Regaining Lost Knowledge through Dynamic Analysis and Aspect Orientation

— An Industrial Experience Report —

Andy Zaidman[1], Bram Adams[2], Kris De Schutter[2], Serge Demeyer[1], Ghislain Hoffman[2], and Bernard De Ruyck[3]

[1]Department of Mathematics and Computer Science, University of Antwerp, Belgium
{Andy.Zaidman, Serge.Demeyer}@ua.ac.be
[2]SEL, INTEC, University of Ghent, Belgium
{Bram.Adams, Kris.DeSchutter, Ghislain.Hoffman}@UGent.be
[3]Koninklijke Apothekersvereniging Van Antwerpen (KAVA), Antwerp, Belgium
bdr@kava.be

Abstract

This paper describes our experiences of applying dynamic analysis solutions on an industrial legacy application written in C, with the help of Aspect Orientation (AO). We use a number of dynamic analysis techniques that can help in alleviating the problem of (1) establishing the quality of the available regression test and (2) regaining lost knowledge of the application. We also show why our aspect language for C, aspicere, *is well-suited for using dynamic analysis in legacy environments. Finally, we present the case study itself, the results we have obtained and the validation thereof by the original developers and current maintainers of the application. We also mention some typical pitfalls that we encountered while dealing with legacy applications in a reengineering context.*

1 Introduction

Legacy software is omni-present: software that is still very much useful to an organization – quite often even *indispensable* – but a burden nevertheless [4]. 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 [20, 5, 17, 15, 7, 8].

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

1. Rewrite the application from scratch, from the legacy environment, to the desired one, using a new set of requirements [4].
2. Reverse engineer the application and rewrite the application from scratch, from the legacy environment, to the desired one [4].
3. Refactor the application. One can refactor the old application, without migrating it, so that change requests can be efficiently implemented; or refactor it to migrate it to a different platform.
4. Often, in an attempt to limit the costs, the old application is ”wrapped” and becomes a component in, or a service for, a new software system. In this scenario, the software still delivers its useful functionality, with the flexibility of a new environment [4]. This works fine 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 or service in the new system, is trusted for what it does, but nobody knows – or wants to know – what goes on internally (white box).
5. A last possibility is a mix of the previous options, in which the old application is seriously changed before being set-up as a component or service in the new environment.

Certainly for all scenarios but the first, the software en-

gineer would ideally like to have:

- a good understanding of the application in order to start his/her reengineering operation (or in order to write additional tests before commencing reengineering)[19, 7]
- a well-covering (set of) regression test(s) to check whether the adaptations that are made are behavior-preserving [8]

However, in practice, legacy applications seldom have up to date documentation available [17], nor do they have such a set of tests.

In this paper we describe the application of a number of dynamic-analysis-based reverse engineering solutions in the context of an industrial legacy C software system [6, 17]. Our goals are to (1) regain lost knowledge, (2) determine test coverage and (3) identify problematic structures in the source code. We build upon a number of recent dynamic analysis techniques that were originally developed for object-oriented software [22, 21].

However, applying dynamic analysis in a legacy context is not without its pitfalls. This is where we think that *Aspect Oriented Programming* (AOP) can make a difference [11]. 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 able to perform dynamic analysis in a clean and efficient way.

This paper is organized as follows: Section 2 starts off with a description of the dynamic analysis solutions we apply. Section 3 introduces our AOP implementation, while Section 4 introduces the case study. The approach we followed is explained in Section 5, while the results and validation thereof are shown in Section 6. Section 7 mentions some typical legacy environment pitfalls we had to deal with. Section 8 concludes and points to future work.

2 Dynamic analysis

Recently a number of novel dynamic analysis techniques that deal with program comprehension have been developed [9, 10, 22, 21]. Most of these techniques have been developed in the context of object-orientation, but we considered it worthwhile to verify whether these techniques could be "transplanted" to the context of procedural systems. Furthermore, the techniques proposed in [22, 21] have previously been validated using publicly available documentation from open source software. In the context of this industrial case study however, we will be able to validate the results with the original developers and current maintainers (see Section 6).

This section will briefly discuss two techniques that aim at improving the program comprehension process (sections 2.1 and 2.2) and a simple algorithm that allows to calculate test coverage (section 2.3).

