AOP for Legacy Environments, a Case Study

Bram Adams, Kris De Schutter and Andy Zaidman

{kris.deschutter,bram.adams}@UGent.be
SEL, INTEC, Ghent University, Belgium

andy.zaidman@ua.ac.be
LORE, University of Antwerp, Belgium

ABSTRACT

We present a case study where we applied Aspect Orientation (AO) against an industrial, legacy, non Object Oriented application, in an effort to regain lost knowledge thereof. We start off by briefly discussing some of the problems which arise in such software, as well as how aspects might aid in alleviating them. We then discuss the implementation of an aspect language for C, aspicere, which is based on declarative pointcuts and meta information. Finally, we present the case study itself, as well as preliminary results of regained knowledge of the system.

Keywords

Aspect Oriented Programming, Legacy Software, C, Evolution, Case Study

1. INTRODUCTION

Legacy software is all-around: software that is still very much useful to an organization – quite often even *indispensable* – but a burden nevertheless. A burden because the adaptation, integration with newer technologies or simply maintenance to keep the software synchronized with the needs of the business, carries a cost that is too great. This burden can even be exaggerated when the original developers, experienced maintainers or up-to-date documentation are not available.

Apart from a status-quo scenario, in which the business has to adapt to the software, three scenarios are frequently occurring:

1. Often, in an attempt to limit the costs, the old application is ”wrapped” and becomes a component in a newer software system. In this scenario, the software still delivers its useful functionality, with the flexibility of a new environment. This works great and the fact that the old software is still present is slowly forgotten. This leads to a phenomenon which can be called the *black-box syndrome*: the old application, now component in the new system, is trusted for what it does, but nobody knows – or wants to know – what goes on internally (white box).
2. Another scenario can be a total rewrite of the application, from the legacy environment, to the desired one.
3. A third possibility is a mix of the previous two, in which the old application is seriously changed before being set-up as a component in the new environment.

For all three scenarios and certainly for scenarios two and three, the software engineer needs a good basic understanding of the application in order to start his/her reengineering operation. This is where we think that *Aspect Oriented Programming* (AOP) can make a difference. With the general ability to insert *advice* in all functions and procedures or the fine-grained injection of code with the help of precisely defined pointcuts, we are now able to perform dynamic analysis in a clean and efficient way.

At Ghent University, a framework has been developed to introduce AOP in legacy languages like Cobol [18] and C [2]. The latter is called *aspicere*[1]. This paper applies it on an industrial case study, provided by one of our partners in the ARRIBA (Architectural Resources for the Restructuring and Integration of Business Applications) research-project[2].

2. THE CONCEPT

2.1 Legacy code and AOP

AO is a relatively new paradigm, which has grown from the limitations of Object Orientation (OO) [13], and a fortiori those of older paradigms. OO takes an object-centric view to software development, where a programmer describes objects, how they should behave and in what ways they should interact with other objects.

When faced with crosscutting concerns, i.e. concerns which don't walk nicely along the lines set out by inheritance, association, etc., OO degrades to the procedural programming style it ought to replace. When embracing AOP however, aspects manage to contain this extra complexity in a module of its own, and to reroute any interaction with the interface exposed by the base program[3] to this module and back. More formally, aspects allow us to *quantify* (through *pointcuts*) which events in the flow of a program (*join points*) interest us, and what we would have to happen at those points (*advice*). Hence we can ‘describe’ what some concern means to an application and have the *aspect-weaver* take care of the hard and repetitive bits for us.

Note however that, although AO has rarely been applied outside of OO, none of the commonly known crosscutting concerns are inherently tied to Object Oriented environments. Yet there seems to be more than enough reason to backport AO's ideas to programming languages and paradigms which had been given up for high tech language research. We mustn't forget that while the academic world lives at the edge of knowledge, their ideas and formalisms (once they've matured enough) get adapted by the industry who expect their new expensive acquisition to serve them for many years.

[1] ”aspicere” is a Latin verb and means ”to look at”. Its past participle is ”aspectus”, so the link with AOP is pretty clear.

[2] Sponsored by the IWT, Flanders

[3] A rather fuzzy concept, as there is no widely accepted definition for it yet. This is a consequence of the relatively young age of AOP and its rather pragmatic development (ie. there is no all-encompassing theorethical foundation). It has now developed into a hot research topic [14, 3, 23].

Conversely, this also means that they're stuck with a huge portfolio of legacy technologies they can't afford to just give up, but which suffer from the same symptoms regarding crosscutting concerns as, say, the relatively younger Java applications.

Knowing this, AOP in the context of Cobol, C, PL/1, ... really makes sense. Aspects might be used for such basic things as patching of existing software, as well as more complex things which aid in the integration of business applications. Aspects could be used for mining business rules [19] and program logic in order to refactor legacy systems to more sound architectures. They can be useful for enforcing good coding practices. E.g. you could prohibit programmers from calling other modules' internal functions. Performing automatic checks for memory leaks, logging, etc. through aspects also frees up the programmer's mind for the really interesting concerns. There are lots of opportunities here. And the non-intrusive nature of aspects makes all this ever more attractive.

