Control Flow Normalization for COBOL/CICS Legacy Systems

Mark van den Brand, Alex Sellink, Chris Verhoef *

University of Amsterdam, Programming Research Group
Kruislaan 403, NL-1098 SJ Amsterdam, The Netherlands

markvdb@wins.uva.nl, alex@wins.uva.nl, x@wins.uva.nl

Abstract

We propose a practical incremental approach to perform control flow normalization of COBOL/CICS legacy systems using a software renovation factory. The normalization improves the maintainability of such systems. A consequence of our normalization is that we reengineer the interface so that such systems can be accessed via Intranet or Internet. Moreover, the performance of such systems is improved. We applied our approach to a mortgage system written in COBOL and CICS.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring;
Additional Key Words and Phrases: Reengineering, System renovation, Interface reengineering, Control flow normalization, Language migration, COBOL, CICS.

1 Introduction

Flow analysis is a fundamental prerequisite for many important types of code improvement [14]. Indeed, many proposals for control flow normalization algorithms have been reported in the literature. Some of them make use of code duplication while others use complicated technology to prevent this as much as possible [2]. What they all share is that they globally eliminate unstructured code. It is our experience that in practice it is not desirable to use such algorithms for our purpose. This is due to the fact that the type of code improvement that we strive for lies in increasing maintainability of the code, whereas the goal of the proposed algorithms is to improve performance of the code. Another difference with such algorithms is that the output of their control flow normalization affects all the so-called unstructured code. We do not want this. In practice, code containing certain types of jump instructions can be quite structured, for example due to systematic use of naming conventions, whereas the normalized code could be far less structured from a viewpoint of maintainability. Therefore, we propose in this paper an approach where we have complete control over the parts of the code that we want to restructure in order to improve their maintainability and leave other parts untouched that are not unstructured in the practical sense. We propose to use pattern recognition and program transformations to perform partial control flow normalizations on parts of the code that have to be restructured while leaving other parts of the code untouched.

We use a software renovation factory approach to implement the control flow normalization of a mortgage system that is in both use at Postbank, a bank, and ABP, a pension fund for civil servants. The mortgage system is implemented in the COBOL85 dialect VS COBOL II and CICS/ESA for MVS/ESA. COBOL stands for *common business oriented language*, VS stands for *virtual system* and the II indicates that it is a dialect of ANSI COBOL85 [1]. CICS stands for *customer information control system*, it is an application server that provides online transaction processing and transaction management for mission-critical applications. MVS/ESA is an indication for the hardware platform.

The combination of COBOL and CICS is quite natural: COBOL takes care of the batch part of the system and CICS deals with the interactive part. This means that reengineering the interface for such legacy systems amounts to reengineering its CICS part. This has been recognized by many vendors: IBM developed for COBOL/CICS systems a Java to CICS gateway. In the brochure we read that the CICS Internet gateway provides an interface between a Web server and a CICS application and data, allowing conversion of 3270 data streams into HyperText Markup Language format used by the World Wide Web. The CICS application can then be accessed by any Web browser with no change to either the browser or the CICS application. From people at IBM we heard that this is almost true: the CICS Internet gateway does not support handlers in CICS [23]. We eliminate such handlers since they contain (implicit) jump instructions. Consequently, after restructuring the CICS part of such a system, its interface is ready for connection to the Intranet of the company or the Internet for direct customer interaction. So, the interface is reengineered as a byproduct. We believe this to be an important practical aspect of our work.

Another consequence of our restructuring is that the performance of the system improves. Namely, when using the CICS handlers this implies that internal memory will be allocated to create a table that can be accessed by other CICS statements to perform the specified control-flow. Moreover, when the control flow goes to another program via the CICS `LINK` statement, the contents of this table are stored in case

*Chris Verhoef was supported by the Netherlands Computer Science Research Foundation (SION) with financial support from the Netherlands Organization for Scientific Research (NWO), project *Interactive tools for program understanding*, 612-33-002.

the control will return to the original program. So in fact when avoiding handlers there is no need for allocating this memory. the handlers have systemwide impact. Since the system uses about 550 such statements, their elimination will improve the performance.

Scope The scope of this paper is purely technical. Given a reengineering problem we explain how we solve it using a factory approach. There is no space left to discuss the peopleware side of this paper. For more architectural views of our approach we refer to [9] for generative matters. We refer to [7] for an overview of core technology we use for system renovation. We refer to [19] for the component-based approach we use, and to [3] for an organizational view on component factories that is complementary to [19].

1.1 Related Work

In [13] we can find a control-flow normalization tool for COBOL74 developed using the TAMPR system [5] – this is a general purpose program transformation system. The control flow normalization is performed by means of translating a COBOL program into an intermediate language on which again transformations can be performed.

The components that we discuss in this paper can presumably also be implemented using, for instance, Software Refinery [21]. However, our generative approach reported on in [9] is new as far as we know. Also our generalized LR parsing technology is new as far as we know; see [8] for elaborate discussions on parsing technology in reengineering. Reasoning and Semantic Designs are working on *GLR* paring as well.

Sneed [24] uses wrapping technology to connect a legacy system to Internet/Intranet. Since it is our purpose to improve maintainability, his approach does not apply to our case.

1.2 Organization of the Paper

In Section 2 we discuss control flow mechanisms specific for COBOL. In Section 3, we provide details on the used implementation methods. In Section 4 we give an overview of the process of eliminating implicit and explicit jump instructions. This is our elimination assembly line. In the subsequent sections we focus on the functionality of the elimination itself: we discuss its functionality concept by concept. So, in Section 5 we discuss the restructuring of while constructs, in Section 6 we discuss restructuring of conditional constructs. Then in Section 7 we discuss the control flow of CICS and in Section 8 we address the restructuring of CICS. Then, we explain in Section 9 how to treat comments in the code while transforming code. Finally, we conclude in Section 10.

