Restructuring of COBOL/CICS legacy systems

Alex Sellink[a], Harry Sneed[b], Chris Verhoef[c,*]

[a]*Quack.com, 1252 Borregas Ave, Sunnyvale, CA 94089, USA*
[b]*SES Software-Engineering Service GmbH, Germany*
[c]*Department of Information Management and Software Engineering, Free University of Amsterdam,
De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*

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. © 2002 Elsevier Science B.V. All rights reserved.

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 [35]. Typical changes are migration to client/server, accessibility via Intranet/Internet, changed business needs, 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 [35]. Many such applications have been

---

* Corresponding author.
  E-mail addresses: slinky@quack.com (A. Sellink), harry.sneed@t-online.de (H. Sneed), x@cs.vu.nl (C. Verhoef).

implemented using COBOL, PL/I, or Assembly/370 for the batch part and customer information control system (CICS) for the interactive part. In [10] 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 [54]. In this paper we continue the treatment of restructuring COBOL/CICS systems. We move the emphasis from CICS code to COBOL code, since the CICS parts have been published already [10], but not the COBOL parts.

1.1. Unknown facts on the economics of restructuring

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 1 year, $5.00 after 2 years, $6.00 after 3, and $8.00 after 4 years [30]. The findings of Jones [30] are confirmed by several studies. We epitomize on some of them, since the subject is of utter importance for the area of reengineering but quantitative data is unknown by many.

Restructuring of legacy code eases maintenance significantly. In a 1987 study on COBOL, sponsored by the Federal Software Management Support Center of the US General Services Administration [16] 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 in [16]: 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 called VISION:Recode and owned by Sterling Software at the time of writing this paper. The recoded versions demonstrated a 44 percent reduction in maintenance and test time [16]. 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 improved significantly.

For people who do not have access to [16], there is a two page summary of the 61 page report in ComputerWorld [3]. 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 [28].

A series of 26 interviews by Capers Jones in the 1970s 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 [28], a clear sign of improved maintainability. For, the bad fix rate for even one-line maintenance changes can be considerably high: in [18] 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 [16] is reported on in [35]. 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 consisted of three senior maintainers and three junior maintainers. After restructuring, the system is maintained by three to four junior maintainers [35]. The authors of [35] 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 [28]. Compare this with the frequently heard phrase that 70 percent of the annual budget is spent on maintenance and adjustments to new requirements and new operating environments. These cost distributions are confirmed in many studies [39,44,4,29,43]. McConnell [40] gives a 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 [28]. So there is enough evidence that restructuring significantly improves maintenance and significant enhancements. One can say that major restructuring is a prerequisite to large-scale renovation.

1.2. Restructuring and mixed language applications

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]: 30 percent of the US software applications contain at least two languages. Jones [30] moreover states that most Year 2000 search engines come to a halt when multiple programming languages are used. 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 [11]. We refer to [9] for a method to obtain a reengineering grammar by hand and we refer to [49,50,38] to obtain such grammars using extensive

tool support. The tools that the authors constructed and applied can deal with mixed languages (see [11] 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 [22] a formal specification in Z [57,58] of the HANDLE CONDITION revealed counter-intuitive behavior of this construct. Hayes (the author of [22]) mentions that the exception handling mechanism is so complex that most readers of either the manual [23] or the Z specification [22] 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 [19] 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 [19]. 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 [26] states that *all* HANDLE commands should be eschewed entirely. Also in his comprehensive textbook on CICS command level programming [27], 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. 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 [31] CICS is characterized as being stable, mature and feature-rich. This richness turns into mean complexity 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 [31]. Apparently, the problem is so intricate that [31] 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 [31]. On the other hand, the authors of [56] indicate that off-loading is a very difficult problem. Jones reports that the assignment scope of migration to new platform is 1800 LOC per month [29,56, p. 600], which is another indication that the problems are huge and that extensive tool support is of economic relevance. Needless to say that reengineering CICS applications is a difficult problem.

This paper deals with the intricate problem of restructuring COBOL/CICS systems. We automatically eliminated the exception handling by CICS, we constructed tools to automatically structure the processing logic, the code is automatically repartitioned so that maintainability improves and enhancements (manual, tool supported, or fully automatic) become feasible. Our tools make the coordination of candidate business logic transparent. As soon as this is the case, we can modify the coordination. Moreover, we can insert new code, modify old code, and delete code without affecting the control-flow. This means that the programs have become modular, change-enabled, and in a sense componentized since only part of the functionality can be invoked at wish.

1.3. Related work

In [16] restructuring of COBOL is used to study whether restructuring improves productivity. CICS is not mentioned in this study. In [21] transformations to restructure and reengineer COBOL programs are discussed. Again, CICS is not incorporated. In [35] the reengineering of on-line transaction systems is discussed from a more organizational view that can be used as complement to the technical view presented by us. In the United States Patent [15] an algorithm is discussed to restructure pure COBOL programs by coalescing paragraphs, procedure chunking, and the creation of a so-called super-procedure. To do this, jump instructions have to be removed. The goal of their work is to bring COBOL programs more in line with the idiom of C, C++, or Fortran programs so that compiler optimization technology that is geared towards these languages can be used fruitfully for COBOL as well. In our paper, we also restructure towards the creation of a simulated main procedure, and a set of subroutines that can be accessed by the super-procedure. In our approach the goal is not to optimize the code, but to enable it for change. This implies that understandability is a crucial aspect of our contribution. Therefore, the code does not look like a large procedure from which all code in the subroutines is accessed. We believe that our design attributes do not necessarily apply to the work reported in [15], for it deals with compiler optimization issues. Also, the second author has implemented the idea of a super-procedure in his commercial reengineering workbench [54], and knows from personal experience that many programmers hate this kind of restructuring. In [10] COBOL/CICS legacy systems are restructured. In that paper, solutions are proposed for the CICS HANDLE ABEND and HANDLE CONDITION commands. Sneed implemented these ideas and additionally a solution for the CICS HANDLE AID command in his reengineering workbench [54].

1.4. Organization of the paper

In Section 2, we will provide a rationale for restructuring. In Section 3, some background information on teleprocessing is included. Furthermore, the control-flow of CICS is explained in detail. In Section 4, the original code is first treated with the reengineering workbench of Sneed [54]. During a first restructuring phase, the code is freed from HANDLE commands. The ubiquitous GO TO and GO TO DEPENDING ON logic is eliminated in Sections 5 and 6 using a COBOL/CICS software renovation factory, designed and implemented to solve this particular problem. The architecture of this factory has been described in several papers [12,48,47,7]. The code is transformed using a complex assembly line developed to incrementally eliminate all GO TO logic. We discuss this algorithm by example in Section 6. Then the GO TO free code is again processed by tools from Sneed in Section 7. It is 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. Then we are in a position to convert the programs to a stateless version so that we can use them as components. We discuss this in Section 8. Finally, we conclude in Section 9.

2. A rationale for restructuring

It is highly unlikely that users will finance a restructuring project solely for the purpose of improving the internal quality of a product, i.e., its maintainability, even if case studies such as the ones we mentioned in the introduction demonstrate the long-term benefits. For the typical IT shop, the benefits of improving maintainability are still too low compared to the costs and risks involved [53]. It has always been an illusion of the academic community that users are really interested in quality, whereas, as a fact, they are mainly interested in functionality and only partially in quality. Even then, quality is perceived as the quality of the external behavior of a system and not in its internal construction. For this reason, there has never been a significant market for restructuring projects and products, as such. Another reason is that any code modification bears the risk of failures. What happens if the restructured code has different semantic behavior than the original code? We note that restructuring often reveals erroneous and dangerous code, so that it is tempting to repair that. Or sometimes it is automatically repaired, due to the used restructuring algorithms. Such corrections can, in turn, trigger errors to occur that due to the original errors, laid dormant in the application. This is a risk that the owners of the systems do often not want to take only for improvement of internal quality.

Migration is another issue. Users are often compelled to move from one environment to another or, moreover, they need to reuse old programs in a new context. The latter objective has become very important in connection with component-based software development [14,1,60]. Then restructuring is accepted as a part of the migration process since the programs have to be altered anyway. However, the main objective is to get the old system to work in the new environment and not necessarily to make it better. Existing programs can be reused as components of a newly developed system provided they meet certain conditions. They must fit into the new architecture, that is, their external interface must be remotely accessible via various calling mechanisms such as remote procedure calls, object request broker connections, or Java's remote method invocations [20]. Besides that they must be made independent of their environment, i.e, made self-contained, their behavior must be predictable and they should produce no side-effects. This is where restructuring comes into play. Restructuring of legacy programs is done to

- provide remotely accessible interfaces,
- insulate the program from its environment,
- isolate individual functions from one another, and
- prevent undesired side-effects.

To provide remotely accessible interfaces, the existing interfaces must be modified. In the case of batch programs these are the read and write operations performed on transaction files managed by the operating system. In the case of on-line programs these are the send and receive operations performed on the maps managed by the teleprocessing monitor. The latter is particularly difficult because of the many interactions between the application program and the transaction monitor. The application program

is essentially reduced to a subprogram of the transaction monitor which is reacting to events triggered by the transaction monitor. For the sake of wrapping this is a suitable solution, but the interactions have to be altered to fit the wrapper. Therefore, interface reengineering is one of the targets we deal with in this paper.

To insulate the program from its environment, all references to services provided by that environment must be redirected. In the case of CICS many services are provided by the transaction processing monitor including memory allocation, error handling and program to program linkage. If the user program is to be taken out of the CICS environment, then these service requests have to be redirected to the wrapper, i.e., converted to CALLs to user-provided or standard routines which perform the same services as provided by CICS. This entails converting all service requests, another objective of the work presented here.

To isolate functions of the program from one another, all direct interconnections between functions must be capped. That means removing all jump instructions. Functions or subroutines in COBOL are packed in paragraphs or sections. A GO TO from one paragraph to another joins the functions of those paragraphs. If programs are to be reused as components, we should be able to invoke their functions independently of one another. The paragraphs or, at least, the sections of a COBOL program should resemble methods in order to provide discrete functionality to potential clients. The program needs to be modularized, that is, partitioned into reusable parts. To achieve this, it has to be restructured. Then, the sections of the program are no longer embedded in a hard wired control structure imposed by that particular application, but are independent functions which may be invoked in any order. This gives the flexibility required in a modern component-based environment. To this end restructuring is a necessary prerequisite to wrapping.

Finally, there is the goal of eliminating side-effects. In CICS most side-effects stem from the use of a common global data area. Addressing errors can create havoc, as one program overwrites the data of another. To prevent this, one program should not be able to directly access the data of another. All data should be passed as a parameter by value. To achieve this, parameter lists must be generated for all global data used by the program. The program itself becomes stateless. It has no own memory, but only processes data passed to it as a message. The capping of global data references is the last stage of the transformation process described in this paper.

3. 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 [23] and IMS-DC [24]. 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 [22] we can read that there is a large number of CICS systems around the world. Twelve years later, Kozaczynski and Wilde [35] 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.

The role of a teleprocessing monitor is essential to on-line 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 [17,27].

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 [23]. 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/I was still PL/I, 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 continuous 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, replacing the standard CALL interfaces with macros which could be processed by a preprocessor. In Section 3.1 we will see some code plus the output of such a preprocessor. The embedded code is first preprocessed and then turned into the host language, and then compiled. So in Section 3.1 we will see translations of CICS to COBOL code. You cannot first preprocess embedded CICS code, and then start reengineering: the mixed language code is the source code for a reengineering tool. So we have to deal with COBOL with embedded CICS as source code. This phenomenon makes parsing for reengineering a challenge for the compiler construction community [6].

For IMS-DC the domain-specific language is called DL/I (Data Language/I), 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/I constructions embedded in the host language. Thus, COBOL programs were 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 30–50 percent of CICS or DL/I 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.