2.1 Dynamic coupling based — webmining

The basis for this technique is the measurement of runtime coupling between modules of a system. Modules that have a high level of runtime export coupling, are often modules that contain important control structures, and request other modules to do work for them. As such, these are ideal candidates to study during early program comprehension.
To overcome the typical problem of coupling measurements —each is between two classes or modules— we add webmining techniques. This makes sure that not only coupling between two separate modules is taken into account, but also a transitive measurement is used for determining the most important modules of a system [21].

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 important web pages can be identified based solely on the hyperlink structure. By interpreting call graphs as web graphs, we show how to apply these *"webmining"* techniques to call graphs, in order to uncover important classes.

Based on the call graph of an execution trace of the application, the HITS webmining algorithm [12] allows us to identify so-called *hubs* and *authorities*. Intuitively, on the one hand, hubs are pages (classes) that refer to other 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 first step to perform is to *compact* the call graph, which removes duplicate tuples from the form *(call origin, call destination)* and replaces it with a weight on the edge indicating the number of calls. For more details about this process, please consult [21].
Next we apply the HITS algorithm: every node in the graph i gets assigned to it two numbers; a_i denotes the authority of the page, while h_i denotes the hubiness. Let $i \rightarrow j$ denote that there is a calling relationship between modules i and j, and let $w[i, j]$ be the number of unique calls between i and j. The recursive relation between authority and hubiness is captured by formulas (1) and (2).

$$h_i = \sum_{i \rightarrow j} w[i, j] \cdot a_j \quad (1)$$

$$a_j = \sum_{i \rightarrow j} w[i, j] \cdot h_i \quad (2)$$

After each update, the values are normalized[1] (for simplicity, not done in the example of Table 1), The HITS algorithm is known to converge to stable sets of authority and hub weights after around 11 iterations [12].

The resultset obtained from this analysis is a list of classes, ranked according to their importance.

From previous case studies in the context of object-oriented systems, we have 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 [21].

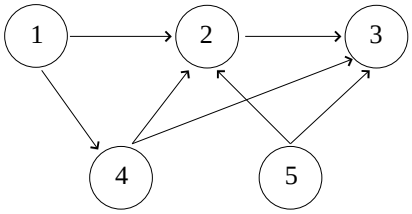

Figure 1. Example web-graph

Example Consider the graph given in Figure 1. Table 1

		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. Tuples have the form (H,A).

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

2.2 Frequency of execution based analysis

Thomas Ball [3] introduced the concept of "Frequency Spectrum Analysis", a way to correlate procedures, functions and/or methods through their relative calling frequency. The idea is based around the observation that a relatively small number of methods/procedures is responsible for a huge event trace. As such, a lot of repeated calling of procedures happens during the execution of the program. By trying to correlate these frequencies, we can learn something about (1) the size of the inputset, (2) the size of the outputset and —most interesting for us— (3) calling relationships between methods/procedures. In [22] we built further upon this idea, by proposing a visualization of the trace that allowed for visual detection of parts of the event trace that showed tightly collaborating methods.

For our case study, we will use this technique to uncover coupling relations between procedures by looking at their calling frequencies.

2.3 Test coverage analysis

Refactoring, migrating, wrapping or limited forms of reengineering require that the basic functionality of the application remains unchanged or at the very least similar. The best way to enforce behavior preservation, is to make sure that a well-covering regression test is available [8].

As such, before commencing a refactoring operation it is worthwhile to verify the coverage of the tests, as to get an idea whether the subsystems that need adaptation are indeed tested.

When static and dynamic information about the application are available, measuring code coverage becomes easy [18]. Equation 3 shows how to calculate procedure coverage for a particular module.

$$coverage(A) = dynamic(A)/static(A) * 100 \qquad (3)$$

where
$dynamic(A)$: # called procedures in module A
$static(A)$: # procedures in module A

We also performed a similar calculation for establishing the statement coverage percentage.

3 AOP for legacy environments

3.1 Introduction

AOP is a relatively new paradigm, grown from the limitations of Object Orientation (OO) [11], and a fortiori those of older paradigms. 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. AOP, however, allows to isolate these crosscutting concerns in dedicated modules (called *aspects*). More formally, aspects allow us to select by *quantification* (through *pointcuts*) which events in the flow of a program (*join points*) interest us, and what

[1] In this context, this means mapping everything to the range $[0, 1]$.

we would have 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.

