What does aspect-oriented programming mean to Cobol?

Ralf Lämmel [1] and Kris De Schutter [2]

[1] Free University & CWI, Amsterdam, The Netherlands
[2] SEL, INTEC, University Ghent, Belgium

ABSTRACT

We study AOP in the context of business programming with Cobol. We face the following questions: What are join points in Cobol programs? What is advice? Does classic Cobol provide any constructs that hint at AOP? (Yes!) What are typical crosscutting concerns in the Cobol world? How do otherwise typical crosscutting concerns make sense for Cobol? How does AOP for Cobol align with classic re-engineering transformations for Cobol? We deliver an AOP language design for Cobol. Codename: AspectCobol. While we adopt several ideas from AspectJ and friends, we also devise original techniques for join-point identification and context capture. We briefly discuss a prototypical implementation of AspectCobol.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.1.m [**Programming Techniques**]: Aspect-Oriented Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

Keywords

Aspect-Oriented Programming, Business Programming, Cobol

1. INTRODUCTION

Cobol is dead; long live Cobol!

Daniel Sabbah's keynote at AOSD 2004 [28] was very encouraging for AOP aficionados with a Cobol hat on. Sabbah explained IBM's strategy towards the deployment of aspect-oriented software development. One issue pervaded his presentation: existing software assets are *heterogeneous* with regard to languages and platforms. So any new IT technology must deal with this heterogeneity.

Cobol is part of this heterogeneity problem. In fact, Cobol embodies a heterogeneity problem on its own — given the inflation of Cobol dialects and embedded languages. There are 180–200 billion lines of COBOL code in use worldwide. Most transactions in this world are Cobol-based. Billions of lines of business-critical Cobol code are still written every year. The existing code volume and the persistence of Cobol provide a strong incentive to understand *what AOP means to Cobol*. Let's not discredit nor circumvent Cobol. *Cobol also means something to AOP*.

AOP in the 1960s and 1970s

Cobol is perhaps the oldest aspect-oriented programming language. The procedure division of a Cobol program can hold so-called *declaratives* with special USE statements, which *provide a method of invoking procedures that are executed when some distinguished condition occurs during program execution*. The USE verb is in all Cobol standards, e.g., in the Cobol 74 standard [1]. It was already present in some form in CODASYL's Cobol 60 [29, p. 142].

Some 40 years ago grandpa wrote this aspect in Cobol:[1]

```
DECLARATIVES.
 HANDLE-F0815-ASPECT SECTION.
  USE AFTER ERROR ON FILE-F0815.
 HANDLE-F0815-ADVICE.
  MOVE "F0815"      TO PANIC-RESOURCE.
  MOVE "FILE ERROR" TO PANIC-CATEGORY.
  MOVE FILE-STATUS  TO PANIC-CODE.
  GO TO PANIC-STOP.
END DECLARATIVES.
```

Cobol is English, but we clarify details. Each file I/O error related to `FILE-F0815` is caught to be handled according to the Cobol sentences in paragraph `HANDLE-F0815-ADVICE`. Control is ultimately transferred to the procedure `PANIC-STOP`, which will display a decent error message and then stop execution.

This 40 years old aspect allows the *many* file I/O statements for `FILE-F0815` in the program to be *oblivious* of the potential need for error handling. "AOP is obliviousness *and* quantification" [9]. The above example performs *point-wise quantification* with regard to the affected file. The following example performs *universal quantification* with regard to the affected file, while it comprises one declarative section per file-processing mode:

```
DECLARATIVES.
 HANDLE-INPUT  SECTION.
  USE AFTER ERROR ON INPUT  CONTINUE.
 HANDLE-OUTPUT SECTION.
  USE AFTER ERROR ON OUTPUT CONTINUE.
 HANDLE-I-O    SECTION.
  USE AFTER ERROR ON I-O    CONTINUE.
 HANDLE-EXTEND SECTION.
  USE AFTER ERROR ON EXTEND CONTINUE.
END DECLARATIVES.
```

[1] We adopt the convention to highlight existing and AspectCobol-specific keywords. (We note that Cobol's language definition readily offers a distinction of globally reserved versus context-sensitive keywords.)

This paper appeared in the proceedings of AOSD'05, March 14–18, 2005, Chicago, USA.

This is a common, compiler-specific idiom.[2] The declaratives imply that file I/O errors are vacuously handled by no-op (i.e., CONTINUE), thereby avoiding that the runtime system panics. As a result, the programmer can place error handling code somewhere after the file I/O statements. This code can still observe file I/O errors because the runtime system communicates the errors via a file-status field. The given encoding is somewhat clumsy in that Cobol's syntax forced us to construct one separate USE statement per file-processing mode. In a later section, we will discuss a feature of the latest Cobol standard, which supports much more versatile error handling for file I/O and others.

More hints at AOP in Cobol'74 & '85 & 2002

One reviewer wondered whether we are "*stretching the point [...] about how 'aspect-oriented' COBOL was*" adding that "*many languages provide generic code introduction for error handling*".

Cobol'74 & '85 had interceptors for procedures! Again, declaratives offer this support: USE FOR DEBUGGING. Here, the keyword DEBUGGING hints at an *intended* usage pattern. One could intercept *specific* procedures or *all* procedures. (In AOP terms, this is a bit of quantification.) There is (was) also a special register, DEBUG-ITEM, providing access to the name of the intercepted procedure (and other useful information). (In AOP terms, this is a bit of join-point reflection.) Ironically, these language elements were obsoleted in the Cobol standard.

There are further standardised or vendor-specific forms of USE statements. Another classic form, USE BEFORE REPORTING, allows for customisation of Cobol's report generation. Again, this form is not about error handling. Cobol 2002 [13] offers two new forms of the USE statements that support exception handling.

How heavily is USE used? We scanned 21 Cobol'74 and '85 systems that were at our avail. These systems originate from 16 'shops', from 4 countries, most from the Netherlands, totalling 10 MLOC, with 7 systems $<$ 100 KLOC, and with some large systems, one as large as 2.7 MLOC. We found that 8 systems contained declaratives, all used for error handling. There was not a single USE FOR DEBUGGING aspect. There were altogether 1654 USE statements. There was one code base with 666 USE statements for a code volume of 0.5 MLOC. We observed that some shops do not use declaratives at all. Also, there is no correlation between the size of code bases and the fact whether declaratives are used.

Cobol 2008 = AspectCobol?

Admittedly, all known forms of declaratives are not fit for general crosscutting concerns. With an AOP hat on, the incompleteness of Cobol's accidental pointcut language is evident: we cannot advise *successful* file I/O statements, subprogram calls and field access. Also, Cobol's accidental join-point control and join-point reflection are very limited. For instance, we cannot re-execute an offending statement within a handler section, which hampers error repair.

Cobol is not a static language. For instance, OO constructs are covered by the current Cobol 2002 standard, with compiler support as early as the mid-1990s [21]. In this paper, we turn Cobol into a state-of-the-art AOP language. Codename: *AspectCobol*. For the record, we adopt several concepts from AspectJ [17]:

- Name-based pointcut designators (PCDs).
- Boolean connectors for PCDs.
- BEFORE, AFTER, AROUND and THROWS advice.
- The dynamic join-point model with CFLOW and friends.

[2] A related thread: http://www.talkaboutprogramming.com/group/comp.lang.cobol/messages/129352.html (comp.lang.cobol)

The release of the next Cobol standard is scheduled for 2008; a respectable list of extensions has been identified including native XML support, a class library for collections, function pointers and dynamic tables. We are too late with AOP for the Cobol 2008 standard. However, the standardisation procedure provides additional mechanisms. In particular, AOP could be covered in an *addendum* to the Cobol 2002/2008 standards. Hereby requested!

Plan of the paper

In Sec. 2, we collect illustrative logging teasers that are encoded in *AspectCobol*. Every AOP language should allow for versatile logging. In Sec. 3, we collect interesting examples of aspectual error handling. In Sec. 4, we will review typical crosscutting concerns just to see whether they make sense for *AspectCobol*: some do, others don't. We will also hint at some original applications of AOP in the Cobol world. In Sec. 5, we will discuss general language design issues for *AspectCobol*. In Sec. 6, we will discuss implementation issues on the basis of a prototypical implementation of *AspectCobol*. In Sec. 7, we conclude.

2. *AspectCobol* IN EXAMPLES: OBLIGATORY LOGGING TEASERS

We will use logging examples to become familiar with *AspectCobol*. In due course, we will touch upon various Cobol specifics. More specifically, we choose the theme of *logging file access*. Cobol applications tend to perform a lot of file access!