3.1. 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, it 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 [19]).

In fact, the exception handling mechanism contains implicit jump instructions. In order to make this visible we show the output of a preprocessor used to translate the CICS code into COBOL code in Fig. 1. We note that such code 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 this cannot be seen in the unexpanded code, we call this CICS code an implicit jump instruction. 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 so that we can conveniently display the code. 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.

It is not important what the CICS statements in Fig. 1 exactly mean, or what the equivalent preprocessed code means. Important is that the code contains implicit jumps. As a consequence, the scope of the control-flow 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. We note that the exit status of CICS commands is stored in a record called `DFHEIGDI`, that is why the `GO TO DEPENDING ON` logic is looking for that variable.

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 [27,19, p. 143]. In order to show how complex such code can be, we

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

Fig. 1. Expanded CICS code reveals intricate jump instructions.

give an example. In [10] 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. This indicates that implicit jump instructions are endemic in this system and are a frequent source of errors during maintenance and enhancements.

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 [19]. Later IBM included after preprocessing the return codes in the WORKING-STORAGE SECTION (we did not depict this in the preprocessed code of Fig. 1). 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 [59]. That means the HANDLE commands must be removed [10,55].

4. 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 or VV-710. 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 be done to preserve the behavior of the program. The exception code

that was expressed with 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, 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.
```

5. Components for 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. Indeed in the above code fragment, the GO TOs are gone, however, they were jump instructions due to elimination of CICS code. The next step is to remove the explicit jump instructions that were already in the code. Moreover, we remove redundant code.

Fig. 2 illustrates 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,

```
┌──────────────────────────────────────────────────────────────┐
│ ◉ COBOL-plus : /home/alex/research/cics/DBRIP08.new.trm    ▓▓  回 │
├──────────────────────────────────────────────────────────────┤
│ □ tree text expand help                                        │
├─────────────┬─┬──────────────────────────────────────────────┤
│   CountGo   │↑│  HAUPTVERARB SECTION.                          │
├─────────────┤ │  HV-000.                                       │
│ PrettyPrint │ │       IF CA-SCHRITT = ZERO                     │
├─────────────┤ │          GO HV-500.                            │
│   AddEndIf  │ │  HV-050.                                       │
├─────────────┤ │  %****    HANDLING PF-FUNKTIONEN               │
│   RemDots   │ │      ┌GO HV-81                                 │
├─────────────┤ │      │    ▲  HV-82                             │
│   FlowOpt   │ │      │       HV-83                             │
├─────────────┤ │      │       HV-84                             │
│ ElimDeadCode│ │      │       HV-85                             │
├─────────────┤ │      │       HV-86                             │
│   AddBarSec │ │      │       HV-87                             │
├─────────────┤ │      │       HV-88                             │
│  ElimGoDep  │ │      │       HV-89                             │
├─────────────┤ │      │       HV-90                             │
│    ElimGo   │ │      │       HV-91                             │
├─────────────┤ │      │       HV-92                             │
│   MovePar   │ │      │       HV-93                             │
├─────────────┤ │      │       HV-94                             │
│  SwitchPars │ │      │       HV-95                             │
├─────────────┤ │      └       HV-96 DEPENDING SWPF.             │
│  Distribute │ │  HV-81.                                        │
├─────────────┤ │  %****    PF1-TASTE (HELP-PROGRAMM)            │
│   Cluster   │ │          GO HV-95.                             │
├─────────────┤ │  HV-82.                                        │
│   ElimCont  │ │  %****   PF2-TASTE / PRINTER-AUFRUF            │
├─────────────┤ │          MOVE 2 TO CA-SWPF.                    │
│   NormCond  │ │          MOVE 0 TO CA-SCHRITT.                 │
├─────────────┤ │          EXEC CICS LINK PROGRAM ('D154'        │
│  RemComment │ │          END-EXEC.                             │
├─────────────┤ │          GO HV-999.                            │
│  ReplacePar │ │                                                │
├─────────────┤ │                                                │
│  RemDoubles │ │                                                │
├─────────────┤ │                                                │
│  UnfoldPar  │ │                                                │
├─────────────┤ │                                                │
│  RemExitPar │ │                                                │
├─────────────┤ │                                                │
│ RemEmptySec │ │                                                │
├─────────────┤ │                                                │
│             │↓│                                                │
├─────────────┼─┴──────────────────────────────────────────────┤
│             │⬍◄▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓►│
└─────────────┴──────────────────────────────────────────────────┘
```
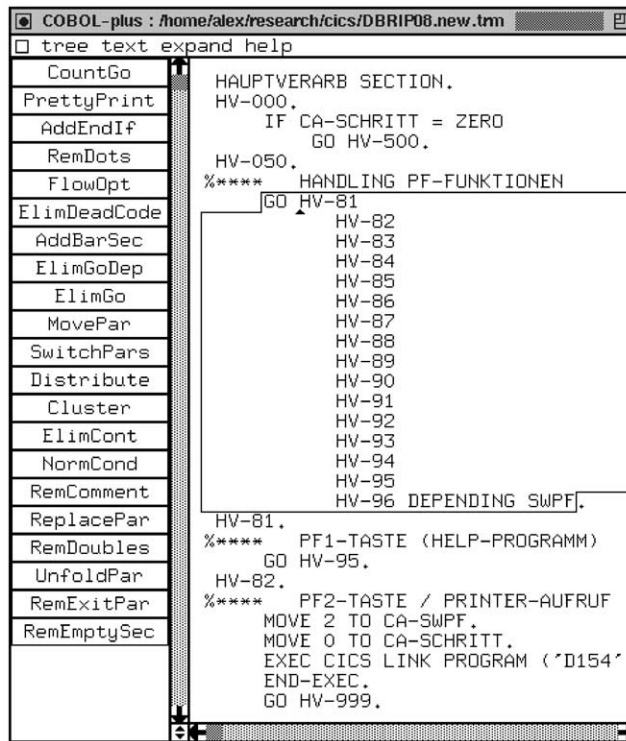
Fig. 2. Restructuring assembly lines containing an original code fragment.

is the start of the main program (HAUPTVERARB SECTION in German). We see three 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 5.1 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. Finally, we compare in Section 5.2 the input with the output of the algorithm. The rather complex coordination of the components is subject to discussion in Section 6.

5.1. 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 [10]. Of course, all reuse speeded up development. We briefly discuss each component of our algorithm implemented in our renovation factory. In fact, each button of Fig. 2 represents a component.

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 (we discuss it below).

PrettyPrint. This is a basic component that we reused. We generate pretty printers using technology discussed in [13]. The only thing that needs to be done by hand is to adapt the formatting to company-specific output.

AddEndIf. We reused this preprocessing component. It has been discussed in detail in [12]. In short, it is a component that adds the explicit scope terminator `END-IF` to `IF` statements. This makes the code more structured. We use this kind of preprocessing also 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. In COBOL there is a lot of syntactic freedom to implement exactly the same functionality. Many statements in COBOL can be optionally separated by periods. We remove all unnecessary ones, so that the code is syntactically more uniform. As a consequence, there is no inconsistent use of separator periods. Also this has the advantage that it makes the patterns that we have to construct simpler than if we can expect dots on arbitrary locations. 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 [48] 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. Dead code is source code that is in a program but that is never executed. This is a useful component that is like `RemDots` both useful for preprocessing as for postprocessing. Also when it is likely that other components will expose obviously dead code, we call this component internally to clean that up. It removes statements that can never be executed, due to a preceding unconditional (explicit) jump instruction. We will see examples of this later on. We like to stress that this tool does not necessarily remove all dead code. Since the algorithm that we developed makes use of moving code from one place to another, dead code becomes apparent due to the code restructuring.

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 PERFORMs—the COBOL way to invoke a subroutine. We note that this component is only necessary to add SUBROUTINES SECTIONs if the code is essentially unstructured. This means that the GO TOs are not simulating conditionals or loops. We note that the PARAGRAPHs in the SUBROUTINES SECTIONs 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 (see examples later on in Section 3.1). 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 TOs are removed. We will see examples in Sections 5.2 and 6.

ElimGo. This component removes a number of GO TOs 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 [10]. We say that a GO TO is almost structured if the GO TO logic is, in fact, simulated structured logic (like a while, an if, en else, branch and so on). Note that the example program discussed in this paper is far from being almost structured. However, removal of almost structured GO TOs simplifies the complexity of other GO TOs, i.e., enables the creation of new almost structured GO TOs. Thus, we can actually replace essentially unstructured code by structured code by properly coordinating our components. We recall that this coordination is explained in full in Section 6.

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 PERFORMs and an additional GO TO to not disturb the control flow. We call this GO TO shifting. 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 new simple jump instructions that we introduced by moving code around. We apply ElimGo and MovePar alternatingly to remove all GO TOs (we discuss this in detail in Section 6).

SwitchPars. This component is a variation of MovePar. This tool 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 it as a typical postprocessing operation. In Sections 5.2 and 6 we will see examples of optimized EVALUATEs.

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 distinct cases appeared to be equal. Cluster takes care of these clones.

ElimCont. This is a component that eliminates CONTINUE statements. We reused this component (discussed in detail in [10]). 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 [10]). 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. Natural means here as prescribed by the local company standard.

The above components are used to eliminate the GO TO logic from COBOL with embedded CICS. 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 [9]. We use this component to *test* whether or not code fragments are equal modulo their comments. Of course, we do *not* remove comments in the actual system.

ReplacePar. This component recognizes pieces of code that are equal in the SUBROUTINES SECTIONs. We restrict ourselves to the SUBROUTINES SECTIONs since there are no jump instructions to the paragraphs in those locations. As soon as this component locates PARAGRAPHs that are equal, it replaces *all* references to both PARAGRAPHs to one of them. We say that two different paragraphs are equal if they are syntactically identical after removal of comment and layout characters. Of course their access labels 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 note that we can only remove such double paragraphs in the SUBROUTINES section since there we have abolished fall-through, that is, when the paragraph occurs twice it will not be executed twice. With fall-through execution we cannot just remove identical copies.

We give a simple example. Here are two paragraphs that should be considered equal.

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

This code is converted to the following lines of code with the `ReplacePar` component. Of course, all occurrences of HV-84 are replaced by references to HV-95 by this component.

```
 HV-85.
****                           PF4-TASTE
   PERFORM HV-95.
 HV-85.
****                           PF5-TASTE
   PERFORM HV-95.
```

The resulting code is subsequently changed into the following code by the `RemDoubles` component.

```
HV-85.
****                           PF4-TASTE
****                           PF5-TASTE
PERFORM HV-95.
```

`ReplacePar` finds the `PARAGRAPH`s 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 there to emphasize that at this location the program flow should exit 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 SECTION`s that were made by the `AddBarSec` component, but appeared not to be necessary. In such, usually small `SECTION`s, the code is almost structured, so that application of the `ElimGo` component cleans the code. In the Internet program we constructed 16 `SUBROUTINES SECTION`s and we needed 8.

5.2. Comparing input and output

The code changes dramatically when we apply our algorithm to the example program. So before we discuss the actual coordination of the components we give an impression of the effect the algorithm has on the code. We will discuss a few pieces of code that can be traced back to the original code that we displayed in Fig. 2.

In the original code (Fig. 2) 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.
```

In the full program (see [52]) the HAUPTVERARB SECTION contained 206 lines of code and restructuring it contains only 7 lines of code. Paragraph HV-050 is performed in the HAUPTVERARB SECTION (see above). It originally contained one large GO TO DEPENDING ON. This paragraph is changed into an EVALUATE statement. We depicted the code in Fig. 3.

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 happened to be one case. Paragraph HV-81 (see Fig. 2) is gone after restructuring, since it only jumped to HV-95. This indirect code is replaced by the (partially restructured) code in Fig. 4.

