An Architecture for Automated Software Maintenance

Alex Sellink and Chris Verhoef
University of Amsterdam, Programming Research Group
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
`alex@wins.uva.nl, x@wins.uva.nl`

Abstract

We developed an assembly line to implement certain specific changes in a stockbroking system written in COBOL with embedded SQL. The changes were proposed by the maintenance team of the system. Using our architecture, it took a few hours to implement the conditional transformations from the code examples we obtained from the maintenance team. Then we could carry out the tasks completely automated. We report on the transformations, their implementation and the architecture we used. It is the intention of the company that owns the COBOL/SQL to use our architecture for similar tasks. This study was carried out in order to give the company that owns the code an indication of the effort it takes, the development process of the components that carry out such tasks, and the process to change software using our architecture.

Categories and Subject Description: D.2.6 [**Software Engineering**]: Programming Environments—Interactive; D.2.7 [**Software Engineering**]: Distribution and Maintenance—Restructuring;
Additional Key Words and Phrases: Reengineering, System renovation, Software renovation factory, Automated software maintenance, Automated redesign, COBOL, embedded SQL.

1 Introduction

During software maintenance in large companies, it appears to be the case that many and diverse changes to entire systems have to be made on a daily basis. Those tasks may seem relatively simple, but not that simple that they can be automated using ad-hoc tools that have no knowledge of the grammar of the code that has to be changed. Due to this fact, this type of maintenance has to be done by hand. Consequently, such maintenance uses a lot of precious capacity. First, since the task has to be done by hand and, second, since it is easy to overlook a case or to make mistakes. Let us give a few examples of typical tasks. The addition of explicit scope terminators in COBOL applications is a task that improves the readability of code. We learned from reliable sources that someone at a Dutch bank worked a year on adding explicit scope terminators manually. To make a comparison, we added in a mortgage system approximately 27 `END-IF`s per minute using a component that took less than an hour to implement using our architecture. More important, the automated process does not miss cases, makes no typographical errors, has uniform layout, etc. Similar advantages have been reported on in [31]. Another example is the adaptation to new standards. It is our experience that large companies have evolving internal standards for coding. As a consequence, they have more than one standard simultaneously. Depending on the time the programmers are working for the company, a certain standard is used. So automated adaption to the latest standards is a useful automated maintenance process. It can also be the case that decision structures need an update, since during the evolution of the system more cases have been added. At the company that we looked at, the tenth bank of the world, such changes are made to their systems on a daily base. It is recognized by the teams working on these issues that these tasks are very time consuming, not very challenging, although its hard to do it in a correct way without tool support other than compile-link-edit cycles.

Maintainers have to look for the problematic code throughout the entire system. This is time consuming and error-prone. Once they located this code they have a good local comprehension of what to do. We stress that for many of these daily tasks it is not necessary to completely comprehend such systems. Local comprehension suffices in order to specify tools that search for the typical patterns they are looking for and once they are found, we automatically change them according to their wishes. The search and replace process is 100 % automated. It will be clear that tool support for carrying out such tasks saves money and makes the maintenance process less error prone [1]. Since many such tasks are very company specific, it is not realistic to assume the existence of tools that carry out these tasks. To give the reader an idea, a typical task that we carried out was to restructure code so that it invokes a company specific routine. We discuss this tool in more detail in Section 3. Obviously, such tools do not exist, hence the proposed architecture.

Using the technology that we developed it is easy to implement such specific tasks by maintenance teams although we are convinced that a short course to learn the tools is necessary (we come back on this issue in Section 5). In order to give an idea of the process that can be carried out by maintenance teams in the future, we carried out such tasks. We report on them in this paper. We obtained from the bank a typical system: a 100 KLOC system written in OS/VS COBOL, a COBOL 74 dialect, that contains embedded SQL. Some parts are 20 years old, and some parts are brand new. Moreover, we obtained some typical tasks that had to be carried out on this system. The maintenance team selected representative tasks that were too difficult to construct an ad-hoc disposable tool for to carry out the task. We implemented the tasks using an architecture, that we

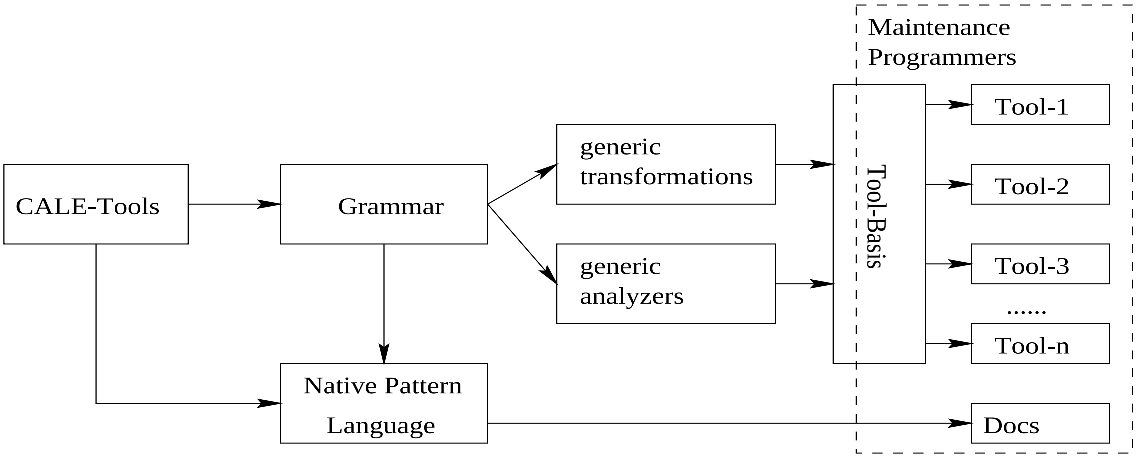

Figure 1: Architecture of the software maintenance factory

call a software renovation factory in a few hours and we were able to carry out the tasks completely automated on the system. The specific tools we implemented are together called an assembly line.

The advantages of our approach are that the changes are made in a controlled way: the tools that we made are in fact the requirement specification of the tasks. This means that it is now possible to follow the history of changes in detail, that the process can be reproduced, moreover the process can be applied to other systems as well. In this way it is possible to set up such maintenance in an automated, controlled, and documented manner. Moreover, the evolution of the system and the many changes are now not only known by certain maintenance teams but they are documented in an executable way that is accessible for other parties, that can reuse the tools as well. This approach has advantages over an approach where the change history is only in the mind of the programmer.