3.2 Justification

Thusfar, we only mentioned OO environments and that is also the direction AOP research was heading until recently. Nevertheless, it is important to recognize that crosscutting concerns have been in existence for many years without adequate solutions. In practice, this has led to lots of scattered and tangled code, making badly documented code very hard to maintain. Having said that, it is now clear that backporting new modularisation techniques like AOP to legacy environments is a sound research area.

Why AOP? Having a technology at one's disposal able to adapt the functionality of an application's code without actually changing it physically, is very beneficial in the business climate described above. At a very low level, this allows to factor out tracing, null pointer checks, bug patching, ... without breaking anything. At a higher level of abstraction, one can define a set of coding [14] or quality conventions and enforce them. Even more abstract, mining for business rules [16] and program logic in order to refactor legacy systems to more sound architectures is a viable possibility. Knowing all this, AOP in the context of Cobol, C, PL/1, ... really makes sense and that is why we chose it for this case study.

The authors at Ghent University have developed a framework for introducing AOP in legacy languages. *Cobble* [13] leverages this for Cobol applications, whereas *aspicere*[2] [2, 1] instantiates it for C. This paper applies the latter 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[3].

As our industrial partner has a large codebase, mainly written in C, we used *aspicere* for our experiments[4].

3.3 Aspicere

It is easiest to explain *aspicere* by a simple example (in section 5, we'll see another one). The C-libraries on our industrial partner's (see section 4) old system were very fault-tolerant with regard to string handling. When e.g. *atoi*() was passed a null pointer, the particular implementation of this standard function didn't throw up a segmentation fault, but gracefully 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*(), ... disappears, resulting in random segmentation faults.

There are numerous solutions to this problem, ranging from tedious manual editing to pure hacking (define macro's named *strcpy*(), *atoi*(), ...). Using AOP, we get a very elegant, cleanly modularised treatment: surround calls to infected methods by a null pointer check and only if everything is alright, the originally called method should be invoked. This is implemented in the following advice:

```
1 ReturnType around safe_ato (Src,ReturnType)
2  on (Jp):
3   call(Jp,"ato.",[Src])
4   && type(Jp,ReturnType) {
5     ReturnType dst;
6
7     if (Src == NULL)
8       dst = 0; /* compiler does the cast */
9     else
10      dst = proceed();
11
12    return dst;
13 }
```

Here, we defined an advice called *safe_ato*. It catches calls to all *ato.*()-functions[5], and guards the argument against null pointers. If everything is safe, the original call to *ato.*() proceeds, otherwise zero is returned. This advice also illustrates *aspicere*'s binding mechanism. 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[6]. So, our bindings are comparable to C macro's or C++ template parameters, enriched with the extra power of Prolog unification.

Advice is the only AOP-specific construct in *aspicere*. Aspects are merely normal compilation units, with their own methods, variables, ...

4 Case study

The industrial partner that we cooperated with in the context of this research experiment is *Koninklijke Apo-*

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

[3] Sponsored by the IWT, Flanders. Also see: http://www.iwt.be

[4] *aspicere* is freely available from http://users.ugent.be/~badams/aspicere/

[5] The "." shows that we allow full regular expression support. Note that we also advise *atok*(), *atom*(), ... if these exist, but we can circumvent this using metadata.

[6] It is also perfectly possible to alter *Src* before doing the **proceed**()-call, although the latter notation doesn't make this clear.

thekersvereniging Van Antwerpen (KAVA)[7]. 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 in its industry to realize the need for an automated tarification process, and have taken it on themselves to deliver this service 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 and technology changes they are very much aware of the necessity to adapt and reengineer this service.

Kava has just finished the process of porting their applications to fully ANSI-C compliant versions, running on Linux. Over the course of this migration effort, it was noted that documentation of these applications was outdated. This provided us with the perfect opportunity to undertake our experiments.

As a scenario for our dynamic analysis, the developers told us that they often use the so-called *TDFS* application as a final check to see whether adaptations in the system have any unforeseen consequences. As such, it should be considered as a functional application, but also as a form of regression test.

The TDFS-application finally produces a digital and detailed invoice of all prescriptions for the healthcare insurance institutions. This is the end-stage of a monthly control- and tariffing process and acts also as a control-procedure as the results are matched against the aggregate data that is collected earlier in the process.