As an added bonus, by seeing how AO lives and breathes inside these legacy environments we might gain a better understanding of what Aspect Orientation really is all about.

As part of our research for AOP in legacy systems, we have therefore developed a framework capable of extending procedural languages like Cobol and C with AOP-constructs. As cobble has been described and situated in this framework in [18], we spend this paper on its sibling aspicere and an industrial case study on which we applied it.

2.2 The case study

The industrial partner that we cooperated with in the context of this research experiment is Kava[4]. Kava is a non-profit organization that groups over a thousand Flemish pharmacists. While originally safeguarding the interests of the pharmaceutical profession, it has evolved into a full fledged service-oriented provider. Among the services they offer is a tarification service —determining the price of medication based on the patient's medical insurance. As such they act as a financial and administrative go-between between the pharmacists and the national healthcare insurance institutions.

Kava was among the first to realize the need for an automated tarification process, and have taken it on themselves to deliver this to their members. Some 10 years ago, they developed a suite of applications written in non-ANSI C for this purpose. Due to successive healthcare regulation changes and forthcoming changes in the dataflow, they feel the necessity to reengineer their applications.

Kava has just finished the process of porting their application to a fully ANSI-C compliant version, running on Linux. Over the course of this migration effort it was noted that documentation of the application was outdated. This provided us with the perfect opportunity to undertake the following re-documentation experiment. Techniques akin to search engines like Google™, are able to detect tight couplings between components. This mined knowledge could then be validated with the original design as known/understood by the programmers. We'll show that to collect the needed data in an unobtrusive way, AOP fits perfectly.

What's more, we also found some nasty residues left behind by the original non-ANSI C implementation, on which we will elaborate in section 3.1 and 6.1.

2.3 Plan of the paper

We will start off with a short overview of our aspect language for C, codenamed aspicere. It is not our intention to give an authorative overview of its design, but rather to give the reader a feel for the expressivity and complexity of the language. (If you're interested to learn more we refer you to [2] for aspicere and [18] for cobble, as well as the websites[5] for these projects.). We then describe the transformation framework which is the driving force underlying our aspect languages. Next, the techniques used to measure program couplings will be described as well as their application at Kava. We also show our results and findings from the case study itself, and close with some concluding remarks.

3. ASPICERE

In [5], the authors proposed a language extension for C to provide AOP-functionality. Their work was based on AspectJ, still the best known AOP-implementation, because a legacy language like C can be regarded as some sort of subset of a modern OO-language. Practical tools for this newly designed aspect language have been hard to find and, even then, difficult to master.

As there exists a large C codebase and C still is a very popular language (Unix/Linux/BSD kernels, Open Source software, ...), we decided to turn our efforts and tools to C. During a Master's thesis [24] a first prototype of aspicere[6] got developed, using a very limited subset of AspectC. Although it turned out to be useful for some purposes, the AspectC-like pointcut language seemed too restrictive. In [2], the authors illustrate this bold statement and propose a pointcut language based on Prolog (inspired by the work of [11]). In fact, this turns out to be a generalization of the approach taken in cobble.

3.1 An Aspect for Safe String Handling

We'll present aspicere's syntax and its features alongside the following task, a residue of Kava's recent migration to the Linux-platform:

> *The C-libraries on the old system were very fault-tolerant with regard to string handling. When atoi() was passed a null pointer, the particular implementation of this standard function didn't throw up a segmentation fault, but gracely returned 0. This behaviour was the compiler vendor's own decision, and not illegal as his system was not ANSI-compliant. When converting to Linux and its GCC-compiler, suddenly the safety net around strcpy(), atoi(), ... disappear, resulting in random segmentation faults.*

How to solve this problem?

- Searching all infected functions and surrounding them with proper null pointer detection code. This is the most tiresome and error-prone approach, producing very tangled code.
- Redirecting the original call to new, custom wrapper functions. This still requires the manual inclusion of the right header files and serious modification of makefiles.
- By cunningly declaring the relevant function names as macros mapped to the names of corresponding wrapper functions, and expanding them right before every compilation, the second solution is emulated somehow in an easier way (the header and makefile remark still hold in theory). Biggest drawback is the extremely untransparent nature of this approach: if it is not documented in a very attention-seeking way, then this mechanism will pass on undetected.
- Identifying this concern as a crosscutting one and encapsulating it into an aspect, yields a better solution: equally modularized as the previous one, yet much more transparent and self-documenting. Yes, the fault tolerance becomes a proper

[4]http://www.kava.be/

[5]aspicere: http://allserv.ugent.be/~kdschutt/aspicere/
cobble: http://allserv.ugent.be/~kdschutt/cobble/

[6]http://allserv.ugent.be/~kdschutt/aspicere/

part of the application instead of being enforced by a particular compiler configuration. We'll illustrate (a part of) aspicere 's capabilities while elaborating some more on this issue.

Although this is a classic AOP example, it's interesting to know on which solutions one has to fall back in C systems without AOP and to face them with the intriguing simplicity of AOP's approach.