Organization of the paper In Section 2 we start with an overview of the architecture of the software renovation factory. We also discuss the implementation technology. In Section 3 we elaborately treat the assembly line that we implemented. In Section 4 we discuss an input and output program in order to show the effect of the entire assembly line. In Section 5 we address peopleware issues that are concerned with our approach. In Section 6 we conclude.

Applications of our approach The technology discussed in this paper has been applied in various other projects. We mention control flow normalization of COBOL/CICS legacy systems [10] and a difficult leap year correction transformation [44]. In [40] a nontrivial restructuring problem for a Swiss bank is solved using this architecture. In [13] a COBOL 85 to COBOL 74 transformation is carried out using this technology (for the same bank as the case study of the current paper).

Related work In [31] also automated tools are implemented to carry out software maintenance tasks. In [46] an architecture for a reengineering workbench is discussed. In that paper the focus is on what functionality is desired, and in our paper the focus is on how to easily create such functionality. In [2] an organizational view on (component) factories is given that can be used as complement to the technical view presented by us. Related implementation platforms for software renovation factories are for instance TXL [14], REFINE [37] and COSMOS [19]. For a comparison of the parsing technology used by these approaches in comparison with our approach we refer to [11]. For a comparison of our implementation platform (the ASF+SDF Meta-Environment) and REFINE we refer to [16]. For an overview of related systems, their history, and comparisons of REFINE and the ASF+SDF Meta-Environment by users of both systems we refer to [7, Section 3.3]. This paper is part of a larger effort. We refer to [5, 27, 6, 18] for an overview of how this paper relates with other papers by the authors.

2 The Architecture

In this section we give an idea of the architecture that we call a software renovation factory. In fact this is a development environment that is suited to implement maintenance and reengineering tasks to be carried out in an automated fashion. It consists roughly of two parts. There is one part that has to be set up by developers of such factories and there is the part that can be used by maintenance teams so that they can develop assembly lines for automated maintenance. In Figure 1, we depicted the architecture of the factory. We discuss Figure 1 from left to right. CALE stands for computer aided language engineering, and CALE-Tools help in constructing a grammar or even generating one from some electronic source, like a standard, or an on-line language manual. CALE-Tools can assess the quality of existing language descriptions, with CALE-Tools we can develop language descriptions, and we can reengineer them. In fact, CALE-Tools form a factory of their own. For a more thorough discussion on CALE-Tools we refer to [42, 45]. Depending on the existence of such documentation we benefit from CALE-Tools in the construction of grammars. In some cases we can generate a grammar from an electronically available language description manual. As soon as we have a grammar, we generate from this grammar its native pattern language. For our example system written in

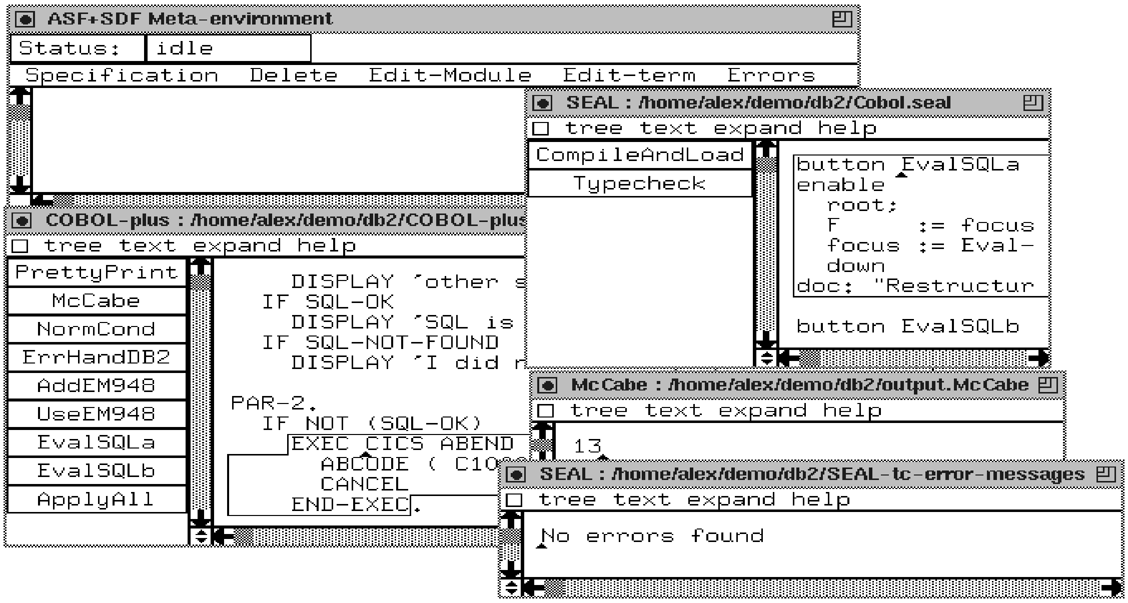

Figure 2: The ASF+SDF Meta-Environment in action

OS/VS COBOL and embedded SQL this means that a real program is also a pattern: the pattern that matches only this program. In the generated pattern language we have for all sort names that are present in the grammar three kinds of variables available. For instance, `Paragraph1` matches exactly one arbitrary paragraph in OS/VS COBOL possibly containing embedded SQL. `Paragraph1+` matches one or more such paragraphs, and `Paragraph1*` matches zero or more such paragraphs. The * stands for zero or more occurrences. The generation process from a grammar to a native pattern language is implemented as a CALE-Tool, hence the connecting arrow. For more information on the generation process, a formal definition of native pattern languages, and an example of their use we refer to [44]. In this paper we will use native patterns as well to implement the requested changes of the maintenance programmers. We believe that native patterns are very helpful when the technology that we developed will be ported to companies that use it for maintenance and/or reengineering. Since those programmers know the language quite well, the learning curve for the native pattern language is minimal. In order to have access to the pattern language it is possible to generate the documentation from the executable specification of this pattern language. This is depicted in the arrow to Docs. For OS/VS COBOL with SQL this amounts to a 25 page document, with a table of contents, etc. We use existing technology to do this [12]. From the grammar we generate generic transformations and generic analyzers. Those parts form the basis for which the user, in this case a maintenance programmer, can plug in custom-made tools. In fact, as soon as the user described the requirements specification of a tool, it is immediately executable since the generic components take care of bookkeeping issues. This architecture enables easy, reusable, and robust development of tools. In particular, it ensures maximal reuse of tools in case of different dialects. This technology has been discussed in [8].

The dashed rectangle represents the part that the maintenance team is working with. On top of the so-called Tool-Basis, the tools can be implemented with the aid of documentation of the native pattern language. The Tool-Basis contains the infrastructure that can be shared in custom tools, and takes care of the fact that the grammar, the generic transformations, and the generic analyzers are known in the custom made tools. In this paper we treat a case study that delivers a number of typical tools.

