Restructuring of COBOL/CICS Legacy Systems

Alex Sellink*, Harry Sneed**, Chris Verhoef*

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

**SES Software-Engineering Service GmbH, Germany

alex@wins.uva.nl, Harry.Sneed@t-online.de, x@wins.uva.nl

Abstract

We provide a strategy to restructure transaction processing systems. Such systems are core assets of most modern business operations, so their enhancement is crucial. Before large-scale renovation of transaction processing systems can take place, they need to be restructured. We argue that teleprocessing systems are unstructured by their nature. In this paper we approach the problems from a technical viewpoint and we report on the methods and tools that are necessary to bring structure in transaction systems.

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, COBOL, CICS, Teleprocessing system, Transaction processing system, Control-flow normalization, Repartitioning, Remodularization.

1 Introduction

Many organizations have interactive business-critical applications in use for which it is important that they can be enhanced considerably. Change is not the exception for these systems, it is the norm [27]. Typical changes are migration to client/server, accessibility via Intranet/Internet, new business rules, improving maintainability, to mention a few. In order to give an idea of the significance of this area: the majority of the code maintained in the world is a part of a transaction system and this trend will continue. Keeping such systems in good shape and allowing them to evolve is extremely important for developed society [27]. Many such applications have been implemented using COBOL, PL/1, or Assembly/370 for the batch part and CICS (Customer Information Control System) for the interactive part. In [8] it was pointed out that in order to reengineer such systems, certain problematic CICS constructs should be removed. In that paper a strategy was mentioned in order to automatically remove them in favor of other constructions so that the above typical enhancements can be made. Sneed implemented some of these ideas in his reengineering workbench [36]. In this paper we continue the treatment of restructuring COBOL/CICS systems.

In general, it is cost-effective to restructure applications. The return on investment of every dollar spend to restructuring tools is good: \$2.50 after one year, \$5.00 after two years, \$6.00 after three, and \$8.00 after four years [24]. Apart from that, it has a very large impact on solving the Year 2000 problem: it is rated four stars (on a scale from zero to five) by Capers Jones [24]. The findings of Jones are confirmed by several studies. We epitomize on some of them, since the subject is of utter importance for the area of reengineering.

Restructuring of legacy code eases maintenance significantly. In a 1987 study on COBOL, sponsored by the Federal Software Management Support Center of the U.S. General Services Administration [13] we can read:

> the report provides convincing evidence to support the contention that restructured programs are less expensive to maintain than are unrestructured programs. It clearly indicates that the amount of time associated with the analysis and test of maintenance changes can be reduced significantly through prudent implementation of a restructuring strategy using tools and appropriate support methodologies designed for that purpose.

The following (expensive) experiment was carried out: for a set of programs two versions were created upon which the same maintenance tasks were performed by two separate teams. One team used the original programs and the other team used the restructured programs. The programs were automatically restructured using a tool called Recoder developed by Language Technology of Salem, Massachusetts. The product is now called VISION:Recode, and sold by Sterling Software. The recoded versions demonstrated a 44 percent reduction in maintenance and test time [13]. On the negative side there were severe parsing problems and the implemented algorithms were not always giving output that satisfied the involved programmers: style, structure, and names of identifiers were criticized. Maybe these problems were due to the algorithms that were used during the restructuring. We stress that in spite of these negative reactions, the productivity was significantly improved. In our setting it is possible to use an interactive approach in which the reengineer guides the restructuring. Once there is a satisfactory interactive solution, it is possible to automate it.

For people who do not have access to [13], there is a two page summary of the 61 page report in ComputerWorld [1]. In 1991, Language Technology Inc. carried out another experiment confirming the above numbers: the effort for making the same change was 40 percent less when the change was made to a well-structured base system (cyclomatic complexity < 5) as when made to the same system in a poorly structured (cyclomatic complexity > 20) form [22]. A series of 26 interviews by Capers Jones in the seventies with IBM systems programmers in California (13 were fixing bugs; 13 were adding enhancements) reached a similar conclusion: working with well-structured base code was between one fourth to twice as fast as with poorly structured for any given update. Also the bad fix rate was asserted to be lower than 50 percent [22], a clear sign of improved maintainability. For, the bad fix rate for even one-line maintenance changes can be considerably high: in [14] a research at a software maintenance organization pointed out that 55 percent of one-line changes were in error before code reviews were introduced. Another example that confirms the results of the 1987 study [13] is reported on in [27]. In one project the time required for a new maintainer to learn the code had dropped from about 18 months prior to restructuring to approximately 6 months afterwards. On this system the maintenance staff before restructuring was three senior maintainers and three junior maintainers. After restructuring the system is maintained by three to four junior maintainers [27]. [27] consider the results, although indicative, not generalizable to other organizations and systems, since they looked at one system. A large-scale example is below.

A good example of a company that adopted restructuring before enhancing is Hartford Insurance. They have been exploring maintenance costs for more than 15 years, and have published that their maintenance assignment scope has tripled and their annual maintenance budget is below 19 percent and dropping [22]. Compare this with the frequently heard phrase that 30 percent of the total costs of a system are devoted to its initial construction: the remaining 70 percent are spent on maintenance and adjustments to new requirements and new operating environments. These costs are confirmed in many studies [28, 30]. [29] gives a recent summary of these findings. The decreases at Hartford Insurance were not caused by a decrease in the volume or amount of changes. Rather, the decreases were attributable to their program of restructuring and remodularizing their legacy systems prior to carrying out extensive updates [22]. So there is enough evidence that restructuring significantly improves maintenance and major enhancements. One can say that a prerequisite to large-scale renovation is major restructuring.

In this paper we address the restructuring of programs written in mixed languages: COBOL interspersed with CICS commands. We note that it is far from trivial to deal with mixed languages. An indication of the problems is given by Jones: 30 percent of the U.S. software applications contain at least two languages. Jones moreover states that most Year 2000 search engines come to a halt when multiple programming languages are used [24]. So mixed languages are not uncommon and tool support is not easy to develop, since parser technology normally does not deal with mixed languages. For an elaborate discussion on how to deal with mixed languages and reengineering/reverse engineering we refer to [9]. We refer to [7] for a method to obtain a reengineering grammar by hand and we refer to [32, 33] to obtain such grammars using extensive tool support. The tools that the authors develop can deal with mixed languages (see [9] for details).

