Aspect-orientation for revitalising legacy business software

Kris De Schutter, Bram Adams

{*Kris.DeSchutter,Bram.Adams*}@*UGent.be*
Ghislain Hoffman Software Engineering Lab, INTEC
Ghent University, Belgium

Abstract

This paper relates on a first attempt to see if aspect-oriented programming (AOP) can help with the revitalisation of legacy business software. By means of four realistic case studies covering reverse engineering, restructuring and integration, we discuss the applicability of the aspect-oriented paradigm in the context of two major programming languages for such environments: Cobol and C.

Key words: AOP, LMP, legacy software, evolution.

1 Introduction

This paper addresses the question of whether aspect-oriented programming (AOP) techniques [6] can help with the revitalisation of legacy business software. AOP is an emerging paradigm, leveraging two key principles: *quantification* and *obliviousness* [4]. The first allows one to express non-localised behaviour in a localised way. The latter means one can apply such behaviour to any existing application without any special preparation to it. When considering legacy applications, which resist change [1], this seems a useful property for a re-engineering tool to have. Whether this philosophical notion also holds in practice has, however, not been tested against realistic legacy software yet. This is most likely due to the lack of instantiations of AOP for these environments.

The authors have developed AO extensions for the two major programming languages encountered in legacy business software: Cobol and C. These extensions enable quantification and obliviousness by making use of logic meta-programming (LMP) [10] in their pointcut language. This was found to be an adequate solution to overcome a lack of reflection in Cobol and C, and to allow the generic definition of behaviour [11]. This paper will now take these tools and see if they can be applied to four realistic cases for the revitalisation of legacy software.

```
 1 static FILE* fp;

 3 Type around tracing (Type) on (Jp):
     call(Jp,"^(?!.*printf$|.*scanf$).*$")
 5   && type(Jp,Type) && !is_void(Type)
   {
 7   Type i;

 9   fprintf (fp, "before␣(␣%s␣in␣%s␣)\n", Jp->functionName, Jp->fileName);

11   i = proceed ();

13   fprintf (fp, "after␣(␣%s␣in␣%s␣)\n", Jp->functionName, Jp->fileName);

15   return i;
   }
```

Fig. 1. A generic tracing aspect (excerpt).

2 Enabling dynamic analyses of legacy software

In order to help legacy systems evolve, one needs a thorough understanding of the systems at hand. As in these environments there is most often a lack of (up-to-date) documentation, one is forced into applying reverse engineering techniques. Dynamic analyses offer one approach to this, by analysing the dynamic run-time behaviour of systems [5,12]. The role for AOP here is to easily enable such techniques by applying some tracing aspect to existing applications.

2.1 A generic tracing aspect

In figure 1, we have shown part[1] of a generic tracing aspect written in Aspicere[2]. Aspects are encapsulated in plain compilation units able to hold advice constructs. Advice itself features a signature (lines 3), a pointcut (lines 4–5) and a body (lines 7–15). The advice body is written in C, with some additions for accessing the runtime context (`Jp` variable on lines 9 and 13, and the `proceed` call on line 11). A simple template mechanism is also available to help overcome C's relatively weak typing.

The idea is to trace calls to all procedures except for the `printf`- and `scanf`-families (line 4) and stream output into a file (`fp`, declared on line 1) before and after each call (lines 9 and 13). Opening and closing of the file pointer on line 1 is achieved by advising the `main`-procedure[3]. The return type of the advised procedure call is bound on line 5. This binding is then used in the advice's signature (lines 3) and as a type parameter in its body (lines 7). This way, the `tracing` advice is not limited to one particular type of procedures. The well-known `thisJoinPoint` construct from AspectJ-like languages, can also be accessed through a join point-specific binding (`Jp` on lines 3) and used as such (lines 9 and 13).

[1] We do not show advice for void procedures, as these are equivalent to the advices shown, less the need for a temporary variable to hold the return value.

[2] Website: http://users.ugent.be/~badams/aspicere/.

[3] Not shown here to conserve space.