Implementation of the Architecture We used the ASF+SDF Meta-Environment [26] to implement the complete architecture. SDF stands for Syntax Definition Formalism [21], it can be used to define the syntax of a language. ASF stands for Algebraic Specification Formalism [3], it can be used to describe the semantics of a language. The combination is thus adequate for defining syntax and semantics of languages and the ASF+SDF Meta-Environment is the supporting environment for both formalisms. We note that a conference is devoted to this system (and similar systems). See the proceedings of ASF+SDF97 [39] for a snap shot of the state of the art. The ASF+SDF Meta-Environment is also used as a language prototyping environment, see [17] for an elaborate treatise. For an overview of some industrial applications of the ASF+SDF Meta-Environment we refer to [4, 6].

The ASF+SDF Meta-Environment is an interactive programming environment generator that takes a language definition as input (including a definition of the syntax of this language and optionally other operations on programs in the language such as, for instance, interpretation, compilation or transformation) and generates corresponding tools as output. From the syntax definition of a language in SDF various components are generated: a lexical scanner [23], a Generalized LR parser [22], a syntax-directed editor [28], a pretty printer [12], and optionally traversal functions and program analysis functions [8]. For the operations defined on programs, efficient term-rewrite engines are generated. We mention that Generalized LR parsing is particularly helpful for reengineering and maintenance, see [11] for details.

We can combine tools to obtain larger tools with a more complex behaviour. Using this component based engineering [27] many tools can easily be reused. We glue these components together with a coordination language called SEAL [29]. SEAL stands for Semantics-directed Environment Adaptation Language; it not only takes care of the coordination but also of a graphical user interface [30]. It is possible to change the coordination run-time, and to add functionality run-time, which is helpful in rapid development of custom tools.

In Figure 2 we give an idea of the look-and-feel of the ASF+SDF Meta-Environment in action. This screen dump shows the implementation of the maintenance tasks that we obtained from the maintenance team. The upper window is the ASF+SDF Meta-Environment. You can add and delete specifications. Specifications can be syntax descriptions of languages, or tools. They are called modules, and they can be edited via the edit-module window. We can also open editors that understand the syntax of a certain module. For instance the `COBOL-plus` window is an editor that can parse OS/VS COBOL plus CICS plus SQL. The buttons at the left-side of the `COBOL-plus` window are implemented using SEAL. Part of the coordination script can be seen in the upper-right window: it contains the functionality of some `EvalSQLa` button. This editor understands the SEAL language. We pressed the `Typecheck` button, that checks for type errors in the SEAL script. The `SEAL` window below pops up and tells us whether there are errors. As an example, we also pressed the `McCabe` button of the `COBOL-plus` window. This results in the small window with the `13` in it. We will discuss the functionality of the maintenance tasks in the next section.

3 A typical software maintenance assembly line

We carried out the case study on part of a stockbroking system in order to show the usefulness of the architecture that we propose. The development of the case study can be characterized as evolutionary. The earliest parts are from before 1980. Today, 1998, still new functionality is added to it. The maintenance tasks we discuss are all in the vein of this evolutionary process. They are concerned with restructuring of the error handling process after an SQL statement has approached a DB2-table. The values of the exit status of an SQL statement are modified in the new release of the embedded SQL that is used in this system. In the current situation, some of the return codes are hard-wired coded into the stockbroking system. This information will now be stored in a separate program that should be called. So a transformation that we carried out implements the change from the coded constants to the call. Overtime, the structure of the error handling procedures themselves, has extended. At this moment the decision structure is not optimal anymore. The other transformation that we carried out turns those decision structures into structures that are now more natural.

The remainder of this section is devoted to a more thorough description of the assembly line in order to illustrate the process of automated software maintenance. As we see it, any automated reengineering or maintenance process consists of a number of very small steps that—together—perform a complex task. Identifying those steps requires a factory approach towards the tasks. What exactly this factory approach is, is hard to say in general. One of the fundamental issues is, in our opinion, that the process consists of three major phases. We call them the preprocessing phase, the main phase and the postprocessing phase. It is our experience that whatever problem we had to solve, it always boiled down to these three phases. The phases themselves are combinations of small steps. It is also our experience that the small steps can be reused over and over again. For instance, a postprocessing step in a control flow normalization assembly line [10], has now been used as a preprocessing step in this maintenance process. The common situation is, however, that pre- and postprocessing steps can be reused for the reason they were constructed for in the beginning.

During development of the assembly line we use the ASF+SDF Meta-Environment in an interactive way. In Figure 2 this is expressed. Once the individual steps have been identified and implemented, we use the SEAL coordination architecture to combine all the small steps into one large usually complex task. A glimpse of this can also be seen in Figure 2. In order to modify the complete system, the implemented task can be carried out batch oriented [13]. Next, we discuss the steps that are present in this particular assembly line.

Formatting code A basic step in maintenance and reengineering is (re)formatting the code. This is also emphasized in [46]. We use generic technology to create a formatter, this technology already exists [12]. We generate from the grammar description a specification that we can adapt to company specific formatting standards. We use this formatter as a preprocessing step, in order to format the code properly. The other steps make use of the formatter in the sense that when the transformation is done the output will be showed in a text editor using this formatter. Constructing the formatter is not a maintenance programmers task. It is part of the development of a renovation or maintenance factory. We reused this formatter from other projects, with an hour effort to adapt it to this project.

McCabe Since some of the restructuring should lead to more simple decision structures, we implemented McCabe's cyclomatic complexity index [33] for OS/VS COBOL plus SQL (We learned from Tom McCabe [34] that his company is working on an implementation for COBOL with embedded SQL). We use the measure to show that the restructuring that we carry out decreases the complexity of the code. We think that this measure should be part of a basic set of tools that are readily available to the maintenance group. So we do not see this as a task for maintenance programmers, although implementation of this measure is not at all hard, when using this architecture. It is not part in the strict sense that we use this tool to perform the tasks. We use it to test whether the transformations reduce complexity. So we measure, transform, and measure the resulting code.