In addition to the parsing problems, what else comes into play when restructuring COBOL/CICS systems? A major problem is the exception handling mechanism of CICS: the so-called CICS `HANDLE` commands. In [17] a formal specification in Z [40, 41] of the `HANDLE CONDITION` revealed counter-intuitive behavior of this construct. Hayes (the author of [17]) mentions that the exception handling mechanism is so complex that most readers of either the manual [18] or the Z specification [17] do not discover the subtle behavior that was revealed during the formalization process. Hayes advises in 1985 revision of exception handling to be more intuitive. In 1987 Goldstein [15] also addresses the problems with the CICS `HANDLE` commands. CICS programs are very dependent upon the `HANDLE CONDITION` and `HANDLE AID` commands for error detection and special key use determination [15]. Goldstein advices to restrict the use to a single `HANDLE ABEND` to avoid unpredictable results from the program. In a reaction on Goldstein's paper, Jatich [20] states that all `HANDLE` commands should be eschewed entirely. Also in his comprehensive textbook on CICS command level programming [21], Jatich discourages the use of `HANDLE` commands, because of the unstructured logic.

Despite the many warnings reported in the literature, the use of `HANDLE` commands is still omnipresent in modern and legacy COBOL/CICS systems. We are not surprised since there seems to be a natural resistance to new technology [12]. When we wish to improve maintainability, or when we wish to migrate COBOL/CICS systems, we must remove the `HANDLE` commands. Note that this is in accordance with the successful Hartford Insurance strategy: their first step towards major enhancements is extensive restructuring.

In [25] CICS is characterized as being stable, mature and feature-rich. This richness turns into mean complex when reengineering comes into play. The expressiveness of CICS does not always translate easily to new environments, leading to sleepless nights wondering how to implement replacement functionality [25]. Apparently, the problem is so intricate that [25] proposes to use a dedicated processor (the Personal/370 adapter card), in order to run mainframe applications without a change on PCs. In this way off-loading is fast and thus cost-effective [25]. On the other hand, the authors of [39] indicate that off-loading is a very difficult problem. Needless to say that reengineering CICS applications is a difficult problem.

This paper deals with the intricate problem of restructuring COBOL/CICS systems where the exception handling is eliminated, the processing logic is structured, the code is repartitioned so that maintainability improves and migrations to client/server architecture become feasible. Jones reports that the assignment scope of migration to new platform is 1800 LOC per month [23, loc. cit. p. 600]. Another indication that the problems are huge and that extensive tool support is of economic relevance.

Related work In [13] restructuring of COBOL is used to study whether restructuring improves productivity. CICS is not mentioned in this study. In [16] transformations to restructure and reengineer COBOL programs are discussed. However, CICS is not mentioned. In [27] the reengineering of online transaction systems is discussed from a more organizational view that can be used as complement to the technical view presented by us. In [8] COBOL/CICS legacy systems are restructured. In that paper, solutions are proposed for the CICS `HAND ABEND` and `HANDLE CONDITION` command. Sneed implemented these ideas and additionally a solution for the CICS `HANDLE AID` command in his reengineering workbench [36].

Organization of the paper In Section 2, some background information on teleprocessing is included. Furthermore the control-flow of CICS is explained in detail. In Section 3 the original code is first treated with the reengineering workbench of Sneed [36]. During a first restructuring phase, the code is freed from `HANDLE` commands. Then the code travels to Amsterdam. The ubiquitous `GO TO` and `GO TO DEPENDING ON` logic is eliminated in Section 4 using a COBOL/CICS software renovation factory. The architecture of this factory has been described in several papers [5, 31, 34, 4]. The code is transformed using a complex assembly line developed to incrementally (possibly interactive) eliminate all `GO TO` logic. Then the `GO TO` free code travels back to Germany in Section 5. There is it repartitioned so that subroutines from the original source can be extracted. This results in programs where the transaction processing logic is separated from the business logic. We briefly discuss the pros and cons of our approach in Section 6. Finally, we conclude in Section 7 that when this kind of extensive restructuring is performed, it is feasible to migrate COBOL/CICS legacy code.

2 The IBM Solution to On-line Transaction Processing

In order to realize on-line transactions on the mainframe, IBM users had the choice between the teleprocessing monitors CICS [18] and IMS-DC [19]. Both teleprocessing monitors were originally developed in the late 1960s and have gone through several revisions since then. Throughout the 1970s and 1980s these two teleprocessing monitors dominated the industry. In [17] we can read that there is a large number of CICS systems around the world. Twelve years later, [27] report that the majority of code to be maintained in the world is part of a transaction system. So many on-line applications developed on a mainframe were developed with the help of CICS. Another indication of the influence of CICS to this day is the presence of the journal CICS Update and an annual conference: CICS Update '98, 16–17 December in London, United Kingdom.

The role of a teleprocessing monitor is essential to online transactions. The teleprocessing monitor is itself a complex program which fulfills all of the necessary functions required to run a user program in on-line mode. It provides most of the standard functions required by application programs for communication with remote and local terminals and subsystems. It establishes the connection to a user terminal, queues the user messages, allocates memory space, copies the next user message into the program input buffer, picks up the output messages from the program output buffer, processes file or database accesses, takes the user program in and out of a waiting state, intercepts all service requests by the programs, handles error exceptions, and establishes connections between programs. It takes care of the control for concurrently running user application programs serving many on-line users. In effect, the teleprocessing monitor is an operating system within an operating system. Only the system calls are of a higher level since they also provide security checking, logging and error recovery and such [11, 21].

Besides this, teleprocessing monitors have their own interface between application programs and terminal devices. IMS-DC uses Map Format Service (MFS) and CICS offers Basic Mapping Support (BMS). The screen handling facility displays the maps which are specified with an Assembler type macro language, extracts the variable field contents, creates an input data stream from these contents, inserts the variable field contents from the output data stream and manipulates the appearance of the map based on the attributes of the fields [18]. The screen services are the old solution to customizing the user interface. Originally the user programs requested the services provided by the teleprocessing monitor by means of CALL commands. This had the advantage that inclusion of teleprocessing code did not introduce mixed languages in the source: COBOL remained COBOL, PL/1 was still PL/1, and Assembler remained Assembler, which made it compliant with all tool support available for these languages. The problem at that time was to recognize by means of the parameters used what kind of function was being invoked. In the end, this CALL interface solution turned out to be very clumsy for users to write and was a continual source of errors since bad parameters could only be detected at run time. So IBM decided to design two domain specific languages dealing with on-line transactions, by replacing the standard CALL interfaces with macros which could be processed by a preprocessor. In Section 2.1 we will see some code plus the output of such a preprocessor. Note that the embedded code is first preprocessed and then turned into the basic language, and then compiled. When reengineering such code, it is not possible to first preprocess embedded CICS code, and then continue: the mixed language code is the source code of a reengineering tool. This phenomenon makes parsing for reengineering a challenge for the compiler construction community [3].

