SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:1605–1642
Published online 8 August 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.745

Cobol minefield detection

Niels Veerman*,† and Ernst-Jan Verhoeven

Department of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

SUMMARY

In Cobol, procedures can be programmed in ways that lead to unexpected behaviour and reduced portability. This behaviour is recognized as so-called 'mines': programming practices containing hidden dangers and requiring extreme caution. Cobol mines can be created intentionally or by a programming error, and can be tripped at an unforeseen moment. This leads to minefields in source code with unseen hazards, which complicate understanding and maintenance, and which can lead to costly breakdowns of business critical software systems. We discuss Cobol mines and the dangers that come with them, having implemented a mine detector for Cobol. Our detector was deployed in an industrial situation, and a number of minefields were found in production systems. By restructuring a complex legacy application, we argue that code restructuring can be used to combat minefields. Copyright © 2006 John Wiley & Sons, Ltd.

Received 7 October 2005; Revised 21 February 2006; Accepted 22 February 2006

KEY WORDS: Cobol; defects analysis; error-prone code; coding standards; code restructuring

1. INTRODUCTION

Programming errors are present in every software system, and are very often a source of trouble. Errors can lay dormant in an application, ready to show up at any moment in time. When a dormant error arises during production, the system can suddenly break down. The origin of the error is often hard to find, causing great expense. Sometimes, a programmer uses an awkward programming style intentionally to implement certain logic. Reasons for this can be, for instance, the absence of a proper language construct, or a preferred programming style. Such programming practice can eventually lead to problems when another programmer, who is not familiar with such logic, has to modify the code; or when the code is migrated to a different environment and the behaviour of the application becomes entirely different.

*Correspondence to: Niels Veerman, Department of Computer Science, Vrije Universiteit Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands.
†E-mail: nveerman@cs.vu.nl

Contract/grant sponsor: Dutch Ministry of Economic Affairs via contract SENTER-TSIT3018; contract/grant number: CALCE: Computer-aided Life Cycle Enabling of Software Assets

The use of coding standards can aid the prevention of awkward programming styles. For example, a common coding standard in Cobol requires that dependencies between procedures are indicated by an alphabetic and numeric hierarchy of their names. All procedures appearing at the same level start with the same letter. This means that the main procedure in a program starts with `A1_MAIN` and all procedures that are directly referenced from that procedure start with `B01_..`, `B02_..`, and so on. Procedures that are approached from several different levels often fulfil common tasks such as error handling, and start, therefore, with a letter from the end of the alphabet (e.g. `Y01_LOG_ERROR`). A coding standard is often developed at the initial stage of the coding process, and is therefore hardwired into the code; it is part of the applications' architecture [1,2]. Consequently, coding standards can be regarded as *source code level architectural decisions*, guarding the quality of the source code. A violation of a standard may indicate a programming error, or potential programming malpractice causing error-prone code. With the help of analysis tools, one can detect intentional and unintentional violations of a standard, which can then be tracked down, mapped, and possibly repaired before a crash appears during a production run.

In this paper, we investigate the phenomenon 'mines' in Cobol. Mines represent potential hazardous programming constructs. In an industrial setting, we implemented a mine detector for Cobol and applied it to several systems. Furthermore, we show how code restructuring can demarcate Cobol minefields and other complicated code in a program.

1.1. Contributions

First, we discuss different implementations of the Cobol perform statement (Cobol procedure call), and we illustrate the variation in compilers by example programs. This confirms that there is no single semantics for Cobol, and that problems can arise when code must be migrated. Second, we present definitions for potential hazardous, error-prone, and incompatible programming practices in Cobol: Cobol mines. These mines can pose severe threats to the proper operation of a system. Third, we present a case study on Cobol minefield detection, which includes an analysis of five industrial Cobol systems from different companies and an evaluation of the results. Finally, we discuss the demarcation of minefields, which is illustrated by the restructuring and visualization of an evolved legacy application.

1.2. Related work

The notion of mines in Cobol is mentioned in [3]. In that paper, an approach is described to identify procedural structures in Cobol programs. An elaborate treatment on the behaviour of perform statements is given, and an algorithm is presented to identify non-structured ranges of perform statements; that is, ranges containing a mine. In our work, we elaborate on different types of mines, the (possible) consequences of a mine, and how one can counteract minefields. Although the authors of [3] recognize that the semantics of the perform statement is not precisely defined, they believe that the implementation of the perform statement in the IBM Cobol compiler [4] is similar to other modern Cobol compilers. According to our findings, this is not the case. We show that the semantics of a perform statement is not fully defined and we demonstrate differences between various compilers.

In [5], a formal semantic description of control constructs of Cobol is given. Because the semantics of a perform statement is not completely defined by the latest Cobol standard, the description holds only for a certain compiler, configuration and environment. In our analysis, we also chose a certain

DOI: 10.1002/spe

semantics for the perform statement, and elaborate on its consequences for the detection of Cobol mines.

In [6], a case study on the analysis and visualization of the control-flow of Cobol is presented. Although they elaborate on the flow of control in Cobol, they do not deal with the goto statement or the so-called fallthrough logic. These language features are very prominent characteristics for the control-flow in Cobol and the creation of error-prone code, such as mines. In our work, we do take these control-flow features into account while detecting hazardous code.

There are a number of commercial tools for analysing Cobol programs, but we could not find one that suited our projects' needs. In particular, an analysis of the fallthrough logic between Cobol paragraphs is not available in many tools. IBM's Cobol structuring facility has capabilities to detect intricate code, including several of the mines we define in this paper, but the tool appears to be no longer in service, and we could not find any case studies that used it for mine detection.

The classification we use for certain hazardous programming practices are quite specific for the Cobol language. Similar problems appear in other languages. For example, the existence of side-effects in the C language that have implementer-defined semantics [7] can cause portability problems. Another related topic is code smells [8]. Code smells are a kind of anti-pattern for programming [9], indicating that the structure and comprehension of a particular piece of code may be improved by refactoring. A number of code smells are proposed in [8]; some smells are specific to object-oriented programming, but there are also smells that can be applied to other paradigms. The relation to Cobol mines is that both smells and mines can be an indication of error-prone code.

2. COBOL CONTROL-FLOW AND COBOL MINES

2.1. Control-flow revisited

In order to understand the control-flow in Cobol, a few things need to be explained first. A Cobol program consists of four divisions: one for a description of the program, one for external dependencies, one for variable declarations, and one for the programming logic. The division for the logic is called the `PROCEDURE DIVISION` and is ordered similar to a text in natural language, using sections, paragraphs and sentences. A section is divided into paragraphs, and a paragraph consists of sentences. A sentence is a sequence of statements, ended by a period. Sections and paragraphs usually start with a label, which can be used to reference them from elsewhere in the program.

When a program is executed, control-flow starts with the first statement in the `PROCEDURE DIVISION`. As soon as the control-flow reaches the end of the last statement in the program, the execution ends. There are also a number of statements that end a program immediately: the `STOP RUN`, `GOBACK` and `EXIT PROGRAM` statements. The precise operation of these statements depends on whether they are executed in a main program or in a subprogram. The `EXIT PROGRAM` statement should not be confused with the plain `EXIT` statement, which has no effect on the execution of a program.

There are a number of constructs that control the program flow. The most famous control-flow construct in Cobol is probably the unconditional `GO TO` statement [10,11], which jumps to a location indicated by a label. In addition, there is a conditional `GO TO` statement, which jumps to a certain

DOI: 10.1002/spe

```
PROCEDURE DIVISION.
MAIN SECTION.
LABEL1.
    PERFORM LABEL2 THRU LABEL-EXIT
    STOP RUN.
LABEL2.
    DISPLAY 'EXECUTING LABEL2'
    IF FLAG
      GO TO LABEL-EXIT
    ELSE
      DISPLAY 'FLAG NOT SET'
    END-IF.
LABEL-EXIT.
    EXIT.
```

Figure 1. A Cobol procedure division.

label depending on a condition. This construct is used to simulate a case-like structure. A `GO TO`-related statement is the `ALTER` statement, which modifies the destination of a special `GO TO` statement at run-time. Because the `ALTER` statement can lead to very incomprehensive programs, it has been removed from the Cobol standard. Nevertheless, the `ALTER` statement still occurs in production code. Furthermore, a `GO TO`-like statement is the `NEXT SENTENCE` statement, which transfers control to the next sentence in the code; because a period indicates the end of the current sentence, the `NEXT SENTENCE` statement can lead to programming errors when a period is removed or added without proper care. An example of this problem is shown in [12, p. 232]. The `NEXT SENTENCE` statement is commonly used in production systems, but is considered archaic in the latest Cobol standard.

A procedure call in Cobol can be made using a `PERFORM` statement, which is used to execute a sequence of paragraphs or sections. In Figure 1, a `PERFORM` statement in `LABEL1` executes a sequence of paragraphs. From `LABEL2`, the control-flow can jump to `LABEL-EXIT` by the explicit `GO TO LABEL-EXIT` statement, or control can continue to `LABEL-EXIT` at the end of `LABEL2` by implicit fallthrough. The `EXIT` statement is purely for the programmer to indicate the end of a sequence; it does not affect the execution of the program. After `LABEL-EXIT`, the control-flow is transferred to the `STOP RUN` statement in `LABEL1`, and the program is terminated. We emphasize here that implicit fallthrough between paragraphs and sections can severely complicate program understanding, as such behaviour cannot be seen from a paragraph or section itself; it depends on the way a paragraph is reached. Furthermore, control-flow can be affected by Cobol's exception handling mechanism for files and the possibility to execute statements from a `SORT` statement or a `MERGE` statement; the operation of these mechanisms resembles a `PERFORM` statement, but we do not consider them in this paper. Hence, there are basically three ways to transfer the control-flow in a program: by a `GO TO` statement, by a `PERFORM` statement, and by the implicit fallthrough logic.

There are several statements, then, that can contain other statements, such as the `IF`, `EVALUATE` (a case-like construct), and the in-line `PERFORM` (a while-like construct). In addition to these well-known types of branching statement, there are several other statements that allow conditional execution

DOI: 10.1002/spe

of statements. This can be when an error arises from an operation, or, conversely, when no error arises. For instance, an arithmetic error can arise from an addition or division. A simple example of this mechanism is the following ADD statement:

```
ADD A TO B
   ON SIZE ERROR       PERFORM OVERFLOW-ERROR
   NOT ON SIZE ERROR   PERFORM COMPUTATION
END-ADD
```

In case of variable B being unable to store the value of variable A (e.g. the value of A is too big for B), an error routine is performed. If no error occurs, a computation routine is executed. Such structured statements are very similar to an IF statement: a condition is evaluated and then one of the branches is executed.

The Cobol 2002 standard [13] extended the Cobol language with a number of constructs that are also available in several newer languages. Important extensions for the flow of control are the new variants of the EXIT statement, which transfer control to the beginning or the end of a loop (EXIT PERFORM), the end of a paragraph (EXIT PARAGRAPH), or to the end of a section (EXIT SECTION). These variants resemble statements like the continue, break and return statements which are common in languages like C and Java. In addition, object orientation [14] is supported and exception handling has been improved, but we do not consider those extensions in this paper. The majority of the production code does not make use of these new features, and, since this paper is about minefield detection in existing systems, this restriction does not limit the applicability of our approach.

2.2. Some semantics of perform statements

The semantics of perform statements differs between Cobol dialects. In particular, the Cobol 2002 standard [13, p. 494] is undefined on nested perform statements:

> Cobol 2002 Standard
>
> The results of executing the following sequence of PERFORM statements are undefined and no exception condition is set to exist when the sequence is executed:
>
> - a PERFORM statement is executed and has not yet terminated, then
> - within the range of that PERFORM statement another PERFORM statement is executed, then
> - the execution of the second PERFORM statement passes through the exit of the first PERFORM statement.
>
> NOTE Because this is undefined, the user should avoid such an execution sequence. On some implementations it causes stack overflows, on some it causes returns to unlikely places, and on others other actions can occur. Therefore, the results are unpredictable and are unlikely to be portable.

So, the behaviour of multiple active overlapping perform statements is undefined by the latest Cobol standard, and it does not forbid the use of recursive perform statements. We consulted a few reference

DOI: 10.1002/spe

manuals from several Cobol dialects on these issues. We summarize our findings here, illustrated by text snippets from the manuals.