2.1 Basic AOP expressiveness

We face the following assignment:

> *Log all access to all files in a given program. We count as file access each statement that starts with one of these verbs: READ, WRITE, REWRITE, and DELETE.*

Such logging is out of reach for classic Cobol since it lacks statement interceptors. We need an aspect of the following shape:

```
DECLARATIVES.
 MY-LOGFILE-ASPECT SECTION. *> This is an optional line.
  USE AFTER
   *> The pointcut designator, i.e., the description of the join points.
  MY-LOGFILE-ADVICE.
   *> The advice, i.e., the paragraph to be executed at each join point.
END DECLARATIVES.
```

In fact, this is the overall structure of *any* intra-program aspect with AFTER advice. (We will later also exercise BEFORE, AROUND and THROWS advice.) A pointcut designator (for short: PCD) is a simple or compound Boolean expression that defines a pointcut, i.e., a set of relevant join points. Advice code boils down to a normal Cobol paragraph (i.e., a named series of sentences, each of which in turn is a series of statements). We assume that such advising paragraphs are executed according to the semantics of a PERFORM statement. That is, when control reaches the join point, then the advising paragraph is performed; when control reaches the end of the advising paragraph, then control is returned to the join point.

The PCD for MY-LOGFILE-ASPECT starts as follows:

```
(    EXECUTION OF READ    STATEMENT
  OR EXECUTION OF WRITE   STATEMENT
  OR EXECUTION OF REWRITE STATEMENT
  OR EXECUTION OF DELETE  STATEMENT )
*> PCD to be continued.
```

The disjunction lists all statement verbs for file access. In Cobol, it is normal to distinguish optional and obligatory keywords. The abbreviated version of the disjunction looks as follows:

```
(READ OR WRITE OR REWRITE OR DELETE)
```

We are not yet done with the pointcut. Logging should report on some details of the intercepted file I/O statement. *AspectCobol* allows us to extract such details from the join point. The extraction is described as part of the PCD, while the results are bound to variables for subsequent use in the advice code. These are the bindings needed for our logging aspect:

```
*> Continued from above.
AND BIND VAR-VERB   TO VERB   *> The given statement verb
AND BIND VAR-FILE   TO FILE   *> The relevant file
AND BIND VAR-RECORD TO RECORD *> The relevant file record
AND BIND VAR-NAME   TO NAME   *> Alphanumeric file name
                    OF VAR-FILE.
```

We extract data from the *shadow* [12] of the join point, i.e., the static program context that belongs to the join point. (This is a form of static join-point reflection.) The first three selectors VERB, FILE and RECORD are applied to the *root* of the shadow. The fourth selector, NAME, is applied to VAR-FILE. Note that *AspectCobol*'s variables are scoped placeholders shared by PCD and advice. They are different from Cobol's 'storable' and global data items, which are explicitly declared in the working-storage section.

The advice code for logging is encoded in normal Cobol as follows:

```
MOVE VAR-VERB   TO LOGFILE-VERB.
MOVE VAR-NAME   TO LOGFILE-NAME.
MOVE VAR-RECORD TO LOGFILE-DATA.
CALL "TOOLS/LOGFILE" USING LOGFILE-ENTRY.
```

The advice delegates most of the actual work to the subprogram TOOLS/LOGFILE, which writes appropriate entries to the logging file. This subprogram is in no way special; see Fig. 1 for an implementation. The subprogram adds one line per call (i.e., per file access) to a sequential file. We have taken measures such that the logging file only needs to be opened once — when the logging subprogram is called for the first time. The bottom part of the figure also shows the parameter area that is filled in the advice code above.

2.2 Intra- vs. inter-program aspects

The scope of classic declaratives is restricted to the hosting program, thereby resembling *intra-program* aspects. *AspectCobol* must also deal with *inter-program* aspects because crosscutting concerns are unlikely to align with subprogram boundaries. Inter-program aspects affect some or all Cobol programs in a given project.

AspectCobol provides a new form of compilation unit for inter-program aspects, which complements the preexisting forms for programs and classes. Let's rephrase the earlier logging aspect. The inter-program version requires a compilation unit as follows:

```
IDENTIFICATION DIVISION.
 ASPECT-ID. ASPECTS/LOGFILE.
```

An aspectual compilation unit applies to all programs in a project. Of course, a program is *affected* only in case its execution exhibits relevant join points. The environment and data divisions of the aspectual unit extend the affected programs. By default, *AspectCobol* separates the name-spaces of aspectual unit and affected program. Thereby, we avoid *unintended name capture*.

In our running example, we note that the logging subprogram performs file access, i.e., it exhibits relevant join points by itself. To

```
IDENTIFICATION DIVISION.
 PROGRAM-ID. TOOLS/LOGFILE.
ENVIRONMENT DIVISION.
 INPUT-OUTPUT SECTION.
  FILE-CONTROL.
   SELECT LOGFILE ASSIGN TO "FILES/LOGFILE.TXT",
   ORGANIZATION IS SEQUENTIAL.
DATA DIVISION.
 FILE SECTION.
  FD LOGFILE DATA RECORD IS LOGFILE-RECORD.
  01 LOGFILE-RECORD PIC X(2048).
 WORKING-STORAGE SECTION.
  01 LOGFILE-STATUS     PIC 9 VALUE ZERO.
     88 LOGFILE-IS-OPEN       VALUE 1.
 LINKAGE SECTION.
  COPY "BOOKS/LOGFILE.DD".
PROCEDURE DIVISION USING LOGFILE-ENTRY.
 IF NOT LOGFILE-IS-OPEN
    OPEN EXTEND LOGFILE
    SET  LOGFILE-IS-OPEN TO TRUE.
 MOVE  LOGFILE-ENTRY TO LOGFILE-RECORD.
 WRITE LOGFILE-RECORD.
 GOBACK.
```

```
*> This is the contents of the copy book "BOOKS/LOGFILE.DD".
01   LOGFILE-ENTRY.
 05  LOGFILE-VERB  PIC X(12).
 05  LOGFILE-NAME  PIC X(32).
 05  LOGFILE-DATA  PIC X(1024).
```

Figure 1: A simplified subprogram for logging file access

avoid *non-terminating interception*, we must restrict logging such that the logging subprogram is removed from the pointcut. This is implemented in the following extension of the earlier PCD:

```
NAME OF PROGRAM IS NOT EQUAL "TOOLS/LOGFILE"
AND ... *> Continue with PCD as before.
```

That is, the join-point shadow must not reside in TOOLS/LOGFILE. Here we take advantage of the fact that each Cobol program defines a PROGRAM-ID in the identification division. Selecting this program name is again a trivial form of join-point reflection.

2.3 Dynamic join points

Let's consider a refined assignment: logging is to be restricted to the program MAINS/P088 and its subprograms — be it for the purpose of debugging. That is, we need to constrain the pointcut in terms of the dynamic calling relationships. In *AspectCobol*, this is expressed as follows:

```
CFLOW PROGRAM "MAINS/P088"
AND ... *> Continue with PCD as before.
```

The CFLOW PCD states that a join point of interest must be *in the control flow* of the program MAINS/P088. (It should be obvious that we simply adopt AspectJ's *dynamic* join points [17].)

2.4 Stateful aspects

Let's reconsider the initial logging program. This program maintains the potentially opened logging file complete with a flag that states whether the file was already opened or not. This example illustrates the basic idea of *stateful aspects*, i.e., *aspects maintaining data along subsequent executions of advice*. We will now illustrate the elimination of CFLOW by means of a stateful aspect. This is a folklore exercise, which we instantiate for Cobol — just to show how well (Aspect)Cobol can cope with stateful aspects.

We exclude the CFLOW predicate from the PCD. To simulate this predicate, we advise the program MAINS/P088 according to a helper aspect; see the upper part of Fig. 2. This trivial aspect makes

An aspect that patches the execution of the program "MAINS/P088".

```
IDENTIFICATION DIVISION.
 ASPECT-ID. ASPECTS/PATCH088.
DATA DIVISION.
 WORKING-STORAGE SECTION.
  COPY "BOOKS/LOGFILE.DD".
PROCEDURE DIVISION.
 DECLARATIVES.
  MY-PATCH-ASPECT SECTION.
   USE BEFORE EXECUTION OF PROGRAM "MAINS/P088"
   MY-PATCH-ADVICE.
    INITIALIZE LOGFILE-ENTRY.
    CALL "TOOLS/LOGFILE" USING LOGFILE-ENTRY.
 END DECLARATIVES.
```

A revised subprogram for logging which observes starting shots. (See Fig. 1 for the initial version.)

```
PROCEDURE DIVISION USING LOGFILE-ENTRY.
  *> Check whether file is to be opened.
  IF  NOT LOGFILE-IS-OPEN
  AND LOGFILE-ENTRY = SPACES
     OPEN LOGFILE
     SET  LOGFILE-IS-OPEN TO TRUE.
  *> Check whether entries are to be written.
  IF LOGFILE-IS-OPEN
  AND LOGFILE-ENTRY NOT = SPACES
    MOVE  LOGFILE-ENTRY TO LOGFILE-RECORD
    WRITE LOGFILE-RECORD.
  GOBACK.
```

Figure 2: Using state for control-flow tracking

sure that the execution of MAINS/P088 will be preceded by a special call to the logging subprogram. This call is special in that an *initialised* LOGFILE-ENTRY is used. All other calls use a *filled* entry. The logging subprogram can observe these two configurations so that log entries are not recorded — unless control flow went through MAINS/P088 previously. A corresponding revision of the logging subprogram is shown in the lower part of Fig. 2.