For IMS-DC the domain specific language is called DL/1 (Data Language/1), for CICS it is the CICS command mode language. All of the operations required from the teleprocessing monitor are addressed by means of CICS or DL/1 constructions embedded in the host language. Thus, COBOL programs are no longer really COBOL programs. They still had a basic COBOL frame and included many COBOL statements, but the average on-line program was made up by 33 to 50 percent of CICS or DL/1 commands.

It would have been easier if both teleprocessing monitors had been implemented in a similar fashion, however there is a different concept to each one. IMS is implemented as a set of subroutines similar to a class library. The user program directs the flow of control. If there is a loop or a selection to be made, it is implemented in the host language. Therefore, in the case of IMS-DC, the teleprocessing monitor operations can be handled as function invocations.

CICS is implemented as a main program similar to a framework in an object oriented environment. It is event driven. The user does something with the keyboard and the system reacts to it. This leads to an inversion of the program. The user program itself is only a set of functions which are waiting to be invoked by CICS.

2.1 CICS and the flow of control

The main flow of control is outside of the program. Therefore, if there is a sequence of user functions to be executed this sequence must be driven by the teleprocessing monitor. For this purpose IBM introduced an exception handling mechanism by means of which the user could communicate to CICS where to go next. This resulted in a pure `GO TO` driven flow of control, which makes CICS programs unstructured (as also stated in [15]).

In fact, the exception handling mechanism contains implicit jump instructions. In order to make this apparent we show the output of a preprocessor used to translate the CICS code into COBOL code that is not for human inspection (it may contain unprintable characters, for instance). In the generated code of the exception handlers `GO TO DEPENDING ON` logic is generated. Since they cannot be seen in the unexpanded code, we call them implicit jump instructions. The problematic CICS commands are `HANDLE ABEND`, `HANDLE AID`, and `HANDLE CONDITION`. We depict from these statements their result after preprocessing them with the CICS preprocessor (module `DFHECP1$`). We removed unprintable characters for the sake of ease. The preprocessor turns the original CICS code into comments using the comment marker `*`. Note the `GO TO DEPENDING ON` logic that pops up in the preprocessed code.

```
*EXEC CICS HANDLE ABEND
*   LABEL       (A)
*END-EXEC.
      MOVE '00073   ' TO DFHEIV0
     CALL 'DFHEI1' USING DFHEIV0
      SERVICE LABEL
      GO TO A DEPENDING ON DFHEIGDI.
*EXEC CICS HANDLE CONDITION
*   DUPKEY      (A)
*   ENDFILE     (B)
*   NOTFND      (C)
*END-EXEC.
      MOVE '00077   ' TO DFHEIV0
     CALL 'DFHEI1' USING DFHEIV0
      SERVICE LABEL
      GO TO A B C DEPENDING ON DFHEIGDI.
*EXEC CICS HANDLE AID
*   PF1         (A)
*   CLEAR       (B)
*   ANYKEY      (C)
*END-EXEC.
      MOVE '023400   ' TO DFHEIV0
     CALL 'DFHEI1' USING DFHEIV0
      SERVICE LABEL
      GO TO A B C DEPENDING ON DFHEIGDI.
```

It is not important what the CICS statements mean exactly, or what the equivalent preprocessed code means. Important is that the code contains implicit jumps. As a consequence, the control flow scope of `HANDLE` statements is global. This implies that once a `HANDLE` command has been given, all the subsequent CICS statements can influence the control flow and suddenly jump to the specified paragraphs (`A`, `B`, `C` in the above examples), depending on their exit status. When the programmer forgets about an active `HANDLE` command given somewhere else in the code (or in an included file), the exception handling may lead to unwanted looping behavior ([21, loc. cit. p. 143]). In order to show how complex such code can be, we give an example. In [8] a 100 KLOC COBOL/CICS system is mentioned with over 600 `HANDLE` commands, and some of them are in an include file that is in 9 percent of the files the first statement of a program.

A solution to avoid the use of `HANDLE` commands, is to make use of the return codes that are used by CICS to deal with the exception handling. In 1987 it was still necessary to write a conversion routine to convert the return codes from hexadecimal to display characters [15]. Later IBM included after preprocessing the return codes in the `WORKING-STORAGE SECTION` (we did not depict this in the preprocessed code because of its length). The return code tells the user program what event has been invoked and allows it to make a decision as to what to do next. This is, of course, much more in time with structured programming because it allows the user program to invoke subroutines rather than jumping into specified labels with no automatic return.

Unfortunately the majority of old CICS programs were already implemented by means of the HANDLE commands long before the new return code was introduced (newer systems also use HANDLE commands). As a consequence most of the legacy CICS programs are unstructured. If they are to be made more maintainable or to be reused as objects in a distributed environment or even converted to another language such as Object-COBOL or Java, they must first be restructured. That means the HANDLE commands must be removed [8, 38].

3 Removing HANDLE Commands

The method proposed here is a two step source transformation. We explain the process with an example containing a HANDLE AID and a HANDLE CONDITION command. In the first step both HANDLE operations are commented out from the code and the decision as to what to do after each receive message is made immediately after the message is received. This is implemented by means of a COBOL EVALUATE statement which checks the states of the function keys and passes control to the labels which were contained in the original HANDLE commands. In the code fragment below, the labels are the expressions in brackets, e.g., VV-860. The first executable statement is that the RECEIVEN SECTION is to be executed. The HANDLE commands are commented out using the comment marker * by Sneed's reengineering workbench.

```
 VV-700.
****                           HANDLE AID UND CONDITION
*
*    EXEC CICS HANDLE CONDITION MAPFAIL (VV-860)
*    END-EXEC.
*
****                            HANDLE AID 1. TEIL
*
*    EXEC CICS HANDLE AID       PF1     (VV-710)
*                               PF2     (VV-720)
*                               PF3     (VV-730)
*                               PF10    (VV-800)
*                               PF11    (VV-810)
*                               PF12    (VV-820)
*                               CLEAR   (VV-840)
*                               ANYKEY  (VV-850)
*    END-EXEC.
****                           R E C E I V E
     PERFORM RECEIVEN.
****                           ENTER-TASTE
     MOVE 13 TO SWPF.
     GO TO VV-999.
```

In order to be able to use the EVALUATE it is necessary to introduce some new state variables or conditional values to denote the function keys. These conditional values are declared in a copy data structure which is included in the WORKING-STORAGE SECTION. Thus, the external events, e.g. the use of function keys, are treated by the new program as a return code.