The structure of an ordinary aspect looks like this:

1. inclusion of header files
2. declarations of static/global variables
3. auxiliary method declarations and definitions
4. various advice to do the crosscutting work

An aspect is just an ordinary C compilation unit with variables, methods, ..., except for the advice structures. What's the scope of the variables and plain methods? Static ones hide themselves from everything outside the enclosing aspect. The others are in fact visible from all other modules and can be interpreted as application-level InterType Declarations (ITD) in AspectJ. They are injected into the global namespace, which requires careful thinking to avoid name collision or unwanted exposure of data and/or behavior. More fine-grained ITD is one of our future plans.

The blueprint of our various advices is pretty simple. Surround calls to infected methods by a null pointer check and only if everything is alright, the originally called method should be invoked.

```
ReturnType around safe_ato (Src,ReturnType) on (Jp):
 call(Jp,"ato.",[Src])
  && type(Jp,ReturnType) {
   ReturnType dst;

   if (Src == NULL) {
     dst = 0; /* compiler does the cast */
   } else {
     dst = proceed();
   }

   return dst;
}
```

Here, we defined an advice called *safe_ato*. It catches calls to all *ato.*()-functions[7], then there's a check on null pointers. If everything is safe, the original call to *ato.*() proceeds, otherwise zero is returned. This perfectly illustrates our binding mechanism[8]. Indeed, thanks to the type variable "ReturnType", we only need to write down one advice to check three functions. Another use of bindings is the capturing of function call arguments like *Src* does. It's perfectly possible to alter *Src* before doing the **proceed**()-call, although the latter notation doesn't make this clear. In 4.4, we will see that our weaver treats these bindings as weave-time macro's.

Looking a bit closer to the syntax of said advice, we see that an (around[9]) advice definition resembles that of a method quite a lot. Indeed, we see:

return type Just as in AspectJ, around-advice must have the same return type as the advised method, because its semantics dictate to surround the advised join point entirely. As illustrated, aspicere allows a logic variable called a binding as return type, yielding a kind of template functionality.

name This is rather unnecessary at the moment (except for documenting purposes), because advice is never called directly (which is in fact the most discriminating characteristic of advice compared to ordinary methods). However, some people have argued in favor of turning advice into real methods and mapping them to the right join points using a binding specification construct [20], effectively yielding more loose coupling between advice and pointcut. We hope to examine this later on.

parameters These are in fact the externally visible bindings of the current pointcut definition. They are comparable to parameters of a real method, but they disappear after weaving. They aren't typed, because typing constraints can be incorporated separately as Prolog predicates in the pointcut definition. Note that the names of these bindings as well as those of the internal ones, all start with a capital letter, in line with common Prolog conventions.

body Cf. methods, but bindings can occur and there are also two special functions available. *proceed*() continues execution of the original, advised join point, and join point context can be retrieved through *this_joinpoint*().

The biggest differences lie in the **on**-clause and in the pointcut definition. In the former, we make the name of the chosen join point explicit, because it's in fact a variable. As stated before, the latter captures those points in the base program that should be advised. It's important to realise that each pointcut expression really corresponds to one Prolog rule and that all primitive pointcuts like *call* are predicates too instead of fixed keywords. So, there's an implicit correspondence between the composition of a pointcut expression using "||", "&&" and "!", and a genuine Prolog predicate. We'll illustrate this in 3.3.

The examples we show are representative for the general predicates aspicere offers, in the sense that they're currently all static: they only work if we can perfectly say that a join point will match a certain PCD. This means that there aren't any residual checks left in the woven code to find out whether advice is applicable to a join point. Dynamic pointcuts like *cflow* aren't excluded a priori from aspicere, but they are part of our future work.

3.2 An Aspect for database recovery

To illustrate Prolog's ability to build predicates for navigating through the static structure of a base program, we'll tackle the next problem (unrelated to the Kava case):

Consider an application which accesses a database. We'd like to catch database errors in such a way that, depending on the specific error code, failed SQL-queries are retried a number of times. This recovery procedure should only be active if the error occurs in an important method.

This concern could be handled with AOP in two ways:

- We could write down one advice with in its PCD a disjunction of all possible (important) enclosing methods and in its body a giant switch-case construct to select the right number of recovery attempts. This is not a satisfiable solution, as important metadata (error code and corresponding number of retry attempts) is hidden in the advice body and can get out of date.
- As the concept of recovery is the same across all error codes and only depends on some metadata, we can easily write this meta-knowledge down in Prolog facts and use these in our PCDs. Even better, we could write Prolog rules which (thanks to the Java-nature of our Prolog engine, see 4.4) can access at weave-time any data source to find all relevant error

[7]The "." shows that we allow full regular expression support.

[8]Of course we also advise methods like *atok*(), *atom*(), ... if these exist. This can be restricted by using Prolog facts as metadata in a pointcut definition (PCD; see 3.2).

[9]**before** and **after** can easily be emulated by putting the call to **proceed**() at the beginning or at the end of the advice body.

codes and the appropriate number of recovery attempts. So, using Prolog, we can make use of modularized metadata in our PCDs.