2 Control Flow in COBOL

We discuss some typical control flow mechanisms in COBOL programs. We have two types of jump instructions in COBOL/CICS systems: explicit and implicit ones. The explicit ones are `GO TO` statements. We discuss implicit ones in Section 7. In this section we will explain the normal use of explicit jump instructions that occur in the mortgage system.

Already in the early days COBOL contained a `GO TO` statement. It has been used to simulate while constructs, conditional constructs, procedure calls, exit statements and more constructions that are nowadays standard in many programming languages, including COBOL85 dialects. In code that we inspected, we found indeed many occurrences of simulations and not arbitrary use of `GO TO` statements. Since this was the case we estimated that it should be possible to use patterns to locate those simulated constructs and to replace them with more natural COBOL code that is available in more modern versions of COBOL. In this way the control flow of the code becomes more clear for the maintainers of the code. In the subsequent sections we will see many examples of these patterns with their replacement pattern in which the `GO TO` statements are eliminated. We wish to stress that the replacement patterns are just one of many possibilities. What is the best solution depends on the demands of the people owning the code. In this paper we show *that* such a restructuring can be carried out automatically and that we have complete control over the precise form of the replacement pattern.

3 Implementing a Software Renovation Factory

A software renovation factory consists of a number of assembly lines. An assembly line is a modular processing unit that performs a number of tasks in a fixed order. An assembly line usually starts with parsing the input resulting in an annotated abstract syntax tree (also called a code base). The modular processes are conditional transformations on the code base. Then the code base is turned into a program listing during unparsing. We generate GLR parsers [22], unparsers [11] and generic conditional transformations [9] from the grammar of the code that has to be renovated [10].

Next we describe the framework we used for our implementation. First, we briefly discuss the development system that we use, the ASF+SDF Meta-Environment, and its two accompanying formalisms, ASF and SDF. Then we focus on the software renovation factory we are implementing. For more details on ASF+SDF we refer to [4, 15] and for more details on the ASF+SDF Meta-Environment we recommend [18].

3.1 The Implementation System

ASF+SDF is a modular algebraic specification formalism for the definition of syntax and semantics of (programming) languages. It is a combination of two formalisms ASF (Algebraic Specification Formalism [4]), and SDF (Syntax Definition Formalism [15]). The ASF+SDF formalism is supported by an interactive programming environment, the ASF+SDF Meta-environment [18]. This system is called *meta-environment* because it supports the design and development of programming environments. ASF+SDF can not only be used for the formal definition of a variety of (programming) languages but also for the specification of software engineering problems in diverse areas. See [6] for details on industrial applications.

ASF is based on the notion of a module consisting of a signature defining the abstract syntax of functions and a set of conditional equations defining their semantics. SDF allows the definition of concrete (i.e., lexical and context-free) syntax. Abstract syntax is automatically derived from the concrete syntax rules.

ASF+SDF specifications can be executed by interpreting the equations as conditional rewrite rules or by compilation to C. For more information on conditional rewrite systems we refer to [20] and [17]. It is also possible to regard the ASF+SDF specification as a *formal* specification and to implement the described functionality in some programming language.

3.2 A Factory Approach

We use the ASF+SDF Meta-Environment to develop a software renovation factory. This factory consists of several parts. First, we used the ASF+SDF Meta-Environment to specify a grammar of the code that had to be reengineered. We used the methods described in [10] to construct a grammar that understands both the VS COBOL II code and the embedded CICS code. This method is modular, which implies that we can reuse parts of grammars that were developed before. For the VS COBOL II grammar we could reuse large parts of a COBOL/370 grammar. We had to construct the CICS grammar from scratch. It took two hours to construct it using the source code; to integrate it with the VS COBOL II module; and to test it on all the CICS constructs in the system.

Another part was to develop a structured way to construct components for a software renovation factory. The components should be easy to make, maintainable, and should be reusable for other dialects, as well. We used the ASF+SDF Meta-Environment to develop a method described in [9] where from an arbitrary context-free grammar we can generate components with such properties. We use this generative technology to obtain the necessary transformations for the control flow normalizations to restructure the mortgage system. It was no effort to generate the generic components that we will instantiate in this paper for implementing our particular reengineering task.

Using the above generative technology, we develop with the ASF+SDF Meta-Environment useful components for a software renovation factory. In the next section we give an overview of the elimination assembly line we constructed.

4 The Elimination Assembly Line

In this section we will give an overview of the components that we constructed in the assembly line that eliminates jump instructions. We note that all the components are constructed using the technology that is discussed in [9]. We use a running example to clarify the various treatments on the code. We discuss the components pointwise.

• `aei` This stands for add `END-IF`. This component is part of the pretreatment phase of the raw material (the original code). All the conditional statements that still use implicit scope terminators like a separator period or a higher `ELSE` are transformed so that they will be ended by the explicit scope terminator `END-IF`. This makes the code uniform with respect to conditional statements. Consequently, the number of patterns for eliminating jump instructions decreases, which is important for the performance of the components invoked after `aei` and for keeping their construction as simple as possible. We stress that the use of `aei` is not essential, it is just sensible to use it. We refer to [9] for more information on `aei` where we discuss this component in detail. Below we see an original code fragment at the left-hand side and at the right-hand side the output of `aei`.

```
PAR-1.                   PAR-1.
  IF X > 1 GO PAR-2.       IF X > 1 GO PAR-2 END-IF.
  DISPLAY 'X'.             DISPLAY 'X'.
  DISPLAY 'Y'.             DISPLAY 'Y'.
PAR-2.                   PAR-2.
```