2.2 Problem: the build system

As source code is the most portable representation of C programs across several platforms, Aspicere relies on a source-to-source weaving strategy, and as such acts as a preprocessor to a normal C compiler. More specifically, it transforms aspects into genuine C compilation units by converting the advices into (multiple) procedures. This enables the normal C visibility rules in a natural way, i.e. the visibility of `fp` on figure 1 is tied to the module containing the aspect. To accomplish this modularisation, we need to link this single transformed aspect into each advised application. Because the original makefile hierarchy drives the production of object files, libraries and executables, using a myriad of other tools and preprocessors (e.g. embedded SQL), and all of these potentially process advised input, it turns out that *Aspicere's weaver crosscuts the makefile system*. We therefore need to find out what is produced at every stage of the build and unravel accompanying linker dependencies. In case all makefiles are automatically generated using, for instance, automake, one could try to replace (i.e. alias) the tools in use by wrapper scripts which invoke the weaving process prior to calling the original tool. The problem here is that this is an all-or-nothing approach. It may be that in some cases weaving is needed (e.g. a direct call to `gcc`), and in others not (e.g. when `gcc` is called from within `esql`). Making the replacement smart enough to know when to do what is not a trivial task.

In [11], we applied the tracing aspect of figure 1 to a large case study (453 KLOC of ANSI-C) to enable dynamic analyses. The system consisted of 267 makefiles, not all of which were generated. Without intimate knowledge of the build system, it was hard to tell whether source files were first compiled before linking all applications, or (more likely) whether all applications were compiled and linked one after the other. As such, our weaving approach was not viable. As an ad hoc solution, we opted to move the transformed advice into the advised base modules themselves. This meant that we had to declare `fp` as a local variable of the `tracing` advice, resulting in huge run-time overhead due to repeated opening and closing of the file.

2.3 Conclusion

Applied to reverse-engineering contexts, the use of AOP, LMP and a template mechanism allows non-invasive and intuitive extraction of knowledge hidden inside legacy systems, *without* prior investigation or exploration of the source code [11]. One does not have to first extract all available types and copy the tracing advice for all of them, as was experienced in [3].

While dynamic analyses can be enabled in this way without the need to prepare the source code of legacy applications in any way, one is still faced with having to prepare the build system for these applications (once). As many such applications rely on custom defined and sometimes complex makefile hierarchies (or similar), any real use of AOP for revitalising legacy software will depend on a solution to this problem.

3 Mining business rules in legacy software

When implemented in software, business knowledge, information and rules tend to be spread out over the entire system. With applications written in Cobol this is even more the case, as Cobol is a language targeted at business processing [4] but without modern day modularity mechanisms. This information then tends to get lost over time, so that when some maintenance is required one is again forced into reverse engineering. We argue that AOP can provide a flexible tool for such efforts.

We will now revisit a case from [7], in which Isabel Michiels and the first author discuss the possibility of using dynamic aspects for mining business rules from legacy applications. The case, put briefly, is this:

> "*Our accounting department reports that several of our employees were accredited an unexpected and unexplained bonus of 500 euro. Accounting rightfully requests to know the reason for this unforeseen expense.*"

We will now revisit this case, showing the actual advices which may be used to achieve the ideas set forth in that paper.

We start off by noting that we are not entirely in the dark. The accounting department can give us a list of the employees which got "lucky" (or unlucky, as their unexpected bonus did not go by unnoticed). We can encode this knowledge as facts:

```
META-DATA DIVISION.
  FACTS SECTION.
    LUCKY-EID VALUE 7777.
    LUCKY-EID VALUE 3141.
    *> etc.
```

Furthermore, we can also find the definition of the employee file which was being processed, in the copy books (roughly similar to header files in C):

```
DATA DIVISION.
FILE SECTION.
  FD EMPLOYEE-FILE.
  01 EMPLOYEE.
     05 EID PIC 9(4).
     *> etc.
```

Lastly, from the output strings we can figure out the name of the data item holding the total value. This data item, `BNS-EUR`, turns out to be an edited picture. From this we conclude that it is only used for pretty printing the output, and not for performing actual calculations. At some time during execution the correct value for the bonus was moved to `BNS-EUR`, and subsequently printed. So our first task is to find what variable that was. We go at this by tracing all moves to `BNS-EUR`, but *only while processing one of our lucky employees*:

```
FIND-SOURCE-ITEM SECTION.
  USE BEFORE ANY STATEMENT
  AND NAME OF RECEIVER EQUAL TO "BNS-EUR"
  AND BIND LOC TO LOCATION
  AND IF EID EQUAL TO LUCKY-EID.
```

[4] Cobol = Common *Business* Oriented Language

```
6  MY-ADVICE.
    DISPLAY EID, ":␣", LOC.
```

In short, this advice states that before all statements (line 2) which have BNS-EUR as a receiving data item (line 3), and if EID (id for the employee being currently processed; see data definition higher up) equals a lucky id (runtime condition on line 5), we display the location of that statement as well as the current id. Amongst several string literals (which we can therefore immediately disregard) we find a variable named BNS-EOY, whose name suggests it holds the full value for the end-of-year bonus.