This gives us the following advice for the current problem:

```
void around recover(ErrNr, Retry, ErrStr) on (Jp):
 call(Jp,"_iqcftch",_)
 && critical_call(Jp)
 && sql_redo(ErrNr,Retry)
 && sql_code(ErrNr,ErrDescr)
 && stringify(ErrDescr,ErrStr){
  /* Check if SQL-engine returned error with ID
   ErrNr and retry at most Retry times.   */
}
```

The extra metadata can be defined as follows:

```
sql_code(-666,'Time limit exceeded.').
/* some other codes with their description */

sql_redo(-666,2).
/* some other codes with the relevant number of
 retry attempts */

critical_method('monthlyPayment').
critical_call(Jp) :-
  enclosingMethod(Jp,EncMethod),
  name(EncMethod, Name),
  critical_method(Name).
```

The descriptions could just as easily be fetched from the database vendor's website by rewriting the first list of predicates into a more complex rule. Note that this metadata approach is very compatible with the XPI-theory of [23].

3.3 Duality of PCDs

To end our discussion of aspicere, we just want to show that we could equally have written our last advice as:

```
void around recover(ErrNr, Retry, ErrStr) on (Jp):
 recoverableSQLCall(Jp,ErrNr,Retry,ErrStr)
   ...
```

with the following Prolog predicate defined in a separate module:

```
recoverableSQLCall(Jp,ErrNr,Retry,ErrStr):-
   call(Jp,"_iqcftch",_),
   critical_call(Jp),
   sql_redo(ErrNr,Retry),
   sql_code(ErrNr,ErrDescr),
   stringify(ErrDescr,ErrStr).
```

This is exactly what happens behind the scenes of our prototype weaver.

4. THE FRAMEWORK

4.1 Up-front considerations

According to [10], we face two generic obstacles when trying to implement AOP support for a legacy language. First, we need to get a handle on a front-end for the relevant legacy language. For C, this is not that impossible, but it is indeed a major challenge in the case of Cobol, as has been argued elsewhere [17]. 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). This will be the topic of this section.

The framework, codenamed *Yerna Lindale*[10], operates at the level of source code and uses XML for the internal representation of that code's AST. As it is generally accepted that programs can be represented efficiently as trees, XML seems a natural fit for this because of its inherent navigability and the myriad of existing and emerging technologies like XPath, XSLT, XQuery, ... The instantiation of Yerna Lindale in the context of aspicere is illustrated in Figure 1.

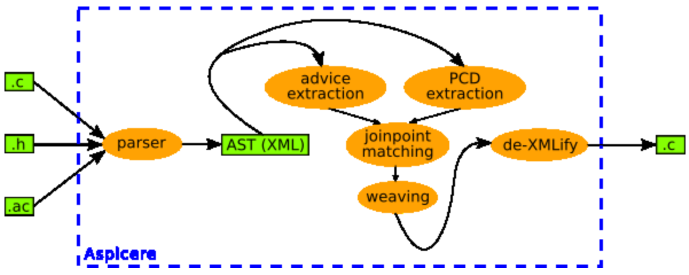

Figure 1: The weaving process.

As we don't know of the existence or feasibility of any system capable of extracting the AST-representation of an executable (possibly at run-time), transforming it and feeding it back, it's obvious that our approach is tied to a static weaver (the tranformation engine). But aren't there better alternatives?

Weaving compiled code implies commitment to a specific vendor including dialect and object format. Language-independent load-time weavers [16] (for .NET, Java or otherwise) are challenged by the distance between byte- or machine code and aspicere's or cobble's pointcut descriptions. Language-independent weavers at the source-code level, such as SourceWeave.NET [12], require specific language front-ends that appeal to the underlying source-code models (i.e., CodeDOM for SourceWeave.NET). Recently, some aspect languages for C emerged (Arachne [6] and TOSKANA [7]) featuring run-time weaving. Although this dynamic (de)weaving opens lots of opportunities, they encounter problems inherent to today's executable formats and runtime environments. In compiled code, a lot of useful implementation is stripped, so that it's hard to define more complex pointcut expressions. There's also a performance penalty, as a running executable can't yet be recompiled together with the advice to perform (compile-time) optimizations. The people of TOSKANA now try to solve these issues by turning to a virtual machine [8]. Conclusion: our approach is not redundant.

The use of XML for the intermediate representation of source code is not uncommon either; it is practised for 'normal' languages (such as Java [4] and C [26]) and also for languages with a weaving semantics [9, 21]. The important advantage of this approach is that standard APIs and tools for XML processing can be readily used. This buys us great flexibility and opportunities for experimentation with different technologies. 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 [22]. In our prototype, we currently neglect these issues. We can report that the ratio *XML format to concrete syntax* (both in text representation) lies typically around 12 to 13 with maximum values below 30 and above 6, which is still quite tractable. We haven't obfuscated the XML-tags, so we have some leeway there.