In paragraph HV-82 of Fig. 2, 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 developed and implemented is correct. This is just to give an indication of the dramatic impact the relatively simple components can have on the input when applied many times in a complex coordination. In the next section we show in detail how this impact is achieved.

```
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.
```

Fig. 3. The recovered EVALUATE statement.

```
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.
```

Fig. 4. The partially restructured code.

6. Systolic structuring in steps

In this section we will discuss the coordination of the components that we introduced in Section 5. Recall that we introduced some 20 small and simple components, each with a very specific but clear task. The crux of the algorithm is that by using the components over and over again in a complex coordination we evolve a program

from an essentially unstructured one to a well-structured program that is suited for componentization.

The algorithm resembles the architecture of so-called systolic arrays: the behavior of identical components in the small used many times resembles the same behavior but then in the large. In our case, we have several simple components, only capable of solving the most elementary form of a problem. But due to the reiterated application of such simple components together, we are able to solve the problems on a larger scale. Such an approach is also known as a systolic algorithm. There is no formal definition of systolic algorithms [41], but characteristics of it are the use of many similar cells which rhythmically compute and pass data through a system [36,37].

As an illustration, we provide a typical nonVLSI example of a systolic algorithm: palindrome recognition. A palindrome is a word or phrase that reads the same backward or forward, like *sexes*. By connecting a lot of simple palindrome recognizers that can only deal with length 2 palindromes, it is possible to create a palindrome recognizer that can handle palindromes of arbitrary length [34].

Our approach is also systolic in the above sense: we have components that can remove obviously dead code, or eliminate very easy jump instructions in favor of other more natural constructions of the language. The power of connecting those simple components and applying them over and over again leads to the removal of dead code that is much harder to detect and the elimination of GO TOs that cannot be removed straightforwardly by introducing IFs or WHILE constructs.

To illustrate the working of the algorithm we apply it on a small but representative example program. This program is derived from a more involved program that is published on the Internet [52]. On that location, an entire production program that we restructured is present with all 70 intermediate steps and a summary of the differences between each step as generated by the Unix utility diff. Of course, we cannot include 5000 LOC files into a research paper. So to keep the paper fully self-contained, we derived from the large program a small sample that shows the working of the algorithm. Now and then we will make comparisons to the full program to give the reader an idea of how the work scales for larger programs.

6.1. A detailed example

In Fig. 5 we depicted a very small example containing a snippet of the real program that we published on the Internet [52]. Of course, the code snippet in Fig. 5 is not a complete program, but perfectly fit to illustrate the systolic structuring algorithm. For example, it does contain a limited version of the GO TO DEPENDING ON as we depicted earlier in Fig. 2. How many GO TOs could you remove yourself at first glance? We think that the code fragment contains several jump instructions that are not trivial to remove so we guess that you found only a few jump instructions that can be easily removed.

The algorithm starts with preprocessing the code such that it is in good shape for the systolic part. The first step is to introduce a subroutine section for every section in the program. In Fig. 6 this is illustrated. We can see that indeed not much is changed in the code, except that we precooked it to be able to contain subroutines that we

Fig. 5. The original fragment.

extract later on. We added a section with a name derived from the original section. It is suited to store subroutines in it. We put a first paragraph in it with a STOP RUN, to prevent execution of the subroutines via fall-through. Notice that the program transformation that we carried out did not introduce any semantical difference with the original program code.

   In the next step we eliminate the GO TO DEPENDING ON logic. Since this statement is in fact nothing more than a case statement with as only possible cases GO TO statements, this is merely a syntax swap. The logic is not improved, it is only represented in a different manner. We have now 8 jump instructions in total in Fig. 7. Note again that the semantics of the new program did not change.

   The code is now massaged in such a manner that we can start to remove jump instructions. In theory, pressing the ElimGo button once, could remove all jump instructions. This only happens, however, if the use of GO TOs is restricted to simulation of conditional and iterative constructs. In practice, this is hardly ever the case. To give you an idea, in the code published on the Internet [52] the maximum number of GO TOs is 102 and after pressing the first time the ElimGo we still have 63 GO TOs left. This means that about 40 jump instructions were rather easy. In fact, we think that the

```
┌──────────────────────────────────────────────────────────┐
│ ● COBOL-plus : /home/x/cics/Example.02 ▓▓▓▓▓▓▓▓▓    回     │
├──────────────────────────────────────────────────────────┤
│ □ tree text expand help                                    │
├─────────────┬──────────────────────────────────────────────│
│  CountGo    │↑│                                             │
│ PrettyPrint │ │  HV-050.                                    │
│  AddEndIf   │ │     GO                                      │
│  RemDots    │ │        HV-81                                │
│  RemThen    │ │        HV-82                                │
│  FlowOpt    │ │        HV-83                                │
│ElimDeadCode │ │        HV-95 DEPENDING SWPF.                │
│  AddBarSec  │ │                                             │
│  ElimGoDep  │ │  HV-81.                                     │
│   ElimGo    │ │     MOVE 2 TO CA-SWPF                       │
│   MovePar   │ │     GO HV-999.                              │
│  SwitchPars │ │                                             │
│  Distribute │ │  HV-82.                                     │
│   Cluster   │ │     GO HV-95.                               │
│  ElimCont   │ │                                             │
│  NormCond   │ │  HV-83.                                     │
│  RemLabels  │ │     GO HV-95.                               │
│  RemComment │ │                                             │
│  ReplacePar │ │  HV-84.                                     │
│  RemDoubles │ │     MOVE 4 TO CA-SWPF.                      │
│  UnfoldPar  │ │                                             │
│  RemExitPar │ │  HV-95.                                     │
│  RemEmptySec│ │     IF SWFEHL > ZERO                        │
│   ApplyAll  │ │        GO HV-999                            │
│  COBOLtoRSF │ │     END-IF                                  │
│             │ │     PERFORM MODULE.                         │
│             │ │                                             │
│             │ │  HV-999.                                    │
│             │ │     EXIT.                                   │
│             │ │                                             │
│             │ │  BAR SECTION.                               │
│             │ │                                             │
│             │ │  BAR-PARAGRAPH.                             │
│             │ │     STOP RUN.                               │
│             │↓│                                             │
│             │ │  HAUPTVERARB-SUBROUTINES SECTION.           │
└─────────────┴──────────────────────────────────────────────┘
```

Fig. 6. Addition of a simulated subroutine location.

average reader found only a few GO TOs in the code snippet that could be naturally removed. The rest is not so easy.

Since our ElimGo component is simple, it indeed only removes a single GO TO. The result is presented in Fig. 8. This difference between the former and this code is that a jump instruction has been eliminated in favor of a conditional statement (in HV-95).

Now we have to do something smart so that we can reuse the simple GO TO eliminator once more. What we do is we move paragraphs that do not contain jump instructions to the subroutine location. Of course we cannot just move such code around without destroying the control-flow. Therefore, we preserve the control-flow by modifying calls to this piece of code. We note that this task is not entirely trivial. In order for the reader to appreciate this, we will discuss the control-flow mechanisms of COBOL briefly. The labels introducing a paragraph in a COBOL program serve a dual purpose: first of all they are the entry-points for jump instructions. Second, they serve as scope-terminators for the previous paragraph when it is called by a subroutine call. In a way, paragraphs thus serve as subroutines. However, there is no natural mechanism like in C or Pascal where we can store subroutines and then invoke them from a main program. In COBOL, all "subroutines" are executed in the order they appear in the code. Although some of the readers may think that this is strange, it reflects one of the original design goals

Fig. 7. Elimination of the GO TO DEPENDING ON.

of COBOL, namely that it should be as close as possible to English (more informa-
tion on the design goals of COBOL and maintenance problems is provided in [45]).
Paragraphs in this text also flow from top to bottom. We all know that in order to
edit a text, we can write little signs in the text and put remarks in the margins. Then
we have to rewrite the text and also repair the flow when modifications are not local.
The same thing happens with COBOL programs, only the signs in the code are GO
TOs, and since there are no margins, the remarks become new paragraphs. Since the
old version is not cleaned up as an English text, a COBOL program will contain the
entire edit history of all the modifications of the previous versions. We believe that this
patch mechanism is a possible cause for entangled programs: it encourages the use of
jump instructions and inclusion of dead code. Where to put a new subroutine without
destroying the current control-flow is not at all a trivial task. So, when we start to
move code around in an automated fashion we have to take care of this idiosyncratic
control-flow mechanism of COBOL, too.

We proceed looking at our example. In Fig. 9 we see that two paragraphs were
transported to the subroutine section. They are HV-84 and HV-95. We discuss the effects
on HV-95 since they are visible in Fig. 9. ElimGoDep converted the GO TO HV-95 in
the EVALUATE statement of Fig. 8 to a PERFORM HV-95 statement followed by a GO

```
┌─────────────────────────────────────────────────────────────┐
│ ● COBOL-plus : /home/x/cics/Example.02                     回 │
│ □ tree text expand help                                       │
│ ┌───────────┐ ▲   HV-050.                                     │
│ │  CountGo  │     EVALUATE SWPF                               │
│ │ PrettyPrint│      WHEN 1 GO HV-81                           │
│ │  AddEndIf │       WHEN 2 GO HV-82                           │
│ │  RemDots  │       WHEN 3 GO HV-83                           │
│ │  RemThen  │       WHEN 4 GO HV-95                           │
│ │  FlowOpt  │     END-EVALUATE.                               │
│ │ElimDeadCode│                                                │
│ │ AddBarSec │     HV-81.                                      │
│ │ ElimGoDep │       MOVE 2 TO CA-SWPF                         │
│ │  ElimGo   │       GO HV-999.                                │
│ │  MovePar  │     HV-82.                                      │
│ │ SwitchPars│       GO HV-95.                                 │
│ │ Distribute│                                                 │
│ │  Cluster  │     HV-83.                                      │
│ │ ElimCont  │       GO HV-95.                                 │
│ │ NormCond  │                                                 │
│ │ RemLabels │     HV-84.                                      │
│ │ RemComment│       MOVE 4 TO CA-SWPF.                        │
│ │ ReplacePar│                                                 │
│ │ RemDoubles│     HV-95.                                      │
│ │ UnfoldPar │       IF SWFEHL > ZERO                          │
│ │ RemExitPar│         CONTINUE                                │
│ │ RemEmptySec│      ELSE                                      │
│ │  ApplyAll │         PERFORM MODULE                          │
│ │ COBOLtoRSF│       END-IF.                                   │
│ └───────────┘     HV-999.                                     │
│                     EXIT.                                     │
│                   BAR SECTION.                                │
│                   BAR-PARAGRAPH.                              │
│                     STOP RUN.                                 │
└─────────────────────────────────────────────────────────────┘
```

Fig. 8. First round of easy GO TO removal.

TO HV-999. MovePar uses this so-called GO TO-shifting, so that the paragraphs in the SUBROUTINES SECTION are not accessible via jumps. Thus simulating a real subroutine area comparable to other programming languages. The GO TO-shifting was necessary since after the original jump instruction the control-flow would not jump back to the case statement, but continue via fall-through to the EXIT paragraph. MovePar did this on all other locations and for other labels like HV-84. Moreover, the locations that formerly contained the code of the moved paragraphs are now represented by mere PERFORM statements. See the two PERFORM statements in paragraph HV-83 (Fig. 9).