- Micro Focus Cobol [15]:
  '. . . an active PERFORM statement, whose execution point begins within the range of another active PERFORM statement should not allow control to pass to the exit of the other active PERFORM statement; furthermore, two or more such active PERFORM statements should not have a common exit.
  These restrictions are not enforced. PERFORM statements can be freely nested, and recursion (a PERFORM statement performing a procedure containing it) is allowed. Only the exit point of the innermost PERFORM statement currently being executed is recognized. These rules can be changed by use of the PERFORM-TYPE Compiler directive.'
- IBM Cobol [16,17]:
  'As an IBM extension, two or more active PERFORM statements can have a common exit.'
  'A PERFORM statement must not cause itself to be executed. A recursive PERFORM statement can cause unpredictable results.'
- Compaq Cobol [18] (formerly DEC Cobol) and DEC Cobol [19]:
  'Two or more active PERFORM statements cannot have a common exit.
  Use the check compiler option with the perform keyword to verify at run time that there is no recursive activation of a PERFORM.'
- AcuCorp Cobol [20]:
  'When this (compiler) switch is used, the PERFORM verb is modified so that return addresses are stored on a stack. Only the most recent PERFORM statement has an active return address. When this option is used, a paragraph under the control of a PERFORM statement may (directly or indirectly) PERFORM itself. See the configuration option PERFORM-STACK.'
- PERCobol [21]:
  'PERCobol supports a return-stack implementation of PERFORM.'
- Fujitsu Cobol [22]:
  'PERFORM statements within the range of containing statements cannot have a common exit. In this compiler, however, one or more PERFORM statements within the range of containing statements can have the common exit.'
- Siemens Nixdorf Cobol [23]:
  '. . . two or more active PERFORM statements must not have a common exit.'
- RM Cobol [24]:
  '. . . two or more active PERFORM statements may not have a common exit.'

These reference manuals have no uniform semantics of a nested perform statement, and its behaviour can often be changed by setting compiler switches. To examine the behaviour of perform statements in practice, we compiled and ran three small test programs using different compilers and examined the output. The three test programs are shown in Figures 2, 3 and 4. Display statements are used to track the control-flow when a program runs. We briefly explain the programs here.

- **Program P1.** This program has a nested overlapping perform statement. In `LABEL1`, a perform statement performs `LABEL2` through `LABEL3`. Then in `LABEL2`, a second perform statement references `LABEL3` through `LABEL4`, thereby passing control through the exit of the first perform statement.

DOI: 10.1002/spe

```
LABEL1.
    DISPLAY '1'
    PERFORM LABEL2 THRU LABEL3
    STOP RUN.
LABEL2.
    DISPLAY '2'
    PERFORM LABEL3 THRU LABEL4.
LABEL3.
    DISPLAY '3'.
LABEL4.
    DISPLAY '4'.
```

Figure 2. Program P1, with nested overlapping perform statements.

```
LABEL1.
    DISPLAY '1'
    MOVE 1 TO A
    PERFORM LABEL3
    DISPLAY 'END'
    STOP RUN.
LABEL2.
    DISPLAY '2'.
LABEL3.
    DISPLAY '3'
    IF A = 1
      MOVE 0 TO A
      GO TO LABEL2
    END-IF.
```

Figure 3. Program P2, where a goto statement jumps out of a performed paragraph.

```
LABEL1.
    MOVE 1 TO A
    PERFORM LABEL2
    STOP RUN.
LABEL2.
    DISPLAY A
    IF A < 3
      ADD 1 TO A
      PERFORM LABEL2
    END-IF
    DISPLAY 'END'.
```

Figure 4. Program P3, with a recursive perform statement.

DOI: 10.1002/spe

- **Program P2.** In this program, LABEL3 is performed. The first time this paragraph is entered, variable A has value 1 and thus a goto statement jumps to LABEL2 before the end is reached. After LABEL2, the control-flow can fall through to LABEL3. The second time this paragraph is entered, variable A has value 0 and thus the end of the paragraph is reached. When the perform statement in LABEL1 terminates, the program displays 'END'.
- **Program P3.** This program has a recursive perform statement. In LABEL1, variable A is initialised with value 1 and LABEL2 is performed. In LABEL2, the value of A is displayed. Then, if the value of A is less than 3, it is incremented and a recursive perform of LABEL2 is made. If A is greater than or equal to 3, the program prints 'END'.

We compiled these three programs with a number of different compilers on different platforms. We used AcuCobol [25], Compaq Cobol [26], Fujitsu Cobol [27], IBM Cobol [28], Micro Focus Cobol [29], and TinyCobol [30] (an open-source Cobol compiler), the platforms were Windows, OpenVMS, OS/400, Unix and Linux. The compilers were initiated with no additional directives, except for the Micro Focus compiler, which we used also with the directive PERFORM-TYPE=OSVS. This directive provides compatibility with the mainframe behaviour of OS/VS Cobol. The directive PERFORM-TYPE=RM provides compatibility with the behaviour of RM Cobol and the directive STICKY-PERFORM also influences the behaviour; we have not tried these options in the experiment. We mention that none of the compilers issued any warning during the compilation of the test programs.

The output of the three test programs is shown in Table I. The output of Program P1, with the nested perform statements, differs between compilers. In AcuCobol, Micro Focus Cobol and TinyCobol, the perform statements do not interfere with each other. First, LABEL2 is performed, which performs LABEL3 and LABEL4. Then, after the perform statement in LABEL2 terminates, LABEL3 is executed once more before the program ends. In Compaq Cobol, Fujitsu Cobol, IBM Cobol, and Micro Focus Cobol with the OS/VS directive, the nested perform statements do interfere. The first perform statement modifies the continuation point of LABEL3, and as soon as control-flow reaches that point, control-flow is transferred to the stop run statement after the first perform statement; LABEL4 is not reached.

The output of Program P2 also illustrates the use of the perform continuation point. All compilers except TinyCobol produce similar results for this program. The perform statement sets the continuation point of LABEL3 to DISPLAY 'END' in LABEL1. As long as the end of that paragraph is not reached, the continuation point remains active. If the program flow reaches that point at a later moment, the flow is transferred to DISPLAY 'END' in LABEL1. Therefore, the continuation point is preserved, even though the control-flow has left the performed paragraph. This can make comprehension and modification of programs difficult.

The output of Program P3 shows that Fujitsu Cobol and IBM Cobol do not support recursion by default, as well as Micro Focus Cobol with the OS/VS directive. The programs printed 'END' continuously and did not terminate. Compilers supporting recursion yielded programs that printed 'END' three times. The Compaq Cobol compiler also does not support recursion, but the test program terminated after it printed 'END' twice. We briefly discuss the IBM and Compaq implementations of the perform statement in more detail.

The implementation of the IBM Cobol compiler is discussed in [3], stating that each perform statement can save exactly one pending continuation point, i.e. a continuation that was set by an earlier perform statement and is still active. This works as follows: when a perform statement is executed, the pending continuation point of the performed paragraph is stored in an area specific for

DOI: 10.1002/spe

Table I. Output of the three test programs.

	AcuCobol[a]	Compaq[b]	Fujitsu[c]	IBM[d]	Micro focus[e]	Micro focus[f]	Tiny Cobol[g]
P1	1	1	1	1	1	1	1
	2	2	2	2	2	2	2
	3	3	3	3	3	3	3
	4				4		4
	3				3		3
P2	1	1	1	1	1	1	1
	3	3	3	3	3	3	3
	2	2	2	2	2	2	2
	3	3	3	3	3	3	END
	END	END	END	END	END	END	
P3	1	1	1	1	1	1	1
	2	2	2	2	2	2	2
	3	3	3	3	3	3	3
	END	END	END..	END..	END	END..	END
	END	END	loop	loop	END	loop	END
	END				END		END

[a]AcuCobol development suite 4.3 for Windows and Development System 6.2 for Linux.
[b]Compaq Cobol V2.7 for OpenVMS.
[c]Fujitsu NetCobol V7.0 & Cobol 85 V3.0 for Windows.
[d]IBM Cobol/400 & ILE Cobol/400 for OS/400.
[e]Micro Focus ObjectCobol 4.1 for Unix.
[f]Micro Focus ObjectCobol 4.1 for Unix, with compiler directive `PERFORM-TYPE=OSVS`.
[g]TinyCobol V0.61 for Linux.

that perform statement, and a new continuation point of the performed paragraph is set to the current perform statement's successor. When control is returned after the current perform statement terminates, the stored pending continuation point is restored and control continues with the perform statement's successor. However, when a perform statement is executed for a second time before its first execution has terminated, the pending continuation point is overwritten before it is restored. So, to see what happened in our test Program P3 we reconstructed its execution. The `PERFORM LABEL2` statement in `LABEL1` set the continuation of `LABEL2` to the stop run statement in `LABEL1`. Then, the first time `PERFORM LABEL2` in `LABEL2` was executed, the pending continuation of `LABEL2` to the stop run statement was saved and the new continuation of `LABEL2` was set to `DISPLAY 'END'`. However, when `PERFORM LABEL2` in `LABEL2` was executed for the second time, the pending continuation point to the stop run statement was overwritten before it was restored. Then, each time the end of `LABEL2` was reached, its continuation was restored from `PERFORM LABEL2` in `LABEL2`, but that was always the `DISPLAY 'END'` statement; hence the program did not terminate.

The Compaq Cobol compiler also does not support recursion, but uses a different model than the other compilers. It produced a program that printed '`END`' only twice before it terminated, whereas the other compilers either did not terminate or printed it three times because `LABEL2` was recursively

DOI: 10.1002/spe

executed three times. We examined the output of the debugger and it turned out that recursive calls can be made, but a problem occurs as soon as the end of LABEL2 is reached for the second time. At the end of LABEL2, first the continuation to LABEL1 was stored, and then overwritten twice by continuations to the DISPLAY 'END' statement, before the end of LABEL2 paragraph was reached for the first time. When the end was finally reached, there was one continuation, referring to DISPLAY 'END'. After the execution of that statement, it seemed that another continuation was expected at the end of LABEL2 but not found, and the program was terminated. The Compaq compiler can detect recursive calls by using a compiler option, but that option was not enabled and thus the problem was skipped without a warning. We assume that when using the Compaq compiler, precisely one continuation can be stored for each paragraph, whereas with the IBM compiler precisely one *pending* continuation can be stored for each perform statement. This means that in Compaq Cobol, one cannot have two or more active perform statements with the same exit, whereas in IBM Cobol one cannot have two or more simultaneous activations of the same perform statement. This is in line with the manuals we consulted. The subtle difference was exposed by the recursion in the test program, but could also be exposed by a program containing two active performs with a common exit.

Hence, by using three small test programs, we demonstrated that there are at least three different implementations to store the continuations for perform statements.

2.3. Hazardous control-flow: Cobol minefields

The test programs in the previous section illustrate that seemingly simple procedure calls in Cobol can cause non-intuitive behaviour and portability problems. We demonstrated that there exist different implementations to store the continuations of performed paragraphs (or sections).

Some of our findings are also discussed in [3], where a Cobol mine is considered to be an active continuation of a perform statement that is skipped owing to a goto statement. The mine can be tripped unexpectedly at a later moment. However, whether a mine can be tripped at a later moment depends on the Cobol compiler. For instance, if the mine is reached at a later moment during the execution of a second perform statement, the mine does not have to be active as some compilers have only one active continuation at a time. In the example program in Figure 5, which we adopted from [3], a mine can be activated and subsequently tripped. The first perform statement sets the continuation of paragraph E to B. If ERROR evaluates to true, control-flow is transferred to B by a goto statement but the continuation to B at the end of E remains active. Then, in B, paragraphs C–F are performed. Subsequently, if the error in D does not occur for the second time, the mine at the end of E can be tripped; however, this depends on the compiler. If the code is compiled with a compiler with only one active continuation at a time (e.g. AcuCobol, Micro Focus, Tiny Cobol), the mine cannot be tripped. If the code is compiled with a compiler with multiple active continuations, the mine *can* be tripped (e.g. Compaq, IBM, Fujitsu). On the other hand, if the mine is reached by means of a goto statement or fallthrough logic, (e.g. in B there is a goto statement to C instead of the perform statement) it is tripped regardless of the used compiler; we illustrated such behaviour with test program P2 in Figure 3.