5 Approach

To analyse the dynamic behaviour of the TDFS application, we needed a detailed trace of some representative runs of the system. As already explained in section 3, this is very easily and unintrusively done using aspects. Here, we show one of the two advices we used [8]:

```
1 RetType around tracing (RetType,FileStr)
2  on (Jp):
3   call(Jp,"^(?!.*printf$|.*scanf$).*$")
4   && type(Jp,RetType)
5   && !str_matches("void",RetType)
6   && logfile(FileName)
7   && stringify(FileName,FileStr) {
8     FILE* fp=fopen(FileStr,"a");
9     RetType i;
10
11    fprintf (fp,"before ( %s in %s ) \n",
12      Jp->functionName,Jp->fileName);
13    fflush(fp);
14    i = proceed ();
15    fprintf (fp,"after ( %s in %s ) \n",
16      Jp->functionName,Jp->fileName);
17    fclose(fp);
18
19    return i;
20 }
```

In the body of the tracing advice (between braces), normal C code is set up to write out tracing statements to a file. However, we need some extra things, like access to join point context (names of method and current file) and some means to continue the original function call. To make the advice code generic, we also need to know the correct return type of the advised function (RetType). All this information is accessible either from the logic variables in the binding list (the comma-separated list on line 1) or from the join point context (line 2). Both have gotten their values (are *bound*) in the pointcut (lines 3–7). It suffices to say that this is in fact a Prolog query denoting *all calls to functions whose name does not end in either "printf" or "scanf" and which return something useful (i.e. non-void).* In the meantime a variable is bound to this return type and out of some metadata repository the name of the log file is fetched (and quoted).

Again, this advice perfectly confirms the points we made in 3. Instead of modifying directly the base applications with the imminent danger of sabotaging them, AOP allows to isolate new concerns in their proper modules, giving more confidence and courage to system maintainers. The aspects are written in almost plain C (no steep learning curve), while the pointcut predicates and metadata can be provided in libraries or designed by domain experts. Abuse of this extra power is still possible though, and that is why the tooling community still has a lot of work to do.

Once we got our traces, we fed them into analysis-specific scripts. For further details about their implementation, see Section 2.

6 Results

This section will cover the results we have obtained from applying each of the three dynamic analysis techniques.

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

[8] The other one is needed for methods returning **void**.

Furthermore, for each of the techniques we will also present the developers' opinions on the resultset we have confronted them with. For this particular module, the TDFS application, two developers were available at Kava. From now on we will call them D_1 and D_2. Both have a thorough knowledge of the structure and the inner workings of this particular application.

Before we began discussing the results with the developers, we first presented them with a schema consisting of each of the 15 modules and 3 questions:

1. Which is the most essential?
2. Which tends to contain the most bugs?
3. Which is the hardest to debug?

We noted their answers and also asked if there were any particular reasons why they believed a certain module to be important, hard to debug or to contain bugs.

We presented our results, technique by technique, to each of the two developers separately and wrote down their answers. Afterwards, during a short session we discussed the results with both developers and highlighted the similarities and differences in their answers.

6.1 Dynamic coupling based – webmining

6.1.1 Resultset

Our tool needed slightly over 10 hours to give us the results listed in Table 2. The results are ranked according to the *hubiness* values in the right column. This hubiness value lies in the range $[0, 1]$. Some important facts that can be derived from Table 2 are:

- the module `e_tdfs_mut1.c` stands out.
- only 7 out of the 15 modules have a value greater than zero. Modules with a value of zero, do not call other modules.
- the 4 modules that are specific to the TDFS application show up in the 4 highest ranked places.

6.1.2 Discussion with developers

D_1 mentioned `e_tdfs_mut1.c` and `tdfs_mut2.c` as being the most essential modules for the TDFS application. `io.c` and `cache.c` are also important from a technical point of view, but are certainly not specific to the TDFS application, as they are used by many other applications of the system. D_1 was actually surprised at the fact that `cache.c` wasn't catalogued as being more important. `csrout.c` and `csroutines.c` are difficult to debug, but they have only once had to change some details in these file in a time period of 10 years.