The application of the MovePar component, introduced THEN statement connectors, to ease automated code transformations. We built a tool to remove this. In Fig. 10 we depicted the result of removing the THEN statement connectors using RemThen. In fact, this tool could be seen as a language dependent trick. What we actually wanted to do in the former step is to turn a single statement into two statements. For, one GO TO became a PERFORM plus another GO TO. When dealing with tools that work on the underlying tree structure of a program, the result should be a correct tree again. So a next component that is based on the grammar can deal with the tree as well. If we convert one statement into two, the resulting underlying tree of the code does not need to comply with the grammar residing in the next tool. In fact, for any

COBOL-plus : /home/x/cics/Example.02

□ tree text expand help

```
CountGo
PrettyPrint
AddEndIf
RemDots
RemThen
FlowOpt
ElimDeadCode
AddBarSec
ElimGoDep
ElimGo
MovePar
SwitchPars
Distribute
Cluster
ElimCont
NormCond
RemLabels
RemComment
ReplacePar
RemDoubles
UnfoldPar
RemExitPar
RemEmptySec
ApplyAll
COBOLtoRSF
```

```
        PERFORM HV-95 THEN
          GO HV-999
      END-EVALUATE.

HV-81.
    MOVE 2 TO CA-SWPF
    GO HV-999.

HV-82.
    PERFORM HV-95 THEN
    GO HV-999.

HV-83.
    PERFORM HV-95 THEN
    GO HV-999
    PERFORM HV-84
    PERFORM HV-95.

HV-999.
  EXIT.

BAR SECTION.

BAR-PARAGRAPH.
  STOP RUN.

HAUPTVERARB-SUBROUTINES SECTION.

HV-84.
  MOVE 4 TO CA-SWPF.

HV-95.
  IF SWFEHL > ZERO
    CONTINUE
  ELSE
    PERFORM MODULE
  END-IF.
```

Fig. 9. First round of subroutine moving.

conversion step where the number of statements is not an invariant (of which there are many) we would then have to write some special code to still have a correct tree. Instead, we split this problem into two smaller problems: first we use some grammar glue to keep the number of statements invariant, and only in a second step we apply a general-purpose tool that removes the glue so that we can break up any number of glued statements into their natural parts. In this way, the number of statements stays invariant during a program transformation. In the case of COBOL the glue is natively available by means of the THEN statement connector. The language dependence in this solution is, therefore, the use of the THEN. Is it possible to generate for any context-free grammar glue support so that also in case there is no native gluing mechanism available in the language we can still use it to simplify program transformation tasks. For more details on grammar transformations and their use we refer to [51,38]. In those papers tools and technology are discussed that enables grammar extensions such that gluing support can be made available in a completely automated fashion. The usefulness of grammar glue has first been recognized by us in COBOL and the THEN statement connector materialized as a first-order approximation of a solution. Later on we generalized on the problem and recognized that structural language independent glue support is crucial if we want to make a serious case of automated

```
●  COBOL-plus : /home/x/cics/Example.02                        回
□  tree  text  expand  help
┌─────────────┬╥──────────────────────────────────┐
│   CountGo   │▲│⌶IDENTIFICATION DIVISION.          │
│  PrettyPrint│ │                                   │
│   AddEndIf  │ │ PROGRAM-ID. XXXXXXX.               │
│   RemDots   │ │                                   │
│   RemThen   │ │ PROCEDURE DIVISION.                │
│   FlowOpt   │ │                                   │
│ ElimDeadCode│ │ HAUPTVERARB SECTION.               │
│   AddBarSec │ │                                   │
│  ElimGoDep  │ │ HV-050.                            │
│    ElimGo   │ │    EVALUATE SWPF                   │
│   MovePar   │ │       WHEN 1 GO HV-81              │
│  SwitchPars │ │       WHEN 2 GO HV-82              │
│  Distribute │ │       WHEN 3 GO HV-83              │
│   Cluster   │ │       WHEN 4                       │
│   ElimCont  │ │          PERFORM HV-95             │
│   NormCond  │ │          GO HV-999                 │
│   RemLabels │ │     END-EVALUATE.                  │
│  RemComment │ │                                   │
│  ReplacePar │ │ HV-81.                             │
│  RemDoubles │ │    MOVE 2 TO CA-SWPF               │
│   UnfoldPar │ │    GO HV-999.                      │
│  RemExitPar │ │                                   │
│ RemEmptySec │ │ HV-82.                             │
│   ApplyAll  │ │    PERFORM HV-95                   │
│  COBOLtoRSF │ │    GO HV-999.                      │
│             │ │                                   │
│             │ │ HV-83.                             │
│             │ │    PERFORM HV-95                   │
│             │ │    GO HV-999                       │
│             │ │    PERFORM HV-84                   │
│             │ │    PERFORM HV-95.                  │
│             │ │                                   │
│             │ │ HV-999.                            │
│             │ │    EXIT.                           │
│             │▼│                                   │
│             │⇕│ BAR SECTION.                       │
└─────────────┴─┴──────────────────────────────────┘
```

Fig. 10. Removal of THEN statement connectors.

code transformations. That is why we need the tool to remove the THENs. Now we apply the next step.

In Fig. 11 we applied the ElimDeadCode component. This component is just like the simple ElimGo component a basic one. It will not remove code that appears to be dead in a very contrived way, but only removes code of which there is no doubt that it is dead. For instance, the last two lines of paragraph HV-83 are directly preceded by a non-conditional GO TO HV-999 (in Fig. 10). Obviously the two lines of code are dead and can be removed. Typically, the ElimDeadCode component will operate on code fragments that are created by other components participating in the systolic algorithm. However, the presence of dead code in the output of components means that there actually was dead code in the original program, although not easily seen (see paragraph HV-84). Step-by-step restructuring in this manner always led to exposition of dead code so that it was easily removable using the simple dead code eliminator.

This answers the question that some readers might have at this point: why is that dead code over there in the program? It is so obviously dead, how can someone do this? Note that although those pieces of code will never be executed via fall-through at that particular location, the code itself is not dead: it is executed from a possibly distant GO TO in the program, and happens to be located at a location where it cannot

Fig. 11. Removal of obviously dead code.

interfere with earlier written code. In fact, this is an example of an ad hoc subroutine: it is preceded by an unconditional GO TO so there is no fall-through. We converted this ad hoc solution to a more structural solution.

Now that we have removed the obviously dead code, we have massaged the code in such a manner that again a very simple GO TO has emerged that is a perfect candidate for our simplistic ElimGo tool. The jump instruction that we refer to is the GO HV-999 which is the last line of paragraph HV-83. Since HV-999 is the adjacent paragraph, the GO TO statement is in fact totally superfluous. This is the type of GO TO that ElimGo can handle. So after pressing the button ElimGo we end up with the code as we depicted it in Fig. 12.

But now ElimGo created a new opportunity to move a paragraph that does not contain jump instructions to the subroutine section. It is the HV-83 paragraph. In Fig. 13 we show the result of using MovePar again: indeed, the paragraph is now in the subroutine section and on all locations where the paragraph was called Move Par modified the code with a PERFORM and a GO TO.

We remove the THEN statement connectors again and we remove the obviously dead code. This results in the code that is depicted in Fig. 14. The tools have now massaged the code again such that trivial jump instruction emerged. It is the GO HV-999 in
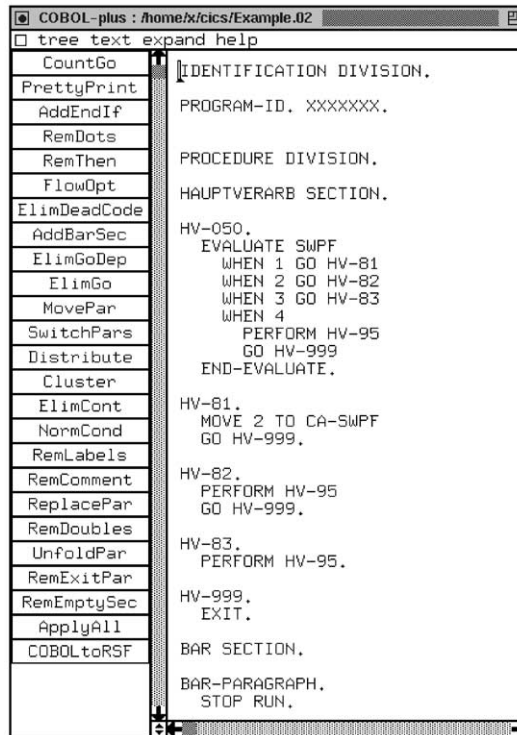
```
┌──────────────────────────────────────────────────────┐
│ ● COBOL-plus : /home/x/cics/Example.02          回     │
│ □ tree text expand help                                │
│ ┌────────────┐┌───────────────────────────────────┐▲  │
│ │  CountGo   ││IDENTIFICATION DIVISION.           │   │
│ │ PrettyPrint││                                   │   │
│ │  AddEndIf  ││PROGRAM-ID. XXXXXXX.               │   │
│ │  RemDots   ││                                   │   │
│ │  RemThen   ││PROCEDURE DIVISION.                │   │
│ │  FlowOpt   ││                                   │   │
│ │ElimDeadCode││HAUPTVERARB SECTION.               │   │
│ │  AddBarSec ││                                   │   │
│ │  ElimGoDep ││HV-050.                            │   │
│ │   ElimGo   ││   EVALUATE SWPF                   │   │
│ │   MovePar  ││      WHEN 1 GO HV-81              │   │
│ │ SwitchPars ││      WHEN 2 GO HV-82              │   │
│ │ Distribute ││      WHEN 3 GO HV-83              │   │
│ │  Cluster   ││      WHEN 4                       │   │
│ │  ElimCont  ││         PERFORM HV-95             │   │
│ │  NormCond  ││         GO HV-999                 │   │
│ │ RemLabels  ││   END-EVALUATE.                   │   │
│ │ RemComment ││HV-81.                             │   │
│ │ ReplacePar ││   MOVE 2 TO CA-SWPF               │   │
│ │ RemDoubles ││   GO HV-999.                      │   │
│ │ UnfoldPar  ││                                   │   │
│ │ RemExitPar ││HV-82.                             │   │
│ │RemEmptySec ││   PERFORM HV-95                   │   │
│ │  ApplyAll  ││   GO HV-999.                      │   │
│ │ COBOLtoRSF ││                                   │   │
│ │            ││HV-83.                             │   │
│ │            ││   PERFORM HV-95.                  │   │
│ │            ││                                   │   │
│ │            ││HV-999.                            │   │
│ │            ││   EXIT.                           │   │
│ │            ││                                   │   │
│ │            ││BAR SECTION.                       │   │
│ │            ││                                   │   │
│ │            ││BAR-PARAGRAPH.                     │   │
│ │            ││   STOP RUN.                       │▼  │
│ └────────────┘└───────────────────────────────────┘   │
└──────────────────────────────────────────────────────┘
```

Fig. 12. Second round of easy `GO TO` removal.

paragraph `HV-82`. We note that it is not a coincidence that we end up with removing these trivial jumps each time. For, the original COBOL coders were not programming in an unstructured manner: they were deliberately coding to reach the single `EXIT` of a section. After numerous stages of maintenance, it took many intermediate steps to get there, and the more maintenance had been done on such code, the more intricate the jumping behavior tended to become. To add new code, you have to put it somewhere. This often implies that it is put on some arbitrary hopefully safe location and that with the use of jump instructions the original control-flow is maintained. What the tools do, is to roll back the coding history and saving the cleansed paragraphs in a subroutine section and thus simulating a programming language that has native support for subroutines. As soon as the tools place the subroutines in a logical place where ordering does not matter since the code cannot be accessed via fall-through, we are on our way to solve the code location problem. Moreover, by unraveling the code in such a way, more and more structured code will emerge, and the simple `GO TO`s will emerge, one at a time. And those can be removed in a trivial way using `ElimGo`.

