with JCL, file layouts, and program or application hardcopy specifications or other documentation. Supporting information generally allows the recovery technician to provide data names and paragraph names that are either the original names or are at least in the style of the client's programming staff. The example shown in Figure 9, in fact, was

completely recovered when we were able to match the recovered code with a COPYBOOK supplied by the client. This enabled us to replace the generic data names with the client's original data names. Another COPYBOOK match for the same recovery revealed an addition to the COPYBOOK after the program was coded. COPYBOOK matches will sometimes also reveal deletions from the current version of the COPYBOOK that must be accounted for when the recovered source code is placed back into the application development source code library.

Supporting information is not always directly related to source code or the execution of the program. Occasionally, the business the client is engaged in provides clues to the meaning and uses of sections of code or data items that appear in recovered source code.

Providing original or meaningful names in the recovered source code *has no impact on the accuracy of the recovery* but undoubtedly is of great importance to the programmer. Most programs are recovered because changes must be made and changing source code that is in generic format is not a simple task. When meaningful names are provided, the application programmers charged with making the changes have a head start.

Step 4. Validation

The last step of the source code recovery methodology is simply to prove that the source code has been recovered accurately.

In order to do this, we take the resulting recovered source code and compile it using the same version of the compiler (and compile options) that created the original executable program. The compile options which were originally specified are key for validation. These options are identified as a part of the first step of the recovery process, Delink. The newly created object module is then disassembled and compared to the disassembly of the original executable. While a 100% identical object module is seldom expected, the technician must compare the original object to the newly created object and verify the resulting program is "functionally equivalent.

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250

Summary

The process of recovering source code from object code is a formal process that results in source code that is guaranteed to be 100 percent functionally equivalent to the original code. Regardless of the original COBOL dialect; the options under which the program was compiled; or the operating system that the application is running on, the pattern description language, pattern compiler, and decompiler form the basis for detecting and recreating the corresponding source code.

While missing source code was the original impetus for the development of this methodology, other uses have subsequently been derived. Source code currency (also called Version Matching) is the process of validating that a given version of source code does in fact generate a given load module. Likewise, taking an older version of source code and modifying it necessary to produce a given version of an executable (termed “source code reconciliation”) can be a beneficial byproduct of this methodology.

Note: This white paper is derived from a transcript of a speech originally delivered by Fred Brandes at Share in September, 1998. The technology resulting from this process is called, ReSource™, a patented product marketed by Essential Systems Technical Consulting. For more information, please call 770 479-2250.

Copyright © 2000, Essential Systems Technical Consulting ▪ 5020 Hickory Flat Highway ▪ Canton, GA 30115
770 479-2250