In the code fragment below, the RECEIVEN SECTION is depicted. Of course, this section depended on the HANDLE commands that are now commented out, so something must change. The exception code that was taken care of by the HANDLE commands is now being added directly below the CICS code. It is an EVALUATE that uses explicitly the COBOL programming GO TO logic. It jumps exactly to the labels that were present in the HANDLE commands.

```
RECEIVEN SECTION.
 RE-000.
*
     MOVE LOW-VALUES TO DBRIM8DI.
     IF CA-SPR = 2
         GO TO RE-100.
*    EXEC CICS RECEIVE MAP    ('DBRIM8D')
*                      MAPSET ('DBRIS8')
*                      INTO   (DBRIM8DI)
*    END-EXEC.
     MOVE 'RC' TO X-CICS-FUNCTION
     MOVE DBRIM8DI TO X-CICS-MAP
     EXEC CICS LINK PROGRAM ('XTPINP')
                   COMMAREA (X-CICS-PARAM)
                     LENGTH (X-CICS-PARAM-LNG)
     END-EXEC.
     MOVE X-CICS-RETCODE TO EIBRESP
     EVALUATE TRUE
       WHEN X-MAPFAIL
         GO TO VV-860
       WHEN X-PF1
         GO TO VV-710
       WHEN X-PF2
         GO TO VV-720
       WHEN X-PF3
         GO TO VV-730
       WHEN X-PF10
         GO TO VV-800
       WHEN X-PF11
         GO TO VV-810
       WHEN X-PF12
         GO TO VV-820
       WHEN X-CLEAR
         GO TO VV-840
       WHEN X-ANYKEY
         GO TO VV-850
     END-EVALUATE.
     GO TO RE-999.
```

We note that if there are more HANDLE commands that reuse labels, that the above approach should be modified. The solution becomes then a bit more complex, but the ideas behind it are the same. This concludes the first step of removing the HANDLE commands.

Obviously, after this first step the HANDLE commands have been removed, but the program is still unstructured since the GO TO branches remain. After all, we only made the implicit CICS jump instructions explicit in the COBOL programming logic. To remove them requires a second step. In this second step the flow of control going out of an EVALUATE is converted to a sequence of subroutine calls (PERFORMs in the COBOL syntax), which execute all statements on the path from the point where a message is received (the entry point) to the point where the control is passed back to the teleprocessing monitor (the exit point). This entry-to-exit path is equivalent to a control flow slice. The slices of a CICS program are both initiated and terminated by a CICS I/O operation. In principle it is possible to use binary trees to depict all possible paths leading out of an input operation. In practice it may be better to use an interactive approach in which the reengineer guides the replacement of the GO TO branches by selection and repetition structures. This second step is carried out in the next code fragment. Important to note is that the GO TOs are removed from the EVALUATE and that the relevant subroutines are listed.

```
RECEIVEN SECTION.
 RE-000.
*
     MOVE LOW-VALUES TO DBRIM8DI.
     IF CA-SPR = 2
         GO TO RE-100.
*    EXEC CICS RECEIVE MAP    ('DBRIM8D')
*                      MAPSET ('DBRIS8')
*                      INTO   (DBRIM8DI)
*    END-EXEC.
     MOVE 'RC' TO X-CICS-FUNCTION
     MOVE DBRIM8DI TO X-CICS-MAP
     EXEC CICS LINK PROGRAM ('XTPINP')
                  COMMAREA (X-CICS-PARAM)
                    LENGTH (X-CICS-PARAM-LNG)
     END-EXEC.
     MOVE X-CICS-RETCODE TO EIBRESP
     EVALUATE TRUE
       WHEN X-MAPFAIL
         PERFORM VV-860
         PERFORM HAUPTVERARB
         PERFORM FEHLMELD
         PERFORM SENDFEHL
       WHEN X-PF1
         PERFORM VV-710
         PERFORM HAUPTVERARB
         PERFORM SENDEN
       WHEN X-PF2
         PERFORM VV-720
         PERFORM HAUPTVERARB
         PERFORM SENDEN
       WHEN X-PF3
         PERFORM VV-730
         PERFORM HAUPTVERARB
         PERFORM SENDEN
       WHEN X-PF10
         PERFORM VV-800
         PERFORM HAUPTVERARB
         PERFORM SENDEN
       WHEN X-PF11
         PERFORM VV-810
         PERFORM HAUPTVERARB
         PERFORM SENDEN
       WHEN X-PF12
         PERFORM VV-820
         PERFORM HAUPTVERARB
         PERFORM SENDEN
       WHEN X-CLEAR
         PERFORM VV-840
         PERFORM HAUPTVERARB
         PERFORM SENDEN
       WHEN X-ANYKEY
         PERFORM VV-850
         PERFORM HAUPTVERARB
         PERFORM SENDEN
     END-EVALUATE.
*    GO TO RE-999.
     EXEC CICS RETURN TRANSID  (CA-TRANS)
                      COMMAREA (COMMAREA)
                      LENGTH   (CA-LENGTH)
     END-EXEC.
```

4 Extensive Restructuring

In the previous section we have seen that the first two steps took care of removal of the HANDLE commands and that the implicit GO TO logic was removed. The next step is to remove the explicit jump instructions and remove redundant code. In this section we will discuss these steps. Since there are so many jump instructions to remove, it is not possible to elaborately describe all the details. However, we give an overview of the used technology.