4.2 Front-end Setup

Development of the front-end for aspicere (first step in Figure 1) started after that of cobble as a proof of concept of its weaving ar-

[10]Quenya (High Elvish) for '*old music*' — this for fans of J. R. R. Tolkien.

```
functionDef {...}:
  (
   (functionDeclSpecifiers) => ds:functionDeclSpecifiers
   |  //epsilon
  )
  declName = d:declarator[true] {...}
  ( decl:declaration {...} )*
  ( VARARGS )?
  ( SEMI )* {...}
  com:compoundStatement[declName] {...}
  ;
```

Figure 2: Grammar fragment for function definitions.

chitecture. As is explained in [18], it's nearly impossible to write out a correct, unambiguous grammar for Cobol, and then there's still the fact that there are dozens of similar dialects out there. For this reason, we decided to use an ambiguous grammar together with btyacc, a backtracking version of the better-known yacc. This way, any ambiguities are resolved by running over all possible alternatives until a rule is satisfied. In line with AspectJ, we added AOP constructs directly to Cobol (better: its grammar).

For aspicere, the same strategy was followed. The base of the grammar was found online[11], and was subsequently transformed to comply to the format used by some other necessary tools [18]. Just as with cobble, AOP-extensions were added to the grammar and then we generated a backtracking parser using btyacc.

During the experiment at Kava, it became clear that the flexibility of our grammars had some serious downsides too:

- When parsing complicated statements like moderately-sized switch-case constructs, speed proved to be a real bottleneck. Apparently, there is too much ambiguity, which results into massive backtracking (up to several hours).
- Because C code can be very cryptic and tangled at times (as opposed to the very verbose Cobol style), the language itself gives rise to such a degree of ambiguity that the parser eventually produces wrong output (XML). This is not to say that we get complete nonsense, but part of our XML just leads to unreliable results.

Fortunately, we were experimenting at that time with an ANTLR-parser found at `http://www.antlr.org/grammar/cgram`. Because it is a non-backtracking parser, we get much faster parsing times. The downside is that this approach is not amenable to cobble, as Cobol is scattered across dozens of dialects, of which none can be described in a traditional yacc- or ANTLR-manner. Similarly, we had to be much more attentive on integrating AOP-constructs in it. Now the parser does its job in less than half a minute in all cases. Figure 2 shows a small part of our grammar.

As mentioned before, the output of the parser is an XML-file representing the abstract syntax tree of the parsed source file. In fact, we go through an expensive text representation of XML, which could be avoided indeed, using a tool-to-tool XML API [22]. As our ANTLR-parser is written in Java, this could now be much easier to do.
The XML representation encodes all details of layout and comments, which is less important for AOP, since a user is not supposed to study the woven code. It is important however when using the framework for re-engineering transformations and it also simplifies our unparser (step 3 in Figure 1), which maps XML data to concrete Cobol or C syntax. In Figure 3, we illustrate the XML representation corresponding to the grammar rule of Figure 2.

So, it's clear that Yerna Lindale doesn't rely on one particular type of parsers. As long as we have a parser which can generate XML dumps of its ASTs, our framework will be able to handle things. That's why aspicere's ANTLR-parser could be easily fit in.

[11] `http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`

```
<NFunctionDef>
  <NFunctionDeclarationSpecifiers>
    <NTypeSpecifier>
      <string>int</string><!--   -->
    </NTypeSpecifier>
  </NFunctionDeclarationSpecifiers>
  <NDeclarator>
    <NIdentifier>
      <string>main</string><!--   -->
    </NIdentifier>
    <string>(</string><!--   -->
    <NParameterDeclaration>
      <NDeclarationSpecifiers>
        <NTypeSpecifier>
          <string>void</string><!--   -->
        </NTypeSpecifier>
      </NDeclarationSpecifiers>
    </NParameterDeclaration>
    <string>)</string><!--   -->
  </NDeclarator>
  <NCompoundStatement>
    <string>{</string><!--   -->
      ...
    <string>}</string><!--   -->
  </NCompoundStatement>
</NFunctionDef>
```

Figure 3: XML element for int main(void){ ... }

4.3 XML-based source-code manipulation

The second obstacle for AOP support [10] concerns the physical weaving framework. Basically, the weaver processes XML via DOM, and generates new XML (step 2 in Figure 1). It is pretty straightforward to locate pointcuts, advice and potential join-point shadows in a DOM tree. For instance, to find a specific function definition in a DOM tree, the following XPath expression does the job[12]:

```
//NFunctionDef[
  string(NDeclarator/NIdentifier/string)='main']
```

We have to note that the XML format is tied to a certain degree to the grammar used. This means that all functionality operating on the XML representation does not resist grammar changes. Also, we cannot immediately serve multiple language-dialects. One solution would be to follow a model-driven approach, where the grammar structure is mapped to a more abstract format. We expect that existing work on language-independent source-level weavers [12], and, more generally, on language implementation will be of use in this context. Alternatively, one could build aspicere into GCC 4.0 and alleviate its new language-independent intermediate representation [1].

4.4 Technology details for the weaver

Now, we'll look a bit deeper into the details of the weaving itself. Looking at Figure 1, we see a high-level overview of the whole process. Required inputs are one source code file at a time together with all the considered aspects. Because we need preprocessed source files to begin with, either the input files should be preprocessed already or aspicere attempts to do this implicitly before parsing (not shown on Figure 1). After applying aspicere, we get a woven source code file which can then be fed to GCC[13].