All in all, the `MovePar` tool transports the next piece of structured code to a safe haven. Again the `RemThen` tool removes the `THEN`s and `ElimDeadCode` scraps the obviously dead code. The components that can introduce dots, like the `MovePar` component,
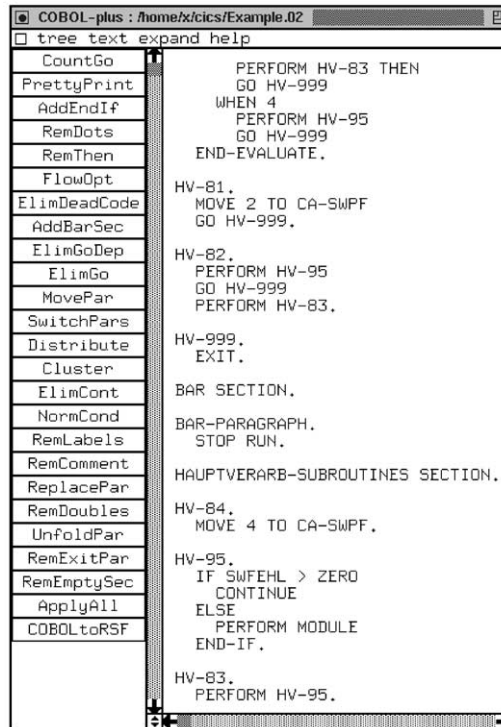
Fig. 13. Second round of subroutine moving.

internally call the RemDots component to clean up the small things like separator periods. We did not mention this before since we want to show the heartbeat of the algorithm and not focus on the smallest details simultaneously.

The result is visible in Fig. 15. As we can see in this display, we have again a simple jump instruction that we can just remove with the ElimGo button. The result is displayed in Fig. 16.

Again, we can fruitfully apply the MovePar button to move paragraph HV-81 to the subroutine section. This is shown in Fig. 17.

At this moment we cannot just continue with the systolic algorithm and remove more jump instructions: there are no more easy GO TOs in the code. As we can see in Fig. 17 the only jumps that are left all reside inside the EVALUATE statement. In Fig. 17 the paragraph moving operations revealed that the distinct jump instructions in the GO TO DEPENDING ON that we originally started with, are all reduced to GO TO HV-999. The intermediate steps that were put in the program, are now represented by PERFORM statements. In the case of this small example program for each case there is one PERFORM statement.

Next we distribute the GO HV-999 out of the case statement: after all, in all cases this statement will be applied. We press the Distribute button. The effect is that the

```
COBOL-plus : /home/x/cics/Example.02
□ tree text expand help
  CountGo        IDENTIFICATION DIVISION.
  PrettyPrint
  AddEndIf       PROGRAM-ID. XXXXXXX.
  RemDots
  RemThen        PROCEDURE DIVISION.
  FlowOpt
  ElimDeadCode   HAUPTVERARB SECTION.
  AddBarSec
  ElimGoDep      HV-050.
  ElimGo            EVALUATE SWPF
  MovePar              WHEN 1 GO HV-81
  SwitchPars           WHEN 2 GO HV-82
  Distribute           WHEN 3
  Cluster                 PERFORM HV-83
  ElimCont                GO HV-999
  NormCond             WHEN 4
  RemLabels               PERFORM HV-95
  RemComment              GO HV-999
  ReplacePar         END-EVALUATE.
  RemDoubles
  UnfoldPar      HV-81.
  RemExitPar        MOVE 2 TO CA-SWPF
  RemEmptySec       GO HV-999.
  ApplyAll
  COBOLtoRSF     HV-82.
                    PERFORM HV-95
                    GO HV-999.

                 HV-999.
                    EXIT.

                 BAR SECTION.

                 BAR-PARAGRAPH.
                    STOP RUN.

                 HAUPTVERARB-SUBROUTINES SECTION.
```

Fig. 14. Second round of `THEN` and dead code removal.

clauses of the `EVALUATE` are all combined, and adjacent to the case statement an `IF` is plugged in containing the common `GO TO` statement. The result of the transformation is shown in Fig. 18. Of course we have to remove `THEN` statement connectors and the obviously dead code (if any), before we can continue with the `GO TO` removal as usual.

In Fig. 19 we have depicted the result of pressing the `ElimGo` button again. As can be seen from the code: the very first `GO TO` that was eliminated in this example had exactly the same pattern as the one we removed here. So at this point we have removed all `GO TO` statements. In the case of the program that we put on the Internet [52], we needed about 60–70 steps to come to the point where all `GO TO`s were gone.

Of course, our ultimate goal was not to just remove jump instructions (as indicated in Section 2), but to massage the code in such a way that we can reuse it in a component-based context. We have seen that the control-flow of the code was not at all optimal. Of course, this has not been fully solved by our systolic algorithm. In fact, the code suffers still from a lot of indirect jumping, albeit that it is no longer expressed by `GO TO`s. Consider, for example, in the original program (see Fig. 5) paragraph `HV-82`. In that paragraph we only jump to `HV-95` which we could have

Fig. 15. Third round of THEN and dead code removal.

done that more directly, by jumping to that paragraph from the start. We did not solve that issue in the beginning since we wanted to focus on one problem at a time. But now the time has come. We implemented a simple tool called UnfoldPar that removes such indirect code and replaces it with more direct code. The tool was relatively simple to implement since we do not need to take the possibility of weird jumps into account. The resulting output of this tool is presented in Fig. 20. As can be seen, from the four cases in the EVALUATE there are only two left: HV-81 and HV-95. In the large program we put on the Internet [52] we went from 16 cases to 7 (see Fig. 3). Also note that in the subroutine section there are just two paragraphs left (see Fig. 20 again). Both paragraphs contain something real, one could call such paragraphs candidates for business logic. In the simulated main section the coordination of the business logic resides. In fact, our algorithm is a prerequisite to detect business logic in source code without going to a higher abstraction level. We make the coordination of candidate business logic transparent. Then we can inspect all the paragraphs one by one. We can modify the coordination. Moreover, we can insert new subroutines, modify old ones, and delete subroutines without affecting the control-flow. This means that the programs have become modular, change-enabled, and in a sense componentized since only part of the functionality can be invoked at wish.
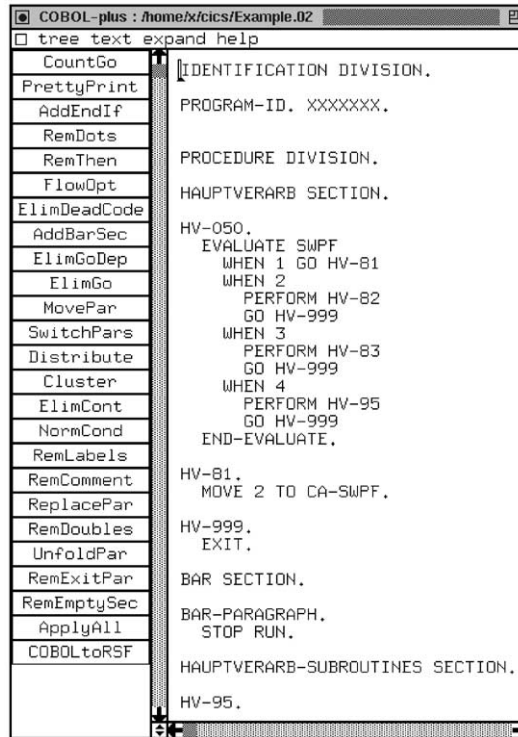
Fig. 16. Fourth round of easy GO TO removal.

A special-purpose transformation, called `Cluster`, cleans up the EVALUATE statement and eliminates case clones by combining the WHEN clauses. The result is displayed in Fig. 21.

We postprocess the code and shape it up as much as possible. The `ElimCont` tool removes all CONTINUE statements that were introduced during the other program transformations. The result is visible in Fig. 22.

Of course, `ElimCont` affected conditionals, and also earlier program transformations made changes to the conditionals, just think of the WHEN clause clustering. We shape up these logically challenged conditionals. The result of pressing the `NormCond` button is depicted in Fig. 23. For instance, WHEN 4 OR 3 OR 2 was converted to 2 THRU 4, and NOT (SWFEHL > ZERO) normalized to SWFEHL <= ZERO, and so on.

Finally, we can get rid of the EXIT paragraph. We recall that the EXIT has no semantic meaning. It is introduced as a type of comment so that programmers could keep track where to jump to in the end so that each section would have one exit location. We have converted to a completely different paradigm for control-flow; namely, in the HAUPTVERARB section we only have coordination code and in the subroutines section we have calculations. So the control-structure is entirely different. The EXIT paragraph has no role anymore, and since it has no semantics as well, we can just
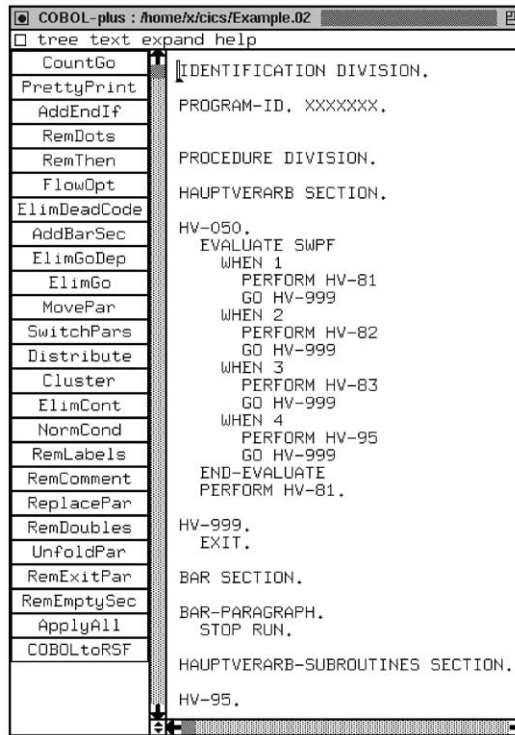
Fig. 17. Fourth round of subroutine moving.

remove it. We do this with the `RemExitPar` button; the resulting code is depicted in Fig. 24.

6.2. The coordination of the components

Until now the coordination of the components was discussed in an implicit manner. We discuss here the full picture. Of course, we do not want to interactively restructure all the programs, so we take the automation of the restructuring a step further by giving the coordination of the components for the general case. It will turn out that the coordination is so regular that it can be automated. The coordination of the components has three main phases: preprocessing, main processing and postprocessing. We discuss them below.

preprocessing. In this phase we do all the work to shape up the code as much as possible prior to start the real work. To enable the systolic flow in the algorithm we need to run the following tools in the order below:

- PrettyPrint
- AddEndIf

```
┌──────────────────────────────────────────────────────┐
│ ● COBOL-plus : /home/x/cics/Example.02          ▓▓▓ 凹│
├──────────────────────────────────────────────────────┤
│ □ tree text expand help                               │
├─────────────┬──────────────────────────────────────┬─┤
│   CountGo   │█IDENTIFICATION DIVISION.             ▲│ │
│ PrettyPrint │                                        │ │
│   AddEndIf  │ PROGRAM-ID. XXXXXXX.                   │ │
│   RemDots   │                                        │ │
│   RemThen   │ PROCEDURE DIVISION.                    │ │
│   FlowOpt   │ HAUPTVERARB SECTION.                   │ │
│ ElimDeadCode│                                        │ │
│  AddBarSec  │ HV-050.                                │ │
│  ElimGoDep  │    EVALUATE SWPF                       │ │
│    ElimGo   │       WHEN 1 PERFORM HV-81             │ │
│   MovePar   │       WHEN 2 PERFORM HV-82             │ │
│ SwitchPars  │       WHEN 3 PERFORM HV-83             │ │
│  Distribute │       WHEN 4 PERFORM HV-95             │ │
│   Cluster   │    END-EVALUATE THEN                   │ │
│   ElimCont  │    IF SWPF >= 1 AND SWPF <= 4          │ │
│   NormCond  │       GO HV-999                        │ │
│  RemLabels  │    END-IF                              │ │
│  RemComment │    PERFORM HV-81.                      │ │
│  ReplacePar │                                        │ │
│  RemDoubles │ HV-999.                                │ │
│  UnfoldPar  │    EXIT.                               │ │
│  RemExitPar │ BAR SECTION.                           │ │
│ RemEmptySec │                                        │ │
│   ApplyAll  │ BAR-PARAGRAPH.                         │ │
│  COBOLtoRSF │    STOP RUN.                           │ │
│             │                                        │ │
│             │ HAUPTVERARB-SUBROUTINES SECTION.       │ │
│             │                                        │ │
│             │ HV-95.                                 │ │
│             │    IF SWFEHL > ZERO                    │ │
│             │       CONTINUE                         │ │
│             │    ELSE                                │ │
│             │       PERFORM MODULE                   │ │
│             │    END-IF.                            ▼│ │
├─────────────┴──────────────────────────────────────┴─┤
│  ≑ ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓      │
└──────────────────────────────────────────────────────┘
```

Fig. 18. Distribution of GO TO over EVALUATE.