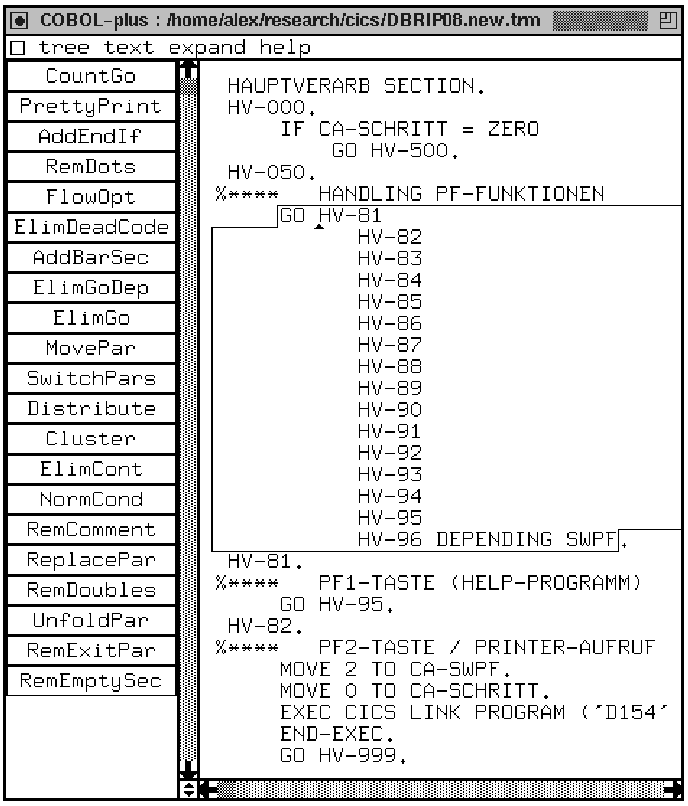

Figure 1: Restructuring assembly lines containing an original code fragment.

Figure 1 show two issues: an indication of the problematic structure of the code and an assembly line that we developed for dealing with it. The code that we see, is the start of the main program (HAUPTVERARB SECTION in German). We see 3 normal GO TOs and one DEPENDING ON jump instruction. In fact, the latter is shorthand for 16 GO TOs. So we see 19 jump instructions in this short fragment. This fragment is not exceptional for this system. Before we show the restructured code we discuss the assembly line. In Section 4.1 we discuss the components. We like to stress that the components have a complex coordination. This coordination is in fact the algorithm to remove all the GO TOs. Since the coordination is a general approach to reengineering code we will discuss this in more detail in a separate paper [35]. Finally, we compare in Section 4.2 the input with the output of the algorithm.

4.1 The Components

The assembly lines took 4 days of effort from development to implementation. Some pre- and postprocessing components were reused, moreover, we extended a GO TO elimination component that we discussed in [8]. We briefly discuss each button of Figure 1. In fact, each button is a component that we use in an algorithm to remove the GO TO logic.

CountGo A useful tool is a counter that measures the number of jump instructions present in the code. For interactive restructuring this tool is convenient since it gives the user an idea of the effect of the followed restructuring strategy. We use this tool also as a tester for other components: sometimes it is useful to measure whether some parts of the code are already free of jump instructions. We use it, e.g., in the `MovePar` component.

PrettyPrint This is a basic component that we reused. We generate pretty printers using technology discussed in [10]. The only thing that needs to be done by hand is to adapt the formatting to company specific output. This was not necessary in this case.

AddEndIf We reused this preprocessing component. It has been discussed in detail in [5]. In short, it is a component that adds the explicit scope terminator `END-IF` to `IF` statements. We note that it makes the code more structured. We use this kind of preprocessing ourselves as well: in order to minimize the number of patterns that we need in order to carry out certain transformations. If, for instance, sometimes a construction is implemented with explicit scope terminators and sometimes not, this imposes extra work. Therefore, we *uniformize* the syntax.

RemDots This is a useful component both for preprocessing and for postprocessing. It is a component that removes separator periods between statements. There is a lot of syntactic freedom to implement exactly the same functionality. Many statements in COBOL can be ended with a separator at wish. We remove all unnecessary ones, so that the code is syntactically more uniform. As a consequence, there is no inconsistent use of separator periods. This has the advantage that it reduces the number of patterns that we have to take into account for further processing the code. During processing code it can be the case that we introduce unnecessary separator periods (this is caused by moving code around). Such components invoke `RemDots` as a postprocessing step, so that the next application of a component still can rely on the fact that there is no inconsistent use of separator periods. So this component is used internally in other components as well.

FlowOpt This useful preprocessing component optimizes the control-flow of `IF` statements. This is a typical evolutionary restructuring component: during maintenance, the control-flow becomes less and less optimal. This component checks for unnecessary complicated control-flow and repairs it. We note that in [31] a special purpose control-flow optimizer is discussed that restructures COBOL/SQL systems for the same reason: change over time degraded the control-flow.

ElimDeadCode This is a useful component that is like `RemDots` both useful for preprocessing as for postprocessing for certain components. It removes dead code that appears below explicit jump instructions. Since the algorithm that we developed makes use of moving code from one place to another, we introduce now and then dead code (due to moving code around). Before we continue, we remove it. So this component is also used internally by other components.

AddBarSec This is a special purpose component. It adds a new COBOL `BAR SECTION` with a special COBOL paragraph in it that contains a `STOP RUN` command. Such a `SECTION` is not accessible via fall through, since then the `STOP RUN` will finish the program. We use this `SECTION` as a place to store paragraphs that are in fact subroutines (for each `SECTION` we create a corresponding `SUBROUTINES SECTION`. In the restructured program they are accessed via `PERFORM`s—the COBOL way to invoke a subroutine. We note that this component is only necessary to add `SUBROUTINES SECTION`s if the code is essentially unstructured. This means that the `GO TO`s are not simulated conditionals or loops. In this case they are necessary. `PARAGRAPH`s in the `SUBROUTINES SECTION`s can be put in *any* order. This makes further restructuring, remodularization, or repartitioning more convenient.

ElimGoDep This is a component that eliminates the `GO TO DEPENDING` logic in two steps like in the previous section was done with the `HANDLE` commands. This is not a surprise since the `HANDLE` commands contain `GO TO DEPENDING ON` logic, as illustrated in Section 2.1. First, the `GO TO DEPENDING ON` is turned into an `EVALUATE` statement containing for every label in the `GO TO DEPENDING ON`, a `WHEN` clause in the `EVALUATE`. Then, the `GO TO`s are removed. We will see an example in Section 4.2.

ElimGo This component removes a number of `GO TO`s that are easy: simulated structured constructs. We mention the use of jump instructions to simulate while loops and we mention the use of jump instructions to implement conditional code. We extended the assembly line that was discussed in [8]. We say that a program is almost structured if all the `GO TO` logic is, in fact, simulated structured logic. In the case that we discuss now, the code is not at all almost structured. On the contrary, it is totally unstructured. Still this component removes all the `GO TO`s due to the special coordination of the components. We refer to [26] for more information on component-based software engineering.