[12] For the moment, XPath alone suffices for our purposes, but we're well aware that XQuery is much more powerful.

[13] As a matter of fact, aspicere is largely platform-independent by using Java-based languages in its implementation. Some minor plug-and-play components are written for Unix/Linux platforms -

The first steps of the weaving process consist of looking up all available advices and separating their bodies from their PCDs. As illustrated, these actions are straightforward using XPath. As a matter of fact we use a full-blown XSLT-transformation for the PCD-part, as we transform them into Prolog rules. That's because as argued in [2], aspicere features a Prolog-based pointcut language in which a PCD is really just a syntactically more polished way to write down a genuine Prolog rule. This is illustrated in 3.3.

The next step in our weaver immediately makes use of the Prolog features to match advised join points in the base program. Here, we opted for backward chaining. This means that instead of first instantiating all join points as facts to check for each one in turn which advices apply (rather imperative way of thinking), we query each advice to know which join points are affected. Once we have found all mappings, we need to transfer them from the Prolog engine to the main component of the weaver.

As a sidenote, to allow easy integration of a large collection of technologies or formalisms (Java, Prolog, shell scripts, . . .), we built all our tools on top of a custom integration framework for Java-based technologies, called lillambi[14]. Its concept is illustrated in Figure 4. The main component booting everything up is a script written in BeanShell[15], which offers more flexibility than the exclusive use of compiled Java. Using a system of agents with associated properties, all sorts of technologies can be combined easily, like the Prolog component for matching pointcuts to join point shadows (TuProlog: [16]). As all languages are Java-based, all data being processed is in the form of Java objects, and communication between components requires no exotic technology.

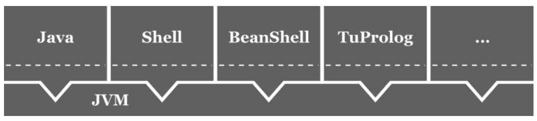

Figure 4: Component integration framework.

Returning to the main discussion, we now have everything at our disposal to start the physical weaving process. For every join point, all its advices are woven by a weaver assigned to the current join point type. Call join points get another weaver than execution join points[17]. Basically, each advice is transformed into a set of regular functions (one instance per set of bindings), and each aspect into a regular C module with its own static and global variables. The woven base program then contains the necessary residue code. When compiling it, the transformed aspects need to be linked in. The actual weavers are implemented in Java using XML APIs for DOM and XPath, as weaving merely boils down to restructuring of a DOM tree.

One of the more interesting features of the weaving process (and the Prolog connection), is the expansion of logic variables in the advice code. As we have seen in section 3, we can expose (bind) logic variables used in the PCD of an advice, and integrate them at the "right" places in the advice body (cf. macro or C++ template). By "right", we mean that as these variables are *not* typed (the usual case in Prolog), their usage should be considered wisely. This feature makes up for C's lack of decent type relationships (other than **`typedef`**), reflection or any other metaprogramming tricks. As such, it's a necessity to be able to write sufficiently generic advices.

our test platforms - but these could easily be rewritten (and enabled).

[14]This means "multiple languages" in Quenya.

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

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

[17]Currently, our prototype only handles call join points, but this is only temporary.

5. REVERSE ENGINEERING TECHNIQUES

Maintaining an application requires a certain level of understanding of the system under consideration. Studies have shown that attaining this level of understanding can take up to 50% of the time budget [25]. As such, a technique that points to those parts of the system that are critical – i.e. need to be understood – can be very helpful when trying to familiarize oneself with an application. The technique presented in this section tries to alleviate the program comprehension problem.

5.1 Webmining

The basis for the technique we used for this case study is the measurement of runtime coupling between modules of a system. To overcome the typical problem of coupling measures – each measure is between two classes or modules – we add webmining techniques to make sure that not only coupling between two separate modules is taken into account, but also a transitive measure for coupling is used for determining the most important modules of a system [25].

In datamining, many successful techniques have been developed to analyze the structure of the web. Typically, these methods consider the Internet as a large graph, in which, based solely on the hyperlink structure, important web pages can be identified. In this section we show how to apply these webmining techniques to a call graph of a program, in order to uncover important classes.

Based on the call graph of an execution trace of the application, the HITS webmining algorithm [15] allows us to identify so-called *hubs* and *authorities*. Intuitively, on the one hand, hubs are pages that refer to pages containing information rather than being informative themselves. Standard examples include web directories, lists of personal pages, ... On the other hand, a page is called an authority if it contains useful information.

The recursive relation between authority and hubiness is captured by the following formulas.

$$h_i = \sum_{i \to j} w[i,j] \cdot a_j \tag{1}$$

$$a_j = \sum_{i \to j} w[i,j] \cdot h_i \tag{2}$$

The algorithm starts with initializing all h's and a's to 1, and repeatedly updates the values for all pages, using the formulas (1) and (2). If after each update the values are normalized, this algorithm is known to converge to stable sets of authority and hub weights.

From previous case studies in the context of Object Oriented systems, we've learned that the classes that are catalogued as "hubs" by the algorithm are the most critical components of the system and are thus excellent candidates for early program understanding.