In this paper, we consider a number of potential hazardous programming practices such as Cobol mines; code containing these structures is regarded as a Cobol minefield. A mine involves a range of performed labels combined with one of the three main ways to transfer control-flow in Cobol: perform, goto and fallthrough. Potential mines are performed ranges that can be entered or exited by goto or fallthrough logic, or performed ranges that overlap with other performed ranges. Hence, we identify

DOI: 10.1002/spe

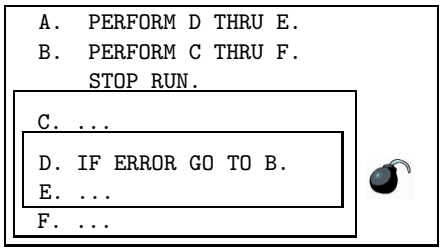

Figure 5. This code example was adopted from [3]. In case of error in paragraph D, the first perform statement leaves a mine at the end of paragraph E, which the subsequent perform can trip. However, this can only happen in IBM-like implementations of the perform statement.

three main types of mine, classified as a *perform mine*, a *goto mine* or a *fallthrough mine*. For the goto and fallthrough mines there are two distinct subtypes, indicated by *Into* (e.g. a goto jumps into a range) and *Out* (e.g. a goto jumps out of a range). The description and definition of mines is shown below. Each mine involves a performed range R combined with one or more labels denoted by $L_1..L_n$, a single label denoted by L, or a statement denoted by S.

- *perform mine*

  one or more labels $L_1..L_n$ that appear in two dissimilar performed ranges R_1 and R_2 (overlapping ranges, multiple entry and exit points)
  $L_1..L_n : L_1..L_n \in R_1 \wedge L_1..L_n \in R_2 \wedge R_1 \neq R_2$

- *goto mine*

  Into: a goto statement S that is not contained in a performed range R, which references a label L contained in R (jump into a range, multiple entry points) $S : S = gotoL \wedge S \notin R \wedge L \in R$
  Out: a goto statement S contained in a performed range R, which references a label L that is not contained in R (jump out of a range, multiple exit points) $S : S = gotoL \wedge S \in R \wedge L \notin R$

- *fallthrough mine*

  Into: the first label L in a range of performed labels R that can be entered by fallthrough logic from its syntactic predecessor (multiple entry types)
  $L : L = first(R) \wedge fallthrough(predecessor(L), L)$
  Out: the last label L in a range of performed labels R, which can be exited by fallthrough logic to its syntactic successor (multiple exit types)
  $L : L = last(R) \wedge fallthrough(L, successor(L))$

Some paragraph labels may be classified as several types of mine, which can be an indication of very intricate control-flow. For example, a label can be classified as both a perform mine and a fallthrough mine. Note that the above definitions for fallthrough mines are not complete, but they are sufficient to describe mines. The predicates *successor*() and *predecessor*() can be determined with

DOI: 10.1002/spe

little effort. The predicate *fallthrough*() requires more effort, since it cannot be seen from a paragraph itself. In fact, it is required to track the control-flow paths from the beginning of a program to the specified paragraphs. Therefore, the use of automatic tools to carry out large-scale minefield detection is inevitable.

The mines we defined can cause unexpected behaviour and portability problems, and complicate program understanding and modification; code with mines leads to multiple entry and exit points in a segment of code. Although the use of multiple and single entry and exit points in programming can be subject to debate, in Cobol one has to be especially careful with different types of entry point because these influence the way the control continues. One also has to be careful with multiple exit points because of the peculiar preservation of continuation points. Moreover, when we read the fine prints of the Cobol standard [13, p. 491] we found for a perform statement referencing 'procedure-name-1' through 'procedure-name-2' that: '*the flow of execution should eventually pass to the end of procedure-name-2*'. Hence, if a program has a path that jumps out of a performed range and does not reach the end of the last paragraph or section before it terminates, it is not in line with the Cobol standard.

Therefore, we believe it is good practice to disallow mines in a coding standard because mines violate best practices and can be harmful to the operation and maintenance of a system. By detecting mines in advance, one can prevent a system from high costs for error correction, updates, migration and system failure.

3. COBOL MINEFIELD DETECTION

We implemented a mine detector to identify Cobol minefields and applied it in a case study on minefield detection. The case study contained five business-critical Cobol systems from different companies, covering 830 000 lines of code in total. We present our tools, some implementation details, and then our case study.

3.1. Tools and implementation

3.1.1. Mine detection and control-flow analysis

In order to detect Cobol mines, control-flow analysis is required. We implemented a control-flow analysis tool for Cobol. Using the output of the analysis tool, we implemented a mine detector. The control-flow analysis tool visits and collects all possible control-flow paths in a program. For proper control-flow analysis, one needs to have precise semantics of the control statements. As the semantics for the perform statement differ among implementations, we chose a semantics: when several perform statements are active at the same time, only the continuation of the innermost perform statement is recognized. If the continuation of the innermost perform is reached, control is returned and the continuation of the outer perform becomes active again. Our semantics therefore resembles the behaviour we encountered with the Micro Focus, AcuCorp, and Tiny Cobol compilers. We believe that this semantics is easier to understand and more in line with the concepts of structured programming than a semantics with multiple active continuations, which we found in the Compaq, Fujitsu and IBM Cobol compilers.

DOI: 10.1002/spe

```
LABEL1.  PERFORM LABEL4 THRU LABEL5.
LABEL2.  DISPLAY 'THIS CAN BE DEADCODE'.
LABEL3.  STOP RUN.
LABEL4.  PERFORM LABEL5 THRU LABEL6.
LABEL5.  ...
LABEL6.  DISPLAY 'THIS CAN BE DEADCODE'.
         GO TO LABEL3.
```

Figure 6. Different perform statement semantics can cause different program points to be reachable.

Different semantics can yield different reachable points in a program. In some rare cases, code can be reachable when one continuation is active, whereas it is unreachable when multiple continuations are active, and *vice versa*. This means that we can miss a mine with our mine detector because the mine occurs in code that is considered not to be reachable. We illustrate this with a code sample in Figure 6. In the code, if only a single continuation can be active at a time, the code in `LABEL2` is not reachable. If multiple continuations can be active at the same time, the code in `LABEL6` is not reachable. The code sample shows that we may miss mines that are located in code that is unreachable according to the semantics we use for the control-flow analysis. The differences in reachable code under different semantics are caused by the existence of mines (e.g. the code in Figure 6 has overlapping perform ranges and a jump out of a perform range). Clearly, this is precisely the idea of minefield detection: the discovery of code that behaves differently when it is compiled with different compilers. Hence, in the cases where we miss a mine owing to the above circumstances, there is at least one other mine in the program that is responsible, and that mine is detected.

3.1.2. Visualization tool

We implemented a visualization tool that translates the output of the control-flow analysis to graphs in Dot format. Dot is part of the Graphviz package [31] and can be used to draw directed graphs. A directed graph is convenient for visualization of the control-flow in a program. Our visualization tool generates control-flow graphs that provide a comprehensive overview of the control-flow among paragraphs and sections in a program. This way, minefields can be visualized. In Figure 7, an example program with its generated graph is shown. The blocks represent sections and the ellipses represent paragraphs. The dashed arrow represents a perform statement and thick arrows goto statements[‡]. The number next to an arrow indicates in which order the control statements appear in a paragraph; in this case there are two goto statements in the paragraph `SUBPAR`, the first one jumps to the exit paragraph and the second one is a local loop. The thinner arrows represent fallthrough. Since `PERFORM SUBROUTINE` in `MAINPAR` can return control at the end of `SUBROUTINE`, there exists a fallthrough arrow from `MAINPAR` to `ENDPAR`.

[‡]The online electronic version of this article uses green arrows for perform statements and red arrows for goto statements.

DOI: 10.1002/spe

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
PROCEDURE DIVISION.
MAIN SECTION.
MAINPAR.
    PERFORM SUBROUTINE.
ENDPAR.
    STOP RUN.
SUBROUTINE SECTION.
SUBPAR.
    IF FLAG
        GO TO SUB-EXIT
    ELSE
        DISPLAY 'FLAG NOT SET'
    END-IF
    GO TO SUBPAR.
SUB-EXIT.
    EXIT.
```

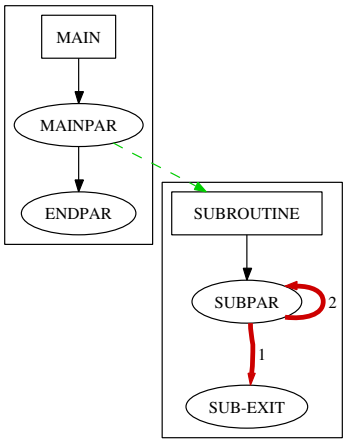

Figure 7. Example program and its generated control-flow graph. Dashed arrow: perform; thick arrow: goto; thin arrow: fallthrough; number: order of appearance in the code.

3.1.3. Implementation details

To carry out source code analyses fast and reliably, one can use automatic tools [32–34]. In our case, we used the ASF+SDF Meta-Environment [35–37] to implement the control-flow analysis tool and the mine detector, as well as the tool to visualize graphs. The ASF+SDF Meta-Environment is a development environment for the automatic generation of interactive systems for manipulating programs, specifications or other texts written in a formal language. Its development has been stimulated by research on industrial software renovation [38–40]. Syntax Definition Formalism (SDF) [41] supports the definition of both lexical and context-free syntax (production rules). In our case study, we used a Cobol grammar in SDF for parsing the Cobol programs. The grammar was derived from the online IBM VS Cobol II grammar [42–44] and modified [45] to be able to parse the Cobol dialects in the case study. To aid the adaptation of the grammar, we used the grammar deployment kit [46]. We employed a preprocessor for the Cobol programs that was adopted from [47], and the architecture we used for carrying out automated analysis and transformation of the Cobol programs is described in [33,34,39]. Furthermore, we constructed a Dot grammar in SDF using the grammar from the Dot manual [48, p. 34]. The Dot grammar was used for our visualization tool.

For a grammar in SDF, the ASF+SDF meta-environment generates a GLR parser [49]. The parser parses a program and yields a parsetree, which can be manipulated by transformations specified in ASF. Algebraic Specification Formalism (ASF) [50] supports the definition of equations (conditional rewrite rules). ASF has sufficient expressive power to describe typechecking, program translation and program execution. The control-flow analysis, the mine detector and the visualization tool were implemented using ASF rewrite rules. In Figure 8, an overview of the tools is given. We briefly describe the operation of the tools, using the program in Figure 7 as a running example.

DOI: 10.1002/spe

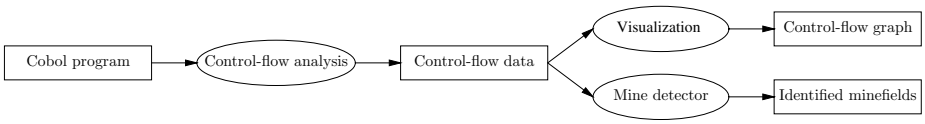

Figure 8. Overview of the tools.

The control-flow analysis explores all possible paths in a program, starting at the first statement in the procedure division. Only control statements are considered (i.e. goto statements, perform statements and statements terminating the execution of the program, and branching statements containing a control statement); all other statements are ignored. During the traversal of the program, each visited statement is stored in a list, representing the current path. In addition, a list of active perform statements is maintained. At the end of a paragraph, the list is queried to see if the paragraph is the current active continuation. If it is, control continues with the statement after the active perform statement. Otherwise, control falls through to the next paragraph and continues with its first statement. If there is no next paragraph (i.e. the end of the program is reached), the traversing stops and the current path is returned. A branching statement splits the current path, resulting in two or more possible paths. If a loop is encountered, the traversing stops and the path so far is returned. A loop occurs if a statement has already been visited within the scope of the current active continuation (i.e. a goto transfers control to a previously visited statement), or if a perform statement is already in the list of active perform statements (i.e. a recursive perform). The example program from Figure 7 has the following two possible paths (the scope of an active continuation is indicated by the curly brackets):

1. `PERFORM SUBROUTINE { BRANCH GO TO SUB-EXIT } STOP RUN`
2. `PERFORM SUBROUTINE { BRANCH GO TO SUB-PAR BRANCH }`

Each path starts at the first statement in the program, which is `PERFORM SUBROUTINE`. When the if statement is encountered (indicated in the path by `BRANCH`), the first path takes the if-branch containing the `GO TO SUB-EXIT` statement. At the end of `SUB-EXIT` (the exit statement is ignored), the active continuation of `PERFORM SUBROUTINE` returns control, and the next executable control statement is the `STOP RUN` statement, which ends program execution and thus the analysis of the first path. The second path takes the else-branch and encounters the `GO TO SUBPAR` statement (the display statement is ignored). Control is again directed to `SUBPAR`, and the if statement is visited for the second time within the scope of the same active continuation. That indicates a loop, and the analysis of the second path ends and the path that has been constructed so far is returned. The algorithm always terminates because the number of statements in a program is finite. Note that our control-flow analysis must use a little more information than that shown in the two example paths. In order to detect a loop, each statement is qualified by a unique identifier. We left this out in order to provide a concise example.

The retrieved control-flow data is used by the visualization tool to draw a control-flow graph, depicting the dependencies between the paragraphs. For example, according to the first path, there is fallthrough from the `PERFORM SUBROUTINE` statement in `MAINPAR` to the `STOP RUN` statement

DOI: 10.1002/spe

in ENDPAR. We implemented a transformation to construct a graph in Dot format from the retrieved information. Each edge is represented by a left-hand side and a right-hand side, and a number of attributes. Attributes are used to manipulate edges; for instance, attributes can specify styles, colours and labels. To draw a labelled bold red line representing the goto statement from SUBPAR to SUB-EXIT, the following Dot statement is sufficient:

```
"SUBPAR" -> "SUB-EXIT" [label = "1", style = bold, color = red]
```

Similarly, edges for fallthrough and perform statements are drawn, as well as the structures for the sections and paragraphs. Using Dot, we were able to specify the graphs concisely; the entire Dot specification for the example program from Figure 7 has less than 15 lines of code.

The main goal of this paper is, of course, to detect minefields. The mine detector uses the control-flow data to detect mines in a program. In order to detect perform mines, we gather all perform ranges (i.e. a list of paragraph labels) from the control-flow data and search for overlapping ranges. According to our definition, this means that we detect paragraph labels that occur in dissimilar ranges. In the example program, there is only one perform range, containing labels SUBPAR and SUB-EXIT. To detect goto mines, we proceed in a similar way. For the goto Into mines, for each perform range, we detect goto statements that occur outside the range, which refer to a label in the range. For goto Out mines, for each perform range, we search goto statements that occur in the range, which refer to a label outside the range. For the fallthrough mines, we look for the first label in a perform range that can also be reached by fallthrough from its predecessing label, and the last label in a perform range that has an outgoing fallthrough edge to its successor. The example program from Figure 7, which we used as a running example so far, has no mines. In the next section, we present several real programs containing harmful mines.

3.2. Case study: minefield detection

We applied our minefield detection to five industrial Cobol systems that are in production and being maintained by different companies. In total, the systems covered approximately 830 000 lines of Cobol code in nearly 1900 programs. We show some statistics in Table II to illustrate the diversity of the systems in terms of lines of Cobol code, number of programs, sections, paragraphs and control statements perform and goto. Consider, for instance, the amount of goto statements and the number of lines of code in System 3 compared to System 4. We started with System 1, a medium-sized system of nearly 80 000 lines of code, and evaluated the results with a system expert of that system. After that, we deployed our minefield detector on the rest of the systems. In Table III, the results are summarized. An interesting finding is that in System 4, which is almost 200 000 lines of code, there are only four mines. We assume that this has to do with the low number of goto statements in that system, which can be an indication that the chances on mines are lower in a system with less goto statements. Another characteristic that appears from the data in the tables is that the detected mines are not equally distributed over a system. In fact, a relatively small number of programs in a system are responsible for mines. This characteristic is in line with the findings of others [51–53]: a small number of modules in a system are responsible for the majority of the errors. Such a characteristic is known as a Pareto distribution, or the 80:20 rule: 80% of consequences originate from 20% of the causes. In our case, if we sum up all program of the systems, 93 of the 1886 programs are responsible for mines, which corresponds to 5% of the programs. We will now discuss our findings with System 1 in more detail. After that, we present the results of the other four systems.

DOI: 10.1002/spe

Table II. Statistics of the systems in the case study.

System	Loc Cobol	Programs	Sections	Paragraphs	Performs	Gotos
1	78 253	92	1406	2892	2308	872
2	17 168	31	434	940	825	105
3	79 867	85	1286	3723	2040	2633
4	198 384	833	3089	6311	5079	12
5	457 310	845	9979	21 124	17 513	7724
Total	830 982	1886	16 194	34 990	27 765	11 346

Table III. Results of the minefield detection.

System	Reported programs	Total mines	Perform mines	Goto		Fallthrough	
				Into	Out	Into	Out
1	14	491	—	30	35	213	213
2	9	19	18	—	—	1	—
3	6	43	8	—	1	18	16
4	3	4	2	—	—	2	—
5	61	413	153	26	42	96	96
Total	93	970	181	56	78	330	325

3.2.1. System 1

In the first system, we detected 491 mines in 14 programs. Most mines were fallthrough mines: a performed range that can also be entered or exited by fallthrough logic. In all detected cases, each range that can be entered by fallthrough was preceded by a range that can be exited by fallthrough. Therefore, the number of fallthrough Into mines is equivalent to the number of fallthrough Out mines. The cause of all the fallthrough mines was the presence of goto mines: a jump into or out of a range. We illustrate this with the control-flow graph of one of the reported programs in Figure 9, which also illustrates the problems associated with these mines. The program reads records from a file and updates the database if necessary. Section `B11_INIT` uses `X01_READ_FILE` to read the first record and establishes a connection with the database. Then, `B12_PROCESS` is executed until there are no more records in the file or an error occurs; `C121_PROCESS_AGR` is used to determine if the database must be updated. If all records are processed, `B13_CLOSE` closes the file, prints the number of read records, and commits the changes to the database. If an error occurs, `B13_CLOSE` performs a rollback operation and a subprogram is called. The subprogram handles and logs the error, and, when it terminates, control is returned to `B13_CLOSE`. After `B13_CLOSE` terminates, the program is ended by a stop run statement in `A1_MAIN`. However, in this program, there is a goto statement from paragraph `B11_01` to paragraph `B12_99` in section `B12_PROCESS`. This jump creates fallthrough between all

DOI: 10.1002/spe

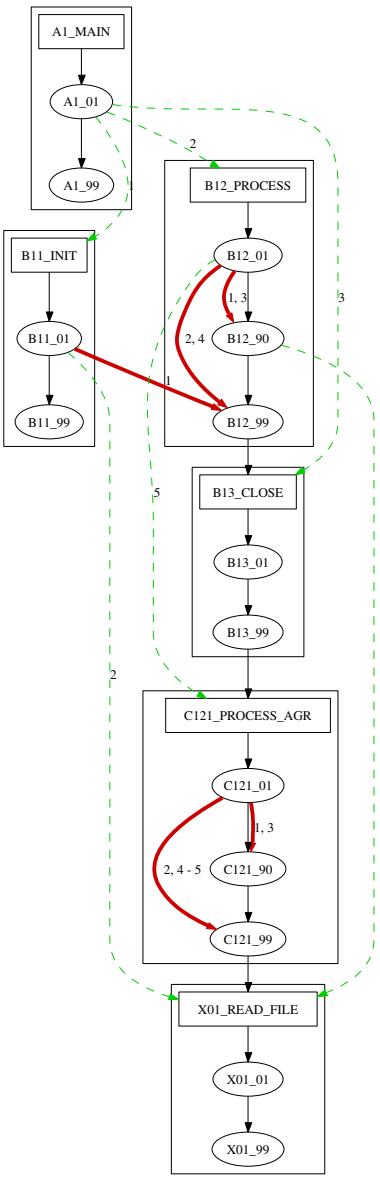

Figure 9. A Cobol minefield in System 1. The erroneous goto statement from section `B11_INIT` to section `B12_PROCESS` violates the corporate coding standard and causes unintended fallthrough between all subsequent sections.

DOI: 10.1002/spe

subsequent sections until the end of the program. In total, this program contains eight mines: two goto mines and six fallthrough mines.

An interesting finding in System 1 was the following one: since a jump out of a range will often result in a jump into another range, one would expect that the number of goto Into mines (jump into of a range) is equivalent to the number of goto Out mines (jump out of a range). However, we detected 35 goto Out mines but only 30 goto Into mines. A closer examination of the code revealed that some of the reported goto mines jumped to a label inside a section, which was not reachable otherwise. Hence, if the jump was programmed by mistake, it accidentally revived some dead code. This results in *zombie code*: code that used to be dead but which has been revived.

As we suspected that the mines in this system were not implemented intentionally, they created possible execution paths that were never intended to exist and may result in severe errors. For instance, incorrect data can be stored or displayed, or the program can terminate abnormally. Also it can be very difficult to trace the error, the system may be down for a while after such errors occur. We wanted to know how many of the detected mines were implemented intentionally and how many where implemented unintentionally. However, that may be difficult to determine without proper knowledge of the system. All we saw from the code was that there were jumps out of a performed section, which can be intentional as well as unintentional. Therefore, we consulted one of the system experts of System 1, and asked him his opinion about the 14 reported programs. It transpired that jumps between sections were not allowed in the corporate coding standard, and the system expert was confident that all cases were programming errors. Often, this was due to a sloppy copy-paste of code. Hence, in System 1, all mines were implemented unintentionally and thus the fallthrough execution paths and zombie code were also created unintentionally.

To have a better sense of the degree to which a minefield causes errors, we tried to asses the severity of the mine from Figure 9 in more detail. In `B11_INIT`, the input file is opened, and if something is wrong with the file, an error routine is called and an error switch is set. After this switch is set, the erroneous goto statement is executed and transfers control to `B12_99` unintentionally. This paragraph has only an exit statement, and control continues with `B13_CLOSE` because the continuation of `B12_PROCESS` is not active. In `B13_CLOSE`, the input file is not closed because it was not properly opened, and no committing to the database is done because no connection was established. The number of read records is displayed, which is zero, and, because the error switch was set, a subprogram is called to handle the error. After that, control falls through to `C121_PROCESS_AGR`. In that section, it is attempted to retrieve data from the database but that fails because no connection exists. This results in an error which is handled by the error routine for the database connection. The error switch, which was already set, is set again, and after the error routine terminates, control is transferred to `C121_99`, which contains only an exit statement. At the end of `C121_PROCESS_AGR`, there is no active continuation and control falls through to `X01_READ_FILE`. In that section, a record is read from the input file, and after that the end of the program is reached, which terminates the execution.

To summarize, the program in Figure 9 has a valid execution path that is clearly erroneous. If something is wrong with the input file in this program, control-flow proceeds in an unintended way through the program. Along the way, several subprograms are called and error logs are created that are not relevant to the actual error. This can obstruct proper maintenance and operation of the system severely. The error has either shown up already or not yet. We suspect that, in case this error has shown up already, the maintainers have been confused by the output and the error logs. At first sight, it appeared that something is wrong with the database. After several error logs are examined

DOI: 10.1002/spe

and some tests are run, it transpires that there was an error with the input file. The maintainer fixes the problem with the file, reruns the program, and the error has disappeared. In either case, the mines remain dormant in the program until they show up, threatening the operation of the system and causing unnecessary maintenance expenses.

Hence, the errors associated with the minefields in this system either have not yet occurred or a system expert knows what to do when these errors arise. The example illustrated that mines can cause code to be executed in an unintended order, resulting in unexpected output that is difficult to debug. A dormant error can also be activated when the program is modified. Errors, such as those found in this system, can be fixed in advance if the erroneous labels are corrected. We illustrate this in Figure 10, which depicts the graph of the repaired program from Figure 9. The jump from `B11_01` to `B12_99` was replaced by a jump from `B11_01` to `B11_99`. This way, the fallthrough mines in the subsequent sections are also removed. Hence, if the erroneous jump is repaired in this program, all mines are cleared. Nevertheless, a mine should always be cleared with care. For example, there can already be a work-around for a mine, and by removing the mine we can actually introduce a new error.

We delivered a report to the company with the detected mines in the system and the control-flow graphs. Some of the system experts were especially interested in the fallthrough analysis between sections because they had been confused by this behaviour during maintenance. This indicates that they had already suffered from minefields. After we reported the mines in System 1 and evaluated the results, we also analysed four other systems. For each system, we discuss the results separately, illustrated with some appealing examples.

3.2.2. System 2

System 2 is a small system of only 31 programs, covering about 17 000 lines of code. In total, 19 mines were detected in 9 programs; 18 were classified as perform mines and one as a fallthrough Into mine (fallthrough into a performed range). The fallthrough mine appeared in a small program of only two sections, which is shown in Figure 11. The first section performs the second one, but after that the control-flow executes the second one again by fallthrough; this was an error. The first section should have been terminated with an exit program statement but, instead, there was just an exit statement (which has no effect on the execution of a program). Consequently the code in the second section is unintentionally executed twice instead of once.

The 18 overlapping perform ranges were caused in a performed section, which performs some of its own inner paragraphs. Although this is perfectly legal in Cobol, this way of programming may lead to errors. If an internal range of performed paragraphs must only be reached by perform statements, a goto statement is required to skip the range when reached by fallthrough inside the section. If the inner range can also be reached by fallthrough (e.g. under different input conditions), we have a fallthrough mine: the flow of control after the inner range is determined by the way the range was reached. This complicates comprehension of the code, and we found an example of such a situation in System 3. In addition, a performed range inside a performed section can pose portability problems, as there is more than one active perform range inside the same section; a programmer might be tempted to implement a jump from the inner range to the outer range because the inner range is directly contained within a section. We will show examples of such code later. Furthermore, we read in one of the compiler manuals [54, p. 28] that when an end of a perform range lies within another range, the compiler produces very inefficient code. To summarize, the overlapping perform ranges in this system are programmed intentionally and cause no error at the moment, the coding style is prone to errors.

DOI: 10.1002/spe

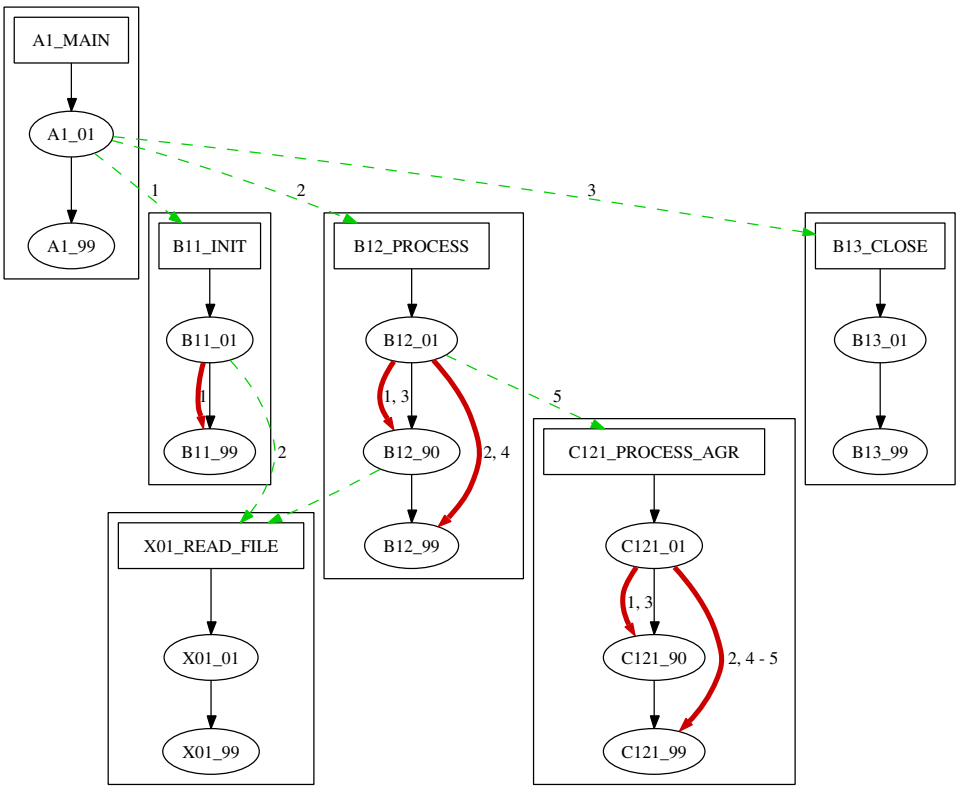

Figure 10. The repaired System 1 program from Figure 9.

We reported our findings to the company that maintained the system. The results were handed to the application manager of the system. The manager told us that she was eager to investigate our findings with the system experts, but that maintenance of the system was frozen at that moment, and thus no further preventive maintenance could be carried out at the time of writing this paper.

3.2.3. System 3

In System 3, also a medium-sized system of about 80 000 lines of code, 43 mines were reported. The perform mines and many of the fallthrough mines were caused by a performed section that performs some of its own paragraphs. Similar practice was also found in System 2, discussed above, but now the inner ranges are also reachable by fallthrough logic, and there were jumps from the inner range to the outer range. This means that the first paragraph of the inner range is classified as both a perform mine and a fallthrough mine, which results in very intricate code. We show a simplified code snippet from System 3 together with its visualization in Figure 12. The code implements a loop in

DOI: 10.1002/spe

