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) and logic meta-programming (LMP) 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 legacy environments: Cobol and C.

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

1 Introduction

This paper addresses the question of whether a combination of aspect-oriented programming (AOP) [8] and logic meta-programming (LMP) [16] techniques can really help with the revitalisation of legacy business software. The hypothesis that this might be so (details in section 2) is not a new one. It is, for instance, at the heart of the ARRIBA [1] project, which the authors are associated with. The reasoning is that using aspects one might instrument, transform, or otherwise modify the original legacy applications from the outside without having to prepare them in any special way. This opens up the road —or rather *another* road— to reverse engineering, restructuring, integration, bug-fixing, maintenance in general, etc.

While the idea of applying AOP to real-life legacy applications is a compelling one, no real case studies exist (sofar) doing exactly this. Though there have been experiments on "legacy" OO code (such as JHotDraw [5]), none exist which have dealt with any kind of Cobol application. We believe this to be most likely due to the lack of usable instantiations of AOP for such environments. While the first author has made progress towards an "AspectCobol" (see [9], work done in association with Ralf Lämmel), a version that is generally usable in practice is not there

[1] ARRIBA: Architectural Resources for the Restructuring and Integration of Business Applications; a GBOU project sponsored by the IWT-Flanders. See http://arriba.vub.ac.be/ for more info.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

yet. The situation for legacy C applications is somewhat better given the many different available AOP incarnations [1], but here too the application of AOP to real problems with industrial legacy code is still in its infancy.

Therefore this paper takes a different approach at finding a validation for our hypothesis: we try to write down aspects for several fictitious (though realistic) cases. The idea is that we should at least be able to tackle this step if the hypothesis is to hold true. As such this paper considers the validity of a *necessary* requirement, rather than a *sufficient* one. In order for this experiment to work best, the examples presented here will each have very different scopes, corresponding to ARRIBA's fields of interest. The first (section 3) will be on reverse engineering, in which aspects are used as an enabling technology. The second (section 4) will be on the recovery of business logic. The third example (section 5) will tackle the encapsulation of business applications, something which should be of use when wanting to integrate these legacy applications into service-oriented (SOA) environments. The fourth and final example (section 6) will focus on a maintenance/bug-fixing problem, using the Y2K bug as an example.

It should be noted that we have been able to test out the first problem on a real case study, and that we will report on some of the major findings in this text. The other ones have not yet had this validation.

2 Hypothesis

Business applications are instantiations of specific business processes, and as such they are highly susceptible to the evolution thereof. With increased globalisation of enterprises, and ever greater demand for interconnectivity between companies, comes increasing pressure to scale up and integrate business applications. Aside from difficulties in integrating the different business *models* and their associated business *processes*, getting the business *applications* to cooperate is a major hurdle: in all but a few cases the documentation and support of these applications is insufficient (or even absent).

In general, the data repositories and the running programs provide the only true description of the information structures and applications they implement. Hence, the actual data and the source code of those applications form the only dependable documentation.

Merging business applications will always require the application of human expertise. Unfortunately, in an environment where its assets are so poorly understood this expertise can never be fully exploited.

As part of the ARRIBA project, we focus on how the emerging paradigm of AOP (combined with LMP) can be applied to this problem. AOP recognizes the problems caused by so-called crosscutting concerns (CCCs) and tries to solve them. Put briefly, some concern A is crosscutting w.r.t. another one (B) if A's implementation is "scattered" throughout the software system and at some point is "tangled" (mixed) with the implementation of B (more details in [14], among others). These symptoms can't be treated natively in traditional paradigms like OO or procedural

programming, such that programs end up as complex, brittle constructions which are hard to maintain, evolve or even understand. Apart from pinpointing the existence of CCCs, AOP addresses them using so-called aspects. These are separate modules (typically) dedicated to *one* CCC. They contain one or more smaller "advice" constructs whose bodies can be written in the base language, constituting the CCC's full implementation. The key issue here is that the advice code will need to be attached to ("woven into") the "base" code at the right places ("join points" identified by the advice's "pointcut") by the time the program execution reaches these. This is taken care of by an "aspect weaver", e.g. at compile time. In short, AOP promises clean modularisation of CCCs using aspects.