The resulting encoding mimics the behaviour of the earlier, CFLOW-based solution perfectly.[3] The encoding reminds us of the potential benefits of native language support for specific pointcut predicates. That is, native CFLOW can potentially avoid runtime join-point checks on the basis of clever static analyses [31]. Also, native CFLOW can potentially avoid noise during debugging, when remaining runtime join-point checks are hidden from the programmer who steps into a potential join point.

3. *AspectCobol* IN EXAMPLES: ERROR CHECKING & HANDLING

We will now look into *error checking*, by which we mean that certain error conditions are identified by aspectual code. We will also look into *error handling*, by which we mean that well-defined error conditions or (exceptions) are handled by aspects that readily intercept those conditions. In due course, we will illustrate further *AspectCobol* concepts and constructs. Incidentally, these examples also demonstrate *AspectCobol*'s strength in an area that is already covered by classic Cobol and OO Cobol.

3.1 Error handling in the past

Cobol readily distinguishes several forms of error. For instance, there are *size errors*, which can occur during the execution of arithmetic statements. (There is plethora of forms.) The syntax of these statements comprises two optional phrases for error-handling code: one is for the case that a size error occurs; another is for the case that *no* such size error occurs. (The latter is not really about error handling in a strict sense.) For instance:

```
ADD 1 TO MY-VERY-SMALL-NUMBER
  ON SIZE ERROR
    *> Panic or repair!
  NOT ON SIZE ERROR
    *> Jump for joy!
```

There are many other Cobol statements that can result in an error while providing similar handling phrases. That is, there are overflows, which can occur as a result of exhausted resources in CALL statements and elsewhere. There are also predefined error forms that are related to file I/O statements, e.g., errors concerned with invalid keys or the end of file.

The many scattered error handlers seem to correspond to a form of *tangling*. Error handling by means of Cobol's declaratives is more in the realm of separation of concerns. Classic Cobol allows us to handle some (but not all errors) by means of USE statements; recall the examples from the introduction. Cobol 2002 allows us to handle *all* possible errors by means of an *exception-handling* USE statement. Here it is assumed that the runtime system translates statement-level errors into OOish exceptions. (Cf. common exception processing [13, § E.14; p. 761].) Let's apply this technique to the SIZE ERROR example:

```
DECLARATIVES.
 HANDLE-ALL-SIZE-ERRORS-ASPECT SECTION.
  USE AFTER EXCEPTION CONDITION EC-SIZE.
 HANDLE-ALL-SIZE-ERRORS-ADVICE.
  *> Panic or repair!
END DECLARATIVES.
```

The name EC-SIZE is Cobol 2002's predefined exception name for size errors. While this is elided here, the handler code could also refer to predefined data names EXCEPTION-STATEMENT, -LOCATION and -STATUS. One might say that these data names serve a form of join-point reflection.

What could we possibly wish for given this beauty of Cobol 2002?

Inter-program scope — declaratives, as of Cobol 2002, affect only the program in which it resides. One might want to specify error-handling concerns that affect some or all programs. (For the record, GLOBAL declaratives of a program will also be considered for all contained programs [13, p. 551].) The inter-program aspects of *AspectCobol* come to our rescue here.

Join-point identification — The exception-handling USE statement can be used to replace ON ... ERROR phrases, but not the *negated* phrases. Also, we cannot handle offending situations before they manifest themselves as exceptions. Furthermore, quantification is limited to exception names as opposed to versatile PCDs.

Join-point reflection — The data names, that can be consulted by handlers, are fixed, and thereby potentially incomplete. Since pointcuts denote *sets of join points* it is imperative that handlers can determine the specifics of each actual join point.

Join-point control — We cannot refer to the offending statement if we want to re-execute it within the handler section, which hampers error repair. (Clearly, in AOP terms, we need *around* advice!)

We will now illustrate *AspectCobol*'s strengths in these areas.

[3] In general, we would also need an AFTER advice to switch off logging at the end of MAINS/P088. However, this is not necessary here if we assume that MAINS/P088 is the top-level main program.

```
IDENTIFICATION DIVISION.
 ASPECT-ID. ASPECTS/SIZEERROR.
DATA DIVISION.
 WORKING-STORAGE SECTION.
  COPY "BOOKS/PANIC.DD".
PROCEDURE DIVISION.
 DECLARATIVES.
   USE AFTER THROWS SIZE ERROR
   AND EXISTS PROCEDURE PANIC-STOP
   AND EXISTS DATA PANIC-FIELD
   AND BIND VAR-LOC TO LOCATION.
  MY-SIZE-ERROR-ADVICE.
   INITIALIZE PANIC-FIELD.
   MOVE "SIZE ERROR" TO PANIC-CATEGORY.
   MOVE VAR-LOC      TO PANIC-LOCATION.
   GO TO PANIC-STOP.
END DECLARATIVES.
```

```
*> This is the contents of the copy book "BOOKS/PANIC.DD".
01  PANIC-FIELD.
 05 PANIC-RESOURCE PIC X(16). *> Optional field
 05 PANIC-CATEGORY PIC X(16). *> Required field
 05 PANIC-LOCATION PIC X(32). *> Implementor-defined
```

Figure 3: An aspect for handling SIZE ERRORs

3.2 Intended name capture

AspectCobol's pointcut language is *complete* with regard to classic error phrases. For instance, the existence of the optional (NOT) ON SIZE ERROR phrase, which is offered by several statements, implies a pointcut predicate (NOT) THROWS SIZE ERROR. Thereby, we can intercept all (non-) size errors without any reference to OOish exceptions. More importantly, we can place the handler code in an *inter-program* aspect; see Fig. 3.

This aspect illustrates that a pointcut can express *intended name capture* (cf. EXISTS), i.e., aspectual unit and affected programs can share procedure names or data declarations. Let's consider the two EXISTS conditions from the figure. The condition EXISTS PROCEDURE `PANIC-STOP` establishes that an affected program provides the procedure `PANIC-STOP`. The condition EXISTS DATA `PANIC-FIELD` establishes that both, aspectual unit and affected program, declare a level 01 field of that name. (We show the intended declaration at the bottom of the figure.)

The declarations of captured data within an aspectual unit serve for early and independent type checking of advice code. That is, the declarations do not contribute to affected programs. To this end, the declarations in the aspectual unit must be the same as those that already reside in the affected programs.[4]

In Fig. 3, we also use a new selector, LOCATION, which extracts location information for the offending statement that throws the exception. (This form of join-point reflection is also adopted from AspectJ [17]: `thisJoinPoint.getSourceLocation()`.) Locations comprise line number and character position in the source file. Locations as such, are already part of the Cobol standard, they are marked as implementor-defined though [13, § E.14; p. 761].

3.3 Advanced quantification and navigation

We will now devise an aspect that determines an error condition — even though Cobol's runtime system does not report any error whatsoever. We face the following assignment:

> *Any read access to a file's record (not to be confused with access to the file itself) is to be guarded by a test for the FILE-STATUS field to be equal to ZERO (meaning no unhandled error occurred previously).*

[4] One could suggest that Cobol's EXTERNAL clause [13, § 13.16.20; p. 283] has a role to play here since it allows us to share data among run units. However, all affected programs would then also need to use the EXTERNAL clause, which is in strong conflict with the obliviousness property of AOP [9].

```
IDENTIFICATION DIVISION.
 ASPECT-ID. ASPECTS/UNSAFEREAD.
DATA DIVISION.
 WORKING-STORAGE SECTION.
  COPY "BOOKS/PANIC.DD".
PROCEDURE DIVISION.
 DECLARATIVES.
   USE BEFORE ANY STATEMENT
   AND BIND VAR-ITEM   TO SENDER
   AND      VAR-ITEM   IS FILE-DATA
   AND BIND VAR-FILE   TO FILE OF VAR-ITEM
   AND BIND VAR-STATUS TO FILE-STATUS OF VAR-FILE
   AND BIND VAR-NAME   TO NAME OF VAR-FILE
   AND BIND VAR-LOC    TO LOCATION
   AND EXISTS PROCEDURE PANIC-STOP
   AND EXISTS DATA PANIC-FIELD.
  MY-UNSAFEREAD-ADVICE.
   IF VAR-STATUS NOT = ZERO
    INITIALIZE PANIC-FIELD
    MOVE  VAR-NAME      TO PANIC-RESOURCE
    MOVE  "UNSAFE READ" TO PANIC-CATEGORY
    MOVE  VAR-LOC       TO PANIC-LOCATION
    MOVE  VAR-STATUS    TO PANIC-CODE
    GO TO PANIC-STOP.
 END DECLARATIVES.
```

Figure 4: An aspect for handling unsafe access to file records

This scenario requires from us to shadow *sending data items* — using Cobol terminology; cf. [13, § 14.5.7; p. 389]. (For instance, x is a sending data item in MOVE x TO y while y is a *receiving* one.) We shadow all statements and bind their sending data items:

```
USE BEFORE ANY STATEMENT
AND BIND VAR-ITEM TO SENDER
*> PCD to be continued.
```

The pointcut must be restricted to data items whose declaration is hosted in the file section. To this end, we use a new condition form (akin to Cobol's class conditions [13, § 8.8.4.1.3; p. 131]):

```
*> Continued from above.
AND VAR-ITEM IS FILE-DATA
```

It remains to select the relevant file-status field, which we want to test in the advice. This selection starts from the sending data item, and walks through the involved file:

```
*> Continued from above.
AND BIND VAR-FILE   TO FILE OF VAR-ITEM
AND BIND VAR-STATUS TO FILE-STATUS OF VAR-FILE
```

The complete aspect is shown in Fig. 4.