• `gte` This stands for `GO TO` eliminator. This component is part of the main phase of the treatment of the code. Here the jump instructions are eliminated. We will discuss this component in detail in the subsequent sections. In this stage it is enough to display its behaviour on the pretreated code:

```
PAR-1.              PAR-1.
  IF X > 1            IF X > 1
    GO PAR-2            CONTINUE
  END-IF.             ELSE
  DISPLAY 'X'.          DISPLAY 'X'
  DISPLAY 'Y'.          DISPLAY 'Y'
PAR-2.                END-IF.
                    PAR-2.
```

• `rsp` This is an abbreviation for remove separator period. Note that in the above `gte`-output the separator periods after the `DISPLAY` statements are gone. If it was still there it would end the `IF` at that place which is incorrect, since then the second `DISPLAY` would be unconditionally executed. Apart from that, a separator period would lead to an `END-IF` without an `IF`, which is incorrect coding anyway. We note that when we do not pretreat the code with `aei`, the presence of the separator periods would have lead to syntactic correct code that has a different semantics than the original code, which is undesirable. The `rsp` component is used in the patterns of `gte`. We will see this when we discuss `gte`.

• `act` This means add `CONTINUE` transformation. Note that in the above output of `gte` we see a `CONTINUE`. This statement has no effect on the execution of a program. Its use is to prevent empty bodies of, for instance, an `IF` statement. Only when `gte` removes a complete body the auxiliary component `act` will substitute a `CONTINUE`. This is the case in the running example, since only a `GO TO` statement is present at the left-hand side `IF` statement. Also `rsp` sometimes creates a `CONTINUE`: if it has to remove a separator period on an empty body.

• `ect` This stands for eliminate `CONTINUE` transformation. Of course, it is not desirable to have code containing `CONTINUE` statements that are not necessary. Therefore, in the finishing phase of the restructuring they will be removed. We stress that we work in this way since otherwise `gte` needs many more patterns to perform the jump instruction elimination. It is much cheaper to add a `CONTINUE` on some places and remove the redundant ones later on in one postprocessing step. For, in many other components of a software renovation factory we will use similar techniques to keep the number of patterns as low as possible. We give the output of the `ect` component:

```
PAR-1.                 PAR-1.
  IF X > 1               IF NOT X > 1
    CONTINUE               DISPLAY 'X'
  ELSE                     DISPLAY 'Y'
    DISPLAY 'X'          END-IF.
    DISPLAY 'Y'        PAR-2.
  END-IF.
PAR-2.
```

• `cnt` This is short for conditional normalization transformation. In the above output we see that the `ect` component changed the condition. This kind of change is also common when restructuring code. To finish the restructuring completely, the `cnt` component evaluates the conditions in, for instance, `IF` statements and represents the conditions in their most natural form. We give the output of the `cnt` component, which gives us the final output of the assembly line that takes care of the elimination of jump instructions.

```
PAR-1.                 PAR-1.
  IF NOT X > 1           IF X <= 1
    DISPLAY 'X'            DISPLAY 'X'
    DISPLAY 'Y'            DISPLAY 'Y'
  END-IF.                END-IF.
PAR-2.                 PAR-2.
```

5 Restructuring While Constructs

From here onwards we will focus on the `gte` component. We recall that `gte` stands for `GO TO` eliminator. In this section we describe the while part of `gte`. The displays to follow contain two code patterns where the left-hand side is the original pattern and the right-hand side is the replacement pattern. Those original patterns match real code found in the mortgage system like the fragments that we discussed above. In the replacement patterns we will see the presence of the auxiliary components that we discussed before.

In COBOL74 dialects there was no while construct available. So in many programs this was simulated with a `GO TO`. Below we depicted a simple pattern simulating a while loop that we found in the code. The right-hand side is its semantically equivalent pattern in a COBOL85 dialect.

```
B-exp1.                  B-exp1.
  IF L-Exp                 PERFORM UNTIL NOT L-exp
     Stat+                   Stat+
     GO TO B-exp1          END-PERFORM.
  END-IF.
```

We explain the symbols that we use in the patterns. `B-exp1` is a variable of the type basic expression, which represents in this case a paragraph label, like `PAR-1`. The variable `L-Exp` represents a logical expression in COBOL, like `X > 1`, and `Stat` is a variable matching a COBOL statement, like `DISPLAY 'X'`. The `+` is used to denote one or more occurrences of the preceding item, for example, `Stat+` matches `DISPLAY 'X' DISPLAY 'Y'`. We will use `*` to denote zero or more occurrences. The left-hand side of the above pattern is a simulated while construction. In COBOL85 dialects this is represented by a so-called in-line `PERFORM` statement. In order to eliminate the `GO TO`, we swap the condition and then we are done (we invoke the auxiliary component `cnt` later on to normalize the conditions).