LMP on the other hand is a particular form of declarative meta-programming, in which meta-programs can be written based on logic rules and facts representing programs, metadata or any other useful knowledge [16]. As such, LMP can be complementary to AOP and we exploit this by using LMP in our advices' pointcuts.

We claim that this combination of LMP and AOP aids in the recovery of business architectures, as well as in the restructuring and integration of business applications, based on two observations. First, by embedding AOP in existing business environments we can empower software developers with a flexible toolchain while avoiding a steep learning curve. In using this toolchain, there is no requirement to move away from the existing development techniques; there is only the incentive to work with something that augments them. This can make for a faster turn-around based on available expertise.

Second, LMP can be used for expressing business concepts and architectural descriptions of business applications in a declarative way. This makes it possible to work with applications at a higher level of abstraction, which will allow better architectural descriptions to emerge. By making these descriptions available for practical use we can actively encourage development and understanding thereof.

In the next four sections, we will now investigate whether the combination of AOP and LMP indeed is able to express solutions to realistic problems in legacy systems.

3 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 traces of the dynamic run-time behaviour of systems [6,17]. The role for AOP which we will be discussing here is to enable such techniques by applying some tracing aspect to existing applications.

```
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",
15    Jp->functionName, Jp->fileName);

17  return i;
  }
```

Fig. 1. A generic tracing aspect: tracing advice.

```
  int around cleanup (Name) on (Jp):
20  execution(Jp,"main")
    && logfile(File) && stringify(File,Name)
22 {
    int i;
24
    fp = fopen (Name, "a"); /* open in append mode */
26  i = proceed ();
    fclose (fp);
28
    return i;
30 }
```

Fig. 2. A generic tracing aspect: initialisation and cleanup.

3.1 The code

Figure 1 shows part [2] of a generic tracing aspect written in Aspicere [3], an aspect language for C we developed [18]. In Aspicere, an aspect is a C module which can also contain advice (e.g. lines 3–18). An advice consists of a signature (line 3), a pointcut (lines 4–5) and a body containing C code (lines 6–18). The advice of

[2] We do not show advice for void procedures, as this is equivalent to the advice of figure 1, minus the need for a temporary variable to hold the return value.

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

figure 1 is activated on calls to all procedures except for the `printf`- and `scanf`-families (line 4), and only for those procedures which do return a value (pointcut condition on line 5 on the type not being `void`). The advice will stream tracing information to a file `fp` (declared on line 1) before and after these events (`fprintf` calls on lines 9 and 14 respectively). In between the tracing advice code, one can invoke the events themselves through the `proceed` call on line 12. Opening and closing of the file pointer `fp` is achieved by advising the sole execution of the `main`-procedure (figure 2) in a similar way.

There are two things to note here which are of importance. The first is the use of an AspectJ-like `thisJoinPoint` construct ([7]) on lines 10 and 15 to retrieve context-specific information on the current runtime event being advised for output to the trace. The second is the use of `Type` as a kind of generic type specifier. It is used inside the tracing advice (line 7) to deal with the various possible return types which may occur, and which C is not able to handle in a uniform way. The value of this type specifier is something which gets extracted by the aspect weaver during the matching of join points in the base program (`type` predicate on line 5), and which is instantiated in the advice like a C++ template parameter. These two additions provide reflective and context information lacking in the legacy base language. Without them, AOP would not be viable in legacy languages, whereas many modern OO languages already offer these features by themselves.

3.2 Evaluation: pro

Applied to reverse-engineering contexts, the use of AOP 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 [18]. One does not have to first extract all available types and write down the tracing advice for all of them, as was experienced in [3].

3.3 Evaluation: contra

As source code is the most portable representation of C programs across several platforms, Aspicere relies on a source-to-source weaving strategy, which corresponds to an extra preprocessing pass before the normal C compilation. More specifically, aspects are transformed 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 fully accomplish this modularisation, this single transformed aspect also needs to be linked into each advised application. Because the makefile system building all applications is a very complex chain of dependencies between object files, libraries and executables produced by a myriad of 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*[4]. However, we need to

[4] This problem is far more widespread than just the application of an aspect weaver: anytime one needs to add a new build step or new resources to a build system, similar issues arise.

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 [18], 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 immediately applicable and we had to resort to an ad hoc solution, resulting in (slightly) degraded performance of the woven application.