3.4 AROUND advice + PROCEED statement

Let's consider an example of error handling that involves repair. We would like to intercept file I/O statements, while making sure that the accessed files are auto-opened, if necessary. To this end, we need an aspect whose advice executes the intercepted file I/O statement under its own control.

We need the following USE statement:

```
USE AROUND
(READ OR REWRITE OR WRITE OR DELETE OR START)
AND BIND VAR-STATUS TO FILE STATUS.
```

Within AROUND advice, one can execute the intercepted join point once, twice, ... (or not at all) by using the special statement PROCEED. (Again, we have adopted this concept and the overall BEFORE, AFTER and AROUND dichotomy from AspectJ [17].) Here is the code sequence for error checking and repair:

```
PROCEED.
IF VAR-STATUS = 42
  OPEN I-O VAR-FILE
  PROCEED.
```

The first PROCEED corresponds to the normal execution of the intercepted file I/O statement. The subsequent IF statement checks for the error condition: "file not opened" (i.e., 42).[5] We are done in case there is no "file not opened" error. Otherwise, the file is opened, and the intercepted statement is tried again via a second occurrence of PROCEED.

4. CROSSCUTTING CONCERNS IN THE COBOL WORLD

Working through AOSD resources, we concluded that a somewhat complete list of typical crosscutting concerns, as of today, is likely to look as follows:

> *logging, tracing, context-sensitive error handling, coordination of threads, remote access strategies, execution metrics, performance optimisation, persistence, authentication, access control, data encryption, transaction management, pre-/post-condition & invariant checking, enforcement of policies for resource access & API usage, implementation of design patterns, test-coverage analysis.*

This list should be revisited with a Cobol hat on. We will deal with three questions. (i) What known crosscutting concerns are definitely meaningful from a Cobol perspective, and what are the Cobol-biased instances of these concerns? (ii) What otherwise meaningful crosscutting concerns are questionable in the Cobol context, and why is that? (iii) Can we enhance this list with crosscutting concerns that suggest themselves in the Cobol context?

4.1 *Confirmed!* Logging and tracing

(What's the difference between logging and tracing anyway? Cf. [4].) Logging certain file operations or subprogram and procedure executions is implemented in Cobol code on a regular basis — with varying degrees of tangling. (Less tangling is present when the operation of interest and the corresponding logging code can be isolated in reusable procedures or subprograms.)

We refer to *program understanding* as a software engineering method that can benefit from AOP support. Program understanding helps with maintaining and migrating Cobol applications. In this context, logging and tracing can be used in the following ways:

- To study resource access policies, or API usage policies.
- To mine business rules [24, 25], or even aspects.
- To gather runtime data about unused (potentially dead) code.

4.2 *Less of an issue!* Synchronisation

We use the term synchronisation in the sense of coordination of threads or concurrency control for the purpose of ensuring the integrity of data (e.g., object state) that is accessible simultaneously by several threads. This is a classic crosscutting concern, which has been spotted early in the history of AOP [23]. Synchronisation is (much) less of an issue in the Cobol context. The Cobol standard does not define expressiveness for multi-threading. Some vendors of Cobol compilers provide support for multi-treading (or multi-tasking, asynchronous programming), but such extensions are not widely used (in business programming with Cobol).[6] By contrast, Cobol applications typically employ a client-server architecture or transaction processors, such as CICS, to enable a multi-user environment with reentrant programs. Transaction processors implement multi-tasking, multi-threading, and integrity of files and other data. Nevertheless, the addition of language support for multi-threading is being considered by the standardisation body for Cobol. So synchronisation could become an issue for Cobol programs if we assume that multi-threading will see widespread use in new or modernised Cobol-based software components.

4.3 *A non-issue!* Persistence

Achieving persistence in an object-oriented application, i.e., managing object state in a database or some other form, is a complex problem, which can benefit from an aspectual approach; see [34, 27]. In the Cobol world, applications tends to be so data-driven that data access is omnipresent in Cobol code — deliberately. These are the typical options for data access in Cobol:

- File I/O for sequential files and keyed files.
- File I/O through a transaction management software.
- Embedded SQL for tables in a relational database.
- Several such options for IMS (a hierarchical database).

Making a Cobol application persistent any further does not seem to be necessary. Trying to modularise Cobol applications with regard to the data-model/data-access layer could be very difficult, and a clear incentive is missing.

4.4 *Risky!* Aspectual middleware

Concerns like synchronisation and persistence suggest a more general topic: middleware concerns. In the Java world, the following tension has been observed. Middleware technologies provide *infrastructural support* for concerns like authentication, secure communication, secure data access, queueing, distribution, transaction processing and persistence, while AOP can provide *language support* for expressing these and other concerns. Codename: aspectual middleware. Once we commit to an application server (perhaps one based on EJB — Enterprise Java Beans), AOP becomes less relevant [18] (and vice versa). The EJB approach, in particular, has been criticised for its complexity and inflexibility. Consequently, it has been argued vehemently [7, 5, 6] that AOP would be able to complement or even replace state-of-the-art application servers.

In the Cobol world, application servers are used heavily for decades already, while some other terms are in use: transaction management system and transaction processing monitor. (Cf. IMS/TM, CICS or Tuxedo.) Commitment to such systems is ingrained in the Cobol software. For instance, the pseudo-conversational style of CICS-type transaction processing has a deep impact on the architecture of the software systems. (This is a form of *software asbestos* [19].) The incentive for a turn from application servers to aspectual middleware might be missing in the Cobol context. The classic architectures readily serve well-defined qualities of service (QoS) such as throughput, workload balancing and recoverability. Also, CICS and friends catch up. (CICS meanwhile supports all sorts of communication protocols; one can deploy EJBs in a CICS installation and vice versa. Hence, additional QoS are obtainable and interoperability is improved.)

(Referring back to the code bases from the introduction, we spotted the use of transactions processors in 12 out of 21 systems; we sensed several idioms hinting at transaction processing for another 4 systems, but we failed to recognise familiar technology.)

[5] Incidentally, 42 is also the answer to the Ultimate Question of Life, the Universe, and Everything: http://en.wikipedia.org/wiki/The_Answer_to_Life,_the_Universe,_and_Everything

[6] A related thread: http://www.talkaboutprogramming.com/group/comp.lang.cobol/messages/130350.html (comp.lang.cobol)

For a turn, we would need guarantees that the traditional QoS are preserved when migrating from infrastructure to aspects. In fact, we would need promises for new QoS that are difficult to achieve with the classic infrastructure. (Which are those QoS?) Finally, we would require an automated approach for the migration from infrastructural middleware to aspectual middleware. Manual migrations are impractical for the typical Cobol code base.

4.5 *How to?* Design by contract

AOP can be used for the modular implementation of DBC — 'design by contract'. (See [16, 33], but there are various white papers that use DBC as a typical or even strong AOP example by now.) Weaving activates condition checking for pre- and post-conditions as well as class invariants. By omitting the deployment of a DBC aspect (or by unweaving), the performance penalty associated with condition checking is avoided.

Design by contract has never been instantiated for Cobol, and such an instantiation will be challenging indeed.[7] How to define pre- and post-conditions for procedures, which have no explicit parameters, which instead operate on the global data division? What to say about files in the contracts? How to model user input validation in contracts? What's the added value of post-conditions when compared to normal Cobol computations and MOVEs? That is, why would we want to encode business rules twice?

4.6 *Promising!* Policy enforcement

While we do not have aspectual DBC for Cobol, the more general theme of software robustness can take advantage of *AspectCobol*. That is, we can devise an aspectual approach to error checking and handling (recall Sec. 3) including policy enforcement. Such checking is a particularly valuable tool for 'shielding' re-engineering activities that involve invasive changes in the sense of transformations for conversion, wrapping or migration.