Our next step is to figure out how the end value was calculated. We set up another aspect to trace all statements modifying the variable BNS-EOY, but again only while processing a lucky employee. We do this in three steps. First:

```
1 TRACE-BNS-EOY SECTION.
    USE BEFORE ANY STATEMENT
3   AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
    AND BIND LOC TO LOCATION
5   AND IF EID EQUAL TO LUCKY-EID.
   MY-ADVICE.
7   DISPLAY EID, ":␣statement␣at␣", LOC.
```

Before execution of any statement (line 2) having BNS-EOY as a receiving data item (line 3), and when processing a lucky employee (line 5), this would output the location of that statement. Next:

```
1 TRACE-BNS-EOY-SENDERS SECTION.
    USE BEFORE ANY STATEMENT
3   AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
    AND BIND SENDING TO SENDER
5   AND BIND SENDING-NAME TO NAME OF SENDING
    AND IF EID EQUAL TO LUCKY-EID.
7  MY-ADVICE.
    DISPLAY SENDING-NAME, "␣sends␣", SENDING.
```

This outputs the name and value for all sending data items (lines 4 and 5) before execution of any of the above statements. This allows us to see the contributing values. Lastly, we want to know the new value for BNS-EOY which has been calculated.

```
  TRACE-BNS-EOY-VALUES SECTION.
2   USE AFTER ANY STATEMENT
    AND NAME OF RECEIVER EQUAL TO "BNS-EOY"
4   AND IF EID EQUAL TO LUCKY-EID.
   MY-ADVICE.
6   DISPLAY "BNS-EOY␣=␣", BNS-EOY.
```

We now find a data item (cryptically) named B31241, which is consistently valued 500, and is added to BNS-EUR in every trace. Before moving on we'd like to make sure we're on the right track. We want to verify that this addition of B31241 is only triggered for our list of lucky employees. Again, a dynamic aspect allows us to trace execution of exactly this addition and helps us verify that our basic assumption holds indeed. We start by recording the location of the "culprit" statement as a usable fact:

```
  META-DATA DIVISION.
```

```
FACTS SECTION.
  CULPRIT-LOCATION VALUE 666.
  *> other facts as before
```

The test for our assumption may then be encoded as:

```
TRACE-BNS-EOY-SENDERS SECTION.
  USE BEFORE ANY STATEMENT
  AND LOCATION EQUAL TO CULPRIT-LOCATION
  AND IF EID NOT EQUAL TO LUCKY-EID.
 MY-ADVICE.
  DISPLAY EID, ":␣back␣to␣the␣drawing␣board.".
```

This tests whether the culprit statement gets triggered during the process of any of the other employees. If it does, then something about our assumption is wrong. Or it may be that the accounting department has missed one of the lucky employees.

Given the verification that we are indeed on the right track, the question now becomes: why was this value added for the lucky employees and not for the others? Unfortunately, the logic behind this seems spread out over the entire application. So to try to figure out this mess we would like to have an execution trace of each lucky employee, including a report of all tests made and passed, up to and including the point where B31241 is added. Dynamic aspects allow us to get these specific traces. First, some preliminary work:

```
WORKING-STORAGE SECTION.
  01 FLAG PIC 9 VALUE 0.
     88 FLAG-SET     VALUE 1.
     88 FLAG-NOT-SET VALUE 0.
```