While dynamic analyses can be enabled using aspects 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 (see e.g. section 4) will depend on a solution to this problem.

4 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 [5] but without modern day modularity mechanisms. This information 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 [10], 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. The code shown here is written in Cobble [6], an implementation of an "AspectCobol" language designed by Ralf Lämmel and the first author [9].

[5] Cobol = Common *Business* Oriented Language

[6] Website: `http://users.ugent.be/~kdschutt/cobble/`.

4.1 The code

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.
2   FACTS SECTION.
      LUCKY-EID VALUE 7777.
4     LUCKY-EID VALUE 3141.
      *> etc.
```

This code reads as "Employees 7777, 3141, etc. got an unexpected bonus". 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):

```
1 DATA DIVISION.
  FILE SECTION.
3   FD EMPLOYEE-FILE.
    01 EMPLOYEE.
5      05 EID PIC 9(4).
       *> etc.
```

Lastly, from the log output we can figure out the name of the data item holding an employee's total end-of-year bonus. 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.
2   USE BEFORE ANY STATEMENT
    AND NAME OF RECEIVER EQUAL TO "BNS-EUR"
4   AND BIND LOC TO LOCATION
    AND IF EID EQUAL TO LUCKY-EID.
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-EOY` in every trace. Before moving on we would like to make sure we are 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.
2   FACTS SECTION.
      CULPRIT-LOCATION VALUE 666.
4     *> 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 figure this out 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.
  USE AFTER READ STATEMENT
  AND NAME OF FILE EQUAL TO "EMPLOYEE-FILE"
  AND BIND LOC TO LOCATION
  AND IF EID EQUAL TO LUCKY-EID.
 MY-ADVICE.
  SET FLAG-SET TO TRUE.
  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 find that `B31241` is part of the following business rule: it is a bonus an employee receives when he or she has sold at least 100 items of the product with number 31241. Apparently this product code had been assigned to a new product the year before. It was once associated to another product which had been discontinued for several years. The associated bonus was left behind in the code, and was never triggered until employees started selling the new product.

4.2 Evaluation: pro

AOP+LMP provided us with a flexible and powerful tool to perform our investigation. Dynamic aspects allow for easy inspection of the behaviour of applications by enabling smart tracing, verification of assumptions and mining of business logic. LMP adds to this the capability of recording and exploiting recovered knowledge.

4.3 Evaluation: contra

First of all, it is required (see section 3.3) that the Cobble weaver has been integrated into the build system. During the weaving process, every new aspect requires reweaving, recompiling, relinking and redeployment of the entire base sys-

```
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. 3. Aspect for basic procedure encapsulation.

tem, which so far comes at a higher cost than, for instance, debugging tools or instrumentation techniques such as DTRACE [4] or ATOM [13]. DTRACE for instance is a scriptable tracing system built into the (Solaris) operating system and is able to extract user-defined information from a running program.

5 Encapsulating procedures

In [12], 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 [11], 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 passed as a program argument. We will look at an aspect-based solution.

5.1 The code for the basic problem

Figure 3 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 liberally exploiting LMP features.

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 ambiguity 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 for every

possible solution to its pointcut, one after the other. However, the DISPATCHING advice will only get triggered when METHOD-NAME matches the name of the selected paragraph (extraction of this name happens 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 immediate 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.:

```
1 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 added to the base program 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 would be no big problem. We could simply define it within an aspect module, which, upon weaving, would augment the target program (modulo some alpha-renaming to prevent unintended name capture):

```
1 IDENTIFICATION DIVISION.
   ASPECT-ID. PROCEDURE-WRAPPING.
3
  DATA DIVISION.
5 LINKAGE SECTION.
    01 METHOD-NAME PIC X(30) VALUE SPACES.
```

From this, it becomes pretty obvious that METHOD-NAME will be used as an input parameter of the *base program*. The concept of a linkage section makes no sense for the external aspect module itself, as an aspect will never be called in such a way.

5.2 The code for the extended problem

The complexity of the problem increases when we consider the fact that paragraphs usually contain various variables used as in- and output. Sneed's tool takes care of

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

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

findall(
  [Name, Para, Wss],
  ( paragraph(Name, Para),
    slice(Para, Slice),
    wss(Slice, Wss)
  ),
  AllInOut
),

max_size(AllInOut, VirtualStorageSize),
{ 01 VSPACE PIC X(<VirtualStorageSize>). },

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

Fig. 4. Full procedure encapsulation (part one).

this in the following way:

> "*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.*" [11]

Put another way, one must find all data items on which the encapsulated procedures depend. These are then gathered in new records (one per procedure), all of which redefine the same "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, the code in figures 4 and 5 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