Let's consider an example for policy enforcement. In Sec. 3, we studied an aspect that caught unsafe read access to file records. This aspect exhibits several shortcomings. Reading from the file record without a prior READ statement appears to be inappropriate, but the aspect will not spot this problem. ("A false negative") Also, file-status fields can be shared among different files, in which case the aspect's insistence on a zero status prior to record access might be unnecessarily restrictive. ("A false positive") What's needed is a watertight file-access policy, where each file I/O statement must be executed in accordance to restrictions on the history of file access and record access in the given program.

The aspect for the general problem is worth a paper on its own. In Fig. 5, we cover part of the problem: we only care about the status of files to be open or closed. The approach scales for more complex state spaces. The aspect in the figure consists of three concerns — OPEN statements, CLOSE statements and other file I/O statements are treated differently. The open concern establishes that the file is not yet open, and marks it as open. The close concern establishes that the file is still open, and marks it as closed. The concern for other statements insists on the file being marked as open.

In Fig. 5, we also use a new selector, IDREF, which delivers a unique reference to the given program entity. In the specific example, we need such unique references to distinguish the various files in the program at hand. We maintain these IDREFs and the file-open status *per file* in a dynamic table.[8]

[7] Cobol 2002 provides the *validate facility* [13, § E.19; p. 792], which allows one to describe validation rules for data fields. This is perhaps a limited form of DBC.

[8] Dynamic tables [26] will be in the Cobol 2008 standard. We could also use Cobol's new collection library instead of dynamic tables [14].

```
IDENTIFICATION DIVISION.
 ASPECT-ID. ASPECTS/CHECKOPEN.

DATA DIVISION.
 WORKING-STORAGE SECTION.
  01  WORK-IDREF      PIC 9(10).
  01  CHECKOPEN-IDX   PIC 999.
  01  CHECKOPEN-MAX   PIC 999 VALUE ZERO.
  01  CHECKOPEN-ENTRY OCCURS ANY TIMES
                      INDEXED BY CHECKOPEN-IDX.
   05 CHECKOPEN-IDREF PIC 9(10).
   05 CHECKOPEN-STATE PIC 9.
      88 CHECKOPEN-CLOSED VALUE 0.
      88 CHECKOPEN-OPEN   VALUE 1.

PROCEDURE DIVISION.
 DECLARATIVES.

  MY-OPEN-CONCERN.
    USE BEFORE OPEN
    AND BIND VAR-IDREF TO IDREF OF FILE.
   MY-OPEN-ADVICE.
    MOVE VAR-IDREF TO WORK-IDREF.
    PERFORM GET-CHECKOPEN-IDX.
    IF CHECKOPEN-OPEN ( VAR-IDREF )
      PERFORM CHECKOPEN-PANICS.
    SET CHECKOPEN-OPEN ( VAR-IDREF ) TO TRUE.

  MY-CLOSE-CONCERN.
    USE BEFORE CLOSE
    BIND VAR-IDREF TO IDREF OF FILE.
   MY-CLOSE-ADVICE.
    MOVE VAR-IDREF TO WORK-IDREF.
    PERFORM GET-CHECKOPEN-IDX.
    IF CHECKOPEN-CLOSED ( MY-IDREF )
      PERFORM CHECKOPEN-PANICS.
    SET CHECKOPEN-CLOSED ( MY-IDREF ) TO TRUE.

  MY-ACCESS-CONCERN.
    USE BEFORE
    (READ OR REWRITE OR WRITE OR DELETE OR START)
    AND BIND VAR-IDREF TO IDREF OF FILE.
   MY-ACCESS-ADVICE.
    MOVE VAR-IDREF TO WORK-IDREF.
    PERFORM GET-CHECKOPEN-IDX.
    IF CHECKOPEN-CLOSED ( MY-IDREF )
      PERFORM CHECKOPEN-PANICS.

 END DECLARATIVES.

 GET-CHECKOPEN-IDX.
  *> Find or allocate entry for VAR-IDREF in dynamic table.

 CHECKOPEN-PANICS.
  *> Print error message and stop execution.
```

Figure 5: Policy checking for the status of files to be open

4.7 *Promising!* Automated regression testing

Perhaps a vital application of *AspectCobol* is automated regression testing. In the Cobol world, we cannot assume good harnesses for regression testing of systems, if we adopt XP standards. In the absence of test harnesses, Cobol applications resist change. It is presumably very expensive to gather such harnesses.

AOP can be employed for automated regression testing as follows. *Recording aspects* are woven to record different kinds of *observable results* along the execution of possibly interactive programs: file access, database access, terminal I/O, subprogram calls, and others. *Replay aspects* are woven to execute programs against recorded results — as a replacement of actual access to data and terminal resources. Recording can be done by the system owner, which will provide meaningful test cases for the system developer. The recording approach implies that each test case is self-contained and reproducible; it is free of user interactions and whatever side effects. (We do not need to store the full file system/database with *each* test case. We can capture test cases in a multi-user and date-

dependent environment.) Most importantly, the captured results (and thereby the test cases) *remain applicable for a changed program or a changed installation*, within limits. For instance, a system adaptation for data expansion [19] must not break any test case. A system adaptation that affects the data-management tier will invalidate some recorded data-access results, but the interaction results should be reproducible. A system adaptation that amends a business rule should not break *unrelated* test cases.

This application of AOP is a promising direction for future work, with potential applications that go beyond the Cobol context.

4.8 *Frontier research.* AOP in context

The large volume of business-critical Cobol code and the longevity of deployed Cobol systems has triggered long-standing, major efforts on software engineering techniques that are different from but related to AOP. Most notably, automated re-engineering transformations are an established means to renovate, migrate, convert and integrate Cobol code [3, 35, 19]. Code transformations readily allow us to implement AOP for Cobol and other languages used in legacy software, as demonstrated in [11]. While it is generally known that code transformation is suitable for many forms of AOP, it is perhaps the only viable approach for legacy software, which is coded in languages that lack compiler support for AOP.

There is an intriguing and poorly understood borderline between more general re-engineering problems and definite AOP problems. Consider web enabling, where a program is transformed and integrated such that dumb terminal I/O is replaced by CGI-based HTML pages, Active Server Pages or others. This scenario can be phrased in AOP terms: we intercept events for terminal I/O and we map them to a web-based user interface. (Think of screen scraping.) The trouble is that a normal AOP language extension will not serve the full scenario. We will need advanced concepts as follows:

- If we want to automate the mapping of legacy forms to HTML content, then we need detailed access to the legacy forms, which however are not necessarily described in native Cobol.
- We might consider non-trivial and invasive restructuring transformations of the program's dialogue model, prior to its comparatively trivial and direct mapping to web content.
- The event-handling scheme of the program might be inappropriate for website generation and field validation, which again suggests non-trivial transformations for preparation.

The tension between re-engineering transformations and AOP is also evident in other contexts. Consider migration of data management from files to a relational database. We could use AOP if there was a direct mapping from files to tables. However, we are likely to face transformations on the data model. Then, the mapping requires invasive changes of program code.

Resolving this tension could lead to a more vital AOP notion, and to a more powerful classification of re-engineering transformations.

5. *AspectCobol*: LANGUAGE DESIGN ISSUES

Let's discuss highlights of *AspectCobol*'s language design.

5.1 Join points

In AspectJ, the most common join point is a method call or a method execution. By contrast, in *AspectCobol*, the typical join point is the execution of a specific statement, a (sub-) program or a procedure, which is summarised in the following EBNF for PCDs:

pcd	→	("**EXECUTION**" "**OF**")? *interceptable*
interceptable	→	(*verb* | "**ANY**") "**STATEMENT**"?
interceptable	→	"**PROGRAM**" *quoted-literal*?
interceptable	→	"**PROCEDURE**" *procedure-name*?

The nonterminal *verb* is a placeholder for Cobol's many statement verbs ACCEPT, ADD, ALTER, ..., WRITE. Aspects for screen I/O deal with the verbs ACCEPT and DISPLAY. Aspects for file I/O deal with all the file I/O statements. Aspects for test coverage analysis deal with the verbs IF and EVALUATE. The EBNF states that the choice of a specific verb can be omitted (cf. ANY); we can also quantify over all programs and all procedures. We note that the syntax, given so far, does not yet accommodate constraints on the on the operands involved in the statements, neither can we access the parameters of intercepted subprogram executions. We will discuss such expressiveness shortly.

By definition, we intercept the execution of a procedure using the PROCEDURE syntax. More subtly, using the STATEMENT syntax, we can intercept (the execution of) the statement that performs a procedure (cf. the PERFORM verb) or jumps to it (cf. the GO verb). Likewise, we can intercept both (the execution of) a CALL statement and the execution of a subprogram. (In AspectJ terms, we have both *execution* and *call* join points.) We also note that we can intercept procedures that are encountered 'by fall-through' rather than by PERFORM and GO — using the PROCEDURE syntax.