Normalize conditions The error handling is treated in IF phrases. They all contain variations of conditions for which transformations should be made. We could make a huge program that takes care of all the possibilities that we may find in the current system and that might occur in other systems that will need this type of maintenance as well. We do not do this. We start with restructuring the code so that we are sure that all the conditions have a certain form. Therefore, we normalize Boolean conditions so that we can be sure of their form. This reduces the complexity of the assembly line drastically. Let us discuss the tasks that we carry out during this preprocessing step. We have to recognize, for instance, the Boolean condition `SQLCODE = -818 OR -904 OR -911 OR -922`. Let us first explain this code, since it may be the case that this code is not familiar for the reader. We recall that this is OS/VS COBOL code with embedded SQL. The product of IBM that processes embedded SQL is called DB2, which stands for DataBase 2 [24]. When DB2 processes an SQL statement, DB2 sets a return code in the `SQLCODE` (and `SQLSTATE`) host variable of the so-called SQL communication area (SQLCA). The return code indicates whether the statement executed succeeded or failed. If `SQLCODE` equals zero, execution was successful, if its larger than zero, execution was successful with a warning, and if it is less than zero, execution was not successful. For instance, `SQLCODE = 100` means that no data was found. The error code `-818` indicates that the database request module and the application program were not the result of the same precompile. This module contains SQL statements extracted from the source program during program preparation. Code `-904` is caused when the resource required for the SQL statement was not available. Code `-911` is returned when a unit of work was the victim of a deadlock or timeout, so it had to be rolled back. The application is rolled back to the previous `COMMIT`, which ends a unit of recovery and commits the relational database changes that were made in that unit of recovery. Return code `-922` means that there was an authorization failure. In all the above cases the SQL statement could not be executed, hence the use of the `OR` connector. The abovementioned `IF` phrase takes care of the error handling.

There are many ways to implement the above Boolean condition, so we normalize all of them to a particular format. Let us give an example, just to give the reader an idea. The tool converts `NOT ( NOT ( SQLCODE = -818 OR (-904 OR -911 ) OR -922 ) )` to `SQLCODE = -818 OR -904 OR -911 OR -922`. We display the two equations that take care of the above normalizations. There are more equations, but they are concerned with other normalizations like `NOT X >= 1` turning into `X < 1`, which are not relevant for the SQL issues that we discuss at the moment.

```
[5] Norm-cond_L-exp(NOT(NOT(L-exp1))) = L-exp1
[9] Norm-cond_L-exp((Id1)) = Id1
```

We called this tool `Norm-cond` and on logical expressions, abbreviated `L-exp` it is called `Norm-cond_L-exp`. We mention that the underscore notation is generated from the grammar as well. Details on the generation process can be found in [8]. Equation `[5]` removes double negations, `L-exp1` is a variable matching exactly one arbitrary logical expression. Equation `[9]` removes superfluous parentheses around identifiers that are logical expressions. We note that in COBOL we can use predicates that are logical expressions that can be given a name. The value of `SQLCODE` is such a predicate. It is not necessary to have a fixed ordering of the return codes. We take care of that in the next analysis function.

Error handling analysis Of course, we do not want to have programs changed that do not contain the error handling patterns. We implemented an analysis tool to check this. In Figure 2 this tool is called `ErrHandDB2`. The idea of this tool is that it returns the number of occurrences of `SQLCODE = -818 OR -904 OR -911 OR -922` in all its variations but then only in `IF` phrases, since when it occurs in another context a transformation is undesirable. This tool depends on the fact that Boolean conditions are normalized first. So it does not take optional brackets and `NOT`s into account. What we do treat in this tool is the degree of freedom that may occur in the ordering of the Boolean condition. Still, the tool is simple. We have two equations for the tool `ErrHandDB2`: one for the `IF` and one for the `IF ELSE` case. We use an auxiliary tool, check return code (`crc`), to check for the correct values of return codes. Let us display the `IF` case of `ErrHandDB2` and two of the 5 equations of the auxiliary tool. The other equations of both tools are similar.

```
[1] crc(Lit1 Lit2 Lit3 Lit4) = true
    ==============================
    ErrHandDB2_Sentence(+,0,
    Statement1*
    IF SQLCODE = Lit1 OR Lit2 OR Lit3 OR Lit4
      Sentence1
    ) = 1
[2] crc(Lit1 Lit2 Lit3 Lit4) =
       contains-818(Lit1 Lit2 Lit3 Lit4)
     & contains-904(Lit1 Lit2 Lit3 Lit4)
     & contains-911(Lit1 Lit2 Lit3 Lit4)
     & contains-922(Lit1 Lit2 Lit3 Lit4)
[3] contains-818(Lit1* -818 Lit2*) = true
```

Equation `[1]` is a conditional one. Above the line, the condition is stated, and if it succeeds the equation below the line is executed. `ErrHandDB2` is an analysis tool. In fact, this means that the output sort is fixed. In this case the output is an integer. We have three arguments to an analysis tool. The first one is the operation that should be used, the second is the default value, and the third is the pattern. Since we wish to count the number of occurrences we use `+` as operator. If we do not find the pattern we wish to return `0` as a default value. If we find the pattern the tool will return `1`. This means that for an entire program every occurrence in any context will be counted and added to the default value. Now we discuss the pattern. It looks for a COBOL sentence that starts with zero or more statements (expressed with the variable `Statement1*`), then an arbitrary `IF` phrase containing four arbitrary literals `Lit1`, `Lit2`, `Lit3`, and `Lit4` in the Boolean condition. So `SQLCODE = -305 OR -502 OR -803 OR -811` matches. In order to prevent that incorrect `SQLCODE` return codes are counted, we have a condition on this equation, which is above the double line. We check whether the return codes are the four we are looking for. The fact that the variables Lit1 – Lit4 are the same above and below the double line means that they are bound to the same value if the rule matches. So if in the code -305 is matched by the first variable Lit1, the crc tool will use -305. The crc tool only returns true if all four literals are correct. This is expressed in equation [2] where we use the & as Boolean connector. In equation [3] we check whether one of the four equals -818. The variable Lit1* matches zero or more literals, then the literal -818 and then again zero or more arbitrary literals matched by Lit2*. The other return codes are checked analogously. It is simple to see that crc returns true only if the four literals are the return values that we are looking for and that there ordering does not matter. We use the ErrHandDB2 and crc analysis tools as conditions for the actual transformations. We discuss them next.