- RemDots
- FlowOpt
- ElimDeadCode
- AddBarSec
- ElimGoDep

   Once we ran all these tools, we have removed a lot of irregularities so that the main processing can commence.

main processing. In this phase, the components do still have relatively clear tasks, but their implementation can be more involved. For instance our simple ElimGo component does recognize over 30 GO TO removal patterns. This is due to the many variations that we found in source code. Therefore, the preprocessing phase is not a luxury issue of beautifying the code: without such a phase the number of different patterns would explode. For instance, the use of a separator dot is often to the discretion of the programmer. We remove all unnecessary ones, so that we do not have to implement for one pattern all its variations with and without all combinations of dots. In this way we cap the exponential growth of the number of patterns in the preprocessing phase. We have already seen that the coordination of the main processing components is not a

```
 ● COBOL-plus : /home/x/cics/Example.02          回
 □ tree text expand help
┌──────────────┬─────────────────────────────────┐
│   CountGo    │ │IDENTIFICATION DIVISION.         │
│  PrettyPrint │                                   │
│   AddEndIf   │  PROGRAM-ID. XXXXXXX.              │
│   RemDots    │                                   │
│   RemThen    │  PROCEDURE DIVISION.              │
│   FlowOpt    │                                   │
│ ElimDeadCode │  HAUPTVERARB SECTION.             │
│   AddBarSec  │                                   │
│  ElimGoDep   │  HV-050.                          │
│    ElimGo    │     EVALUATE SWPF                 │
│    MovePar   │        WHEN 1 PERFORM HV-81       │
│  SwitchPars  │        WHEN 2 PERFORM HV-82       │
│  Distribute  │        WHEN 3 PERFORM HV-83       │
│   Cluster    │        WHEN 4 PERFORM HV-95       │
│   ElimCont   │     END-EVALUATE                  │
│   NormCond   │     IF SWPF >= 1 AND SWPF <= 4    │
│  RemLabels   │        CONTINUE                   │
│  RemComment  │     ELSE                          │
│  ReplacePar  │        PERFORM HV-81              │
│  RemDoubles  │     END-IF.                       │
│  UnfoldPar   │  HV-999.                          │
│  RemExitPar  │     EXIT.                         │
│  RemEmptySec │                                   │
│   ApplyAll   │  BAR SECTION.                     │
│  COBOLtoRSF  │  BAR-PARAGRAPH.                   │
│              │     STOP RUN.                     │
│              │                                   │
│              │  HAUPTVERARB-SUBROUTINES SECTION. │
│              │                                   │
│              │  HV-95.                           │
│              │     IF SWFEHL > ZERO              │
│              │        CONTINUE                   │
│              │     ELSE                          │
│              │        PERFORM MODULE             │
└──────────────┴─────────────────────────────────┘
```

Fig. 19. Fifth round of easy `GO TO` removal.

simple pipeline, but resembles a systolic algorithm. But let us first list the components that are part of the main processing phase:

- ElimGo
- MovePar
- SwitchPars
- RemThen
- RemDots
- ElimDeadCode
- Distribute

In the production version of these components, we combine certain pre- and post-processing tools into more involved components, i.e., we already know that the `MovePar` will probably reveal dead code, so we have built-in this postprocessing component `ElimDeadCode` in the production version of the `MovePar`. In general, the more complex components are themselves small assembly lines: they start with preprocessing, then they do their main operation, and then things are cleaned up in a small postprocessing step. If we in the sequel talk about components like `ElimGo` or `MovePar` we talk about the production versions used in a fully automated software renovation factory.

Fig. 20. Removal of indirect code.

First of all, we alternately run `ElimGo` and `MovePar` until the code does not change anymore. Usually, this does not imply that all the `GO TO`s are gone: it just means that the easy ones are gone. Then we break out of the loop and try the `Distribute` if there was a `GO TO DEPENDING ON` in the original program, which probably moves a common `GO TO` out of its body. Also the `Distribute` component is in the production environment enriched with pre- and postprocessing components. Then we continue with the `ElimGo`, `MovePar` array. If we are again stuck and not ready, and there is no common jump instruction to be extracted from case statements, we use the `SwitchPars` tool. In some cases, there is a simple automated check that ensures we can move paragraphs to other locations. To get the idea: think of pieces of code that are entirely separated by unconditional jump instructions at the end. In fact, we use these ultra bad parts of the code to move a paragraph with a jump back to a distant location nearer to its entry-point label. Often we turn it into a `GO TO` to a location below the label to jump to, so that the `ElimGo` component can probably deal with it. So when all else fails and there are still `GO TO` statements, invoking `SwitchPars` will switch code around in the hope that we can reenter the `ElimGo`, `MovePar` loop. Note that we are not at all claiming that this systolic algorithm will remove all jump instructions from all programs. We only claim that for all the practical cases that we encountered we could

Fig. 21. Clustering of case clones in `EVALUATE`.

remove them using this algorithm. For code we could not handle, we always had the following process to adapt the algorithm: we had to add an extra pattern to the `ElimGo` tool to add yet another variation of the use of `GO TO`s in an almost structured way. To get back to the algorithm, as soon as the `GO`s are gone, we can start to postprocess the code.

postprocessing. In this phase, life is getting a lot easier for us. In the main processing phase we got rid of control-flow problems that are hardly controllable. But now we have a rudimentary structure that already resembles bad C or Pascal programs: there is a "main" for each section, and we have a subroutine section for all of them. The code is now change enabled, and depending on the final application you had in mind it is now possible to either by hand or automatically further process the code. In our case study it was massaging the code so that it could be further componentized. We list the components that we applied to the code in the end:
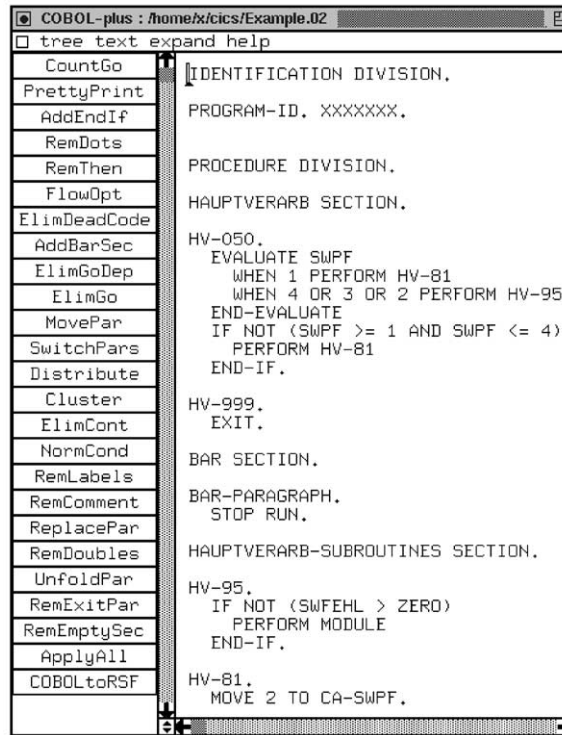
- Cluster
- ElimCont
- NormCond

Fig. 22. Elimination of `CONTINUE` statements.

- FlowOpt
- RemComments
- ReplacePar
- RemDoubles
- ReplacePar
- UnfoldPar
- RemExitPar
- RemEmptySec

Just like the preprocessing components, these postprocessing tools are also applied one by one.

In Fig. 25 the application of the components is at the vertical axis, and the consecutive steps are on the horizontal axis. The curve represents the heartbeat of the systolic structuring algorithm. We call such diagrams "cardiograms". The three main phases can readily be detected from the form of the curve. The irregularities during the repetitive part are clearly visible in the graph. We note that the cardiogram of the full program published on the Internet resembles the curve of Fig. 25 (it is only shorter).

Fig. 23. Normalization of Boolean conditions.

So using the explained coordination of the components we can automate the interactive session that we discussed in this section. We implemented this functionality in the `ApplyAll` button. We used the coordination language SEAL [32,33] for that.

7. 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 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 easy to maintain [56].

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

```
┌────────────────────────────────────────────────────────────┐
│ ◉  COBOL-plus : /home/x/cics/Example.02  ▒▒▒▒▒▒▒▒      回    │
├────────────────────────────────────────────────────────────┤
│ □ tree text expand help                                      │
├─────────────────┬──────────────────────────────────────────┤
│    CountGo       ▲ ║IDENTIFICATION DIVISION.                 │
├─────────────────┤ ║                                          │
│  PrettyPrint     │ ║PROGRAM-ID. XXXXXXX.                      │
├─────────────────┤ ║                                          │
│    AddEndIf      │ ║                                          │
├─────────────────┤ ║PROCEDURE DIVISION.                       │
│    RemDots       │ ║                                          │
├─────────────────┤ ║HAUPTVERARB SECTION.                      │
│    RemThen       │ ║                                          │
├─────────────────┤ ║HV-050.                                   │
│    FlowOpt       │ ║  EVALUATE SWPF                           │
├─────────────────┤ ║    WHEN 1 PERFORM HV-81                  │
│  ElimDeadCode    │ ║    WHEN 2 THRU 4 PERFORM HV-95           │
├─────────────────┤ ║  END-EVALUATE                            │
│   AddBarSec      │ ║  IF SWPF < 1 OR SWPF > 4                 │
├─────────────────┤ ║    PERFORM HV-81                         │
│   ElimGoDep      │ ║  END-IF.                                 │
├─────────────────┤ ║                                          │
│    ElimGo        │ ║BAR SECTION.                              │
├─────────────────┤ ║                                          │
│    MovePar       │ ║BAR-PARAGRAPH.                            │
├─────────────────┤ ║  STOP RUN.                               │
│  SwitchPars      │ ║                                          │
├─────────────────┤ ║HAUPTVERARB-SUBROUTINES SECTION           │
│   Distribute     │ ║                                          │
├─────────────────┤ ║HV-95.                                    │
│    Cluster       │ ║  IF SWFEHL <= ZERO                       │
├─────────────────┤ ║    PERFORM MODULE                        │
│    ElimCont      │ ║  END-IF.                                 │
├─────────────────┤ ║                                          │
│    NormCond      │ ║HV-81.                                    │
├─────────────────┤ ║  MOVE 2 TO CA-SWPF.                      │
│   RemLabels      │ ║                                          │
├─────────────────┤ ▼                                          │
│   RemComment     │                                            │
├─────────────────┤                                            │
│   ReplacePar     │                                            │
├─────────────────┤                                            │
│   RemDoubles     │                                            │
├─────────────────┤                                            │
│   UnfoldPar      │                                            │
├─────────────────┤                                            │
│   RemExitPar     │                                            │
├─────────────────┤                                            │
│   RemEmptySec    │                                            │
├─────────────────┤                                            │
│    ApplyAll      │                                            │
└─────────────────┴──────────────────────────────────────────┘
```
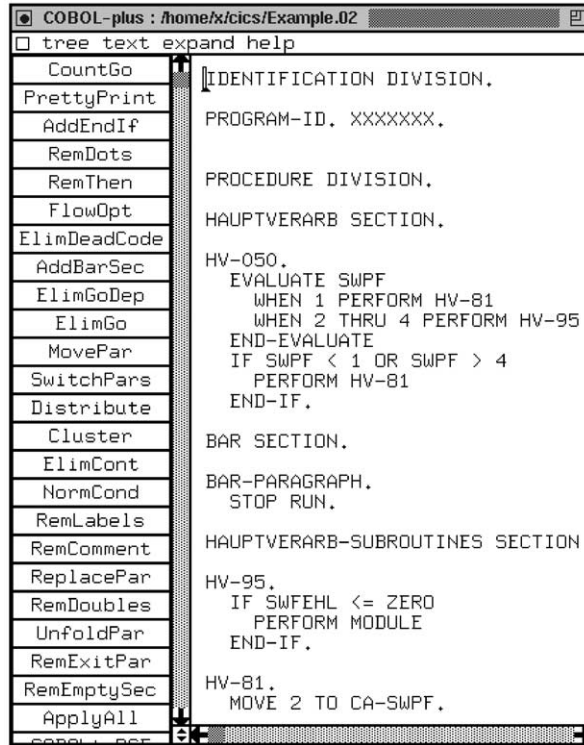
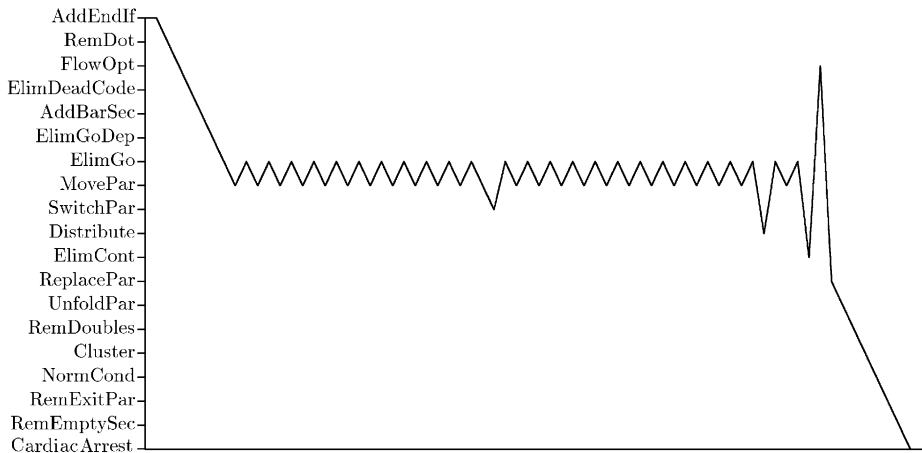Fig. 24. Elimination of EXIT paragraphs.