Since there was no existing program construction for while statements, it is not surprising that we found more than one method of implementing them in the code. Below we display a pattern of a while construct that has special behaviour at the first loop. The right-hand side pattern provides a solution for eliminating the `GO TO`. We note that this is a global pattern: since we wish to make changes on more than one location in the program the pattern below matches an entire COBOL program.

```
Ident-div1                 Ident-div1
Env-div1                   Env-div1
DATA DIVISION.             DATA DIVISION.
File-sec1                  File-sec1
WORKING-STORAGE SECTION.   WORKING-STORAGE SECTION.
Data-desc1*                Data-desc1*
Link-sec1                  01 FIRST-LOOP PIC X(5).
PROCEDURE DIVISION         Link-sec1
Using1.                    PROCEDURE DIVISION
Decl1*                     Using1.
Paragraph1*                Decl1*
Section1*                  Paragraph1*
B-exp1 SECTION.            Section1*
Paragraph2*                B-exp1 SECTION.
B-exp2.                    Paragraph2*
  Sentence1*               B-exp2.
  Stat1*                     MOVE 'true' TO FIRST-LOOP
  IF L-exp1                  PERFORM TEST AFTER UNTIL L-exp1
    Stat1+                     IF FIRST-LOOP = 'false'
  ELSE                           act(Stat2*)
    Stat2*                     END-IF
    GO B-exp2                  rsp(Sentence1*)
  END-IF                       Stat1*
  Stat3*.                      MOVE 'false' TO FIRST-LOOP
  Sentence2*                 END-PERFORM
Paragraph3*                  Stat1+
Section2*                    Stat3*.
                             Sentence2*
                           Paragraph3*
                           Section2*
```

We discuss the notation used above. Strings containing a number are variables. A COBOL program consists of four divisions. `Ident-div1` is a variable matching the entire `IDENTIFICATION DIVISION` of a COBOL program. `Env-div1` matches the entire `ENVIRONMENT DIVISION`. The `DATA DIVISION` is in the pattern unfolded, since we wish to make a change in its `WORKING-STORAGE SECTION`. The variable `File-sec1` matches the entire `FILE SECTION`. The variable `Data-desc1*` matches zero or more records in the `WORKING-STORAGE SECTION`. The variable `Link-sec1` matches the entire `LINKAGE SECTION`. Then we enter the

PROCEDURE DIVISION. The Using1 variable matches the optional presence of the USING phrase in the procedure division header. The variable Decl1* matches the possible presence of declarative procedures. Then we enter zero or more paragraphs indicated by the variable Paragraph1* followed by zero or more sections. Then we enter a section containing our pattern. This is expressed by B-exp1 SECTION, where the variable is the name of the section. Then zero or more paragraphs and then the paragraph named B-exp2 containing pattern we wish to restructure. Sentence1* matches zero or more occurrences of type sentence, like DISPLAY 'X'. DISPLAY 'Y'.. Stat1* matches zero or more COBOL statements. One of the sentences in the pattern contains the special pattern we are looking for. We unfolded this sentence in the pattern as general as possible. It consists of zero or more statements, Stat1*. This is followed by a specific simulated while: if L-exp1 is not true we execute Stat2*, Sentence1*, and Stat1* again, until L-exp1 is true. After the END-IF we end this sentence with zero or more statements Stat3* followed by a separator period. Note that if L-exp1 is true we execute Stat1+, Stat3*, and Sentence2*. Note that we finish the pattern with Paragraph3* and Section2* since we match an entire program.

A possible control flow normalization is to use a flag to indicate whether or not we are doing the first loop. This means that we have to introduce a fresh variable. Therefore, we match on entire programs. In the replacement pattern, we introduce a fresh variable FIRST-LOOP at the appropriate location. Introducing a fresh variable has the advantage that there is no code duplication, which improves maintainability in our opinion. We initially set FIRST-LOOP to true. Then we execute Sentence1* and Stat1* since the first IF fails and L-exp1 is tested after the first loop has ended; this is expressed in the TEST AFTER option of the PERFORM statement. Then we set our fresh variable FIRST-LOOP to false so that we execute Stat2* and Sentence1* and Stat1* again, until L-exp1 is true. We continue with Stat1+, Stat3*, and Sentence2*. Note that act and rsp are invoked at the appropriate locations (cf. Section 4).

Next, we discuss another global pattern based on code that we found in the mortgage system. It already contains a PERFORM and it uses a GO TO statement to implement special behaviour at the last time the loop is executed. Since we use again a fresh variable the pattern is again program oriented. To focus the discussion on the parts in the pattern that do change, we elided the parts that remain the same in the original and replacement patterns. We used [...] to denote elisions in the patterns. We note that the elided parts coincide with the spelled out parts in the first program oriented pattern.

```
                                 [...]
                                 01 LAST-LOOP PIC X(5).
                                 [...]
                                   B-exp2.
                                   Sentence1*
                                   Stat1*
[...]                              MOVE 'false' TO LAST-LOOP
B-exp2.                            PERFORM Vary1 Test1 UNTIL
  Sentence1*                           L-exp1 OR LAST-LOOP = 'true'
  Stat1*                             Stat2*
  PERFORM Vary1 Test1                IF L-exp2
      UNTIL L-exp1                     Stat3*
    Stat2*                             MOVE 'true' TO LAST-LOOP
    IF L-exp2                        ELSE
      Stat3*                           act(Stat4*)
      GO B-exp3                      END-IF
    END-IF                         END-PERFORM
    Stat4*                         IF LAST-LOOP = 'false'
  END-PERFORM                        act(Stat5*)
  Stat5*.                            rsp(Sentence2*)
  Sentence2*                       END-IF.
B-exp3.                          B-exp3.
[...]                            [...]
```

We discuss notations we have not met before. The variables Vary1 and Test1 match the optional presence of special cases of a PERFORM statement: VARYING FROM BY and WITH TEST BEFORE or AFTER options, see [1] for more information. If they are present we use them in the replacement pattern as well. In the above pattern there is a jump to the next paragraph B-exp3. We can exit the PERFORM containing this jump in two ways: either it naturally ends and we continue with the possible statements below, or we end it due to the jump instruction and go directly to paragraph B-exp3. So, the code directly following the END-PERFORM in the left-hand side is in fact conditional code. This is apparent in the right-hand side: depending on whether it was indeed the last loop the code following the END-PERFORM will be executed. We use the switch to decide on that and to prevent code duplication. The components act, and rsp are in the replacement pattern for reasons already discussed.