Modify data At this point we enter the main operation phase. It consist of two parts. In the DATA DIVISION we have to add some issues and we have to change the actual coding in the PROCEDURE DIVISION. Since both parts are not depending on each other as it comes to our transformations, we can split them up into two steps. If that was not the case we could have implemented a global pattern. In some cases this is necessary, such as in complex jump instruction eliminations, see [10] for examples of global patterns. The tool that we discuss in this paragraph modifies the DATA DIVISION if a check in the PROCEDURE DIVISION succeeds. Its functionality is that it adds to the WORKING-STORAGE SECTION a special variable and a COPY member, plus some comments. If there was not yet a DATA DIVISION, it will be created, if the WORKING-STORAGE SECTION did not exist, it will create it. Of course the transformation is only performed when there are more than zero occurrences of the error handling code using the SQLCODE, so that the transformation makes sense at all. We explain the five equations that implement this tool.

```
[1] ErrHandDB2(Program1) = 0
    =====================================
    Add-EM948_Program(Program1) = Program1
[2] ErrHandDB2(Program1) != 0,
    Program1 = COMMENT1* Ident-div1 Env-div1
                         Data-div1 Proc-div1
    ========================================
    Add-EM948_Program(Program1) =
    COMMENT1* Ident-div1 Env-div1
    Add-EM948_Data-div(Data-div1) Proc-div1
[3] Add-EM948_Data-div( ) =
    DATA DIVISION. Add-EM948_Ws-sec( )
[4] Add-EM948_Ws-sec(
    WORKING-STORAGE SECTION. Data-desc1*) =
    WORKING-STORAGE SECTION. Data-desc1*
      01 L-EM948  PIC X(09) VALUE 'EM948   L'.
  * FOR TESTING EM948 SQL-CODES
      COPY A0046075.
[5] Add-EM948_Ws-sec( ) =
    WORKING-STORAGE SECTION.
      01 L-EM948  PIC X(09) VALUE 'EM948   L'.
  * FOR TESTING EM948 SQL-CODES
      COPY A0046075.
```

Equation [1] is a conditional one. Above the line, the condition is stated, and if it succeeds the transformation below the line will be performed. The condition checks whether there are zero occurrences of the error handling code in a program. The expression Program1 is a variable from the native pattern language that represents a complete OS/VS COBOL program that may contain embedded SQL. So it matches any program of the system that we obtained from the bank. If there are zero occurrences, the function Add-EM948 applied to a program, denoted Add-EM948_Program, returns the original program unchanged. Equation [2] is treating the case that there are occurrences of the error handling somewhere in the program. The first equation in the conditions contains != standing for not equal to. So it means that the variable Program1 does contain occurrences. The second condition unfolds the Program1 into optional comments COMMENT1* and four divisions. Those divisions can be empty. If that is the case, they match the empty word. We denote this by a space. Add-EM948_Program states that its only relevant to act on the DATA DIVISION: the completely unfolded program is reiterated. Only at the variable matching the DATA DIVISION, called Data-div1, a modification should be made. What that is, is explained in the next two equations. Equation [3] states that if there is a DATA DIVISION it is only necessary to act on the WORKING-STORAGE SECTION. If there is *no* DATA DIVISION equation [3] creates it: Add-EM948_Data-div( ) returns the word DATA DIVISION including a period followed by Add-EM948 working on the sort Ws-sec. Then we have the same situation: either the WORKING-STORAGE SECTION exists or not. Equation [4] treats the case that it exists. It deals with Add-EM948 applied to the WORKING-STORAGE SECTION. This is denoted as Add-EM948_Ws-sec. The input pattern matches the terminal WORKING-STORAGE SECTION including the separator period. The variable Data-desc1* matches zero or more Data descriptors. The replacement pattern is the right-hand side of the equation. It reiterates the WORKING-STORAGE SECTION and the possibly occurring Data descriptors. Then the special variable is added, then some comment, and the COPY member. Equation [5] is similar to [4] except that there was no WORKING-STORAGE SECTION so it is also created. The rest is just a copy from the other equation. Now we have the right information put in the WORKING-STORAGE SECTION, we can modify the PROCEDURE DIVISION in the next tool.

Modify error handling We modify the error handling code in the PROCEDURE DIVISION. We do this with a tool called Use-EM948. It consists of two conditional equations, one dealing with an IF phrase and one dealing with an IF-ELSE phrase. We treat them both.

```
[1] crc(Lit1 Lit2 Lit3 Lit4) = true
    ==============================
    Use-EM948_Sentence(
      Statement1*
      IF SQLCODE = Lit1 OR Lit2 OR Lit3 OR Lit4
        Sentence1) =
      Statement1*
      MOVE SQLCODE TO SQL-CODE IN LINKAREA-EM948
      CALL 'UT100' USING L-EM948
                         LINKAREA-EM948
      IF RETURNCODE = '9'
        Sentence1
[2] crc(Lit1 Lit2 Lit3 Lit4) = true
    ==============================
```

```
Use-EM948_Sentence(
  Statement1*
  IF SQLCODE = Lit1 OR Lit2 OR Lit3 OR Lit4
    Cond-body1
  ELSE
    Sentence1) =
  Statement1*
  MOVE SQLCODE TO SQL-CODE IN LINKAREA-EM948
  CALL 'UT100' USING L-EM948
                     LINKAREA-EM948
  IF RETURNCODE = '9'
    Cond-body1
  ELSE
    Sentence1
```

Both conditions reuse the auxiliary `crc` tool that checks whether we are dealing with the correct return codes. If those values are correct, the condition is satisfied and we can make the change. The changes that we have to make in the `PROCEDURE DIVISION` are very local: the largest structures that we modify are COBOL sentences. Therefore, our tool has only two equations on this level (of course all the other equations on other levels are generated, as explained in [8]). The tool `Use-EM948` applied to a COBOL sentence is denoted `Use-EM948_Sentence`. The first sentence that we are interested in consists of zero or more arbitrary statements, matched by `Statement1*`, then the special phrase starting with `IF SQLCODE = Lit1 OR Lit2 OR Lit3 OR Lit4` followed by exactly one `Sentence1`. Since OS/VS COBOL is a COBOL 74 dialect, the scope of the `IF` has to be ended with a separator period (there is no `END-IF`). We recall that a sentence is one or more statements ended with a dot. We explain the replacement pattern. We leave the context intact: the `Statement1*` and `Sentence1` are reiterated. Before we change the original `IF` phrase, we insert some new code above it. First, the exit status variable is copied to an auxiliary data item `SQL-CODE` that is a subrecord of `LINKAREA-EM948`. Then a company specific `CALL` follows, which checks the exit status and modifies the data item `RETURNCODE` accordingly. Now we change condition of the original `IF` phrase: it is changed into `RETURNCODE = '9'`. The second equation is a variation of the first: the Boolean condition can also be contained in an `IF ELSE` phrase. So the only difference is the `Cond-body1` that represents anything that can occur between an `IF` and an `ELSE`. The rest of this case is the same as the first equation.