It is worth defending *AspectCobol*'s support for interception of procedure execution. We note that procedure names are built from paragraph and section names whose scope is limited to the hosting program. Hence, advising such private definitions by means of an inter-program aspects could be considered harmful. However, Cobol coding standards tend to deal with the procedure concept as if procedure names had a system-wide scope. For instance, it is common that the programs in a code base implement paragraphs of distinguished names. It is also common that distinguished paragraphs are imported from copy books. We recall the use of 'panic' procedures in earlier examples, but there are many such idioms for file handling, screen I/O processing, report generation, and others. Code owners know and care about these idioms and corresponding mnemonics. Accordingly, system-wide interception of procedure execution must be allowed by *AspectCobol*.

5.2 Deliberate or innocent omissions

Some omissions are motivated by our focus on classic Cobol since there is not much OO Cobol code around anyhow. We could readily add an alternative to *interceptable* so that we can also intercept the execution of a *method*. Likewise, we currently neglect AspectJ's inter-type declarations for classes in OO Cobol. (Admittedly, the composition of a classic Cobol program and an inter-program aspect is akin to inter-type declarations.) Incidentally, we *can* intercept method calls using the STATEMENT syntax: OO Cobol adopts the INVOKE verb for method calls.

There are syntactic categories that are closely related to those covered by *interceptable*. We draw the following line. We do not include sentences into *interceptable* because we have not identified any usage scenario for advising sentences. We do not extend *interceptable* as far as categories of statement operands are concerned because Cobol is a statement-oriented language, and hence advice cannot be attached to operands anyhow. We do not cater for interception of Cobol's various compiler-directing statements because these statements do not directly participate in program execution.

5.3 Selector-based join-point access

AspectJ provides excellent notational support for method-calling patterns, which resemble the syntactical structure of method calls, and which take advantage of the fact that method arguments are *typed* by means of Java's method signatures, and *named* by means of the method headers. In the case of Cobol, we face a much richer syntax, weaker typing, non-flat argument lists in subprogram calls, anonymous operands in statements and other complications. We have found that *selector-based access to program contexts* is more appropriate for Cobol than a plethora of patterns.

Selection starts from the join-point shadow [12], which must be a statement, a procedure or a program due to the grammar in Sec. 5.1. Selection can be nested as in *selector* OF … *selector* OF SHADOW. The final "OF SHADOW" can be omitted. Selectors can also be applied to the bound result of an earlier selection. A selector's *applicability* depends, of course, on the category at hand. For instance, an attempt to extract a name is only valid when we face a *named* entity. Here are *AspectCobol*'s selectors:

- LOCATION – The selected location in the source-code file.
- IDREF – An opaque identity for referring to the selected entity.
- NAME – The alphanumeric literal for the name of the selected entity.
- VERB – The alphanumeric literal for the statement verb at hand.
- TYPE – The type of a selected data item (a picture string).
- PROCEDURE – Navigation to the hosting procedure.
- PROGRAM – Navigation to the hosting program, i.e., to the root.
- LEVEL 01 – Navigation to the hosting top-level data entry.
- FILE – Selection of the file in a file I/O statement.
- RECORD – Selection of the record in a file I/O statement.
- FILE-STATUS – Selection of the file-status field for the selected file.
- SENDER – Selection of a sending data item.
- RECEIVER – Selection of a receiving item.
- PARAMETER – Selection of a parameter for a subprogram or a call.

The list clarifies that there are selectors for the *extraction of basic properties*, such as names or references, and there other selectors, which cater for the *navigation from one program context to another*. We note that some selectors can be ambiguous. For instance, a statement can involve several sending items. We will handle such ambiguous selection shortly.

The list of selectors, as given above, is incomplete. Additional selectors can be derived systematically from the Cobol grammar while limiting the amount of details to be exposed to the *AspectCobol* programmer. We also need a few condition forms that perform *classification tests* on a given syntactical entity, e.g.:

- IS FILE-DATA x – x is a data item declared in the file section.
- IS WORKING-STORAGE-DATA x – dito.
- IS LINKAGE-DATA x – dito.
- IS SENDER x – x is a sending item.
- IS RECEIVER x – dito.

AspectCobol's design strongly suggests that join-point reflection on the join-point shadow should be viewed *as part of the pointcut* — as opposed to using reflection in the advice code.

5.4 Context capture

In AspectJ, we explicitly list the context that can be assumed by the advice code. This resembles a method signature. There are special pointcut designators such as `this`, `target` or `args` that can *capture context*. For instance, the following AspectJ aspect intercepts all invocations of `set` methods, while it uses the PCD `target` to capture the callee for use in the advice:

```
after (Object callee):
 target(callee) && call(* set* (..)) {
  // print class of callee using reflection
  System.err.println(callee.getClass().getName());
 }
```

AspectCobol has been inspired by logic meta-programming and corresponding AOP approaches [8]. In particular, this concerns the use of variables as placeholders complete with a general form of binding. Here is a representative *AspectCobol* example, which mimics the earlier AspectJ code in a non-OO manner:

```
USE AFTER PERFORM STATEMENT
    *> Test procedure name by regular expression matching.
   AND NAME OF PROCEDURE IS LIKE "SET-.*"
   AND BIND VAR-NAME TO NAME OF PROGRAM.
MY-ADVICE.
 DISPLAY VAR-NAME.
```

That is, a variable's purpose to capture context of a certain type is only revealed by its occurrence in the binding position of a BIND phrase.[9] One might suggest to make explicit the captured context in terms of Cobol's syntax for linkage section and the USING clause — just as for subprograms in classic Cobol or methods in OO Cobol. For one thing, this approach would be verbose. More importantly, some of *AspectCobol*'s selectable categories are not first-class citizens in Cobol. For instance, we cannot parameterise in logical file names. By contrast, we can have a placeholder for a logical file name in *AspectCobol* without any extension of Cobol because the placeholder is resolved at weaving time anyhow.

5.5 Multi-valued selection

Some selectors are ambiguous; cf. SENDER or PARAMETER. They should be thought of as returning a *set of possible results*. We can readily handle them by assuming an appropriate semantics for pointcuts. Let's consider an example. Here is an aspect that counts certain sending operands in ADD and SUBTRACT statements:

```
USE BEFORE (ADD OR SUBTRACT)
 AND BIND VAR-ITEM TO SENDER
 *> Disregard the item if it is a receiving data item, too.
 AND NOT IS RECEIVER VAR-ITEM
MY-SENDER-ADVICE.
 *> Count the sending data item if it equals zero.
 IF VAR-ITEM = ZERO
  ADD 1 TO COUNT-ZERO-ITEMS.
```

(That is, this aspect detects superfluous additions and subtractions.) *AspectCobol* assumes a 'for all' semantics for the pointcut. That is, advice is issued for *all successful PCD evaluations*, where ambiguous selections give rise to multiple solutions for the capturing BIND phrases. Consider the following ADD statement:

```
ADD A B TO C.
```

The advised execution of this statement will be preceded by two executions of MY-SENDER-ADVICE because the data items A and B are both admitted by the pointcut.

One might argue that multi-valued selections can be avoided once the join-point model of *AspectCobol* provides corresponding categories of join point shadows. (In the example, we would then pointcut on data items rather than on statements.) We disfavour this option for one major reason. That is, Cobol is a statement-oriented language. Consequently, trying to pointcut on operands of statements causes mind-boggling irregularities. For instance, how to handle AROUND advice for access to data items?

[9] Rules for bindings: conjunctions are evaluated from left to right. Thereby, bindings can be used in subsequent subexpressions of the PCD. For simplicity, we assume that the bindings for the advice are listed in the top-level conjunction of the PCD. We also assume that bindings do not escape from operands of a disjunction.

6. *AspectCobol*: IMPLEMENTATION ISSUES

We will now discuss issues that relate to the implementation of *AspectCobol*. We will also briefly sketch a (so far incomplete) prototype implementation, which is based on a more general, experimental infrastructure for transforming, analysing and generating Cobol programs. Codename: cobble.[10,11]

6.1 Up-front considerations

According to [11], we face two generic obstacles to implementing AOP support for a legacy language. Firstly, we need to get a handle on a front-end for the relevant legacy language. This is indeed a major challenge in the case of Cobol, as we have argued elsewhere [22]. Secondly, we need to get a handle on a suitable weaving framework. At the very least, we require a basic transformation framework, which allows us to express the weaving semantics (by which we mean the elimination of AOP constructs in terms of transformations). Ideally, the weaving framework would even cater for some amount of reuse among different languages or dialects — an ambition that we will not address in this paper.

Our cobble-based implementation of *AspectCobol* is a stand-alone weaver that operates at the level of source code and uses XML for the internal representation of Cobol and *AspectCobol* code.

What about alternatives? Weaving compiled code would imply commitment to a specific vendor including dialect and object format. Language-independent load-time weavers [20] (for .NET or otherwise) would be challenged by the distance between bytecode and *AspectCobol*'s pointcut descriptions. Language-independent weavers at the source-code level, such as SourceWeave.NET [15], would require a specific Cobol front-end that appeals to the underlying source-code model (i.e., CodeDOM for SourceWeave.NET). The source-code model is likely to require an enhancement, too, for coverage of Cobol.