Module	Value
e_tdfs_mut1.c	0.814941
tdfs_mut1_form.c	0.45397
tdfs_bord.c	0.397726
tdfs_mut2.c	0.164278
tools.c	0.164278
io.c	0.12548
csrout.c	0.0321257
tarpargeg.c	0
csroutines.c	0
UW_strncpy.c	0
td.ec	0
cache.c	0
decfties.c	0
weglf.c	0
get_request.c	0

Table 2. Results of the webmining technique

D_2 clearly ranks the `e_tdfs_mut1.c` module as being the most important and most complicated module: it contains most of the business logic. `tdfs_mut2.c` makes a summary of the operations carried out by `e_tdfs_mut1.c` and checks the results generated by `e_tdfs_mut1.c`. `tdfs_mut1_form.c` is mainly responsible for building up an interface for the end-user, while `tdfs_bord.c` is concerned with formatting the output.

6.1.3 Discussion

As such, the opinions of D_1 and D_2 are indeed very similar. D_1 ranks `e_tdfs_mut1.c` and `tdfs_mut2.c` as being most important, D_2 points to `e_tdfs_mut1.c` as being the most important module.

The resultset of our own technique (see Table 2) clearly ranks `e_tdfs_mut1.c` as being the most important module in the system. Furthermore, all modules that are specific to this application appear at the top of the ranking.

A last remark on one of the drawbacks of this webmining technique: container classes or modules are often ranked very low, because of the fact that their export coupling is low [21]. This fact partly explains why `cache.c` – a caching data-structure – which was expected to rank higher according to D_1, is placed quite low.

6.2 Frequency analysis

Due to the huge size of the event trace (90GB $\approx 4.86 \times 10^8$ procedure calls), the visualization we presented in [22], didn't scale up to this huge amount of data. Therefore, we opted for a slightly different solution. We still use frequency of execution as the underlying model, but summarize the results before visualizing.

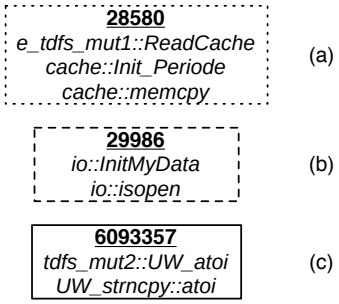

Figure 2. Three frequency clusters from the TDFS application

A fragment of the result is shown in Figure 2. This shows three *"frequency clusters"*. Each cluster shows the total execution frequency, and the procedures that fall into this frequency interval. Figure 2 shows different kinds of boxes, to indicate cohesion within a frequency cluster: a box with a full line (Figure 2.c) indicates that $\leq 50\%$ of the methods in the cluster come from the same module, a dashed line ((Figure 2.b) indicates total cohesion as all procedures belong to the same module. A dotted line (Figure 2.a) meanwhile indicates a level of cohesion within the frequency-cluster of between 50 and 100%.

In total, 237 unique procedures were executed during the scenario. Of these, 160 could be clustered into 25 frequency-clusters. In other words, 67.5% of the procedures could be catalogued in clusters. When considering the cohesion of each of these frequency-clusters, we have the following results: two of these clusters had a full line, i.e. they didn't show cohesion. 12 had a dashed line, meaning that all procedures within a frequency-cluster originated from a single module, while the 11 others had a dotted line, also indicating a strong level of cohesion.

This technique provides an easy way to find procedures that share common goals, because they are related through their frequency of execution. Furthermore, it allows to easily audit the system when it comes to cohesion.

The total time to perform this analysis and compute the results was 5 hours.

6.2.1 Discussion with the developers.

D_1 immediately remarked that one of the two frequency clusters with a full line, i.e. a cluster with a limited degree of cohesion, was actually a wrapper construction they had hastily constructed when performing the migration from UnixWare to Linux. These are exactly the wrappers discussed in 3.3.

The clusters found didn't surprise the developers either.

6.2.2 Discussion

For our particular case study, 48% of the clusters were found to be fully cohesive. These fully cohesive clusters are accountable for 20% of the procedures. 44% were found to be strongly cohesive; these clusters contain 49% of the total number of procedures. The largest non-cohesive cluster had a frequency of execution of *1*, consisting mainly out of procedures with initialization functionality. The other non-cohesive cluster was the one that caught D_1's attention for containing wrapper functionality.

As such, we can conclude that the system is actually well-structured, as most clusters were cohesive and these account for 70% of all procedures.