Redesigning SQL control structures Control structures that were adequate at the start of development, are extended or modified overtime and in the end the structure is not optimal anymore. With the aid of design knowledge about the system it is possible to redesign such control structures. From the maintenance team of the system we obtained a few code examples and special requirements on the desired form of the new control structures. We discuss four transformations that take care of the required changes. We cannot simply make one tool that carries out the transformations, since two changes have overlapping patterns. So, we have two tools to perform this task in the right order. We call them `Eval-SQL-a` and `Eval-SQL-b` respectively, since we evaluate the SQL control structure and change it when its not optimal. We start with the first one.

```
[1] Eval-SQL-a_Sentence*(
    Sentence1*
    Statement1*
    IF NOT SQL-OK
      Sentence1
    IF SQL-OK
      Sentence2
    Sentence2* ) =
    Sentence1*
    Statement1*
    IF SQL-OK
      rsp(Sentence2)
    ELSE
      Sentence1
    Sentence2*
```

Here we turn two `IF` phrases into one. Two `IF` phrases cannot occur in one sentence, since the separator period is also an implicit scope terminator. So, the `Eval-SQL-a` works on zero or more sentences, which is denoted by `Eval-SQL-a_Sentence*`. The pattern consists of a list of arbitrary sentences containing the two `IF` phrases that we are looking for. This is expressed with the (context) variables `Sentence1*` and `Sentence2*`. In between we have the two sentences containing the special `IF` phrases. The first one starts with zero or more arbitrary statements (`Statement1*`), then the conditional `IF NOT SQL-OK Sentence1` (the latter variable expresses that the body of the `IF` does not matter). Then another arbitrary conditional statement with fixed part `IF SQL-OK`. From the code examples we understood that the maintenance team wants this in an `IF ELSE` construct. The replacement pattern is conceptually simple. There is, however, a small complication with the scope termination of the `IF`. In the replacement pattern we use an auxiliary function `rsp`. This stands for remove separator period. For, the variable `Sentence2` represents code that is ended with a period. But since this period is also is also an implicit scope terminator, we have to remove it from `Sentence2`. Of course the separator period of `Sentence1` ends the scope of the new `IF ELSE`.

Next, we discuss the second transformation. Here, three consecutive `IF` phrases are turned into one nested `IF` phrase. In this transformation we make use of design knowledge: `SQL-OK` and `SQL-NOT-FOUND` cannot be true at the same time. This information can also be found in the file that defines the SQL auxiliary data items that are used in the code examples that we obtained from the maintenance team.

```
[2] Eval-SQL-a_Sentence*(
    Sentence1*
    Statement1*
    IF NOT ( SQL-OK OR SQL-NOT-FOUND )
      Sentence1
    IF SQL-OK
      Sentence2
    IF SQL-NOT-FOUND
      Sentence3
    Sentence2* ) =
    Sentence1*
    Statement1*
    IF SQL-OK
      rsp(Sentence2)
    ELSE
      IF SQL-NOT-FOUND
        rsp(Sentence3)
      ELSE
```

```
      Sentence1
   Sentence2*
```

This second equation also works on lists of sentences, for the same reason as the first one. Again we have the context variables `Sentence1*` and `Sentence2*`. The sentences in between start again with zero or more arbitrary statements (matched by `Statement1*`). Then we have three consecutive phrases containing the control structure that has to be changed. In the replacement pattern we use the `rsp` tool that removes separator periods in two cases for the same reason as in the previous equation. So its another pattern, but the approach to deal with it is completely analogous to the first one.

We discuss the third equation. Although this one is very simple, it interferes with the other patterns if not applied in the right order. In order to demonstrate this to the maintenance team of the system we used two buttons in the ASF+SDF Meta-Environment. We give it another name as mentioned above: `Eval-SQL-b`. Apart from updating the control structures there was a requirement to start with positive conditions in all `IF` phrases. As we can see in the above cases, all replacement patterns satisfy this criterion already. So this next tool takes care of the remaining cases. This can be obtained by the following simple transformation:

```
[3] Eval-SQL-b_Sentence(
    Statement1*
    IF NOT SQL-OK
      Sentence1 ) =
    Statement1*
    IF SQL-OK
      NEXT SENTENCE
    ELSE
      Sentence1
```

This transformation swaps the negative condition of the `IF` phrase. So `Eval-SQL-b` works on single COBOL sentences, which is denoted by `Eval-SQL-b_Sentence`. We see that the input pattern is exactly the same as part of the input pattern of the first transformation. Therefore, we apply it in this ordering, since otherwise the pattern above would be broken. The `NEXT SENTENCE` is an empty statement in COBOL comparable to the empty statements `CONTINUE` or `EXIT`.

We discuss the next equation, which is a variation of the third:

```
[4] Eval-SQL-b_Sentence(
    Statement1*
    IF NOT SQL-OK
      Cond-body1
    ELSE
      Sentence1 ) =
    Statement1*
    IF SQL-OK
      rsp(Sentence1)
    ELSE
      asp(Cond-body1)
```

Again it works on a COBOL sentence. The special condition `NOT SQL-OK` is now part of an `IF ELSE` phrase. Since we swap both branches, we have to remove the separator period from `Sentence1` in the `ELSE` branch, using `rsp`, and we have to add a separator period to the body of the `IF` branch. We have a variable `Cond-body1` that matches anything that can be part of an `IF` branch. We have a special sort for that due to the difficult scope termination rules of COBOL, see [9] for more details on these issues. We use an auxiliary tool `asp`, add separation period, in order to end the scope of the swapped `IF ELSE`. We note that sloppiness towards separator periods in COBOL 74 dialects can be disastrous, see [41] for details on a Y2K tool that is erroneous due to sloppiness with separator periods.