```
PROCEDURE DIVISION.
MAIN SECTION.
MN_00.
    PERFORM A01_PROCESS.
MN_99.
    EXIT.

A01_PROCESS SECTION.
A01_00.
    MOVE 5 TO CUST_TYPE.
    ...
A01_99.
    EXIT.
```

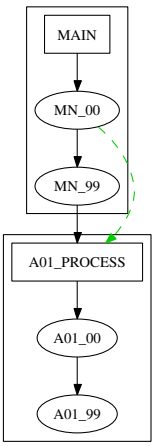

Figure 11. A Cobol fallthrough mine in System 2. The A01_PROCESS section is reachable by both a perform statement and by fallthrough logic.

which a table is searched; the entire section was about 150 lines of code, containing 31 goto statements and 21 paragraphs. The actual searching is done in paragraph C11, but that paragraph can be reached in two ways: by direct fallthrough from paragraph C09 and by a perform statement in paragraph C17. This results in several mines that complicate the code: an overlapping range (C11 THRU C15), a jump out of a range (in C11 GO TO C99), and a range that can be entered and exited by fallthrough (C09 into C11 THRU C15 to C17). In addition, this code has portability problems as it behaves differently on some compilers. On a compiler with a single active perform continuation, if C11 is reached by the perform statement in C17, the continuation of the outer perform statement (PERFORM C-DAT) is not active. Then, if in C11 the goto statement to C99 is executed, control-flow falls through from C99 to the next section in the program and control is not returned to PERFORM C-DAT; hence, there is an additional fallthrough mine at the end of C99 which occurs when certain compilers or compiler flags are used.

It would be interesting to interview the original developer of this code, but he was no longer maintaining this system. Hence, we were unable to receive proper feedback on this particular minefield. We assume that the complexity of the program has evolved over time, causing a tangled structure. For a new programmer, the code is difficult to understand and modify, it is prone to errors, and it has portability problems. One could untangle the code by restructuring it; however, if this particular piece of code is untangled fully automatically, it will result in code that is still difficult to grasp. In addition, to clear this minefield automatically, an automatic tool has to be tailored to some semantics for a perform statement, because the code is not in line with the Cobol standard. A more appropriate solution to such minefields is to detect, record and report such code, and let a system expert restructure the code by hand.

DOI: 10.1002/spe

```
     PERFORM C-DAT.
...
C-DAT SECTION.
...
C09. IF COUNT-BT > COUNT-1
        GO TO C99.
     ...
     IF P-CODE = ZERO
        ...
        GO TO C09.
     ...
C11. ...
     SEARCH TAB-LK
        AT END GO TO C99
        WHEN T-CODE = LK-1
             GO TO C13.
C13. ...
     IF AK-1 =ZERO
        GO TO C15.
     ...
C15. ...
C17. ...
     PERFORM C11 THRU C15
     ...
     GO TO CO9.
...
C99.
     EXIT.
```

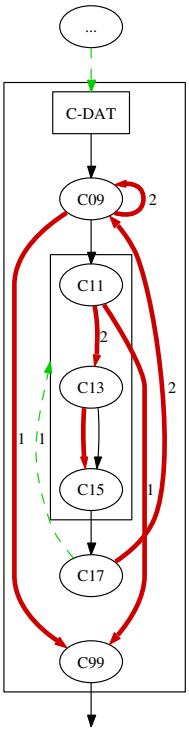

Figure 12. A Cobol minefield in System 3.

3.2.4. System 4

System 4 is a larger system with nearly 200 000 lines of code in 833 programs. A striking statistic is the low number of goto statements: only 12. The system was initiated in the early 1990s, and, apparently, the use of goto statements was restricted during implementation and maintenance. The result is hardly any goto statements, but in return there are some complex loops with deeply nested if statements and flag variables. Our mine detector reported four mines in this system: two perform mines, and two fallthrough Into mines. One of the fallthrough mines was similar to that found in System 2, where a section can first execute a subsequent section by a perform statement and then executes the subsequent section again by fallthrough. In this case, the mine was less dangerous because the perform statement referencing the second section was located in an if statement and thus can only be executed conditionally. The second section contained an unconditional program terminating statement, ending the execution of the program; hence, the second section never returned control and thus cannot be executed two times in a row in a single program execution. Nevertheless, the program contained an error that can cause problems sooner or later.

DOI: 10.1002/spe

```
    PERFORM X01_COMMIT_CHECK
...
X01_COMMIT_CHECK SECTION.
X01_00.
    IF ASPS_REC = ZERO
       PERFORM X11_CONT_CHANGED
    ELSE
       PERFORM X12_ASP_CHANGED
    END-IF.
X01_99.
    EXIT.

X11_CONT_CHANGED.
X11_00.
    ...
X11_99.
    EXIT.

X12_ASP_CHANGED SECTION.
    ...
```

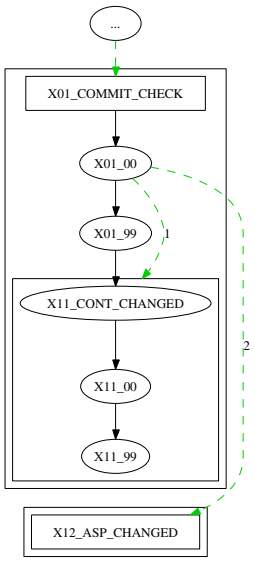

Figure 13. In System 4, an overlapping perform range was created unintentionally by an omitted a SECTION keyword after label X11_CONT_CHANGED.

The two perform mines and the other fallthrough mine in the system were caused by a programming error. We show a code sample in Figure 13 containing this error. In the code, it can be seen from the naming convention and section structure that the original intention was to create two subsequent sections; however, in the second section label X11_CONT_CHANGED, the keyword 'SECTION' was omitted. The compiler now assumes the label X11_CONT_CHANGED is just an empty paragraph in X01_COMMIT_CHECK. If the condition in X01_00 is true, PERFORM X11_CONT_CHANGED executes no statements. When that statement terminates, control-flow continues to X01_99 and then falls through to X11_CONT_CHANGED. Then the statements in X11_00 are executed and thus the behaviour is equivalent to the intended situation (i.e. the 'SECTION' keyword after X11_CONT_CHANGED is not omitted). The real error occurs when the condition in X01_00 is not true, and PERFORM X12_ASP_CHANGED is executed. After that statement terminates, the control-flow also reaches the statements in X11_00 but this time these are executed unintentionally. Our findings were handed to the application manager of this system; he was particularly interested in the sections with a missing key word, as he was surprised that such mistakes were not found earlier.

Although there were only four mines in this large system, the mines represent hazardous programming practices and errors that are a potential threat to the proper operation of the system. The low number of mines can be due to the low number of goto statements: only 12 in nearly 200 000 lines of code. Although mines are not always caused by goto statements, the figures on this system can be an indication that programming with less goto statements reduces the chances on mines significantly. In the next section, we show how the elimination of goto statements can help to fight minefields.

DOI: 10.1002/spe