6 Restructuring Conditional Constructs

Although right from the beginning there was a conditional construct available in COBOL we found many occurrences of conditional code that was simulated using both conditional constructs and GO TO statements. It appeared that this was in accordance with the programming style at that time. This style lead to many jumps to the next paragraph. We will discuss three such patterns with their possible replacement pattern.

```
Paragraph1*       Paragraph1*
B-exp1.           B-exp1.
  Sentence1*        Sentence1*
  Stat1*            Stat1*
  IF L-exp1         IF L-exp1
    Stat2*            act(Stat2*)
    GO B-exp2       ELSE
  END-IF              act(Stat3*)
  Stat3*.             rsp(Sentence2*)
  Sentence2*        END-IF.
B-exp2.           B-exp2.
  Sentence3*        Sentence3*
```

We see the pattern starting with with zero or more paragraphs and then a paragraph split up in sentences and a special sentence containing an IF with a jump instruction to the next paragraph. In the right-hand side we see that the GO TO is eliminated in the IF. Since the GO TO jumps to the next paragraph, it means that all the code below the END-IF in the left-hand part is executed depending on the condition L-exp1. So it is in fact just conditional code.

Therefore, this code is now in the ELSE branch. The components act, and rsp are in the replacement pattern for reasons already discussed.

```
Paragraph1*         Paragraph1*
B-exp1.             B-exp1.
  Sentence1*          Sentence1*
  Stat1*              Stat1*
  IF L-exp1           IF L-exp1
    Stat2*              act(Stat2*)
    GO B-exp2         ELSE
  ELSE                  Stat3+
    Stat3+              Stat4*
  END-IF                rsp(Sentence2*)
  Stat4*.             END-IF.
  Sentence2*        B-exp2.
B-exp2.               Sentence3*
  Sentence3*
```

The above pattern differs from the previous one since it has an ELSE branch. It is treated analogously to the one above: the GO TO is removed and the END-IF is replaced to the lowest possible place.

```
                     [...]
                     01 BOTH-TRUE PIC X(5).
                     [...]
                     B-exp2.
                       Sentence1*
                       Stat1*
                       MOVE 'true' TO BOTH-TRUE
                       IF L-exp1
[...]                    Stat2*
B-exp2.                  IF L-exp2
  Sentence1*               act(Stat3*)
  Stat1*                 ELSE
  IF L-exp1                Stat4*
    Stat2*                 MOVE 'false' TO BOTH-TRUE
    IF L-exp2            END-IF
      Stat3*           ELSE
      GO B-exp3          MOVE 'false' TO BOTH-TRUE
    END-IF             END-IF
    Stat4*             IF BOTH-TRUE = 'false'
  END-IF                 act(Stat5*)
  Stat5*.                rsp(Sentence2*)
  Sentence2*           END-IF.
B-exp3.              B-exp3.
[...]                [...]
```

This pattern contains a jump statement in a nested conditional. Analysis of the original pattern learns that the code under the END-IF is executed in three out of four cases: only if both conditions L-exp1 and L-exp2 are false. A way to deal with such asymmetric code is to use a switch variable BOTH-TRUE like we did before. It is not hard to see that the left- and right-hand side are semantically equivalent.

7 Control Flow in CICS

We discuss some some typical control flow mechanisms in COBOL/CICS programs. They not only contain explicit jumps but also implicit jumps. The implicit ones are hidden jump instructions in CICS statements. In this section we will explain the normal use of implicit jump instructions that we found in the mortgage system.

The CICS concept was introduced to take care of the interactive part of the software system. Embedded CICS code is more readable than if it were directly written in COBOL. Moreover, a large library of predefined functions can be used within CICS. One of the problems with CICS is that CICS statements can take over the flow control of a COBOL program. More specifically, some of the CICS statements contain implicit GO TO statements. Their scope is global which means that they influence the control flow for all the subsequent CICS code depending on their exit status (unless in the CICS command an explicit NOHANDLE is specified, which was nowhere the case in the mortgage system). As a consequence, the control flow of a COBOL/CICS system becomes completely unclear. So elimination of those implicit GO TO statements was one of the targets in the restructuring of the system. The mortgage system contains over 600 implicit GO TO statements. Some of them are in a copybook (an include file) that is in 9 % of the files the beginning statement of a program. Needless to say that the control flow of this system is obfuscated.

It is not a coincidence that COBOL analysis tools are enhanced to also analyze the trace logic of CICS statements. For instance, release 6.4 of Compuware's automated analysis and documentation tool PATHVU for OS/VS COBOL and VS COBOL II systems has this option.

Next we explain what an implicit GO TO statement is. Below we depicted a CICS statement that handles the control flow of all subsequent CICS statements.

```
EXEC CICS HANDLE CONDITION
          QIDERR (PAR-1)
END-EXEC.
```

EXEC CICS tells the CICS preprocessor that embedded CICS should be translated into COBOL. The functionality of the above CICS statement is that if there is an error in the name of a (temporary) queue that is accessed by another CICS statement (following the above one) the control flow will go to a paragraph named PAR-1. We can make this implicit GO TO explicit by giving the output of the CICS preprocessor below:

```
MOVE '01261' TO DFHEIV0
CALL 'DFHEI1' USING DFHEIV0
SERVICE LABEL
GO TO PAR-1 DEPENDING ON DFHEIGDI.
```

We note that this code is generated by the CICS preprocessor (module DFHECP1$), which is not for human inspection. We give the output with the *only* purpose to show that indeed a GO TO statement is generated by the CICS preprocessor. We briefly explain the code. In Appendix D of [16] we can read that each CICS command is replaced by one or more MOVE statements followed by a COBOL CALL statement. This is almost true, since we not only have a CALL to a CICS library program, but we also have statements following the CALL. Note that the generated string '01261' is moved to the generated variable DFHEIV0, which serves as a parameter for the library program DFHEI1. The SERVICE LABEL statement is a compiler-directing statement generated by the CICS preprocessor to indicate control flow (namely, the next statement is a GO TO). It is not intended for general use, so normally not seen in COBOL source code. Finally, we find a GO TO DEPENDING ON statement that jumps to PAR-1 depending on the value of a the CICS reserved variable DFHEIGDI. Due to this CICS statement *all* the subsequent CICS statements may take over the control flow in COBOL, depending on their exit status. We note that DFH is not an acronym, but an IBM code for their CICS product.