The FLAG data item will be used to indicate when tracing should be active and when not. For ease of use we also define two "conditional" data items: FLAG-SET and FLAG-NOT-SET. These reflect the current state of our flag. Our first advice is used to trigger the start of the trace:

```
  TRACE-START SECTION.
2   USE AFTER READ STATEMENT
    AND NAME OF FILE EQUAL TO "EMPLOYEE-FILE"
4   AND BIND LOC TO LOCATION
    AND IF EID EQUAL TO LUCKY-EID.
6  MY-ADVICE.
    SET FLAG-SET TO TRUE.
8   DISPLAY EID, ":␣start␣at␣", LOC.
```

I.e., whenever a new employee record has been read (line 2 and 3), and that record is one for a lucky employee (line 5), we set the flag to true (line 7). We also do some initial logging (line 8). The next advice is needed for stopping the trace when we have reached the culprit statement:

```
TRACE-STOP SECTION.
  USE AFTER ANY STATEMENT
  AND LOCATION EQUAL TO CULPRIT-LOCATION.
 MY-ADVICE.
  SET FLAG-NOT-SET TO TRUE.
  DISPLAY EID, ":␣stop␣at␣", LOC.
```

Then it is up to the actual tracing. We capture the flow of procedures, as well as execution of all conditional statements:

```
TRACE-PROCEDURES SECTION.
  USE AROUND PROCEDURE
  AND BIND PROC TO NAME
  AND BIND LOC TO LOCATION
  AND IF FLAG-SET.
 MY-ADVICE.
  DISPLAY EID, ":␣before␣", PROC, "␣at␣", LOC.
  PROCEED.
  DISPLAY EID, ":␣after␣", PROC, "␣at␣", LOC.

TRACE-CONDITIONS SECTION.
  USE AROUND ANY STATEMENT
  AND CONDITION
  AND BIND LOC TO LOCATION
  AND IF FLAG-SET.
 MY-ADVICE.
  DISPLAY EID, ":␣before␣condition␣at␣", LOC.
  PROCEED.
  DISPLAY EID, ":␣after␣condition␣at␣", LOC.
```

From this trace we can then deduce the path that was followed from the start of processing a lucky employee, to the addition of the unexpected bonus. More importantly, we can see the conditions which were passed, from which we can (hopefully) deduce the exact cause.

This is where the investigation ends. For those curious, we refer to the original paper for the solution [7]. Whatever the cause of the problem, AOP+LMP provided us with a flexible and powerful tool to perform our investigation.

4 Encapsulating procedures

In [9], Harry and Stephan Sneed discuss creating web services from legacy host programs. They argue that while tools exist for wrapping presentation access and database access for use in distributed environments,

"*accessing [...] the business logic of these programs, has not really been solved.*"

In an earlier paper, [8], Harry Sneed discusses a custom tool which allowed the encapsulation of Cobol procedures, to be able to treat them as "methods", a first step towards wrapping business logic. Part of that tool has the responsibility of creating a switch statement at the start of the program, which performs the requested procedure, depending on the method name.

4.1 A basic wrapping aspect

Figure 2 shows how encapsulation of procedures (or "business logic") can be achieved, in a generic way, using AOP and LMP. The aspect shown here, written in Cobble, consists of two advices.

The first advice, `DISPATCHING` (lines 1–7), takes care of the dispatching. It acts around the execution of the entire program (line 2), and once for every paragraph in this program (line 3). The latter effect is caused by the ambiguousness of the `PARAGRAPH` selector. This can be any of a number of values. Rather than just picking one, what Cobble does is *pick them all*: the advice gets activated

```
 1 DISPATCHING SECTION.
     USE AROUND PROGRAM
 3   AND BIND PARA TO PARAGRAPH
     AND BIND PARA-NAME TO NAME OF PARA
 5   AND IF METHOD-NAME EQUAL TO PARA-NAME.
    MY-ADVICE.
 7   PERFORM PARA.

 9 ENCAPSULATION SECTION.
     USE AROUND PROGRAM.
11  MY-ADVICE.
     PERFORM ERROR-HANDLING.
13   EXIT PROGRAM.
```

Fig. 2. Aspect for procedure encapsulation.

for every possible solution to its pointcut, one after the other. Furthermore, the DISPATCHING advice will only get triggered when METHOD-NAME matches the name of the selected paragraph (extraction of this name is seen on line 4). This is encoded in a runtime condition on line 5. Finally, the advice body, when activated, simply calls the right paragraph (PERFORM statement on line 7).