4 Transforming an example program

Now that we have constructed the entire assembly line that carries out the various tasks that were communicated to us, we treat an example program and its output so that the reader gets an idea of the effect of the many small changes together. We modified a program of the system for this purpose. In order to fit the example in this paper, we removed parts of it that are not relevant for explanatory purposes, and we made certain issues anonymous. Here is the modified input program:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. XXXXX.
AUTHOR.     N.N.
DATE-WRITTEN. APRIL 22, 1998.
PROCEDURE DIVISION.
1010-DETERMINE-MAX-SERIALNR.
  MOVE SQLCODE TO W-SQL-CODE.
  IF SQLCODE = -818 OR (-904) OR -922 OR -911
    PERFORM 9999-EM904-FILL
  ELSE
    IF IND-MAX-SERIALNR = -1
      MOVE 1 TO HH-MAX-SERIALNR-MESSAGE
    ELSE
      IF SQL-OK
        ADD 1 TO HH-MAX-SERIALNR-MESSAGE
      ELSE
        MOVE 'XXXXXXXXMESSAGE' TO EM900-TABLENAME.
1015-ADD-TCTMESSAGE-CT590.
  EXEC SQL
    INSERT INTO XXXXXXXXMESSAGE
    VALUE (:DCLXXXXXXXXMESSAGE.TRANSCODE-ORIGIN
           :DCLXXXXXXXXMESSAGE.EVENTTYPE
           :DCLXXXXXXXXMESSAGE.SERIALNR-MESSAGE
           :DCLXXXXXXXXMESSAGE.BUF-MESSAGE)
  END-EXEC.
  MOVE SQLCODE TO W-SQL-CODE.
  IF NOT (NOT (SQLCODE = -818 OR (-904 OR -911) OR -922))
    PERFORM 9999-EM904-FILL
  ELSE
    IF NOT SQL-OK
      MOVE 'XXXXXXXXMESSAGE' TO EM900-TABLENAME.
PAR-1.
  IF NOT (SQL-OK OR (SQL-NOT-FOUND))
    MOVE 'FETCH' TO H-SQL-CODE
    PERFORM 9999-SQL-ERROR.
  IF SQL-OK
    IF WS-EMP-IND-01 = C-SQL-NULL
      MOVE SPACES TO DATE-INC.
  IF SQL-NOT-FOUND
    MOVE 'Y' TO SW-EOF-ORDERS.
PAR-2.
  IF NOT (SQL-OK)
    MOVE 'OPEN' TO EM900-FUNCTION
```

```
    PERFORM 9999-SQL-ERROR.
  IF SQL-OK
    PERFORM 2000-FETCH-TRANSACTION UNTIL STOP-FETCH.
```

In our assembly line, we have also a button that carries out the complete maintenance task. It is the `ApplyAll` button in Figure 2. If we process the above program with our assembly line by pressing the `ApplyAll` button, we obtain the completely transformed program below.