The global scope of a conditional CICS statement can cause undesired looping behaviour of the system. In case of the mortgage system we know that these errors can occur in the system. The reason that they occur is as follows. The condition is mostly meant for the subsequent CICS statement, but not for the CICS statement following it. However, an error in such a statement will lead to a jump to the wrong paragraph. Then either, wrong code is executed, or the systems starts looping. In the first case this can lead to undesired erasure of data or the task will be terminated due to looping behaviour. Our restructuring also solves this problem since by eliminating the conditional CICS statements, we also eliminate their global scope.

For the sake of clarity, we stress that it is *not* our intention to restructure preprocessed CICS code, but to restructure the original CICS code. We address this issue in the next section.

8 Restructuring CICS

In this section we will make a start with the explanation of how to eliminate implicit jump instructions. The patterns are too involved to treat them all in this paper. However, since the issue is important and strongly related to control flow normalization we will treat an intricate pattern to give the reader an idea of the complications.

Let us first give a typical input and output code fragment, and then the patterns that take care of the automatic transformation of the code.

```
  EXEC CICS HANDLE CONDITION
    ITEMERR (PAR-2)
    QIDERR  (PAR-2)
  END-EXEC.
  EXEC CICS READQ TS
    QUEUE  (A)
    INTO   (B)
    LENGTH (C)
    ITEM   (D)
  END-EXEC.
  GO PAR-3.
PAR-2.
  MOVE X TO Y.
PAR-3.
  MOVE Z TO T.
```

```
  EXEC CICS READQ TS
    QUEUE  (A)
    INTO   (B)
    LENGTH (C)
    ITEM   (D)
    NOHANDLE
  END-EXEC.
  EVALUATE EIBRESP
    WHEN DFHRESP(ITEMERR)
        OR DFHRESP(QIDERR)
      MOVE X TO Y
    WHEN NOT DFHRESP(NORMAL)
      CALL ABEND-PROG
  END-EVALUATE.
PAR-3.
  MOVE Z TO T.
```

The above code fragment on the left side should be read as follows. The first CICS statement is telling the second CICS statement to jump to PAR-2 whenever there is an error in the name of the queue that it reads or that the given item number is outside the range of the queue. Then the actual CICS statement reads the queue from temporary storage (TS). If there is no mistake the GO PAR-3 is executed. If there was one of the above errors the control flow will go to PAR-2 and then via fall through to PAR-3. So, the code in PAR-2 is conditional code depending on the exit status of the READQ TS statement. In the right-hand side the code is restructured. The HANDLE CONDITION is eliminated. The READQ TS had an extra option NOHANDLE indicating that it is not permitted to listen to other HANDLE CONDITION statements. The explicit GO PAR-3 is removed plus the PAR-2 label. To take care of the code in PAR-2 a case statement is introduced. In this EVALUATE the two cases considering the errors when reading a queue are treated, plus an extra default action when there is a mistake other than those two. Note that this is extra functionality that is added due to an earlier eliminated HANDLE CONDITION that took globally care of error handling in all the programs. This is restructured to calling a program ABEND-PROGRAM directly after a CICS statement if another error occurs while executing it. ABEND-PROGRAM handles the abnormal end of the CICS statement. The EXEC interface block (EIB) is a data area that contains the field EIBRESP, among many others. The CICS preprocessor modifies the LINKAGE SECTION by inserting the EIB structure as the first parameter. So, for this reason we can use EIBRESP in a COBOL program without declaring it. The variable EIBRESP contains a number between 0 and 94 indicating the exit status of the last executed CICS statement. So, for instance, if a QIDERR occurred the value of EIBRESP is 44, ITEMERR returns 26 and NORMAL returns 0. The built-in function DFHRESP tests the value of the EIBRESP subfield. In this way we can construct the cases in the EVALUATE. We recall that DFH is an IBM code for their CICS product.

We note that the scope of every HANDLE CONDITION is global, which means that all subsequent CICS statements in the program will jump to PAR-2 if one of the errors occurs. In this case it was not the intention to have a global scope, on the contrary, in fact a CICS RESET command should have been implemented to end the scope. This did not happen, and therefore this construction has lead to undesirable looping behaviour. Our restructuring solves that problem as we mentioned in Section 7. For more details on CICS in general we refer to [16]. Below we give the pattern that covers the above situation.

```
Paragraph1*
B-exp1.
  Sentence1*
  Stat1*
  EXEC CICS HANDLE CONDITION
    Cics-opt1*
  END-EXEC
  Stat2*.
  Sentence2*
  Stat3*
  EXEC CICS Cics-command1
    Cics-opt2*
  END-EXEC
  GO B-exp4.
B-exp2.
  Stat1+.
B-exp4.
  Sentence3*
Paragraph3*
```

```
Paragraph1*
B-exp1.
  Sentence1*
  Stat1*
  Stat2*.
  Sentence2*
  Stat3*
  EXEC CICS Cics-command1
    Cics-opt2*
    NOHANDLE
  END-EXEC
  EVALUATE EIBRESP
    c2e(Cics-opt1*,Stat1+)
    WHEN NOT DFHRESP(NORMAL)
      CALL ABEND-PROGRAM
  END-EVALUATE.
B-exp4.
  Sentence3*
Paragraph3*
```

Since its a multi-paragraph pattern, we start and end it with zero or more paragraphs. In one of them we will find the CICS HANDLE CONDITION statement. The variable Cics-opt1* matches all the options that are present in the code, like ITEMERR (PAR-2) and QIDERR (PAR-2).