The use of XML for the intermediate representation of source code is not uncommon; it is practised for 'normal' languages (such as Java [2] and C [36]) and also for languages with a weaving semantics [10, 30]. The important advantage of this approach is that standard APIs and tools for XML processing can be readily used. There are known scalability problems, which require extra effort for compact XML representations or the use of tool-to-tool XML APIs without intermediate textual XML content [32]. In our prototype, we currently neglect these issues. We can report that the ratio *concrete Cobol syntax to XML format* (both in text representation) is approximately 10, which is still quite tractable.

6.2 The basic weaving semantics

AspectCobol appears to be an AspectJ-like language, but an adoption of AspectJ's weaving semantics [12] raises specific issues related to Cobol idiosyncrasies and *AspectCobol*'s pointcut language:

Let's consider the following intra-program aspect:

```
*> Print file name before file is read.
DECLARATIVES.
   USE BEFORE READ
   AND BIND VAR-NAME TO NAME OF FILE.
 FILE-ADVICE.
   DISPLAY "Reading " VAR-NAME ".".
END DECLARATIVES.
```

We are about to weave advice for this statement:

```
READ ORDER-FILE.
```

[10] Etymology: http://dictionary.reference.com/search?q=cobble

[11] Download: http://allserv.ugent.be/~kdschutt/cobble/

The most basic weaving scheme commences as follows. The advice is *cloned* per join-point shadow, and placed within a newly created section. The cloned version is also subjected to substitution such that all variables (cf. VAR-NAME) are replaced by the values that were obtained by pointcut evaluation. Weaving advice for the sample READ statement, we obtain the following clone:

```
AOP42-FILE-ADVICE SECTION.
FILE-ADVICE.
  DISPLAY "Reading " "ORDER-FILE" ".".
```

It remains to extend the actual join-point shadow by a PERFORM statement such that the cloned paragraph is executed:

```
PERFORM AOP42-FILE-ADVICE *> No period. Guess why!
READ ORDER-FILE.
```

The same steps apply to AFTER advice. When it comes to dynamic join points, advice execution is conditional as usual. When it comes to AROUND advice, effort is needed to handle the PROCEED statement. First the original join-point shadow is moved to a newly created section. Then, when cloning the body of the AROUND advice, all PROCEED statements are replaced by a PERFORM statement that executes the moved shadow. The original paragraph for the shadow is re-implemented such that the cloned advice is invoked.

Note that we have to prevent accidental fall-through of the control flow from the original sections into the added sections, which are therefore placed at the end of the procedure division after a special section that is also generated by the weaver:

```
DO-NOT-CROSS SECTION.
DO-NOT-CROSS-PARAGRAPH.
  GOBACK.
```

Let's also consider the problem of weaving an inter-program aspect. We can easily reduce this problem to the intra-program case by preparing each single program as follows. The program is composed with the environment and data divisions of the aspectual unit. This composition can be reverted if we fail to identify any relevant join-point shadow for the program. This composition is subject to alpha conversion so that unintended name capture is avoided.

The weaving semantics, discussed so far, exhibits one scalability problem. Consider an aspect whose pointcut applies to many different shadows and whose advice code is of substantial size. In this (not too unrealistic) case, the pervasive cloning approach, as described above, will imply code explosion. If we want to reuse the advice code, as is, we can try to declare data items for the variables so that parameters are passed through global data items to the advice code. In the example, we would need this data declaration:

```
01 VAR-NAME PIC X(31).
```

This time, the join-point shadow is transformed as follows:

```
MOVE "ORDER-FILE" TO VAR-NAME *> This line is new!
PERFORM FILE-ADVICE
READ ORDER-FILE.
```

This technique does not readily generalise for bindings that cannot be stored in Cobol data items, e.g., symbolic file names, even though the amount of cloning can still be reduced by reusing subparagraphs that do not refer to such problematic bindings. The technique is also challenged by AROUND advice because we would need to parameterise in procedure names. In fact, Cobol offers some related idioms. We might want to rely on vendor extensions for data items with USAGE PROCEDURE-POINTER. We could

```
display :  "DISPLAY"
    (what-to-display where-to-display? display-option*)+
    ("ON"? "EXCEPTION" statement+)?
    ("NOT" "ON"? "EXCEPTION" statement+)?
    "END-DISPLAY"?;

what-to-display : identifier | literal;
```

Figure 6: Grammar fragment for the DISPLAY statement

also use nested subprograms instead of procedures for join-point shadows, while the alphanumeric names of such subprograms can readily be stored.[12] Even simpler, the weaver could just assign literal codes to all paragraphs, and then use a single 'monster switch' to map literal codes to actual PERFORM statements.

6.3 Semi-automatic front-end development

We will now address the first obstacle for AOP support [11]: the front-end. The development of cobble's Cobol front-end started from a consolidated VS Cobol II grammar[13], which was recovered from IBM's language reference in a previous project [22]. Using the grammarware toolkits GRK[14] and GDK[15], we derived deployable grammars for two other Cobol dialects, Fujitsu Siemens Cobol 2000 and AcuCobolGT. The grammars were also extended to offer concrete syntax for all AOP language elements (as opposed to AOP technologies that use XML for join-point description, such as JBoss AOP and AspectWerkz). GRK and GDK interoperate via an EBNF-like grammar formalism, LLL, from which GDK can generate parsers based on different technologies. Fig. 6 shows an LLL excerpt. We use a GDK option for a C-based parser, which relies on a proprietary combinator library. We have implemented *AspectCobol* for the AcuCobolGT dialect because this dialect is dictated by the code base in a proof-of-concept project.

6.4 XML-based source-code manipulation

We will now address the second obstacle for AOP support [11]: the weaving framework. We employ a GDK option, where the generated C-based parser constructs syntax trees according to a proprietary term API. We complete the GDK-generated parser by a serialiser that maps the C terms to XML. (We go through an expensive text representation of XML, which could be avoided indeed, using a tool-to-tool XML API [32].) The XML representation encodes all details of layout and comments, which is less important for *AspectCobol*, since a user is perhaps not supposed to study the weaved Cobol code, but it is important when cobble is used for re-engineering transformations. In Fig. 7, we illustrate the XML representation. We also provide an unparser that maps XML data to concrete Cobol syntax. The various XML element types correspond to the nonterminals in the Cobol grammar.

The cobble-based weaver processes XML via DOM. It is straightforward to locate pointcuts, advice and potential join-point shadows in a DOM tree. For instance, to find all paragraphs in a DOM tree, the following XPath query suffices:

```
//paragraph
```

The immediate definition of the XML format in terms of the Cobol grammar is debatable. All functionality that operates on the XML representation does not resist grammar changes. Also, we cannot immediately serve multiple Cobol dialects. We plan to work on a

[12] A related thread: http://www.talkaboutprogramming.com/group/comp.lang.cobol/messages/112729.html (comp.lang.cobol)

[13] http://www.cs.vu.nl/grammars/vs-cobol-ii/

[14] Grammar Recovery Kit: http://www.cs.vu.nl/grammarware/grk/

[15] Grammar Deployment Kit: http://gdk.sourceforge.net/

```
<display>
 <string>DISPLAY</string> <!--   -->
 <what_to_display>
  <literal>
    <string>&quot;HELLO WORLD!&quot;</string>
  </literal>
 </what_to_display>
</display>
```

Figure 7: XML element for `DISPLAY "HELLO WORLD!"`

model-driven approach, where the grammar structure is mapped to a more abstract format. Codename: aspectual abstract Cobol syntax. We expect that existing work on language-independent source-level weavers [15], and, more generally, on language implementation will be of use in this context.

6.5 Technology details for the weaver

The actual weaver is implemented in Java using XML APIs for DOM and XPath. Weaving boils down to restructuring of a DOM tree. The weaver loops over all potential join-point shadows, attempts to match pointcuts, and performs advice weaving. We use BeanShell[16] scripts for component integration, which offers more flexibility than the exclusive use of compiled Java. We (currently) use a Prolog component for matching pointcuts to join-point shadows. *AspectCobol*'s bindings and multi-valued selection suggests this use of logic programming. We use a specific Prolog implementation, TuProlog[17], which is fully inter-operable with Java. For the record, there is no proper obstacle to reconstructing the component for matching pointcuts in plain Java.

7. CONCLUDING REMARKS

We have described an effort to bring AOP to Cobol. We have focused on classic Cobol, as opposed to OO Cobol, because most existing business-critical Cobol code use some classic Cobol dialect. Also, aspect-oriented extensions of OO languages are comparatively well understood anyhow. We have also neglected more advanced AOP constructs for abstract pointcuts, abstract advice, aspect inheritance, aspect composition whose development will be orthogonal to the basic development of this paper. We have found that Cobol is well prepared for an AOP extension because of its existing concept of declaratives. We claim that an AOP extension of Cobol (perhaps even the proposed *AspectCobol*) is an essential contribution to the deployment of the AOP/AOSD paradigm — if it supposed to cover business programming ... if it supposed to cover the language in which most business-critical software resides.