The second advice, ENCAPSULATION (lines 9–13), serves as a generic catch-all. It captures execution of the entire program (line 10), but replaces this with a call to an error handling paragraph (line 12) and an exit of the program (line 13). The net effect is that whenever the value in METHOD-NAME does not match any paragraph name in the program, the error will be flagged and execution will end. This, together with the first advice, gives us the desired effect.

We are left with the question of where METHOD-NAME is defined, and how it enters our program. The answer to the first question is simply this: any arguments which get passed into a Cobol program from the outside must be defined in a *linkage section*. I.e.:

```
LINKAGE SECTION.
01 METHOD-NAME PIC X(30) VALUE SPACES.
```

Furthermore, the program division needs to declare that it expects this data item as an input from outside:

```
PROGRAM DIVISION USING METHOD-NAME.
```

This begs the question as to how this input parameter METHOD-NAME was inserted in an AOP-like way. Simply: it was *not*. We tacitly assumed our aspect, and the accompanying input parameters, to be defined *inside* the target program (a so-called "intra-aspect"). Of course, for a truly generic "inter-aspect" we need to remedy this. Definition of the METHOD-NAME data item is no big problem. We can simply define it within an aspect module, which, upon weaving, would augment the target program (modulo some alpha-renaming to prevent unintended name capture):

```
IDENTIFICATION DIVISION.
 ASPECT-ID. PROCEDURE-WRAPPING.

DATA DIVISION.
LINKAGE SECTION.
```

```
01 METHOD-NAME PIC X(30) VALUE SPACES.
```

From this, it becomes pretty obvious that `METHOD-NAME` should be used as an input parameter of the *base program*. The concept of a linkage section makes no sense for an external aspect module, as an aspect will never be called in such a way. The hard part lies with the semantics of declaring extra input data items on another program. What do we expect to happen?

- Does the introduction of an input data item by the aspect replace existing input items in the advised program, or is it seen as an addition to them?
- If it is added to them, then where does it go into the existing list of inputs? At the front? At the back?
- What happens when multiple aspects define such input items? In what order do they appear?
- How do we handle updating the sites where the woven program gets called? The addition of an extra input item will have broken these.

Consider the C/Java/... equivalent of this: what does it mean to introduce new parameters on procedures/methods? More to the point, *should* we allow this?

4.2 An extended wrapping aspect

The complexity of the problem increases when we consider another important feature of Sneed's tool (ignored until now):

> "*For each [encapsulated] method a data structure is created which includes all variables processed as inputs and outputs. This area is then redefined upon a virtual linkage area. The input variables become the arguments and the output variables the results.*" [8]

Put another way, we must find all data items on which the encapsulated procedures depend. These are then gathered in a new record (one per procedure), which redefines a "virtual linkage area" (in C terms: a union over all newly generated typedefs). This linkage area must then also be introduced as an input data item of the whole program. Such a requirement seems far out of the scope of AOP. While it has a crosscutting concern in it (cfr. "for *each* method"), this concern can not be readily defined using existing AOP constructs.

Instead, figure 3 shows a different approach to the problem. It is encoded neither in Cobble or Aspicere, opting for a different view on the AOP+LMP equation. Whereas the previous examples were based on LMP embedded in AOP, figure 3 is based on embedding AOP in LMP, similar to the approach in [2]. The code can be read as follows. Whatever you find enclosed in curly brackets ($\{\ldots\}$) is (aspect-)code which is to be generated. This can be further parameterized by placing variables in "fishgrates" ($<\ldots>$), which will get expanded during processing. Everything else is Prolog, used here to drive the code generation.