MovePar This component moves paragraphs that are free of jump instructions to the corresponding `SUBROUTINES SECTION`. It uses `CountGo` to test this. Since it is not possible to just move code, this component also takes care of inserting a `PERFORM` at the location where the paragraph was located. Moreover, it will replace all jump instructions to this paragraph by `PERFORM`s and an additional `GO TO` to not disturb the control flow. Although it may seem senseless to turn one `GO TO` into another `GO TO`, it is not. The label of the introduced `GO TO` is usually more near the paragraph than the former `GO TO` due to the fact that we remove a paragraph. Then the `ElimGo` component can remove the created more structured jump instructions. We apply ElimGo and MovePar alternatingly to remove all GO TOs (we discuss this in detail in [35].

SwitchPars This component is a variation of MovePar. This one is used when a paragraph is not free of jump instructions and the GO TOs are not simple. Think of a GO TO to a distant place back in the code. Such difficult paragraphs can be freely switched with other paragraphs that end in a jump instruction themselves. This step turns a jump to a label far away sooner or later in a jump that is more close to the label. This leads to more structured code and eventually the ElimGo component, that attacks simple GO TOs will eliminate such jumps.

Distribute This component optimizes EVALUATE statements. It distributes common statements that occur in all the WHEN clauses outside the EVALUATE. It is a special purpose component that beautifies the code. We use is as a typical postprocessing operation. In Section 4.2 we see an example of an optimized EVALUATE.

Cluster This component clusters WHEN clauses of evaluates when the cases are equal. During maintenance of systems with many GO TOs, it is difficult to keep track on the exact number of cases, so apparently, maintainers start copying paragraphs and treat them as a new case (using GO TO logic). After the GO TOs have been eliminated it becomes clear that some different cases are equal.

ElimCont is a component that eliminates CONTINUE statements. We reused this component (discussed in detail in [8]) . The CONTINUE statement does not have semantic meaning. Other components will now and then introduce CONTINUE statements in order to make the components themselves as simple as possible. In the final phase of the GO TO elimination we eliminate all introduced CONTINUE statements in one fell-swoop.

NormCond This is also a postprocessing component. We reused it (discussed in detail in [8]). It optimizes the Boolean expressions in the COBOL source. Other components sometimes introduce a NOT in a condition, or they combine certain conditional expressions. In the end, we turn such conditions into their most natural form.

The above components are sufficient to eliminate the GO TO logic from COBOL/CICS systems. After this is done, it becomes feasible to remove redundant code. This can be dead code or indirect code. We discuss six such components.

RemComments This is a tool that is used by other components. It simply removes all comments. We note that comments are part of our reengineering grammar [7]. We use this component to test whether or not code fragments are equal modulo their comments.

ReplacePar This component recognizes pieces of code that are equal in the SUBROUTINES SECTIONs. As soon as this component locates PARAGRAPHs that are equal, it replaces *all* references to both PARAGRAPHs to one of the two. We say that code is equal when their access labels differ and when their comments possibly differ. We use the RemComments to analyze this.

RemDoubles This component looks for PARAGRAPHs that have the same label and the same contents. It removes the doubles but it copies the comments to the remaining one, so that comments are not thrown away.

We give a simple example.

```
 HV-84.
****                          PF4-TASTE
   PERFORM HV-95.
 HV-85.
****                          PF5-TASTE
   PERFORM HV-95.
    --ReplacePar-->
 HV-85.
****                          PF4-TASTE
   PERFORM HV-95.
 HV-85.
****                          PF5-TASTE
   PERFORM HV-95.
    --RemDoubles-->
HV-85.
****                          PF4-TASTE
****                          PF5-TASTE
PERFORM HV-95.
```

ReplacePar finds the PARAGRAPHs and makes their labels equal (and all references in the rest of the program). RemDoubles then removes the duplicate code and merges the comments into the remaining code.

UnfoldPar This component makes the code more direct. As can be seen in the above example, a PERFORM HV-85 actually is a PERFORM HV-95. The UnfoldPar unfolds indirect calls into direct calls.

RemExitPar This component removes superfluent EXIT paragraphs. Since the control-flow is difficult to understand using GO TO logic, COBOL provides statements that have no semantics, but are used to give the reader an idea of the status of the control-flow. The EXIT statement is comment with the intention to mean that when the control is at the EXIT statement, it will leave the current SECTION. Note that putting an EXIT inside a program will *not* cause it to exit, the EXIT statement is only to emphasize that at this location it will leave a SECTION. Since we removed all the GO TO logic, there are no jumps to such paragraphs anymore so we remove them.

RemEmptySec This component removes empty SUBROUTINES SECTIONs that were made by the AddBarSec component, but appeared not to be necessary. In such, usually small SECTIONs, the code is almost structured, so that application of the ElimGo component cleans the code. In this example program we constructed 16 SUBROUTINES SECTIONs and we needed 8.

4.2 Comparing Input and Output

The code changes dramatically when we apply our algorithm to the example program. Note that the program was totally unstructured, so it is not surprising that the program significantly changed in appearance. We will discuss pieces of code that can be traced back to the original code that we displayed in Figure 1.

In the original code (Figure 1) we see that if CA-SCHRITT equals ZERO the control goes to HV-500. If the condition fails we will continue with paragraph HV-050 via fall-through. In the restructured code, the HAUPTVERARB SECTION has become very small: it only contains the statement that either HV-500 or HV-050 is PERFORMed. In our situation, HV-500 is a subroutine that is in the HAUPTVERARB-SUBROUTINES SECTION. Below we depicted the complete main SECTION of the restructured program.

```
HAUPTVERARB SECTION.
HV-000.
  IF CA-SCHRITT = ZERO
    PERFORM HV-500
  ELSE
    PERFORM HV-050
  END-IF.
```

Paragraph HV-050 originally contained one large GO TO DEPENDING ON. This paragraph is changed into an EVALUATE statement.

```
HV-050.
%****   HANDLING PF-FUNKTIONEN
  EVALUATE SWPF
    WHEN 1 PERFORM HV-95
    WHEN 2 PERFORM HV-82
    WHEN 3
      PERFORM HV-83
      PERFORM HV-95
    WHEN 4 THRU 9 PERFORM HV-95
    WHEN 10
      PERFORM HV-90
      PERFORM HV-200
    WHEN 11 PERFORM HV-200
    WHEN 12 PERFORM HV-92
    WHEN 13 PERFORM HV-200
    WHEN 14 PERFORM HV-94
    WHEN 15 PERFORM HV-95
    WHEN 16 PERFORM HV-200
  END-EVALUATE.
```