There are barriers for getting overly enthusiastic. We have revealed that some killer apps of AOP do not carry over to Cobol. We have aimed at a compensation. That is, we have identified new application potential for AOP such as aspects that help re-engineering Cobol code bases, which in turn is the major technical challenge in managing these software assets. For instance, we have argued that AOP can be used for automated testing of pervasive program changes. Another barrier: Does the conservative Cobol developer appreciate *AspectCobol*? We admit that the typical Cobol developer already depends on a ugly cocktail of home-grown, vendor-provided, and third-party tools for analysis, transformation, code generation, preprocessing, and embedded language processing. *AspectCobol* would just add to this plethora. There is an alternative: *A future Cobol development environment shall offer a general transformation framework, which serves automated re-engineering transformations as well as advice weaving.*

[16] BeanShell: http://www.beanshell.org/

[17] TuProlog http://lia.deis.unibo.it/research/tuprolog/

Acknowledgements

The authors are grateful for contributions, comments or support by Jeff Gray, Steven Klusener, David Tas, Barry Tauber, Herman Tromp and Chris Verhoef. Ralf Lämmel received support within the Dutch research project CALCE (*C*omputer-*A*ided *L*ife *C*ycle *E*nabling of Software Assets), sponsored by the Dutch Ministry of Economic Affairs via contract SENTER-TSIT3018. Ralf Lämmel represents the Free University Amsterdam and thereby the Netherlands in the ISO standardisation working group (WG4) on Cobol. The described concepts are in no way approved or recommended by the working group. Kris De Schutter received support within the Belgium research project ARRIBA (*A*rchitectural *R*esources for the *R*estructuring and *I*ntegration of *B*usiness *A*pplications), sponsored by the IWT, Flanders.

8. REFERENCES

[1] ANSI. American National Standard Programming Language Cobol X3.23–1974. American National Standards Institute, 1430 Broadway, New York, New York 10018 (Price $17).

[2] G.J. Badros. JavaML: a markup language for Java source code. *Computer Networks*, 33(1-6):159–177, 2000.

[3] I.D. Baxter. Using transformation systems for software maintenance and reengineering. In *ICSE 2001: Proc. of the 23rd International Conf. on Software Engineering*, pages 739–740. IEEE Computer Society, 2001.

[4] R. Boban. The Difference Between Logging and Tracing, 18 August 2004. SAP Developer Network; http://www.sdn.sap.com/irj/servlet/prt/portal/prtroot/com.sap.km.cm.docs/documents/a1-8-4/The%20Difference%20Between%20Logging%20and%20Tracing.article.

[5] B. Burke and M. Fleury. A Killer App for AOP. *Linux Magazine*, June 2004.

[6] T. Cohen and J. Gil. AspectJ2EE = AOP + J2EE. In M. Odersky, editor, *ECOOP 2004 – Object-Oriented Programming, 18th European Conf., Proc.*, volume 3086 of *LNCS*. Springer, 2004.

[7] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *AOSD 2004: Proc. of the 3rd International Conf. on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[8] K. De Volder and T. D'Hondt. Aspect-Orientated Logic Meta Programming. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2md International Conf., Reflection 1999, Proc.*, volume 1616 of *LNCS*, pages 250–272. Springer, 1999.

[9] R.E. Filman and D.P. Friedman. Aspect-oriented programming is quantification and obliviousness. In M. Akşit, S. Clarke, T. Elrad, and R.E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley, Reading, MA, September 2004.

[10] J. Gray, T. Bapty, S. Neema, and J. Tuck. Handling crosscutting constraints in domain-specific modeling. *CACM*, 44(10):87–93, 2001.

[11] J. Gray and S. Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD 2004: Proc. of the 3rd International Conf. on Aspect-Oriented Software Development*, pages 36–45. ACM Press, 2004.

[12] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD 2004: Proc. of the 3rd International Conf. on Aspect-Oriented Software Development*, pages 26–35. ACM Press, 2004.

[13] ISO/IEC. Information technology — Programming languages — COBOL, 2002. Reference number ISO/IEC 1989:2002(E).

[14] ISO/IEC JTC 1/SC 22 /WG 4. Information Technology — Programming languages, their environments and system software interfaces — COBOL Collection Classes, 22 November 2004. Technical Report, ISO/WDTR 24717, Working Draft Under Consideration.

[15] A. Jackson and S. Clarke. SourceWeave.NET: Source-level cross-language aspect-oriented programming. In G. Karsai and E. Visser, editors, *Proc. of the 3rd International Conf. on Generative Programming and Component Engineering (GPCE 2004)*, volume 3286 of *LNCS*, pages 115–134. Springer, 24–28 October 2004.

[16] M. Karaorman, U. Holzle, and J. Bruno. jContractor: A reflective Java library to support design by contract. In P. Cointe, editor, *Meta-Level Architectures and Reflection, 2nd International Conf. on Reflection 1999, Proc.*, volume 1616 of *LNCS*, pages 175–196. Springer, July 1999.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An Overview of AspectJ. In *Proc. of the 15th European Conf. on Object-Oriented Programming (ECOOP 2001)*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[18] H. Kim and S. Clarke. The Relevance of AOP to an Applications Programmer in an EJB Environment, April 2002. Paper presented at Workshop on Aspects, Components and Patterns for Infrastructure Software (affiliated with AOSD 2002).

[19] S. Klusener, R. Lämmel, and C. Verhoef. Architectural Modifications to Deployed Software. *Science of Computer Programming*, 54:143–211, 2005.

[20] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA 2003: Proc. of the 18th Annual ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–12. ACM Press, 2003.

[21] R. Lämmel. Object-Oriented COBOL: Concepts & Implementation. In J. Wessler et al., editors, *COBOL Unleashed*. Macmillan Computer Publishing, September 1998. 44 pages, 2 chapters.

[22] R. Lämmel and C. Verhoef. Semi-automatic Grammar Recovery. *Software—Practice & Experience*, 31(15):1395–1438, December 2001.

[23] C.V. Lopes. *D: A language framework for distributed programming*. PhD thesis, College of Computer Science, Northeastern University, November 1997.

[24] A.C. Lorents. Mining Legacy Assets with AssetMiner. *CobolReport.com*, May 2001. http://objectz.com/columnists/alden/05212001.asp.

[25] I. Michiels, K. De Schutter, T. D'Hondt, and G. Hoffman. Using Dynamic Aspect to Extract Business Rules from Legacy Code. In *Online proceedings of Dynamic Aspects Workshop at AOSD*, 2004.

[26] J. Piggot. Variable-Size (Dynamic) Tables, 30 April 2003. J4 Technical Report 03-0107.

[27] A. Rashid and R. Chitchyan. Persistence as an aspect. In *Proc. of the 2nd international Conf. on Aspect-oriented software development*, pages 120–129. ACM Press, 2003.

[28] D. Sabbah. Aspects: from promise to reality. In *AOSD 2004: Proc. of the 3rd International Conf. on Aspect-Oriented Software Development*, pages 1–2. ACM Press, 2004.

[29] J. E. Sammet. The early history of COBOL. In *The first ACM SIGPLAN Conf. on History of programming languages*, pages 121–161. ACM Press, 1978.

[30] S. Schonger, E. Pulvermüller, and S. Sarstedt. Aspect-Oriented Programming and Component Weaving: Using XML Representations of Abstract Syntax Trees. In *Proc. of the 2nd German GI Workshop on Aspect-Oriented Software Development*, pages 59 – 64, February 2002. Technical Report No. IAI-TR-2002-1, University of Bonn, Computer Science Dept.

[31] D. Sereni and O. de Moor. Static analysis of aspects. In *Proc. of the 2nd International Conf. on Aspect-oriented Software Development*, pages 30–39. ACM Press, 2003.

[32] S.E. Sim. Next generation data interchange: Tool-to-tool application program interfaces. In *Proc. of Working Conf. on Reverse Engineering (WCRE'00)*, pages 278–283. IEEE Computer Society, November 2000.

[33] T. Skotiniotis and D.H. Lorenz. Cona: aspects for contracts and contracts for aspects. In *OOPSLA 2004: Companion to the 19th annual ACM SIGPLAN Conf. on Object-oriented programming systems, languages, and applications*, pages 196–197. ACM Press, 2004.

[34] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *OOPSLA 2002: Proc. of the 17th ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 174–190. ACM Press, 2002.

[35] N.P. Veerman. Revitalizing modifiability of legacy assets. *Journal of Software Maintenance and Evolution*, 16(4-5):219–254, July-Oct 2004. Special issue on CSMR 2003.

[36] Y. Zou and K. Kontogiannis. A framework for migrating procedural code to object-oriented platforms. In *8th Asia-Pacific Software Engineering Conf. (APSEC 2001), 4-7 December 2001, Macau, China*, pages 390–499. IEEE Computer Society, 2001.