Let us apply this to the code in figure 3. Lines 1 and 2 declare the header of our aspect, while lines 4–6 define the linkage section as discussed before. Lines 8–

```
   { IDENTIFICATION DIVISION.
 2     ASPECT-ID. PROCEDURE-WRAPPING.

 4   DATA DIVISION.
     LINKAGE SECTION.
 6   01 METHOD-NAME PIC X(30) VALUE SPACES. },

 8 findall(
     [Name, Para, Wss],
10   ( paragraph(Name, Para),
       slice(Para, Slice),
12     wss(Slice, Wss)
     ),
14   AllInOut
   ),
16
   max_size(AllInOut, VirtualStorageSize),
18 { 01 VSPACE PIC X(<VirtualStorageSize>). },

20 all(member([Name, Para, Wss], AllInOut), (
     {  01 SLICED-<Name> REDEFINES VSPACE.},
22   all( (record(R, Wss), name(R, RName)), (
       clone_and_shift(R, "<RName>-<Name>", SR),
24     { <SR> }
     ))
26 )),

28 { PROGRAM DIVISION USING METHOD-NAME, VSPACE.
     DECLARATIVES. },
30
   all(member([Name, Para, Wss], AllInOut), (
32   { WRAPPING-FOR-<Name> SECTION.
         USE AROUND PROGRAM
34       AND IF METHOD-NAME EQUAL TO "<Name>".
        WRAPPING-BODY.
36   },
     all( (top_record(R, Wss), name(R, RName)),
38     { MOVE <RName>-<Name> TO <RName>.}
     ),
40   {    PERFORM <Name>.}
     all( (top_record(R, Wss), name(R, RName)),
42     { MOVE <RName> TO <RName>-<Name>.}
     )
44 )),

46 { ENCAPSULATION SECTION.
       USE AROUND PROGRAM.
48    MY-ADVICE.
       PERFORM ERROR-HANDLING.
50     EXIT PROGRAM.
     END DECLARATIVES. }
```

Fig. 3. Full procedure encapsulation.

15 calculate all slices (`slice/2` on line 11) for all paragraphs (`paragraph/2` on line 10). From each of these we extract the working-storage section (`wss/2` on line 12), which gives us the required in- and output parameters, collected in `AllInOut` (line 14). From this we extract the size of the largest one (`max_size/2` on line 17) which is used next in the definition of the virtual storage space (line 18). Next, for each paragraph (i.e. for each member of `AllInOut`), we generate a redefinition of the virtual space to include all data items on which that paragraph

depends (lines 20–26). The redefinition can be seen on line 21, where it is given a unique name (i.e. `SLICED`-*paragraph-name*). Its structure is defined by going over all records in the working-storage section for that paragraph (line 22), cloning each record under a new, unique name while updating the level number (line 23), and then outputting this new record (line 24). This concludes the data definition. Next, the procedure division is put down, declaring the necessary parameters (line 28). We then generate advice similar to that in figure 2, but now they need to perform some extra work. First, they must transfer the data from the virtual storage space as redefined for the paragraph, to the original records defined for the program (lines 37–39). The original paragraph may then be called without worry (line 40). Afterwards, the calculated values are retrieved by moving them back to the virtual storage space, again as redefined for the paragraph (lines 41–43). All that is left is the generic catch-all (lines 46–50), and the closing of the aspect (line 51).

Despite the inherent complexity of the problem, AOP+LMP allowed us to write down our crosscutting concern with certain ease. LMP was leveraged to define our aspect by reasoning over the program. AOP was leveraged to tackle the actual weaving semantics, unburdening us from writing program transformations. Granted, we quite happily made use of a slicing predicate to do most of the hard work (line 11). Still, the use of libraries which hide such algorithms is another bonus we can get from LMP.

5 Year 2000 syndrome

The Y2K-bug is probably the best-known example of unexpected change in legacy systems. It is important to understand that at the heart of this was not a lack of technology or maturity thereof, but rather the understandable failure to recognize that code written as early as the sixties would still be around some forty years later. So might AOP+LMP have helped solving the problem? The problem statement certainly presents a crosscutting concern: whenever a date is accessed in some way, make sure the year is extended.

This presents our first problem: how do we recognize data items for dates in Cobol? While Cobol has structured records, and stringent rules for how data is transferred between them, they carry no semantic information whatsoever. Knowing which items are dates and which are not, requires human expertise. The nice thing about LMP is that we could have used it to encode this. In C, where a disaster is expected in 2038[5] (hence Y2K38), the recognition problem is less serious because of C's more advanced typing mechanisms. A date in (ANSI-)C could be built around the standard time provisions (in "time.h"), or otherwise some (hopefully sensibly named) custom typedef. In the former case, recompiling the source code on a system using more than 32 bits to represent integers solves everything immediately. Whereas all variables in Cobol have to be declared in terms of the

[5] More details on `http://www.merlyn.demon.co.uk/critdate.htm`

same, low-level Cobol primitives, C allows variables to be declared as instances of user-defined types. In this sense, the latter case (custom date type) represents much less of a problem. The check for a date would be equivalent to a check for a certain type.