```
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. 5. Full procedure encapsulation (part 2).

were based on LMP embedded in AOP, this code is based on a generative programming approach, similar to that in [2]. The code can be read as follows. Anything enclosed in curly brackets ($\{\ldots\}$) is (aspect-)code which is to be generated. This can be further parameterized by placing variables between angle brackets ($<\ldots>$), which will get expanded during processing. Everything else is Prolog, used here to drive the aspect generation.

Let us apply this knowledge to the code in figures 4 and 5. Lines 1 and 2 on figure 4 declare the header of our aspect, while lines 4–6 define the linkage section as discussed before. Lines 8–15 calculate all base program slices [15] (`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). Then, 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 (on figure 5), the procedure division is put down, declaring the necessary parameters (line 28). We then generate advice similar to that in figure 3, 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).

5.3 Evaluation: pro

Despite the inherent complexity of the problem, AOP+LMP allowed us to write down our crosscutting concern in a generic way with certain ease. LMP was leveraged to define our aspect by reasoning over the program, while AOP was used to tackle the actual modification of the application. 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.

5.4 Evaluation: contra

The hard part of the above aspects 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 or Java equivalent of this: what does it mean to introduce new parameters on procedures or methods? More to the point, *should* we allow this?

The need for a generative approach gives a firm hint that we are touching on the boundaries of current AOP language technology here, but it is not yet clear whether this means that current language research has not advanced enough or that AOP does not lend itself to solve this problem.

6 Year 2000 syndrome

The Y2K-bug is probably the best-known example of unexpected change in legacy systems, somewhere ahead of the conversion to the Euro currency. 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 cross-cutting concern: whenever a date is accessed in some way, make sure the year is extended.

6.1 The code

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 [7] (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. In the latter case, C allows variables to be declared as instances of user-defined types, whereas in Cobol, variables have to be declared in terms of the same, low-level Cobol primitives (e.g. a sequence of ten digits). These user-defined types in C are most likely sufficiently modularized, allowing for a localized (non-AOP) solution.

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 (based on design information) for data items which hold dates, and that these have a uniform structure (in casu "yymmdd"). Then we might write something like:

```
A-YYMMDD-FIX SECTION RETURNING MY-DATE.
  USE AROUND SENDING-DATA-ITEM
  AND SENDING-DATA-ITEM IS DATE.
MY-ADVICE.
  MOVE PROCEED TO MY-DATE(3:8).
```