As can be seen, there was a lot of code duplication in the old code: many labels in the original GO TO DEPENDING ON statement have disappeared. We turned 16 cases into 7 in this example program, e.g., the cases 4 THRU 9 were all the same. Paragraph HV-81 (see Figure 1) is gone after restructuring, since it only jumped to HV-95. This indirect code is replaced by the (also restructured) code below.

```
HV-95.
  PERFORM CAFILL
  IF SWFEHL > ZERO
    CONTINUE
  ELSE
    MOVE LOW-VALUE TO DBRIM8DI
    MOVE 1 TO
      FEHL-NR
      CA-FEHLNR
    MOVE -1 TO M8SACHTL
  END-IF
  PERFORM HV-900.
```

In paragraph HV-82 of Figure 1, we see embedded CICS code followed by a GO TO HV-999. In the restructured program, this code is the same, except that the GO TO has disappeared. The reason that this is possible, is as follows. In the original code we can jump to HV-82 via the GO TO DEPENDING ON logic. Then we jump HV-999 (an EXIT paragraph). In the restructured code we use only PERFORMs. They turn back to the place where they were invoked, so after the PERFORM HV-82 is executed the control-flow is returned to the EVALUATE statement. Then, the PERFORM HV-050 is finished and the control-flow is back in the IF of the HAUPTVERARB SECTION. Via fall-through, the control-flow goes to the next SECTION. Of course, this control flow comparison of a small fragment of the code is not a proof that the algorithm we use is correct. This is just to give an indication.

5 Repartitioning the Structured Code

Once the CICS code has been restructured to be driven by return codes rather than by branching to given labels it is then possible to repartition it. Repartitioning involves the extraction of subroutines on the SECTION level from the original source text where they are performed in-line and their replacement by subprograms which are called dynamically at run time. This repartitioning has the advantage of making the CICS programs much smaller, more flexible, more testable and more easier to maintain [39].

A repartitioned program is of course much smaller because only the CICS commands and the control logic are left in the main program. The processing routines are removed to separate modules. Such programs are more flexible because it is now possible to change the transaction control without affecting the processing logic. They are more testable because each processing routine, i.e. each sliced segment, can be tested separately. By stubbing out subprogram calls it is also possible to test the transaction control logic separately without the processing. Finally, the program is easier to maintain because the interfaces have been clearly specified and the processing units have been isolated from the transaction control. In effect, repartitioning aims at a complete separation of coordination and computation. In [2] such strategies are considered being core technologies for system renovation. Note that in Section 4 coordination and computation are also separated which enables flexible (re)use of components.

The repartitioning process is accomplished in three passes. In the first pass a data flow analysis is performed to determine which variables are used in what way in each subroutine, i.e. performed PARAGRAPH or SECTION. These variables are stored in a data reference table for each subroutine. In the second pass the subroutines are cut out of the original source code and placed in the procedural part of a module framework. The variables they use are defined as parameters in the LINKAGE SECTION and are listed in the entry command. In this way each subroutine becomes a separate subprogram. In the third pass the main program is processed again to convert all PERFORMs to subroutines into CALL USING commands. The CALL is made to a variable name which is assigned before the name of the subroutine. This type of dynamic linking is better supported by CICS as it avoids having to run a static link job. The parameters of the CALL statement are filled with the variables used by that particular subroutine. We give the shortest example that was available in the code. We see that the VERARB SECTION that was performed is now put in a subprogram. In that subprogram a paragraph dbrip002 is called that uses all the data that was found during the data flow analysis in the first step.

```
*  PERFORM VERARB
     CALL  "dbrip002" USING
      COMMAREA,
      RECHENFELDER,
      ARBEITSDATEN,
      DFHBMBRY,
      DBRIM8DI,
      XM181-P,
      SWITCHES,
      P-XM314,
      KONSTANTEN,
      P-XM200,
      P-P008L.
```

The result of the repartitioning process is a modular CICS program consisting of a main control module with the transaction processing logic and processing submodules which implement the business logic. A further step might be to separate out the access logic, but that might be the topic of another paper [37].

In order to repartition for migration to other platforms, it is important to remove all CICS commands. Therefore, the normal CICS commands are substituted by CALLs like the preprocessor as discussed in Section 2.1. We give an example of a CICS statement that can be equally well expressed using standard COBOL.

```
*  EXEC CICS LINK
*    PROGRAM ( 'D154' )
*  END-EXEC.
   CALL 'D154'.
```

The CICS statement is commented out (using the comment marker *) and a standard COBOL CALL is replacing the CICS. This kind of CICS elimination makes the program more portable to other environments. Of course, CICS is available on many platforms, so this step is not always necessary.

6 Discussion

We briefly discuss the pros and cons of our approach. First of all, we replaced the CICS commands by CALLs to a wrapper. In this case the programs can be executed in any environment provided the wrapper is able to fulfill the requests. Moreover, the user interface can be implemented totally independent of the old program, for instance in Java. The wrapper could be implemented in C++. Everything can be connected with an ORB using IDL to define the interfaces.

Second, we removed all GO TO logic. We have argued in the introduction that this makes the programs easier to maintain and test, irrespective of what environment it is running in. Another benefit is that the code decreases in size.

We stress that the problem with old COBOL programs is that any one section of code will address data elements scattered throughout the DATA DIVISION, that is, they use global data. This results in long parameter lists when we are going to remodularize. There is nothing that can be done to prevent this. If we would pass individual data elements instead of structures, the lists are even longer. This is the problem of dealing with real programs instead of models of programs created for research purposes. So, we create complex interfaces for the subprograms, which may be seen as more problematic than in-line solutions. So for the first two steps we are sure that they are cost-beneficial, as argued in the introduction. This last step is debatable, since the advantage of subprograms might be annihilated by the creation of the complex interfaces.

7 Conclusion

CICS on-line programs have a unique event driven structure irrespective of what host language they use. The CICS commands are normally scattered throughout the source for controlling the interaction with the user as well as for the data access. The host code in which the CICS commands are embedded only serves to describe the data structures and to process them when they are made available by CICS. As a rule this processing logic is unstructured. If the processing logic is to be separated from the presentation and the access logic as is required by modern software architectures, then the CICS commands which return control from the user to specified code locations must be removed and replaced by a return value. In addition, the user code must be restructured to perform subroutines based on this return value rather than by branching directly to given labels. Only when this is done, will it be possible to migrate or wrap CICS on-line programs.