Fig. 25. Cardiogram of the program DBRIP08.COB.

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 [5] such strategies are considered being core technologies for system renovation. Note that in Sections 5 and 6 coordination and computation were 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, that is, 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 (see below). 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 next step could be to separate out the access logic, but that topic deserves a separate paper.

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 3.1. 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.

8. Creating stateless programs

CICS programs communicate with one another via a global data area referred to
as the *communication area*—in COBOL syntax the COMMAREA. Here data may be
deposited by one program and picked up by another one. This kind of global coupling
is dangerous and may produce undesired side-effects, particularly if pointers are being
used. To access the communication area CICS programs are passed a pointer as a
parameter. All references to data within that area are subsequently made by computing
the displacement from that start address. User programs usually include a COPY data
structure describing the entire communication area even though the program only uses
a few variables. Not only has the size of the program exploded, but the program is
given access to data which it should not see. This is a blatant violation of the principle
of information hiding and a frequent cause of undesirable side-effects [42]. The user
can easily overwrite data that does not belong to him or her. The error will only be
recognized later in some successor program.

Such addressing techniques must be eliminated if CICS programs are to be reused
in another environment. For this reason, all of the data referenced within a module
should be placed in an explicit parameter list, even if this list becomes long. It is
the lesser of two evils. Either one has to create a similar global data area in the new
environment or one creates parameter lists. The solution selected here is to generate a
list of parameters, one for each structure and elementary data type referred to by the
module. These parameters can then be set by any client program calling the module in
question.

By placing all of the data used by a module in its linkage section, the module
becomes stateless, that is, it has no own memory. In this way reentrancy is assured
and the module is detached from its data [56]. The data may reside somewhere in a
buffer within the wrapper. From there it should be passed by value, meaning that only
a copy of the values are provided to the module called and only those values which it
really processes. This greatly reduces the risk of side-effects and makes the repartioned
CICS program much more flexible. Later modules of this type can be readily converted
to methods as prescribed in the new Object COBOL standard [2,25]. In the code below
we give an impression of what this looks like. We show the long LINKAGE SECTION

on purpose so that the realities of paradigm shifts become apparent, and note that this
is the lesser of two evils.

```
****************************************************************
*    VERARBEITUNG
****************************************************************
*
METHOD-ID. "dbrip002".
DATA DIVISION.
LINKAGE SECTION.
01  RECHENFELDER.
   05  R-ADRSW  PIC 9(3)        VALUE ZERO.
   05  R-LEERZ  PIC 9(3)        VALUE ZERO.
   05  R-RNW    PIC S9(5)V99    VALUE ZERO.
   05  R-RATE   PIC S9(7)V99    VALUE ZERO    COMP-3.
   05  R-REST   PIC S9(5)V9999 VALUE ZERO.
01 ARBEITSDATEN.
   02  M8CANA           PIC X.
   02  M8XJUMPA         PIC X(8).
01 DFHBMBRY              PIC X(8).
01 DBRIM8DI.
   02  FILLER           PIC X(40).
   02  M8PRIDI          PIC X(8).
   02  M8PRIDL          PIC 99.
   02  M8PRIDF          PIC X.
   02  M8CANI           PIC X(8).
   02  M8CANL           PIC 99.
   02  M8CANF           PIC X.
   02  M8ABS1I          PIC X(8).
   02  M8ABS1L          PIC 99.
   02  M8ABS1F          PIC X.
   02  M8ABS2I          PIC X.
   02  M8ABS2L          PIC 99.
   02  M8ABS2F          PIC X.
   02  M8ABS3I          PIC X.
   02  M8ABS3L          PIC 99.
   02  M8ABS3F          PIC X.
   02  M8ABS4I          PIC X.
   02  M8ABS4L          PIC 99.
   02  M8ABS4F          PIC X.
   02  M8ADRZ1I         PIC X.
   02  M8ADRZ1L         PIC 99.
   02  M8ADRZ1F         PIC X.
   02  M8ADRZ2I         PIC X.
   02  M8ADRZ2L         PIC 99.
```

```
02  M8ADRZ2F        PIC X.
02  M8ADRZ3I        PIC X.
02  M8ADRZ3L        PIC 99.
02  M8ADRZ3F        PIC X.
02  M8ADRZ4I        PIC X.
02  M8ADRZ4L        PIC 99.
02  M8ADRZ4F        PIC X.
02  M8ADRZ5I        PIC X.
02  M8ADRZ5L        PIC 99.
02  M8ADRZ5F        PIC X.
02  M8ADSW1I        PIC X.
02  M8ADSW1L        PIC 99.
02  M8ADSW1F        PIC X.
02  M8ADSW2I        PIC X.
02  M8ADSW2L        PIC 99.
02  M8ADSW2F        PIC X.
02  M8ADSW3I        PIC X.
02  M8ADSW3L        PIC 99.
02  M8ADSW3F        PIC X.
02  M8ADSW4I        PIC X.
02  M8ADSW4L        PIC 99.
02  M8ADSW4F        PIC X.
02  M8SACHTI        PIC X.
02  M8SACHTL        PIC 99.
02  M8SACHTF        PIC X.
02  M8SACH1I        PIC X.
02  M8SACH1L        PIC 99.
02  M8SACH1F        PIC X.
02  M8SACH2I        PIC X.
02  M8SACH2L        PIC 99.
02  M8SACH2F        PIC X.
02  M8ANREDI        PIC X.
02  M8ANREDL        PIC 99.
02  M8ANREDF        PIC X.
02  M8ERDATI        PIC XXXXX.
02  M8ERDATL        PIC 99.
02  M8ERDATF        PIC X.
02  M8MENDZI        PIC XXXXX.
02  M8MENDZL        PIC 99.
02  M8MENDZF        PIC X.
02  M8MENDVI        PIC XXXXX.
02  M8MENDVL        PIC 99.
02  M8MENDVF        PIC X.
02  M8OBJI          PIC XXXX.
02  M8OBJL          PIC 99.
```

```
    02  M8OBJF           PIC X.
    02  M8UEBETI         PIC X.
    02  M8UEBETL         PIC 99.
    02  M8UEBETF         PIC X.
    02  M8KAUTII         PIC X.
    02  M8KAUTIL         PIC 99.
    02  M8KAUTIF         PIC X.
    02  M8ZGUTI          PIC X.
    02  M8ZGUTL          PIC 99.
    02  M8ZGUTF          PIC X.
    02  M8ANZRAI         PIC X.
    02  M8ANZRAL         PIC 99.
    02  M8ANZRAF         PIC X.
    02  M8RATEI          PIC X.
    02  M8RATEL          PIC 99.
    02  M8RATEF          PIC X.
    02  M8FZ1I           PIC X.
    02  M8FZ1L           PIC 99.
    02  M8FZ1F           PIC X.
    02  M8FZ2I           PIC X.
    02  M8FZ2L           PIC 99.
    02  M8FZ2F           PIC X.
    02  M8FZ3I           PIC X.
    02  M8FZ3L           PIC 99.
    02  M8FZ3F           PIC X.
    02  M8XJUMPI         PIC X.
    02  M8XJUMPL         PIC 99.
    02  M8FJUMPF         PIC X.
01  XM181-P.
    05 XM181-1.
    10 FEHL-NR     PIC 9999       VALUE 0.
    10 FEHL-SPR    PIC 9.
    05 XM181-2.
    10 XM181-PROG  PIC X(8).
    05 XM181-3.
    10 FEHL-MELD   PIC X(79).
    05 XM181-4.
    10 FEHL-VAR1   PIC X(20).
    05 XM181-5.
    10 FEHL-VAR2   PIC X(20).
    05 XM181-6.
    10 FEHL-VAR3   PIC X(20).
    05 XM181-7.
    10 FEHL-VAR4   PIC X(20).
    05 XM181-8.
```

```
   10 FEHL-VAR5    PIC X(20).
   05 XM181-9.
   10 FEHL-VAR6    PIC X(20).
01 P-XM314.
   05 P1-XM314.
   10 P1-XM314-EL    OCCURS 5.
   15 P1-XM314-ADRSW PIC X.
   15 P1-XM314-ADRZ  PIC X(30).
   05 P2-XM314.
   10 P2-XM314-1    PIC X.
   10 P2-XM314-2    PIC X.
   10 P2-XM314-3    PIC 9(4) COMP.
   10 P2-XM314-4    PIC 9(4) COMP.
   10 P2-XM314-5    PIC X(8).
01 P-XM200.
   05 P200-1        PIC X(8).
01 P-P008L          PIC S9(4) COMP.
01 P-P008.
   05 P008-1        PIC S9(8) COMP.
PROCEDURE DIVISION    USING
INOUT                     RECHENFELDER,
OUTPUT                    ARBEITSDATEN,
INPUT                     DFHBMBRY,
OUTPUT                    DBRIM8DI,
OUTPUT                    XM181-P,
OUTPUT                    P-XM314,
OUTPUT                    P-XM200,
INOUT                     P-P008L,
INOUT                     P-P008.
**************************************************************
```

9. Conclusion

Embedded 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 methods, 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. So we implemented tools that replace CICS commands by CALLs to a wrapper. In this way, 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 object request broker using OMA's Interface Definition Language to define the interfaces.