Then some intermediate code not containing CICS follows. This will be checked as we will see below. Then another CICS statement follows. The variable `Cics-command1` denotes the kind of command, like `READQ TS`. The variable `Cics-opt2*` matches the options that accompany the command, like `QUEUE (A)` up to and including `ITEM (D)`. If there is a sensible relation between the conditional CICS statement and the second, then we can execute the transformation. We will see below how we do that, but first we explain the replacement pattern. We see that the `HANDLE CONDITION` is gone, a `NOHANDLE` is added to the list `Cics-opt2*`, and `EVALUATE` is added. The `EXEC` interface block (`EIB`) is a data area that contains the field `EIBRESP`, among many others. We use an auxiliary component `c2e` (conditions to evaluate) to construct the when clauses of the `EVALUATE` from the options in the `HANDLE CONDITION` and the conditional code `Stat1+` (`MOVE X TO Y` in the code fragment above). The actual transformation is conditional, as we already announced. We give the ASF equation that is responsible for transforming the above code fragment.

```
nhc(rsp(Stat2*. Sentence2* Stat3*.)) = true,
hcr(Cics-opt1*,Cics-command1)        = true
==========================================
hce(<left-hand-pattern>) = <right-hand-pattern>
```

We briefly explain notations. Above the line we have conditions and below the transformation. `nhc` stands for no `HANDLE CONDITION` and it is an analysis component (also mainly generated) that checks on the code between the `HANDLE CONDITION` and the next CICS statement contains another `HANDLE CONDITION`. `hcr` stands for `HANDLE CONDITION` relation. This component tests whether the options mentioned in the `HANDLE CONDITION` are related to the CICS command in such a way that the control flow can be influenced at all. In the code example above, a queue is read and the options in the `HANDLE CONDITION` are `ITEMERR` and `QIDERR`. Those options are related in the sense that those error can occur while executing the `READQ TS` command. If both conditions are true, then we can apply `hce` (`HANDLE CONDITION` eliminator). The `<left-hand-pattern>` and `<right-hand-pattern>` stand for the patterns we listed above. We will not discuss the briefly mentioned components in more detail, due to space limitations. We refer to [10] and [9] for more details on the use of ASF equations to construct components like the ones above and to [12] for a wealth of information on the use of ASF+SDF in general.

9 Restructuring Comments

When restructuring code, the comments need our attention, as well. A possibility is to consider comments to be layout, in other words, to throw away the comments while restructuring the code. This is not satisfactory in many cases, for instance, parts that are not affected by a restructuring will loose valuable comments. Therefore, we incorporated comments in the grammar so that while restructuring code, we can manipulate the original comments. Of course, some comments will be out of date after restructuring. Therefore, during a redocumentation process such comments need to be updates somehow. During the automatic restructuring we can mark such places due to the fact that the comment is part of the grammar. In the explication of the elimination assembly line we did not treat comment handling. The reason refrain from this is that this would obfuscate the elimination aspects of the restructuring. We stress that the elimination assembly line does treat comments as well. In this section we explain how.

It is not trivial to extend a grammar with a sort that can occur at virtually every spot in a program. So it is not a good idea to add comments to the grammar in an unstructured manner. We added comments in an incremental way: only when we detected comment in real code we extended the grammar with the possibility that comment can be present at that place. We found out that as soon as comment is added to a construct, it can not be added to its right-most subconstruct without causing ambiguities. Therefore, our strategy is to add them after the smallest parts in the grammar, which are the terminals. In practice we needed 43 locations after terminals where we added the sort dealing with comment. We give an example production rule in SDF style:

```
Stat+ "." COMMENT* -> Sentence
```

This means that one or more statements followed by the terminal separator period followed by zero or more comments make up a sentence in COBOL. Now a variable of type `Sentence`, matches one or more occurrences of statements; the terminal separator period; and the possible comments succeeding it. The smallest part in the grammar is here the character preceding the comment, which is the separator period. So in the patterns we have shown, comment in a sentence was included; only not visible since we did not unfold the variables containing it. The same holds for many other pieces of the grammar. However, as soon as we unfold the grammar rules by specifying a pattern in increasing detail, we will use in the end terminals instead of variables like `Sentence1`. At those locations in the pattern, we make the variables of type comment explicit. We omitted them since otherwise it would be confusing to have at some places comments but at other places not and still all comments are treated. We reiterate a pattern from `gte` but now with explicit comments added.

```
Paragraph1*             Paragraph1*
B-exp1. COMMENT1*       B-exp1. COMMENT1*
  Sentence1*              Sentence1*
  Stat1*                  Stat1*
  IF L-exp1               IF L-exp1
    Stat2*                  act(Stat2*)
    GO B-exp2             ELSE
  END-IF                    act(Stat3*)
  Stat3*. COMMENT2*         rsp(Sentence2*)
  Sentence2*              END-IF. COMMENT2*
B-exp2. COMMENT3*       B-exp2. COMMENT3*
  Sentence3*              Sentence3*
```

Since `Paragraph1` consists of sentences and they include comment as we saw above in the production rule, there is no need to add them in the pattern: they are implicitly included. For the other variables in the pattern the same holds (provided *that* we found comment in real code at that location). The more detailed a pattern becomes, the more terminals occur in it. At those locations we add comment variables to the pattern. In the above pattern we see this three times: after a separator period. In the environment of these locations the actual restructuring takes place so it is likely that something special needs to be done with them. Maybe they need to be changed, marked for inspection during a redocumentation process, erased, or moved to another location. Although this kind of manipulation takes some effort in some cases, it is not a problem using our generative approach.