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

```
6   IF MY-DATE(3:4) GREATER THAN 50 THEN
      MOVE 19 TO MY-DATE(1:2)
8   ELSE
      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. [8] 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, e.g. using the generative approach of section 5.2.

The biggest problem lies in the weaving. When committed to a source-to-source approach, as we are with Cobble, weaving anything below the statement level [9] becomes impossible. As Cobol lacks the idea of functions [10], 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.

6.2 Evaluation: pro

If design information is available about the various date formats and if aspect weaving technology would be more mature, application-specific aspects could solve unexpected change problems like Y2K or the conversion to Euro.

6.3 Evaluation: contra

As illustrated by the first two problems of section 6.1, getting access to design information clearly is a prerequisite in weakly typed legacy environments. Even then, vendor dependence locks AOP implementations into specific language dialects, making the last two problems mentioned even worse. This is also one of the main reasons that only the case of section 3 has been put into practice. Other barriers encountered when building an aspect weaver for (some dialect of) Cobol, are its ambiguous grammar, the large variety of statement types and clauses, etc. In this respect, an AOP solution for C is more easily provided and supported.

Problem	OK?	How/Why?
Reverse-engineering	OK	(LMP in) AOP
Business rule mining	OK	LMP in AOP
Encapsulation of logic (basic)	OK	LMP in AOP
Encapsulation of logic (extended)	OK	AOP in LMP
Y2K38 (ANSI-C)	N/A	modular already
Y2K (Cobol)	NO	too weakly typed

Table 1
Summary of our findings.

7 Conclusion

Table 1 summarizes our findings. Briefly put, we discussed reverse engineering, 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 AOP. Encapsulation of procedures in Cobol, a typical legacy integration scenario, required a more generative approach embedding AOP in LMP as we clearly touched some boundary here.

As for the Y2K restructuring problem, the semantics of Cobol, especially its lack of typing, present too much of a limitation for an AOP solution. In C, the Y2K38 problem can still be managed reasonably, precisely because it does feature better typing support. Also, the limited number of C dialects makes it much easier to build a widely useful aspect weaver.

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. As C and Cobol are two ends of the procedural programming spectrum, other legacy languages will likely yield similar results.

8 Future Work

So far, only the dynamic analysis approach using aspects has been tried in practice. In order to perform more elaborate case studies in Cobol, a solution has to be found for the relative disparity between the various major Cobol dialects. Otherwise, one is tied to the applications written in the sole targeted dialect, both for experimental and real use. Likewise it will be needed to investigate other restructuring and in-

[8] Though not in this case, as the structure of the advice body *only* refers to the data item *after* the `PROCEED` statement.
[9] I.e. subexpressions like senders and receivers, etc.
[10] Functions can be written in later versions of Cobol. Our focus on legacy systems, however, rules these out for use here.

tegration problems, as well as the general need for generative AOP programming solutions. The question here would be to find when the generative approach is really required and when/how it can be avoided. Finally, another problem which merits attention is the heterogeinity of legacy systems, not only in the programming languages used, but also in the tools used to build the applications and other software development artifacts like database schemas, etc.

References

[1] B. Adams. AOP on the C-side. In *LATEr '06: Proceedings of the 2nd Linking Aspect Technology and Evolution Workshop*, Bonn, Germany, 2006.

[2] J. Brichau, K. Mens, and K. De Volder. Building composable aspect-specific languages with logic metaprogramming. In D. S. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering (GPCE)*, volume 2487 of *Lecture Notes in Computer Science*, pages 110–127. Springer, 2002.

[3] M. Bruntink, A. van Deursen, and T. Tourwé. An initial experiment in reverse engineering aspects. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, volume 00, pages 306–307. IEEE, 2004.

[4] B. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the USENIX Annual Technical Conference, General Track*, 2004.

[5] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. A qualitative comparison of three aspect mining techniques. In *IWPC '05: Proceedings of the 13th International Workshop on Program Comprehension*, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[6] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *CSMR '05: Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, pages 112–121. IEEE Computer Society, 2005.

[7] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[9] R. Lämmel and K. De Schutter. What does aspect-oriented programming mean to Cobol? In M. Mezini and P. L. Tarr, editors, *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 99–110. ACM, 2005.

[10] I. Michiels, T. D'Hondt, K. De Schutter, and G. Hoffman. Using dynamic aspects to distill business rules from legacy code. In R. Filman, M. Haupt, K. Mehner, and M. Mezini, editors, *DAW '04: Proceedings of the Dynamic Aspects Workshop*, pages 98–102, Mar. 2004.

[11] H. M. Sneed. Encapsulating legacy software for use in client/server systems. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering*, pages 104–120. IEEE, 1996.

[12] H. M. Sneed and S. H. Sneed. Creating web services from legacy host programs. In *WSE '03: Proceedings of the 5th International Workshop on Web Site Evolution*, pages 59–65. IEEE Computer Society, 2003.

[13] A. Srivastava and A. Eustace. ATOM — A system for building customized program analysis tools. In *PLDI '94: Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[14] K. van den Berg and J. M. Conejero. Disentangling crosscutting in AOSD: A conceptual framework. In *EIWAS '05: Proceedings of the European Interactive Workshop on Aspect-Orientation*, EIWAS 2005, Brussels, 2005.

[15] M. Weiser. Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[16] R. Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, 2001.

[17] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR '05: Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, pages 134–142. IEEE Computer Society, 2005.

[18] A. Zaidman, S. Demeyer, B. Adams, K. De Schutter, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and Aspect Orientation. In *CSMR '06: Proceedings of the 10th European Conference on Software Maintenance and Reengineering*, volume 0, pages 91–102, Los Alamitos, CA, USA, 2006. IEEE Computer Society.