Second problem for Cobol: given the knowledge of which data items carry date information, how do we know which part encodes the year? It may be that some item holds only the current year, or that it holds everything up to the day. A data item may be in *Gregorian* form (i.e. "yyddd") rather than standard form ("yymmdd"). Of course, that "standard" may vary from locale to locale (the authors would write it as "ddmmyy"). But again, we could use LMP to encode this knowledge.

Let us assume we can check for data items which hold dates, and that these have a uniform structure (in casu "yymmdd"). Then we might write something like:

```
1 AN-YYMMDD-FIX SECTION RETURNING MY-DATE.
    USE AROUND SENDING-DATA-ITEM
3   AND SENDING-DATA-ITEM IS DATE.
   MY-ADVICE.
5   MOVE PROCEED TO MY-DATE(3:8).
    IF MY-DATE(3:4) GREATER THAN 50 THEN
7     MOVE 19 TO MY-DATE(1:2)
    ELSE
9     MOVE 20 TO MY-DATE(1:2).
```

This advice has two problems. One is the definition of MY-DATE (referred to as a return value on line 1, and assumed to have a "yyyymmdd" format). In Cobol, all data definitions are global. Hence, MY-DATE is a unique data item which gets shared between all advices. While this is probably safe most of the time, it could lead to subtle bugs whenever we have nested execution of such advice.[6] The same is true for all advices in Cobble. It is just that the need for a specific return value makes it surface more easily. Of course, in this case, the fix would be to require duplication of this data item for all advice instantiations. The greater problem lies in the weaving. When committed to a source-to-source approach, as we are with Cobble, weaving anything below the statement level becomes impossible. As Cobol lacks the idea of functions[7], we can not replace access to a data item with a call to a procedure (whether advice or the original kind) as we could do in C. The remedy for this would be to switch to machine-code weaving, but we are reluctant to do so, as we would lose platform independence. Common virtual machine solutions (e.g. as with ACUCobol) are not widespread either.

6 Conclusion and Future Work

We discussed restructuring and integration problems using four issues related to (classic) legacy software, and showed how three of these might be aided through a mixture of AOP and LMP. Reverse engineering based on tracing in C and business rule mining in Cobol went smoothly, employing LMP as a pointcut mechanism in

[6] Though not in this case, as the structure of the advice body *only* refers to the data item *after* the PROCEED statement.

[7] Functions can be written in later versions of Cobol. Our focus on legacy systems, however, rules these out for use here.

AOP. Encapsulation of procedures in Cobol, a typical legacy integration scenario, required a more generative approach embedding AOP in LMP.

As for the Y2K restructuring problem, the semantics of Cobol, especially its lack of typing, present too much of a limitation. In C, the Y2K38 problem can still be managed reasonably, precisely because it does feature such typing. Other legacy languages will likely exhibit the same behavior.

All in all, AOP+LMP proves a useful, flexible and strong tool to tackle the ills of legacy software, limited only by the base language's typing support. More elaborate case studies are needed to provide more feedback about other restructuring and integration problems, and the general necessity of the AOP-in-LMP approach.

References

[1] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1), 1995.

[2] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In *GPCE*, 2002.

[3] M. Bruntink, A. van Deursen, and T. Tourwé. An initial experiment in reverse engineering aspects. In *WCRE*. IEEE, 2004.

[4] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[5] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *CSMR*. IEEE, 2005.

[6] G. Kiczales. Aspect-oriented programming. In *Proceedings of the Eighth Workshop on Institutionalizing Software Reuse*, 1997.

[7] I. Michiels, T. D'Hondt, K. De Schutter, and G. Hoffman. Using dynamic aspects to distill business rules from legacy code. In *Dynamic Aspects Workshop*, 2004.

[8] H. M. Sneed. Encapsulating legacy software for use in client/server systems. In *WCRE*, 1996.

[9] H. M. Sneed and S. H. Sneed. Creating web services from legacy host programs. In *WSE*, 2003.

[10] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *TOOLS USA '98*. IEEE, 1998.

[11] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation - an industrial experience report. In *CSMR*, 2006.

[12] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*. IEEE, 2005.