```
        PERFORM RTC_AC
...
RTC_AC SECTION.
10_01. ...
        IF PRS = 99
           ...
           GO TO 101_99.
10_10. ...
        IF PRS = 99
           ...
           GO TO 101_99.
        ...
        PERFORM 101_20 UNTIL PRS = 02.
        ...
        GO TO 101_99.
10_20. ...
        IF PRS = 99
           ...
           GO TO 101_99
        ...
        IF PRS = 99
           ...
           GO TO 101_99.
        ...
10_99.
        EXIT.
```

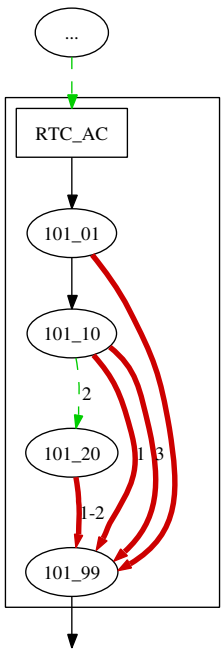

Figure 14. A Cobol minefield in System 5.

3.2.5. System 5

In System 5, we detected 413 mines in 61 programs. Most of the mines were fallthrough mines and these were all caused by unintended goto mines except for one; that fallthrough mine was caused by an omitted exit program statement, just like one of the mines in System 2. The number of goto Into and goto Out mines are not the same, because there were five jumps into the first section of the program, which was not performed itself, and there were 11 jumps from an inner range to a label in the outer range; hence, the jump originates in both the inner and outer range and its target is also located in the outer range. This is illustrated by the minefield in Figure 14. Such minefields cause portability problems, as observed in System 3: as soon as the jump is made out of the inner range, the continuation of the outer range is not active when certain compilers are used. The rest of the goto mines were unintentional programming errors. The perform mines were explained as follows: two of them were caused by forgotten `SECTION` key words (see the similar mines in System 4), and the other 151 were performed sections that perform some of its paragraphs as an internal subroutine.

DOI: 10.1002/spe

3.2.6. Summary

We applied our analysis to five industrial Cobol systems and we were able to determine potential error-prone programming constructs: Cobol minefields. We observed that a mine hardly comes alone, as in almost all cases one mine introduced a number of other mines, creating a minefield. Hence, clearance of one mine often results in clearance of other mines as well. Minefield clearance should always be done with care because a new error can be introduced. It is difficult to determine automatically whether a mine is programmed intentionally or unintentionally; therefore, one needs the help of a system expert to decide. The majority of the mines was caused by unintentional programming errors, and can cause code to be executed in an unintended order. This leads to problems ranging from invalid output to a sudden breakdown of the system during production. In general, unintentional mines can be cleared but this should be done with care.

Nevertheless, a minority of the mines was programmed intentionally, and can hinder maintenance of the code. Although it can be difficult to deal with this type of mine, it is valuable for a system expert to map the minefields of a system and to be aware of their presence. We also observed that in a large system with only a few goto statements, the number of mines can be significantly lower. Although we have little empirical data on this, it can indicate a relation between the number of goto statements and the number of mines in a system. In the next section, we show how we can deal with minefields by eliminating goto statements and other restructuring techniques.

4. DEMARCATION OF COBOL MINEFIELDS

We detected and evaluated intentional and unintentional mines in five industrial systems. Due to problems that can arise from minefields, it can be desirable to clear them. However, mines usually cannot be cleared that easily, owing to the imprecise semantics of the perform statement and the subjective meaning of the code. After all, it can be difficult to determine what the intention of the original programmer was, and the behaviour of a mine can be compiler-dependent. Nevertheless, we showed that a tool can detect and report minefields. In this section, we want to argue that the power of tools lies beyond the detection of minefields. We show how restructuring tools can help to fight minefields. A tool can simplify a complex minefield by demarcation of its perimeter, i.e. a tool can identify, mark and separate areas that are considered save. That way, a programmer is made aware of the code that should be approached with care.

4.1. Styles, standards and structures

In our definitions of mines, mines involve the perform statement. Nevertheless, we believe that the primary sources of programming problems, such as mines, are the use of fallthrough logic and the employment of numerous goto statements to express programming logic. This may not be a novel thought, but we know from experience that there is a great deal of Cobol code in production that makes extensive use of this ‘classic’ style. We also believe that there are valid reasons for that, and we will explain the most common reasons here.

Many Cobol programs were initiated decades ago. At that time, Cobol lacked support for certain ways of programming that are nowadays considered as structured. With the establishment of a new

DOI: 10.1002/spe

Cobol standard in 1985, new features for structured programming were added. Support for local loops was added by the in-line perform statement, replacing the need for a simulated loop by goto statements, and the explicit scoping of nested statements was another contribution that can improve the structure of programs. Furthermore, it was advised not to use certain language constructs, as these easily complicate the flow of control (e.g. self-modifying code using the alter statement). Then, as discussed earlier, the 2002 standard added more features for structured control-flow, common in several newer programming languages. These included constructs for breaking out of local loops, and to end prematurely the execution of paragraphs and sections. The idea is that these features restrict the use of the goto statement to exceptional cases.

However, extension of the standard does not mean that the extensions are immediately incorporated in existing and new code, and that existing compilers provide proper support. Fortunately, many Cobol compilers now have support for the 1985 standard, and a number of compiler vendors have started to implement the 2002 standard. Nevertheless, one can think of several reasons why a great deal of code is developed and maintained using a classic style with goto statements and fallthrough logic. One reason may be the common conception not to touch code that *works*. This is because a change means immediate costs and, more importantly, a change can introduce or reveal errors. Unless it is imminent to apply a change (e.g. owing to keeping in line with changing requirements or to correcting an error), the modification of a program to keep up with the latest standard or to improve its internal structure has usually a low priority. Another reason may be that, when a programmer modifies code, it is common practice to adhere to the original coding style. To avoid a mixture of styles, one may be forced, then, to program using fallthrough logic and goto statements. In addition, experienced Cobol programmers may be more familiar, skilled and productive when using the classic Cobol style; hence they are not tempted to program with newer constructs. Therefore, although there are ways in Cobol to reduce the need for fallthrough logic and goto statements, how to employ them effectively is not a trivial matter.

One approach to obtaining structured programs in a consistent, reliable and cost-effective way is the use of automatic restructuring tools and the establishment and enforcement of coding standards. Tools can detect violations of coding standards, and tools for restructuring Cobol code have existed for decades. These tools can automatically replace complex code by structured alternatives, resulting in a normalized control-flow without goto statements. However, there are several objections to the use of automatic tools. Because a restructured program can be very different from the original program, it is likely that the main objections to the use of restructuring tools come from the original maintainers. Restructured code can become non-intuitive due to the normalization of simulated constructs, the duplication of code, and the creation of meaningless flag variables and paragraph names [55]. In the maintainers' view, their precious code is mutilated by some automatic process, operating beyond their powers. They are not eager to accept and maintain the result. Also, the argument not to change code that works can still be valid. For example, a program with goto mines may behave differently after all goto statements have been removed. This is because the restructuring tool was tailored towards a specific semantics for the perform statement, which can be different from the implementation in the used compiler. Nevertheless, for many programs, some form of restructuring is inevitable sooner or later. The key to successful restructuring is to know what, how, and when to restructure. We summarize some findings here.

- The need for restructuring can become evident when the code or its environment must change [56]. For example, business requirements force the code to be changed, migrated

DOI: 10.1002/spe

or reused. Then, restructuring is done to enable a system to change. In addition, restructuring is then often accepted as part of the process as the code has to be changed anyway.
- The chances on acceptance of (automatically) changed code can be higher when the original maintainers are involved in the change effort, or when they are replaced [57] (e.g. retirement, outsourcing).

Although the internal quality of code is rarely a reason to modify code, we will illustrate how restructuring can aid combatting Cobol minefields; this adds another rationale for code restructuring. To do this, we present a restructuring approach in more detail. We do not state that this is the *best* way to deal with minefields; careful programming and other restructuring approaches can serve as well.

4.2. Restructuring of Cobol

Sellink *et al*. [56] developed a method to restructure Cobol code which eliminates fallthrough logic by transformation of paragraphs into subroutines. The restructured program is divided into a main part and a subroutines part. The main part can contain goto statements and fallthrough logic, whereas the subroutines part contains only paragraphs that are free of fallthrough logic and goto statements. The primary goal is to move as many paragraphs to the subroutines section as possible, which means that a large part of the program is free of fallthrough logic and goto statements. It is then more obvious to add new code that does not interfere with the existing control-flow, which is in-line with common ideas on structured programming. In addition, it is the intention to alter the existing paragraphs as little as possible. Although the approach moves a great amount of code around, changes to the existing paragraphs are kept as small as possible. As there are no flag variables or labels generated to normalize the control-flow, it is not always possible to remove all goto statements. The rationale is that some types of goto statement simulate constructs that were not available in Cobol at the time the code was written. This means that some of the goto statements that remain after the restructuring can be replaced by variants of the extended exit statement, which was introduced in the Cobol 2002 standard. However, the restructuring approach does not introduce features of the 2002 standard as these are not supported by all compilers. Sellink *et al*. [56] presented an industrial Cobol program to illustrate the approach.

We continued their work to see whether the ideas are sufficient for restructuring large Cobol systems [12]. In our case studies, the restructuring approach was able to transform up to 80% of the code of a large system into subroutines (>2 million lines of code). This can be further increased if Cobol 2002 features are introduced. Thus, only 20% of the code remained in the main part, containing goto statements and fallthrough logic. To illustrate what this means, we discuss some of the production code that we restructured.

Figure 15 shows a small code snippet from [12], before and after the restructuring effort. The accompanying control-flow graphs are depicted in Figure 16. First, a goback statement ensures that there is no fallthrough logic beyond paragraph `7299`. Then, the paragraphs are moved one by one. A moved paragraph is replaced by a perform statement that references the moved paragraph. The perform statement is added to the preceding paragraph. Existing perform ranges are taken into account during restructuring to make sure that the behaviour of the program is not modified. This means that not every paragraph can be moved (e.g. owing to the existence of mines or complex perform ranges). Each existing goto statement that references the moved paragraph is replaced by a perform statement and a goto statement. The perform statement references the moved paragraph

DOI: 10.1002/spe

and the goto statement jumps to the paragraph label after the performed paragraph. This way, the existing logic is preserved. Before a paragraph can be moved, it must be untangled from its surrounding paragraphs, which may require elimination of some goto statements. For example, when paragraph `7209` is moved to the subroutines section, `PERFORM 7209` is added to the end of paragraph `7207`. When the condition in the if statement in `7207` is complemented, the `PERFORM 7209` is placed in the if branch. Then, `GO TO 7299` can be removed. That way, `7207` has no external goto statements and can be isolated as well. This process continues until no more code can be moved or goto statements can be eliminated.

Hence, in this example, about 80% of the code is restructured into subroutines, which are performed from a main part that contains about 20% of the code. Paragraphs in the subroutines part have become loosely coupled blocks of code. One can rearrange, reuse or modify the separate blocks more easily as they are no longer tangled with each other. Moreover, one can safely add new paragraphs to the created subroutines part without interfering with the fallthrough logic between existing paragraphs.

4.3. Minefield demarcation

We showed that a Cobol program can be divided into a main part and a subroutines part by restructuring. We now explain what such an approach can do to fight minefields.

With the above restructuring approach, in a restructured program, all remaining fallthrough logic and goto statements are located in the main part, and the subroutines part consists solely of performed paragraphs. Mines are not moved; they are recognized by the restructuring algorithm and left alone (we already explained that perform ranges are taken into account during the restructuring, see [12] for more details). Hence, existing minefields remain in the main part of the program and the subroutines part is a save area without mines. Minefields are therefore automatically demarcated. Moreover, by moving the code surrounding a minefield to the save area, we can shift its perimeter and reduce its size. We demonstrate this with one of the minefields we encountered in our case studies.

The minefield in System 3 that we presented in Figure 12 contains several mines. Although it is a complicated piece of code with perform, goto and fallthrough mines, we can restructure some of its paragraphs into subroutines. This is shown in Figure 17. Two paragraphs are isolated from the minefield, and the perimeter of the minefield is shifted. This way, the size of the minefield is reduced and the size of the intricate code has been reduced to one paragraph. So, by restructuring a program, we can demarcate a minefield. To illustrate the demarcation of minefields and complex code in a large program, we present a case in which we restructured a Cobol legacy application for an IT-service company.

4.4. Demarcation of a legacy application

An IT-service company was requested to improve the maintainability of a large Cobol program, as part of a legacy portfolio modernization project. It concerned a 15 000 lines mainframe program which was initiated more than 25 years ago. The company restructured the program by hand, reducing it to 10 000 lines. The reduction of 5000 lines was due mainly to the removal of code that was considered no longer necessary. However, they wanted to restructure it further and realized that the use of automatic tools was inevitable for a fast and accurate result. They asked us to apply our restructuring tools to their program. The program we received contained only one section with 236 paragraphs, 4690 statements, 501 goto statements, and 173 perform statements. It turned out that the program was initially developed with

DOI: 10.1002/spe