The research described in this paper is intended to solve the problem of restructuring CICS programs in order to make them more maintainable and to facilitate the migration into different environments such as to a client/server platform.

References

[1] C. Babcock. Restructuring eases maintenance. *Computerworld*, page 19 and 22, November 1987.

[2] 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. Available at http://adam.wins.uva.nl/~x/sofsem/sofsem.html.

[3] M.G.J. van den Brand, P. Klint, and C. Verhoef. Reengineering needs generic programming language technology. *ACM SIGPLAN Notices*, 32(2):54–61, 1997.

[4] M.G.J. van den Brand, P. Klint, and C. Verhoef. Term rewriting for sale. In C. Kirchner and H. Kirchner, editors, *Second International Workshop on Rewriting Logic and its Applications*, Electronic Notes in Theoretical Computer Science. Springer-Verlag, 1998. Available at: http://adam.wins.uva.nl/˜x/sale/sale.html.

[5] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. Available at http://adam.wins.uva.nl/˜x/scp/scp.html. An extended abstract with the same title appeared earlier as [6].

[6] 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.

[7] 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. Available at http://adam.wins.uva.nl/˜x/coboldef/coboldef.html.

[8] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Control flow normalization for COBOL/CICS legacy systems. In P. Nesi and F. Lehner, editors, *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, pages 11–19, 1998. Available at http://adam.wins.uva.nl/˜x/cfn/cfn.html.

[9] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of the sixth International Workshop on Program Comprehension*, pages 108–117, 1998. Available at http://adam.wins.uva.nl/˜x/ref/ref.html.

[10] 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.

[11] B. Crownhart. *IBM's Workstation CICS*. McGraw-Hill, 1992.

[12] M. Bauer (ed.). *Resistance to New Technology: Nuclear Power, Information Technology, and Biotechnology*. Cambridge University Press, 1997. reprint edition.

[13] M.A. Colter (ed.). Parallel Test and Productivity Evaluation of a Commercially Supplied COBOL Restructuring Tool. Technical report, Office of Software Development and Information Technology, Falls Church, Virginia, USA, 1987.

[14] D.P. Freedman and G.M. Weinberg. *Handbook of Walkthroughs, Inspections and Technical Reviews*. Dorset House, 3rd edition, 1990. Originally published by Little, Brown & Company, 1982.

[15] L. Goldstein. An alternative to CICS `HANDLE` commands. *CICS Update*, July 1987.

[16] T. Harmer, P. McParland, and J. Boyle. Transformations to restructure and re-engineer COBOL programs. *Journal of Automated Software Engineering*, 5:321–345, 1998.

[17] I.J. Hayes. Applying formal specification to software development in industry. *IEEE Transactions on Software Engineering*, SE-11(2):169–178, 1985.

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

[19] IBM, Kingston, New York, USA. *AOC/MVS IMS Automation Programmer's Reference and Installation Guide*, 1.3 edition, 1994.

[20] A.M. Jatich. CICS `HANDLE` commands versus `RESP/RESP2`. *CICS Update*, November 1987.

[21] A.M. Jatich. *CICS Command Level Programming*. Wiley Professional Computing, 1991.

[22] C. Jones. *Assessment and Control of Software Risks*. Prentice-Hall, 1994.

[23] C. Jones. *Estimating Software Costs*. McGraw-Hill, 1998.

[24] C. Jones. *The Year 2000 Software Problem – Quantifying the Costs and Assessing the Consequences*. Addison-Wesley, 1998.

[25] S.H. King. Mainframe application migration using P/370 technology. *Enterprise Systems Journal*, May 1995.

[26] P. Klint and C. Verhoef. Evolutionary software engineering: A component-based approach. In R.N. Horspool, editor, *IFIP WG 2.4 Working Conference: Systems Implementation 2000: Languages, Methods and Tools*, pages 1–18. Chapman & Hall, 1998. Available at: http://adam.wins.uva.nl/˜x/evol-se/evol-se.html.

[27] W. Kozaczynski and N. Wilde. On the re-engineering of transaction systems. *Journal of Software Maintenance*, 4(3):143–162, 1992.

[28] B.P. Lientz and E.B. Swanson. *Software Maintenance Management—A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Reading MA: Addison-Wesley, 1980.

[29] S. McConnell. *Rapid Development*. Microsoft Press, 1996.

[30] J. Reutter. Maintenance is a management problem and a programmer's opportunity. In A. Orden and M. Evens, editors, *1981 National Computer Conference*, volume 50 of *AFIPS Conference Proceedings*, pages 343–347. AFIPS Press, Arlington, VA, 1981.

[31] M.P.A. Sellink and C. Verhoef. An architecture for automated software maintenance. Technical Report P9807, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/˜x/asm/asm.html.

[32] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. Technical Report P9805, University of Amsterdam, Programming Research Group, 1998. An extended abstract appears in ASE 98. Available at: http://adam.wins.uva.nl/˜x/cale/cale.html.

[33] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. Technical report, University of Amsterdam, Programming Research Group, 1998. In preparation.

[34] M.P.A. Sellink and C. Verhoef. Native patterns. In M. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings of the 5th Working Conference on Reverse Engineering*. IEEE Computer Scociety Press, 1998. Available at http://adam.wins.uva.nl/˜x/npl/npl.html.

[35] M.P.A. Sellink and C. Verhoef. A systolic structuring algorithm. Technical report, University of Amsterdam, Programming Research Group, 1998. In preparation.

[36] H.M. Sneed. Architecture and functions of a commercial software reengineering workbench. In P. Nesi and F. Lehner, editors, *Proceedings of the Second Euromicro Conference on Maintenance and Reengineering*, pages 2–10, 1998.

[37] H.M. Sneed. Encapsulation of IBM-CICS transactions in a distributed object environment, 1998. Work in Progress.

[38] H.M. Sneed. *Object-oriented Software Migration*. Addison-Wesley, October 1998.

[39] H.M. Sneed and E. Nyary. Downsizing large application programs. *Journal of Software Maintenance*, 6(5):235–247, 1994.

[40] J.M. Spivey. *Understanding Z: A Specification Language and Its Formal Semantics*. Cambridge Tracts in Theoretical Computer Science 3. Cambridge University Press, 1988.

[41] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1992.