```
  IDENTIFICATION DIVISION.
  PROGRAM-ID. XXXXX.
  AUTHOR.     N.N.
  DATE-WRITTEN. APRIL 22, 1998.
  DATA DIVISION.
  WORKING-STORAGE SECTION.
  01 L-EM948 PIC X(09) VALUE 'EM948   L'.
* FOR TESTING EM948 SQL-CODES
  COPY A0046075.
  PROCEDURE DIVISION.
  1010-DETERMINE-MAX-SERIALNR.
    MOVE SQLCODE TO W-SQL-CODE.
    MOVE SQLCODE TO SQL-CODE IN LINKAREA-EM948
    CALL 'UT100' USING L-EM948 LINKAREA-EM948
    IF RETURNCODE = '9'
      PERFORM 9999-EM904-FILL
    ELSE
      IF IND-MAX-SERIALNR = -1
        MOVE 1 TO HH-MAX-SERIALNR-MESSAGE
      ELSE
        IF SQL-OK
          ADD 1 TO HH-MAX-SERIALNR-MESSAGE
        ELSE
          MOVE 'XXXXXXXXMESSAGE' TO EM900-TABLENAME.
  1015-ADD-TCTMESSAGE-CT590.
    EXEC SQL
      INSERT INTO XXXXXXXXMESSAGE
      VALUE (:DCLXXXXXXXXMESSAGE.TRANSCODE-ORIGIN
             :DCLXXXXXXXXMESSAGE.EVENTTYPE
             :DCLXXXXXXXXMESSAGE.SERIALNR-MESSAGE
             :DCLXXXXXXXXMESSAGE.BUF-MESSAGE)
    END-EXEC.
    MOVE SQLCODE TO W-SQL-CODE.
    MOVE SQLCODE TO SQL-CODE IN LINKAREA-EM948
    CALL 'UT100' USING L-EM948 LINKAREA-EM948
    IF RETURNCODE = '9'
      PERFORM 9999-EM904-FILL
    ELSE
      IF SQL-OK
        NEXT SENTENCE
      ELSE
        MOVE 'XXXXXXXXMESSAGE' TO EM900-TABLENAME.
  PAR-1.
    IF SQL-OK
      IF WS-EMP-IND-01 = C-SQL-NULL
        MOVE SPACES TO DATE-INC
      ELSE
        NEXT SENTENCE
    ELSE
      IF SQL-NOT-FOUND
        MOVE 'Y' TO SW-EOF-ORDERS
      ELSE
        MOVE 'FETCH' TO H-SQL-CODE
        PERFORM 9999-SQL-ERROR.
  PAR-2.
    IF SQL-OK
      PERFORM 2000-FETCH-TRANSACTION UNTIL STOP-FETCH
    ELSE
      MOVE 'OPEN' TO EM900-FUNCTION
      PERFORM 9999-SQL-ERROR.
```

As we can see, a `DATA DIVISION`, a `WORKING-STORAGE SECTION` is created, and the necessary data item and `COPY` statement are added. The `IF SQLCODE` parts are recognized, regardless extra parentheses, extra negations, and ordering of the SQL return codes. The code is changed to the new `CALL` and the `IF` uses the `RETURNCODE` instead. The control structure redesign is also insensitive to extra parentheses, etc. In the replacement program we can see the effect of equations `[1]` and `[2]` of `Eval-SQL-a`. So all the changes are performed in the right ordering.

Finally, in order to carry out such tasks on complete systems, it is not desirable to do this in an interactive way, even not using the `ApplyAll` facility. We see the interactive part of the ASF+SDF Meta-Environment as the development environment of the assembly line. When the assembly line is designed and implemented, an entire system should be transformed. We use the batch version of the ASF+SDF Meta-Environment to do this. We note that in [13] an elaborate discussion can be found how batch oriented system-wide maintenance and renovation tasks are carried out using the ASF+SDF Meta-Environment.

5 Peopleware Issues

It is hard to come up with an objective measurement for maintainability. In fact, what to one programmer appeals, could feel cumbersome to the other. This is maybe best illustrated by the religious wars between programmers about the use of `GO TO`s, programming language, indentation style, choice of text editor, commenting style, variable naming conventions, etc [35]. So maybe we could argue that there is no such measure at all. Still, the wish from many companies is to improve maintainability—after all major part of the total cost of a system is due to maintenance, as is confirmed in many studies [32, 38]; [36] gives a recent summary of these findings.

It is also known that programmers working on a system for some time, typically maintenance programmers, consider the code to be their own [47]. Due to this fact, special care should be taken towards code inspections in general [20], or temporarily outsourcing maintenance and renovation. So, when management decides to (temporarily) outsource maintenance or renovation of a system there is a serious danger that the maintenance team will reject the code that is returned to them. For, their code has been taken away, which is for them a sign that they are not doing their work properly. Then someone else fiddles around with it, and when it comes back it is broke. Harry Sneed mentioned during his keynote for the fourth Working Conference on Reverse Engineering that he experienced this phenomenon in an off-shore outsourced Year 2000 conversion. On the other hand, outsourcing can be quite effective, for instance, a recent study [25] reports that outsourcing averages about twice the productivity of in-house development in New England banking applications. In [15], a key to success for off-shore outsourcing reengineering projects appeared to be intensive communication. Note that our approach also included intensive communication.

The approach that we propose tries to avoid the abovementioned risks. First of all, the maintenance team provided us with individual changes that we carry out system wide.

Second, it is planned that maintenance teams of our partners are going to make those changes themselves, thus avoiding some of the abovementioned problems and risks. This paper is the first step in this process: we do a study in order to show how such projects are carried out in general. After our presentation of the assembly line we obtained some new transformations that we carried out. A next project that has been carried out for this bank is the automation of a temporary maintenance task where COBOL 85 is transformed back to COBOL 74 (see [13] for details). Ideally, the situation could be that in-house teams can make massive changes in a cost effective way, using tools that make the task a challenge instead of a tedious error-prone time-consuming job. We have done our very best to gear the tools as much as possible towards the normal situation of programmers that are working at this bank, for instance by using native patterns [44] and a grammar that is geared towards their dialect [9]. Moreover the names of the sorts in the grammar are chosen as much as possible from their manuals. At the moment we are constructing a similar maintenance/renovation architecture for Ericsson for a number of their proprietary languages [45].

6 Conclusions

In this paper we proposed an architecture to carry out software maintenance in an automated way. We applied our approach to a real-world COBOL/SQL stockbroking system plus some real-world maintenance tasks to show its effectiveness. We were able to implement the tasks in a few hours so that we could make the changes in a completely automated fashion on the entire system. We believe that this approach towards automated maintenance is promising in the sense that it is easy to apply, fast, and less error sensitive than traditional hand-crafted maintenance.

References

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

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

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

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

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

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

[7] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. Available at http://adam.wins.uva.nl/~x/scp/scp.html. An extended abstract with the same title appeared earlier: [8].

[8] M.G.J. van den Brand, M.P.A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. In I.D. Baxter, A. Quilici, and C. Verhoef, editors, *Proceedings Fourth Working Conference on Reverse Engineering*, pages 144–153, 1997. Available at http://adam.wins.uva.nl/~x/trans/trans.html.

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

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

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

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

[13] J. Brunekreef and B. Diertens. Towards a user-controlled software renovation factory. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 83–90, 1999.

[14] J.R. Cordy, C.D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16(1):97–107, 1991.

[15] G. Dedene and J.-P. De Vreese. Realities of off-shore reengineering. *IEEE Software*, 7(1):35–45, 1995.

[16] A. van Deursen. A comparison of Software Refinery and ASF+SDF. In *Program Analysis for System Renovation*, pages 9–1–9–41. CWI, 1997. Available at: http://www.cwi.nl/~arie/papers/refine.ps.gz.

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

[18] A. van Deursen, P. Klint, and C. Verhoef. Research issues in the renovation of legacy systems. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering*, LNCS. Springer-Verlag, 1999. Available at: http://adam.wins.uva.nl/~x/etaps/etaps99.html.

[19] Emendo Software Group, The Netherlands. *Emendo Y2K White paper*, 1998. Available at http://www.emendo.com/.

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

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

[22] J. Heering, P. Klint, and J. Rekers. Incremental generation of parsers. *IEEE Transactions on Software Engineering*, 16(12):1344–1351, 1990.

[23] J. Heering, P. Klint, and J. Rekers. Incremental generation of lexical scanners. *ACM Transactions on Programming Languages and Systems*, 14(4):490–520, 1992.

[24] IBM Corporation. *DB2 for MVS/ESA V4 SQL Reference*, 4 release 1 edition, 1997.

[25] C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, 1991.

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

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

[28] J.W.C. Koorn. GSE: A generic text and structure editor. In J.L.G. Diets, editor, *Computing Science in the Netherlands (CSN92)*, SION, pages 168–177, 1992.

[29] J.W.C. Koorn. Connecting semantic tools to a syntax-directed user-interface. In H.A. Wijshoff, editor, *Computing Science in the Netherlands (CSN93)*, SION, pages 217–228, 1993.

[30] J.W.C. Koorn. *Generating uniform user-interfaces for interactive programming environments*. PhD thesis, University of Amsterdam, 1994.

[31] W. Kozaczynski, J. Ning, and A. Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, 1992.

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

[33] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-12(3):308–320, 1976.

[34] T.J. McCabe. Personal Communication, March 1998.

[35] S. McConnell. *Code Complete*. Microsoft Press, 1993.

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

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

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

[39] M.P.A. Sellink, editor. *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF97)*, Electronic Workshops in Computing. Springer verlag, January 1998.

[40] M.P.A. Sellink, H.M. Sneed, and C. Verhoef. Restructuring of COBOL/CICS legacy systems. In P. Nesi and C. Verhoef, editors, *Proceedings of the Third European Conference on Maintenance and Reengineering*, pages 72–82, 1999. Available at http://adam.wins.uva.nl/~x/cics/cics.html.

[41] M.P.A. Sellink and C. Verhoef. Reflections on the evolution of COBOL. Technical Report P9721, University of Amsterdam, 1997. Available at http://adam.wins.uva.nl/~x/lib/lib.html.

[42] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions – extended abstract. In B. Nuseibeh, D. Redmiles, and A. Quilici, editors, *Proceedings of the 13th International Automated Software Engineering Conference*, pages 314–317, 1998. Full version in [43].

[43] M.P.A. Sellink and C. Verhoef. Development, assessment, and reengineering of language descriptions. Technical Report P9805, University of Amsterdam, Programming Research Group, 1998. Available at: http://adam.wins.uva.nl/~x/cale/cale.html.

[44] M.P.A. Sellink and C. Verhoef. Native patterns. In M.H. Blaha, A. Quilici, and C. Verhoef, editors, *Proceedings Fifth Working Conference on Reverse Engineering*, pages 89–103, 1998. Available at http://adam.wins.uva.nl/~x/npl/npl.html.

[45] M.P.A. Sellink and C. Verhoef. Generation of software renovation factories from compilers. Technical Report P9901, University of Amsterdam, Programming Research Group, 1999. Available via http://adam.wins.uva.nl/~x/com/com.html.

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

[47] G.M. Weinberg. *The Psychology of Computer Programming*. Van Nostrand Reinhold, 1971.