Note that possible comment after `GO B-exp2` is erased automatically; in case that this comment is still important, it is possible to prevent this by specifying a more detailed pattern. All the other comments remain intact in the restructured code. Only the possible `COMMENT2*` after `Stat3*` is put at the end of the new sentence. The latter is just a choice, it could have been shipped to another location as well. Such decisions are up to the owners of the code. We only want to show that we can manipulate comments and how it works in principle.

10 Conclusions

In this paper we proposed to use a software renovation factory to perform control flow normalization that was focussed towards improving the maintainability of COBOL/CICS systems. We used a technological infrastructure that was designed to develop a software factory. We applied that technology to implement an elimination assembly line in order to execute the normalization process. We based our approach on real code: a mortgage system written in COBOL/CICS. Apart from a decrease of the maintenance problem for such systems an important consequence of the normalization is that the interface is reengineered, as well. After performing the normalization a COBOL/CICS system can be connected to Internet/Intranet without a change to either the system or to the used Web browser. This can be obtained by using existing technology: the IBM Java to CICS Internet gateway. Another improvement is that normalized systems have better performance (due to elimination of certain CICS statements). We believe that our normalization and its implications are of economic importance.

References

[1] American National Standards Institute, Inc. *Programming Language – COBOL*, ANSI X3.23–1985 edition, 1985.

[2] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, 1992.

[3] V.R. Basili, G. Caldiera, and G. Cantone. A reference architecture for the component factory. *ACM TOSEM*, 1(1):53–80, 1992.

[4] J.A. Bergstra, J. Heering, and P. Klint. The algebraic specification formalism ASF. In J.A. Bergstra, J. Heering, and P. Klint, editors, *Algebraic Specification*, ACM Press Frontier Series, pages 1–66. The ACM Press in co-operation with Addison-Wesley, 1989. Chapter 1.

[5] J.M. Boyle. A transformational component for programming language grammar. Technical Report ANL-7690, Argonne National Laboratory, Argonne, Illinois, 1970.

[6] M.G.J. van den Brand, A. van Deursen, P. Klint, S. Klusener, and E.A. van der Meulen. Industrial applications of ASF+SDF. In M. Wirsing and M. Nivat, editors, *Algebraic Methodology and Software Technology (AMAST '96)*, volume 1101 of *LNCS*, pages 9–18. Springer-Verlag, 1996.

[7] M.G.J. van den Brand, P. Klint, and C. Verhoef. Core technologies for system renovation. In K.G. Jeffery, J. Král, and M. Bartošek, editors, *SOFSEM'96: Theory and Practice of Informatics*, volume 1175 of *LNCS*, pages 235–255. Springer-Verlag, 1996.

[8] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. Technical Report P9719, University of Amsterdam, Programming Research Group, 1997. Available at http://adam.wins.uva.nl/˜x/ref/ref.html.

[9] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *proceedings of the fourth working conference on reverse engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/˜x/trans/trans.html.

[10] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Obtaining a COBOL grammar from legacy code for reengineering purposes. In M.P.A. Sellink, editor, *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications*, Electronic Workshops in Computing. Springer verlag, 1997. To appear, Available at http://adam.wins.uva.nl/˜x/coboldef/coboldef.html.

[11] M.G.J. van den Brand and E. Visser. Generation of formatters for context-free languages. *ACM Transactions on Software Engineering and Methodology*, 5:1–41, 1996.

[12] A. van Deursen, J. Heering, and P. Klint, editors. *Language Prototyping: An Algebraic Specification Approach*, volume 5 of *AMAST Series in Computing*. World Scientific Publishing Co., 1996.

[13] T. Harmer, P. McParland, and J. Boyle. Using knowledge-based transformations to reverse engineer COBOL programs. In *11th Knowledge-Based Software Engineering Conference*. IEEE-CS-Press, 1996.

[14] M. S. Hecht. *Flow Analysis of Computer Programs*. Elsevier, Amsterdam, 1977.

[15] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism SDF — Reference manual. *SIGPLAN Notices*, 24(11):43–75, 1989.

[16] IBM, Mechanicsburg, Pennsylvania, USA. *CICS/ESA Application Programming Reference*, 1992.

[17] S. Kaplan. Conditional rewrite rules. *Theoretical Computer Science*, 33(2):175–193, 1984.

[18] P. Klint. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology*, 2(2):176–201, 1993.

[19] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, Berlin, Germany, February 1998. To Appear. Available at: http://adam.wins.uva.nl/˜x/evol-se/evol-se.html.

[20] J.W. Klop. Term rewriting systems. In *Handbook of Logic in Computer Science, Volume II*, pages 1–116. Oxford University Press, 1992.

[21] Reasoning Systems, Palo Alto, California. *Refine User's Guide*, 1992.

[22] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992. ftp://ftp.cwi.nl/pub/gipe/reports/Rek92.ps.Z.

[23] J. Rekers. Consultant at IBM, the Netherlands, personal communication, July 1997.

[24] H.M. Sneed. Program interface reengineering for wrapping. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *proceedings of the fourth working conference on reverse engineering*, pages 206–214, 1997.