```
72-PT-MS SECTION.
7201.
  MOVE M-PT IN TRA-NUM TO M-ACC.
  MOVE 0 TO PT-M-TABLE-R.
7203.
  IF M-ACC < 01
    ADD PERIOD-ACC TO M-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  IF M-ACC < PERIOD-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  IF M-ACC = PERIOD-ACC
    MOVE 1 TO M-TABLE(M-ACC)
    GO TO 7205.
  SUBTRACT PERIOD-ACC FROM M-ACC.
  GO TO 7203.
7205.
  ADD PERIOD-ACC TO M-ACC.
  IF M-ACC > 72
    GO TO 7207.
  MOVE 1 TO M-TABLE(M-ACC).
  GO TO 7205.
7207.
  MOVE PT-M-TABLE-R TO PT-MS.
  MOVE 0 TO DEP-DATA IN TRA-NUM.
  IF P-TYPE
    GO TO 7299.
7209.
  MOVE 0 TO MIN-AMOUNT IN TRA-NUM
            MAX-AMOUNT IN TRA-NUM
            REMAIN-PERC IN TRA-NUM.
7299.
  EXIT.
```

```
72-PT-MS SECTION.
RESTRUCTURE-PAR.
  PERFORM 7201
  PERFORM 7203
  PERFORM 7205
  PERFORM 7207.
7299.
  EXIT.
  ... GOBACK.
72-PT-MS-SUBROUTINES SECTION.
7201.
  MOVE M-PT IN TRA-NUM TO M-ACC
  MOVE 0 TO PT-M-TABLE-R.
7203.
  PERFORM TEST BEFORE UNTIL ( M-ACC < 01 )
      OR ( M-ACC < PERIOD-ACC )
      OR ( M-ACC = PERIOD-ACC )
    SUBTRACT PERIOD-ACC FROM M-ACC
  END-PERFORM
  IF M-ACC < 01
    ADD PERIOD-ACC TO M-ACC
  END-IF
  MOVE 1 TO M-TABLE(M-ACC).
7205.
  ADD PERIOD-ACC TO M-ACC
  PERFORM TEST BEFORE UNTIL ( M-ACC > 72 )
    MOVE 1 TO M-TABLE(M-ACC)
    ADD PERIOD-ACC TO M-ACC
  END-PERFORM.
7207.
  MOVE PT-M-TABLE-R TO PT-MS
  MOVE 0 TO DEP-DATA IN TRA-NUM
  IF NOT ( P-TYPE )
    PERFORM 7209
  END-IF.
7209.
    MOVE 0 TO MIN-AMOUNT  IN TRA-NUM
              MAX-AMOUNT  IN TRA-NUM
              REMAIN-PERC IN TRA-NUM.
```

Figure 15. Code before and after restructuring, taken from [12].

DOI: 10.1002/spe

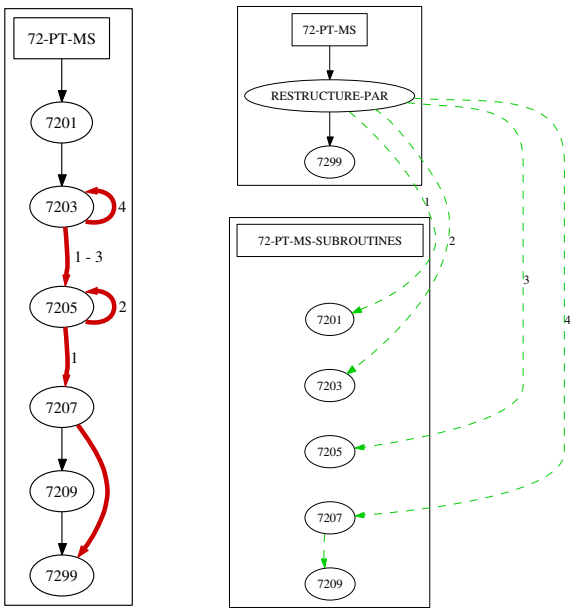

Figure 16. Control-flow graphs of the code from Figure 15 before and after restructuring.

many goto statements, and later on people started adding perform statements. This resulted in about 30 paragraphs near the end of the program that were only reachable by perform statements, resembling a subroutines structure. Still, more than 200 paragraphs were tangled with intertwined logic, and we also detected two intentional goto mines. To obtain an impression of the complexity of the application, we generated a control-flow graph, depicted in Figure 18. We mention that the online electronic version of this article provides insight into the distribution of the perform and goto statements by using coloured lines. In addition, one can read the paragraph labels by enlarging the graph.

At first sight, it may appear to be difficult to draw conclusions from the graph, but this is not true. We explain the graph here. There are 236 nodes, representing the paragraphs in the program. The location of a paragraph in the graph is influenced by its position in the source file and by the attached edges. This means that paragraphs at the top of the graph appear in the beginning of the file and that an edge between two nodes decreases the distance between them. Identical edges have been merged by our visualization tool. From the top of the graph, about two-thirds of the graph is dominated by goto edges and some fallthrough edges. This red majority of the graph represents more than 200 paragraphs that are intertwined with unstructured goto and fallthrough logic. At the bottom of the graph, on the left-hand side, about one-third is covered with green perform edges, representing about 30 paragraphs with a subroutines structure. The conclusion that we can draw from this graph is that we are dealing with a complex and evolved program, containing a mixture of goto, fallthrough and perform logic.

DOI: 10.1002/spe