Example Consider the webgraph given in Figure 5. Table 1 shows three iteration steps of the hub and authority scores (represented by tuples *(H,A)*) for each of the five nodes from Figure 5. From this, we can conclude that 2 and 3 will be good authorities, whereas 4 and 5 will be good hubs, while 1 will be a less good one.

5.2 Expected results

The result from applying the technique is an ordered list of classes or modules, ranked according to their hubiness score. Taking the best hubs from this ordered list, should give you the most critical

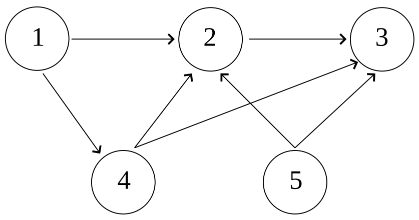

Figure 5: Example web-graph

		Nodes				
		1	2	3	4	5
Iterations	1	(1,1)	(1,1)	(1,1)	(1,1)	(1,1)
	2	(2,0)	(1,3)	(0,3)	(2,1)	(2,0)
	3	(4,0)	(3,8)	(0,5)	(6,2)	(6,0)
	4	...	...	...	...	...

Table 1: Example of the iterative nature of the HITS algorithm

components of the system and as such the modules that need to be understood first.

5.3 Setup of the experiment

To compute hubiness and authority scores, some tools have been developed at the University of Antwerp. Given a detailed execution trace of an application as input, they build up the corresponding call graph and resolve the coupling metrics fairly quickly. To get this trace in an unobtrusive way, is where aspicere comes in[18]. One of the two tracing advices (the other one is needed for **void**-methods) is shown here:

```
RetType around tracing (RetType,FileStr) on (Jp):
 call(Jp,"^(?!.*printf$|.*scanf$).*$")
 && type(Jp,RetType)
 && !str_matches("void",RetType)
 && logfile(FileName)
 && stringify(FileName,FileStr) {
  FILE* fp=fopen(FileStr,"a");
  RetType i;

  fprintf (fp,"before ( %s in %s ) \ n",
   Jp->functionName,Jp->fileName);
  fflush(fp);
  i = proceed ();
  fprintf (fp,"after ( %s in %s ) \ n",
   Jp->functionName,Jp->fileName);
  fclose(fp);

  return i;
}
```

Note that the constant opening, flushing and closing of files is not optimal. Normally, as aspects are transformed into plain compilation modules and advice into ordinary methods of those modules, we could get hold of a static file pointer and use this throughout the whole program. However, as will be explained in 6.1, this would mean we had to revise the whole make-hierarchy to link these uniques modules in. Instead, we added a "legacy" mode to our weaver in which advice is transformed to methods of the modules part of the advised base program. This way, the make-architecture remains untouched, but we lose the power of static variables and methods.

[18]Don't forget the advice of section 3.1 either.

Module	Hubiness score
voorschrift_scherm.c	1.000000
ua.ec	0.838293
kies_apot.c	0.714017
scroll_form.c	0.698913

Table 2: The highest 13% scoring modules according to hubiness.

6. RESULTS

Table 2 shows the results of applying the complete process on a small, but representative application of the Kava application suite, the so-called "ua" application, that allows for the administration of medical prescriptions. The application consists of 30 modules and Table 2 shows those 4 modules that are likely essential in the program understanding process, as these 4 modules are the best hubs. The scores in the second column, are relative hubiness values. *voorschrift_scherm.c* is thus the module that has the most outgoing calls, and the score for *ua.ec* is relative to that of *voorschrift_scherm.c*.

Granted, we have – as yet – not validated these results with the developers. Because we are in the process of conducting a larger-scale experiment at Kava, we didn't want to put a bias on the developers, by presenting them with these preliminary results. As such, for the moment we trust on the earlier validation of the webmining technique from previous case studies [25] [19].

6.1 Difficulties

As alluded to in 2.2, some remains of the non-ANSI implementation are still visible in the new system. In non-ANSI C, method declarations with empty argument list are allowed. Actual declaration of their arguments is postponed to the corresponding method definitions. As is the case with ellipsis-carrying methods, discovery of the proper argument types must happen from their calling context. Because this type-inferencing is rather complex, it's not fully integrated yet in aspicere. Instead of ignoring the whole base program, we chose to "skip" bad join points, introducing some errors in our measurements. To be more precise, we advised 367 files, of which 125 contained skipped join points (one third). Of the 57015 discovered join points, there were 2362 filtered out, or a minor 4 percent.

While preparing the case study, we faced some other problems as well. To fit aspicere into the existing build process, we had to adapt the existing Makefiles. As these were properly structured, most of this work could be automated. Eventually, manual inspection was needed to verify some minor peculiarities. Having a weaver direcly inside GCC would surely make these things easier.

Finally, the efficiency of our weaving process isn't optimal yet. While a normal build of the system takes approximtely 10 to 20 minutes, the integration of our weaver extends this to 17 hours and 38 minutes! Most of this time is spent on the join point matching, an area which needs (and still allows) some serious improvements. Of course, as aspicere is a preprocessor to GCC, the weaving time will always take longer than the original build process.