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

Then we can inspect all the paragraphs one by one. We can modify the coordination. Moreover, we can insert new subroutines, modify old ones, and delete subroutines without affecting the control-flow. This means that the programs have become modular, change-enabled, and in a sense componentized since only part of the functionality can be invoked at wish.

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. Thus, we create complex interfaces for the subprograms, which may be seen as more problematic than in-line solutions. The first two steps are certainly cost-beneficial, as argued in the introduction. But not implemented on a large scale by companies for the reasons mentioned in Section 2. Our last step is not necessarily improving maintainability, since the advantage of subprograms might be annihilated by the creation of the complex interfaces of which we showed a short one above. However, if we have to access COBOL/CICS legacy applications in a modular way from the outside we have little choice but to use the complex interfaces.

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] P. Allen, S. Frost, Component-Based Development for Enterprise Systems, Cambridge University Press, Cambridge, 1998.

[2] E. Arranga, D. Coyle, Object-Oriented COBOL, SIGS Books, 1996.

[3] C. Babcock, Restructuring eases maintenance, Computerworld, 1987, pp. 19–22.

[4] B. Boehm, Software Engineering Economics, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[5] M.G.J. van den Brand, P. Klint, C. Verhoef, Core technologies for system renovation, in: K.G. Jeffery, J. Král, M. Bartošek (Eds.), SOFSEM'96: Theory and Practice of Informatics, Lecture Notes in Computer Science, Vol. 1175, Springer, Berlin, 1996, pp. 235–255. Available at http://www.cs.vu.nl/˜x/sofsem/sofsem.html.

[6] M.G.J. van den Brand, P. Klint, C. Verhoef, Re-engineering needs generic programming language technology, ACM SIGPLAN Notices 32 (2) (1997) 54–61.

[7] M.G.J. van den Brand, P. Klint, C. Verhoef, Term rewriting for sale, in: C. Kirchner, H. Kirchner (Eds.), Second Internat. Workshop on Rewriting Logic and its Applications, Electronic Notes in Theoretical Computer Science, Springer, Berlin, 1998. Available at: http://www.cs.vu.nl/˜x/sale/sale.html.

[8] M.G.J. van den Brand, M.P.A. Sellink, C. Verhoef, Generation of components for software renovation factories from context-free grammars, in: I.D. Baxter, A. Quilici, C. Verhoef (Eds.), Proc. Fourth Working Conf. on Reverse Engineering, 1997, pp. 144–153. Available at http://www.cs.vu.nl/˜x/trans/trans.html.

[9] M.G.J. van den Brand, M.P.A. Sellink, C. Verhoef, Obtaining a COBOL grammar from legacy code for reengineering purposes, in: M.P.A. Sellink (Ed.), Proc. Second Internat. Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshops in Computing, Springer, Berlin, 1997. Available at http://www.cs.vu.nl/˜x/coboldef/coboldef.html.

[10] M.G.J. van den Brand, M.P.A. Sellink, C. Verhoef, Control flow normalization for COBOL/CICS legacy systems, in: P. Nesi, F. Lehner (Eds.), Proc. Second Euromicro Conf. on Maintenance and Reengineering, 1998, pp. 11–19. Available at http://www.cs.vu.nl/˜x/cfn/cfn.html.

[11] M.G.J. van den Brand, M.P.A. Sellink, C. Verhoef, Current parsing techniques in software renovation considered harmful, in: S. Tilley, G. Visaggio (Eds.), Proc. Sixth Internat. Workshop on Program Comprehension, 1998, pp. 108–117. Available at http://www.cs.vu.nl/˜x/ref/ref.html.

[12] M.G.J. van den Brand, M.P.A. Sellink, C. Verhoef, Generation of components for software renovation factories from context-free grammars, Sci. Comput. Program. 36(2–3) (2000) 209–266. (Available at http://www.cs.vu.nl/˜x/scp/scp.html. An extended abstract with the same title appeared earlier: [8].)

[13] M.G.J. van den Brand, E. Visser, Generation of formatters for context-free languages, ACM Trans. Software Eng. Methodol. 5 (1996) 1–41.

[14] A.W. Brown (Ed.), Component-Based Software Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1996.

[15] J.B. Caldwell, H.C. Muttart, D.H. Gross, Automatic compiler restructuring of COBOL programs into a proc per paragraph model, U.S. Patent No. 5,778,232, July 7, 1998.

[16] 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, VA, USA, 1987.

[17] B. Crownhart, IBM's Workstation CICS, McGraw-Hill, New York, 1992.

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

[19] L. Goldstein, An alternative to CICS HANDLE commands, CICS Update, July 1987.

[20] P. Gossam, Accessing legacy systems, Object Expert 5 (3) (1997) 58–60.

[21] T. Harmer, P. McParland, J. Boyle, Transformations to restructure and re-engineer COBOL programs, J. Aut. Software Eng. 5 (1998) 321–345.

[22] I.J. Hayes, Applying formal specification to software development in industry, IEEE Trans. Software Eng. SE-11 (2) (1985) 169–178.

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

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

[25] ISO/IEC/NCITS. Programming Language COBOL, 1.8 Edition, 2000. Available at http://www.ncits.org/tc_home/j4htm/cd18all.pdf.

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

[27] A.M. Jatich, CICS Command Level Programming, Wiley Professional Computing, New York, 1991.

[28] C. Jones, Assessment and Control of Software Risks, Prentice-Hall, Englewood Cliffs, NJ, 1994.

[29] C. Jones, Estimating Software Costs, McGraw-Hill, New York, 1998.

[30] C. Jones, The Year 2000 Software Problem—Quantifying the Costs and Assessing the Consequences, Addison-Wesley, Reading, MA, 1998.

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

[32] J.W.C. Koorn, Connecting semantic tools to a syntax-directed user-interface, in: H.A. Wijshoff (Ed.), Computing Science in the Netherlands (CSN93), SION, Utrecht, 1993, pp. 217–228.

[33] J.W.C. Koorn, Generating uniform user-interfaces for interactive programming environments, Ph.D. Thesis, University of Amsterdam, 1994.

[34] L. Kossen, W.P. Weijland, Correctness proofs for systolic algorithms: palindromes and sorting, in: J.C.M. Baeten (Ed.), Applications of Process Algebra, Cambridge Tracts in Theoretical Computer Science, Vol. 17, Cambridge University Press, Cambridge, 1990, pp. 89–125.

[35] W. Kozaczynski, N. Wilde, On the re-engineering of transaction systems, J. Software Maintenance 4 (3) (1992) 143–162.

[36] H.T. Kung, C.E. Leierson, Systolic arrays (for VLSI), in: I.S. Duff, G.W. Stewart (Eds.), Sparse Matrix Proceedings 1978, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1979, pp. 256–282.

[37] H.T. Kung, C.E. Leierson, Systolic array apparatuses for matrix computations, U.S. Patent No. 4,493,048, January 8, 1985.

[38] R. Lämmel, C. Verhoef, semi-automatic grammar recovery, Technical Report P2000, University of Amsterdam, Programming Research Group, 2000. Available at: http://www.cs.vu.nl/˜x/ge/ge.html.

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

[40] S. McConnell, Rapid Development, Microsoft Press, Redmond, WA, 1996.

[41] J.H. Moreno, T. Lang, Matrix Computations on Systolic-Type Arrays, Kluwer Academic Publishers, Dordrecht, 1992.

[42] D.L. Parnas, The criteria to be used in decomposing systems into modules, Commun. ACM 15 (12) (1972) 1053–1058.

[43] L.H. Putnam, W. Myers, Measures for Excellence—Reliable Software on Time, Within Budget, Yourdon Press Computing Series, Prentice-Hall, Upper Saddle River, NJ, 1992.

[44] J. Reutter, Maintenance is a management problem and a programmer's opportunity, in: A. Orden, M. Evens (Eds.), 1981 National Computer Conference, AFIPS Conference Proceedings, Vol. 50, AFIPS Press, Arlington, VA, 1981, pp. 343–347.

[45] M.P.A. Sellink, C. Verhoef, Reflections on the evolution of COBOL. Technical Report P9721, University of Amsterdam, 1997. Available at http://www.cs.vu.nl/˜x/lib/lib.html.

[46] M.P.A. Sellink, C. Verhoef, Development, assessment, and reengineering of language descriptions—extended abstract, in: B. Nuseibeh, D. Redmiles, A. Quilici (Eds.), Proc. 13th Internat. Aut. Software Eng. Conf., 1998, pp. 314–317. (For a full version see [50].) Available at: http://www.cs.vu.nl/˜x/ase98/ase98.html.

[47] M.P.A. Sellink, C. Verhoef, Native patterns, in: M. Blaha, A. Quilici, C. Verhoef (Eds.), Proc. Fifth Working Conference on Reverse Engineering, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 89–103. Available at http://www.cs.vu.nl/˜x/npl/npl.html.

[48] M.P.A. Sellink, C. Verhoef, An architecture for automated software maintenance, in: D. Smith, S.G. Woods (Ed.), Proc. Seventh Internat. Workshop on Program Comprehension, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 38–48. Available at http://www.cs.vu.nl/˜x/asm/asm.html.

[49] M.P.A. Sellink, C. Verhoef, Generation of software renovation factories from compilers, in: H. Yang, L. White (Ed.), Proc. Internat. Conf. on Software Maintenance, IEEE Computer Society Press, Los Alamitos, CA, 1999, pp. 245–255. Available via http://www.cs.vu.nl/˜x/com/com.html.

[50] M.P.A. Sellink, C. Verhoef, Development, assessment, and reengineering of language descriptions, in: J. Ebert, C. Verhoef (Eds.), Proc. Fourth European Conf. on Software Maintenance and Reengineering, IEEE Computer Society Press, Los Alamitos, CA, 2000, pp. 151–160. (Full version of [46]). Available at: http://www.cs.vu.nl/˜x/cale/cale.html.

[51] M.P.A. Sellink, C. Verhoef, Scaffolding for software renovation, in: J. Ebert, C. Verhoef (Eds.), Proc. Fourth European Conf. on Software Maintenance and Reengineering, IEEE Computer Society Press, Los Alamitos, CA, 2000, pp. 161–172. Available via http://www.cs.vu.nl/˜x/scaf/scaf.html.

[52] M.P.A. Sellink, H.M. Sneed, C. Verhoef, Systolic structuring algorithm in steps, 1998. Available at http://www.cs.vu.nl/˜x/systolic/systolic.html.

[53] H.M. Sneed, Economics of software reengineering, J. Software Maintenance 3 (3) (1991) 163–182.

[54] H.M. Sneed, Architecture and functions of a commercial software reengineering workbench, in: P. Nesi, F. Lehner (Eds.), Proc. Second Euromicro Conf. on Maintenance and Reengineering, IEEE Computer Society Press, Los Alamitos, CA, 1998, pp. 2–10.

[55] H.M. Sneed, Object-oriented Software Migration, Addison-Wesley, Reading, MA, 1998.

[56] H.M. Sneed, E. Nyary, Downsizing large application programs, J. Software Maintenance 6 (5) (1994) 235–247.

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

[58] J.M. Spivey, The Z Notation: A Reference Manual, Prentice-Hall, Englewood Cliffs, NJ, 1992.

[59] A.A. Terekhov, C. Verhoef, The realities of language conversions, IEEE Software 17 (6) (2000) 111–124. Available at http://www.cs.vu.nl/˜x/cnv/S6.pdf.

[60] A.C. Wills, D.F. D'Souza, Objects, Components, and Frameworks With UML—The Catalysis Approach, Object Technology Series, Addison-Wesley, Reading, MA, 1998.