```
      PERFORM C-DAT.
...
C-DAT SECTION.
...
C09.  IF COUNT-BT > COUNT-1
         GO TO C99
      END-IF
      ...
      IF P-CODE = ZERO
         ...
         GO TO C09
      END-IF
      ...
C11.  ...
      SEARCH TAB-LK
         AT END GO TO C99
         WHEN T-CODE = LK-1
              PERFORM C13
              PERFORM C15
      END-SEARCH.
C17.  ...
      PERFORM C11
      ...
      GO TO C09.
...
C99.
      EXIT.
... GOBACK
C-DAT-SUBROUTINES SECTION.
C13.  ...
      IF NOT ( AK-1 = ZERO )
         ...
      END-IF.
C15.  ...
```

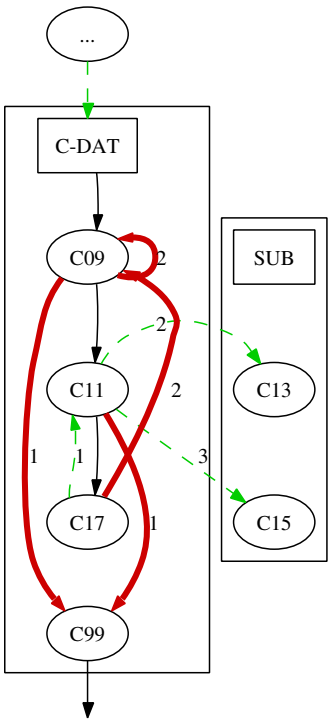

Figure 17. Demarcation of the minefield from Figure 12.

We applied our restructuring tools to the program to increase the number of subroutine paragraphs and to demarcate a save area from the intertwined goto logic. Some statistics of the original and restructured program are given in Table IV. We reduced the number of goto statements to 215 and we increased the number of perform statements to 454. The main part consists of 43 paragraphs and the subroutines part consists of 124 paragraphs. The total number of paragraphs was decreased because dead code was removed and some of the paragraphs were merged when the labels were no longer necessary. We generated a control-flow graph of the restructured program, shown in Figure 19.

Although it is not possible to distinguish the flow of the edges in the graph, the figure clearly shows that the program has been restructured into a main part on the left-hand side and a subroutines part on the right-hand side. The remaining goto statements and the two mines are clustered together in the small main part, and the majority of the paragraphs can be reached only by means of perform statements in the

DOI: 10.1002/spe

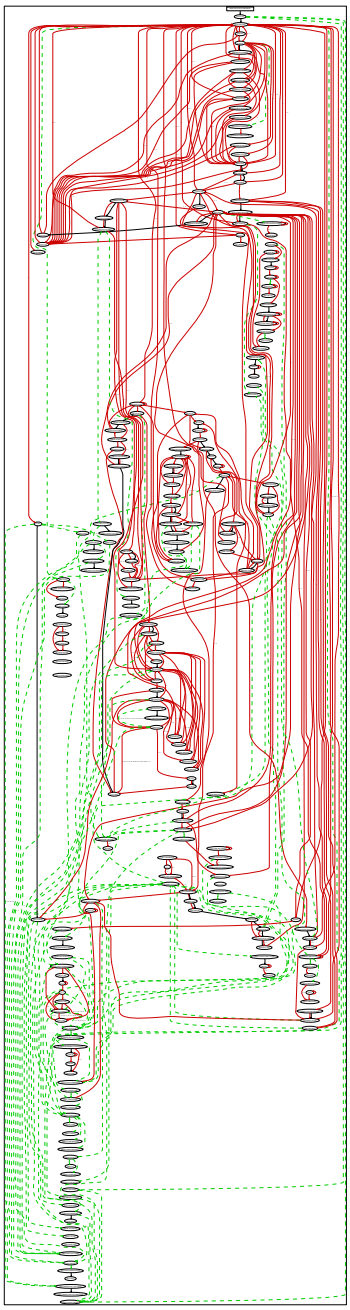

Figure 18. An evolved program.

DOI: 10.1002/spe

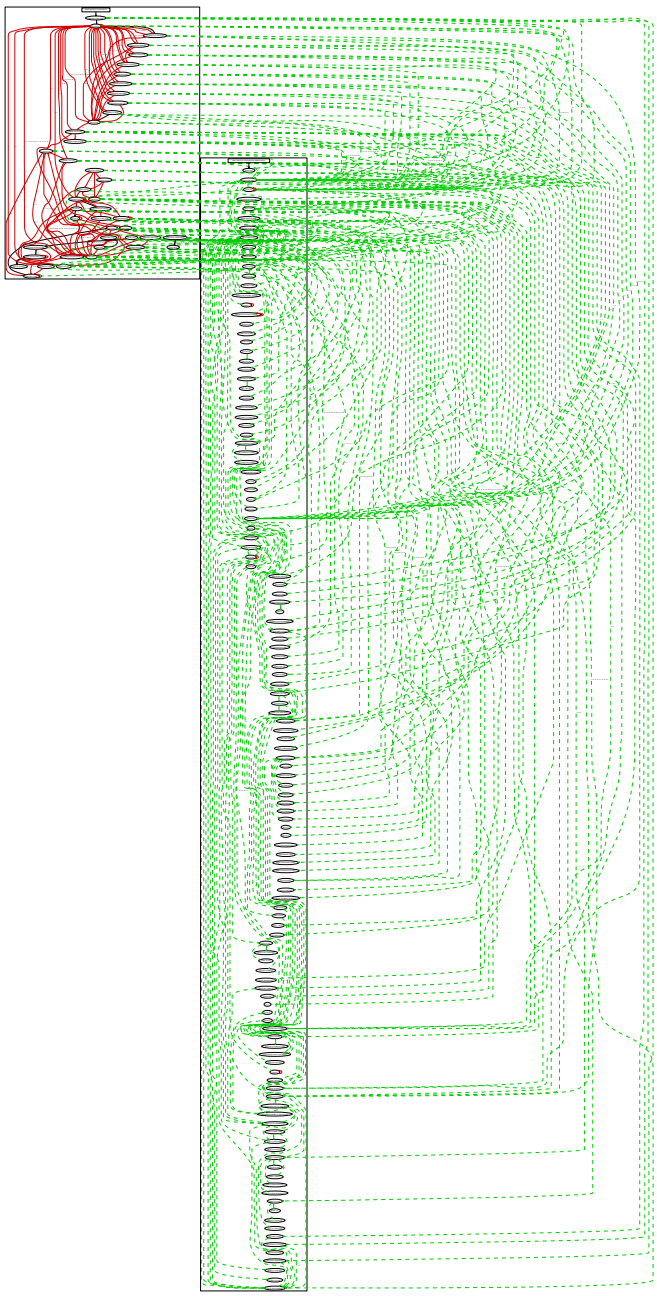

Figure 19. Automatic restructuring of the program from Figure 18.

DOI: 10.1002/spe

Table IV. Statistics of the restructuring effort.

	Original	Restructured	Percentage
Statements	4690	4119	−12
Goto statements	501	215	−57
Perform statements	173	454	+162
Paragraphs	236	167	−29
Subroutine paragraphs	0	124	n/a
Main paragraphs	236	43	−82

subroutines part. The code in the subroutines part can still contain interrelated structures, but there are no goto statements and fallthrough logic among the paragraphs. Hence, there is a clear demarcation line between code containing the intertwined logic (fallthrough and goto statements) and code containing only (non-overlapping) perform statements.

The performance of the restructuring can probably be improved by adjusting the tools. This means that we must add specific goto elimination patterns that occur in the program. Our restructuring effort was a *pro bono* activity, so our time was limited and we did not adjust our tools to this particular application. Nevertheless, we were able to achieve a significant result and help the programmers with their modernization project. After we restructured the program, we sent the code and the control-flow graphs to the company. They were pleased with the result and compiled and tested the program successfully.

4.5. Summary

We argued that restructuring can help to fight minefields and other intricate code. We illustrated this with several examples, including an evolved legacy application. By using automatic restructuring tools, we divided a large program into two parts: a small main part that contains complex code, such as minefields, and a larger subroutines part that consists solely of loosely coupled blocks of code. The code in the main part should be approached with care, but the code in subroutines part can be moved, modified and reused more easily. This way, a clear demarcation line has been established between potentially error-prone code and code that is considered to be secure.

5. CONCLUSIONS

We investigated the behaviour of the Cobol perform statement in theory and practice, and we presented definitions for potentially dangerous programming constructs in Cobol: mines. We analysed five industrial Cobol systems for minefields. We found that one mine often induces other mines, and that a mine can be programmed intentionally or unintentionally. Mines can cause sudden system failure, and can be hard to debug by hand. Fortunately, automatic tools can detect mines. Unintentional mines, representing programming errors that may not yet have occurred, can be repaired by a system

DOI: 10.1002/spe

expert before they do damage. Intentional mines, representing intentional programming logic, are more difficult to deal with, as they often represent certain complex logic in a program. One can implement a tool for automatic removal of mines, but such a tool should be tailored towards specific semantics. Instead, we argued that it is better to detect mines, as they can be an indication of intricate code that requires human inspection. Furthermore, we argued that code restructuring can assist fighting minefields, and we showed that a Cobol program can be restructured into a relatively small area with potentially error-prone code and a larger area containing only loosely coupled blocks of code. This way, a clear demarcation line is established between a save zone and the dangerous minefields.

ACKNOWLEDGEMENTS

We thank the anonymous reviewers of the journal *Software: Practice and Experience* for their valuable comments and suggestions for improvements. We are grateful to Steven Klusener for his support during the project and for his comments on a draft of this paper. Our thanks are due to Chris Verhoef for his ideas and comments on this paper. We thank Peter Bol from Getronics PinkRoccade and Carl Iglesias and Grant LeMyre from Telecom Management Consulting Group for their cooperation. We further acknowledge Wim Ebbinkhuijsen for his comments on an earlier draft of this paper. This research was sponsored by the Dutch Ministry of Economic Affairs via contract SENTER-TSIT3018 *CALCE: Computer-aided Life Cycle Enabling of Software Assets*.

REFERENCES

1. Klusener S, Lämmel R, Verhoef C. Architectural modifications to deployed software. *Science of Computer Programming* 2005; **54**:143–211.
2. McConnell S. *Code Complete*. Microsoft Press: Redwood, WA, 1993.
3. Field J. Ramalingam G. Identifying procedural structure in Cobol programs. *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99)*. ACM Press: New York, 1999; 1–10.
4. IBM Corporation. *IBM Cobol for MVS & VM Language Reference*. IBM Corporation: San Jose, CA, 1995.
5. Baumann P, Fässler J, Kiser M, Öztürk O. Beauty and the beast or a formal semantic description of the control constructs of Cobol and its implementation. *Technical Report ifi-93.39*, Institut für Informatik der Universität Zürich, 1993.
6. van Deursen A, Visser J. Building program understanding tools using visitor combinators. *Proceedings of the 10th International Workshop on Program Comprehension (IWPC '02)*, Washington, DC, 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 137–146.
7. Kernighan BW, Ritchie DM. *The C Programming Language*. Prentice-Hall: Englewood Cliffs, NJ, 1988.
8. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading, MA, 1999.
9. Brown W, Malveau R, McCormick H III, Mowbray T. *Anti-patterns: Refactoring Software, Architecture and Projects in Crisis*. Wiley: New York, 1999.
10. Dijkstra E. Go to statement considered harmful. *Communications of the ACM* 1968; **11**:147–148. Available at: http://www.acm.org/classics/oct95/.
11. Rubin R. 'GOTO considered harmful' considered harmful. *Communications of the ACM* 1986; **30**:195–196.
12. Veerman N. Revitalizing modifiability of legacy assets. *Software Maintenance and Evolution: Research and Practice, Special issue on CSMR 2003* 2004; **16**(4–5):219–254.
13. American National Standards Institute (ANSI). *Information Technology—Programming Languages—COBOL*. ISO/IEC 1989:2002(E), 2002.
14. Wessler J *et al*. *COBOL Unleashed*. Macmillan Computer Publishing: Houndmills, Basingstoke, 1998.
15. Micro Focus. *Micro Focus Cobol Language Reference*. Micro Focus International Limited: Rockville, MD, 2002.
16. IBM Corporation. *IBM Cobol for OS/390 & VM Language Reference* (5th edn). IBM Corporation, 2000.
17. IBM Corporation. *IBM Enterprise COBOL for z/OS Language Reference Version 3 Release 3*. IBM Corporation, 2004.
18. Compaq Computer Corporation. *Cobol Reference Manual Version 2.5*. Compaq Information Technologies Group, 2002.
19. DEC. *Cobol Reference Manual Version 2.3*. Digital Equipment Corporation, 2002.
20. Acucorp. *AcuCobol-85 Reference Manual*. Acucorp Inc, 1999.

DOI: 10.1002/spe

21. LegacyJ. *PERCobol Language Reference Manual* (8th edn). LegacyJ Corporation, 2005.
22. Fujitsu Software Corporation. *Cobol 85 Reference Manual*. Fujitsu Limited, 1996.
23. Siemens Nixdorf. *Cobol 85 Cobol Compiler Version 2.2A Reference Manual*. Siemens Nixdorf Informationssyteme AG, 1996.
24. Liant Software Corporation. *RM/Cobol Reference Manual* (1st edn). Liant Software Corporation: Austin, TX, 2005.
25. Acucorp. AcuCobol Development System. http://www.acucorp.com [1 January 2006].
26. Compaq. Compaq COBOL. http://www.compaq.com [1 January 2006].
27. Fujitsu. NetCobol. http://www.fujitsu.com [1 January 2006].
28. IBM Corporation. IBM Cobol. http://www.ibm.com [1 January 2006].
29. Micro Focus. ObjectCobol. http://www.microfocus.com [1 January 2006].
30. Free Software Foundation. TinyCobol. http://tiny-cobol.sourceforge.net/ [1 January 2006].
31. AT&T and Lucent Bell Labs. Graphviz—graph visualisation software, 2002. http://www.graphviz.org/ [1 January 2006].
32. van den Brand MGJ, Sellink MPA, Verhoef C. Generation of components for software renovation factories from context-free grammars. *Proceedings of the 4th Working Conference on Reverse Engineering*, Baxter ID, Quilici A, Verhoef C (eds.). IEEE Computer Society Press: Los Alamitos, CA, 1997; 144–153.
33. Sellink A, Verhoef C. An architecture for automated software maintenance. *Proceedings of the 7th International Workshop on Program Comprehension (IWPC '99)*, Pittsburgh, PA, 5–7 May 1999. IEEE Computer Society Press: Los Alamitos, CA, 1999.
34. Verhoef C. Towards automated modification of legacy assets. *Annals of Software Engineering* 2000; **9**:315–336.
35. van den Brand MGJ *et al.* The ASF+SDF meta-environment: A component-based language development environment. *Compiler Construction (CC '01)* (*Lecture Notes in Computer Science*, vol. 2027), Wilhelm R (ed.). Springer: Berlin, 2001; 365–370.
36. van den Brand MGJ *et al*. The ASF+SDF Meta-Environment. http://www.cwi.nl/projects/MetaEnv/.
37. Klint P. A meta-environment for generating programming environments. *ACM Transactions on Software Engineering and Methodology* 1993; **2**(2):176–201.
38. van den Brand MGJ, Klint P, Verhoef C. Core technologies for system renovation. *Proceedings of the 23rd Seminar on Current Trends in Theory and Practice of Informatics* (*Lecture Notes in Computer Science*, vol. 1175). Springer: Berlin, 1996; 235–254.
39. van den Brand MGJ, Sellink MPA, Verhoef C. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming* 2000; **36**(2–3):209–266.
40. van Deursen A, Klint P, Verhoef C. Research issues in the renovation of legacy systems. *Proceedings of the 2nd Conference on Fundamental Approaches to Software Engineering (FASE'99)* (*Lecture Notes in Computer Science*, vol. 1577). Springer: Berlin, 1999; 1–23.
41. Heering J, Hendriks PRH, Klint P, Rekers J. The syntax definition formalism SDF—Reference manual. *SIGPLAN Notices* 1989; **24**(11):43–75.
42. Lämmel R, Verhoef C. VS Cobol II grammar version 1.0.3, 1999. http://www.cs.vu.nl/grammars/vs-cobol-ii/.
43. Lämmel R, Verhoef C. Cracking the 500-language problem. *IEEE Software* November/December 2001; 78–88.
44. Lämmel R, Verhoef C. Semi-automatic grammar recovery. *Software—Practice & Experience* 2001; **31**(15): 1395–1438.
45. Klusener S, Lämmel R. Deriving tolerant grammars from a base-line grammar. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2003)*. IEEE Computer Society Press: Los Alamitos, CA, 2003.
46. Kort J, Lämmel R, Verhoef C. The grammar deployment kit. *Electronic Notes in Theoretical Computer Science* 2002; **65**(3).
47. van den Brand MGJ, Sellink MPA, Verhoef C. Obtaining a Cobol grammar from legacy code for reengineering purposes. *Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications (Electronic Workshops in Computing)*, Sellink MPA (ed.). Springer: Berlin, 1997.
48. Gansner E, Koutsofios E, North S. Drawing graphs with *dot*, 2002. http://www.graphviz.org/Documentation/dotguide.pdf [1 January 2006].
49. Visser E. Scannerless generalized LR parsing. *Technical Report P9707*, Programming Research Group, University of Amsterdam, July 1997.
50. Bergstra JA, Heering J, Klint P. The algebraic specification formalism ASF. *Algebraic Specification* (*ACM Press Frontier Series*), Bergstra JA, Heering J, Klint P (eds.). The ACM Press in co-operation with Addison-Wesley: Reading, MA, 1989; 1–66.
51. Boehm B, Basili VR. Software defect reduction top 10 list. *Computer* 2001; **34**(1):135–137.
52. Fenton NE, Ohlsson N. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering* 2000; **26**(8):797–814.
53. Ostrand TJ, Weyuker EJ. The distribution of faults in a large industrial software system. *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM Press: New York, 2002; 55–64.

DOI: 10.1002/spe

54. Micro Focus. *Micro Focus Server Express Program Development*. Micro Focus International Limited: Rockville, MD, 2002.
55. Calliss FW. Problems with automatic restructurers. *ACM SIGPLAN Notices* 1988; **23**(3):13–21.
56. Sellink A, Sneed HM, Verhoef C. Restructuring of COBOL/CICS legacy systems. *Science of Computer Programming* 2002; **45**(2–3):193–243.
57. Sneed HM. Risks involved in reengineering projects. *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press: Los Alamitos, CA, 1999; 204–211.

DOI: 10.1002/spe