7. CONCLUDING REMARKS

We have shown that a declarative approach to the weaving of legacy applications based on an intermediate XML representation is a very

[19]We expect to have the complete set of results and the thorough validation of these results within two months.

flexible and manageable solution. The main concepts in AO (quantification and obliviousness) remain valid within the context of non-OO languages. The differences seem to be limited to the *kind* of events in the flow of an application an aspect may be applied to.

We also argued that in order to make advice as generic as possible, Aspect Oriented extensions to environments which lack reflective capabilities need to provide some form of metaprogramming.

The webmining reengineering solution we chose, allowed us to identify four modules from the preliminary case study (from a total of thirty) that are important from a program comprehension point of view. A validation with the original developers and maintainers of the software is scheduled for the very near future.

8. ACKNOWLEDGEMENTS

We would like to thank Kava for their cooperation and very generous support.
Kris De Schutter and Andy Zaidman 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. Bram Adams is supported by a BOF grant from Ghent University.

9. REFERENCES

[1] Bram Adams. Language-independent aspect weaving. 2005. Extended abstract, GTTSE '05 Summer School (Braga).

[2] Bram Adams and Tom Tourwé. Aspect Orientation for C: Express yourself. In *3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT), AOSD 2005*, 2005.

[3] Jonathan Aldrich. Open modules: modular reasoning about advice. 2005. Submitted for publication.

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

[5] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. 26(5):88–98, 2001.

[6] Rémi Douence, Thomas Fritz, Nicolas Loriant, Jean-Marc Menaud, Marc Ségura-Devillechaise, and Mario Südholt. An expressive aspect language for system applications with Arachne. In *AOSD '05*, pages 27–38, New York, NY, USA, 2005. ACM Press.

[7] Michael Engel and Bernd Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In *AOSD '05*, pages 51–62, New York, NY, USA, 2005. ACM Press.

[8] Michael Engel and Bernd Freisleben. Using a LowLevel Virtual Machine to improve dynamic aspect support in operating system kernels. In *4th AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS), AOSD*, 2005.

[9] Jeff Gray, Ted Bapty, Sandeep Neema, and James Tuck. Handling crosscutting constraints in domain-specific modeling. *Commun. ACM*, 44(10):87–93, 2001.

[10] Jeff Gray and Suman Roychoudhury. A technique for constructing aspect weavers using a program transformation engine. In *AOSD '04*, pages 36–45, New York, NY, USA, 2004. ACM Press.

[11] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03*, pages 60–69, New York, NY, USA, 2003. ACM Press.

[12] Andrew Jackson and Siobhán Clarke. SourceWeave.NET: Source-level cross-language Aspect Oriented Programming. 2004.

[13] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect oriented programming. In *Proceedings ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[14] Gregor Kiczales and Mira Mezini. Aspect Oriented Programming and modular reasoning. In *ICSE '05*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[15] Jon M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[16] Donal Lafferty and Vinny Cahill. Language-independent Aspect Oriented Programming. In *OOPSLA '03*, pages 1–12, New York, NY, USA, 2003. ACM Press.

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

[18] Ralf Lämmel and Kris De Schutter. What does Aspect Oriented Programming mean to Cobol? In *AOSD '05*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[19] Isabel Michiels, Kris De Schutter, Theo D'Hondt, and Ghislain Hoffman. Using dynamic aspect to extract business rules from legacy code. In *Online proceedings of Dynamic Aspects Workshop at AOSD*, 2004. `http://aosd.net/2005/workshops/daw/`.

[20] Istvan Nagy, Lodewijk Bergmans, Gurcan Gulesir, Pascal Durr, and Mehmet Aksit. Generic, property based queries for evolvable weaving specifications. In *3rd Software-Engineering Properties of Languages and Aspect Technologies Workshop*, 2005.

[21] Stefan Schonger, Elke Pulvermüller, and Stefan Sarstedt. Aspect Oriented Programming and component weaving: Using XML representations of abstract syntax trees. In *2nd German GI Workshop on Aspect Oriented Software Development*, pages 59 – 64. University of Bonn, 2002. Technical Report No. IAI-TR-2002-1, Rheinische Friedrich-Wilhelms-Universität Bonn, Institut für Informatik III.

[22] Susan Elliott Sim. Next generation data interchange: Tool-to-tool application program interfaces. In *WCRE*, pages 278–280, 2000.

[23] Kevin Sullivan, William Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. On the criteria to be used in decomposing systems into aspects. In *ESEC/FSE 2005, to appear*, 2005.

[24] Stijn Van Wonterghem. Aspect-oriëntatie bij procedurele programmeertalen, zoals C. Master's thesis, Ghent University, 2004.

[25] Andy Zaidman, Toon Calders, Serge Demeyer, and Jan Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR 2005*, pages 134–142. IEEE Computer Society, 2005.

[26] Ying Zou and Kostas Kontogiannis. A framework for migrating procedural code to object oriented platforms. In *8th IEEE Asia-Pacific Software Engineering Conference (APSEC)*, pages 408–418. IEEE, 2001.