6.3 Test coverage

6.3.1 Resultset

Starting from the event trace, determining the test coverage of the scenario that was handed to us by the developers took around 5 hours. For each module present in the system, we calculated the number of covered procedures and the number of covered statements. We presented the results of this analysis to the developers in the form of a simple HTML-based website, similar to the way code coverage tools for e.g. Java (e.g. Emma[9]) present their results.

Unfortunately, due to non-disclosure agreements we are not able to disclose precise numbers with regard to the test coverage of the application.

6.3.2 Discussion with developers

D_1 knew that certain procedures were not tested within the scenario of the TDFS application and told us that they were tested in other scenarios.

D_2 immediately recognized that the parts of the module that showed up as being *not tested*, were actually functions that were no longer in use. Some 5 years ago, new functions were written for the conversion from the *Belgian Franc* to the *Euro* and each of the functions that shows up as being not tested by our analysis has a Euro counterpart, prepended with *e_*.

[9] http://emma.sourceforge.net/

```
gcc -c -o file.o file.c
```

Figure 3. Original makefile.

```
gcc -E -o tempfile.c file.c
cp tempfile.c file.c
aspicere -i file.c -o file.c \
     -aspects aspects.lst
gcc -c -o file.o file.c
```

Figure 4. Adapted makefile.

6.3.3 Discussion

The results from the test coverage itself were in range with the expectations from the developers. However, our coverage analysis turned out to have a positive side-effect in that the developers at Kava saw that they still had to remove portions of dead code after the switch from the Belgian Franc to the Euro was made[10].

7 Pitfalls of reengineering legacy code using AOP

Applying aspects onto a base program, is intended to happen transparently for the end user. However, while using our experimental legacy AOP tools during our experiments at our industrial partner, we encountered several problems. This section describes some of these.

7.1 Adapting the build process

The Kava application uses `make` to automate the build process. Historically, all 269 makefiles were hand-written by several developers, not always using the same coding-conventions. During a recent migration operation from *UnixWare* to *Linux*, a significant number of makefiles has been automatically generated with the help of *automake*[11]. Despite this, the structure of the makefiles remains heterogeneous, a typical situation in (legacy) systems.

We built a small tool, which parses the makefiles and makes the necessary adaptations. (A typical example is shown in Figures 3 and 4.) However, due to the heterogeneous structure, we weren't able to completely automate the process, so a number of makefile-constructions had to be manually adapted. The situation becomes more difficult when e.g. Informix esql preprocessing needs to be done. This is depicted in Figures 5 and 6.

[10]The switch from Belgian Franc to the Euro was made in the first 6 months of 2002

[11]`Automake` is a tool that automatically generates makefiles starting from configuration files. Each generated makefile complies to the GNU Makefile standards and coding style. See http://sources.redhat.com/automake/.

```
.ec.o:
   $(ESQL) -c $*.ec
   rm -f $*.c
```

Figure 5. Original makefile with esql preprocessing.

```
.ec.o:
   $(ESQL) -e $*.ec
   chmod 777 *
   cp 'ectoc.sh $*.ec' $*.ec
   esql -nup $*.ec $(C_INCLUDE)
   chmod 777 *
   cp 'ectoicp.sh $*.ec' $*.ec
   aspicere -verbose -i $*.ec -o \
      'ectoc.sh $*.ec' -aspects aspects.lst
   gcc -c 'ectoc.sh $*.ec'
   rm -f $*.c
```

Figure 6. Adapted makefile with esql preprocessing.

Using our scripts to alter the makefiles takes a few seconds to run. Detecting where exactly our tool failed and making the necessary manual adaptations took us several hours.

7.2 Compilation

A typical compile cycle of the application consisting of 407 C modules (453 KLOC in total) takes around 15 minutes[12]. We changed the cycle to:

1. Preprocess
2. Weave with *aspicere*
3. Compile
4. Link

This new cycle took around 17 *hours* to complete. The reason for this substantial increase in time can be attributed to several factors, one of which may be the time needed by the inference engine for matching up advice and join points (still unoptimized).

7.3 Legacy issues

Even though Kava recently migrated from UnixWare to Linux, some remains of the non-ANSI implementation are still visible in the 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 is not fully integrated yet

[12]Timed on a Pentium IV, 2.8GHz running Slackware 10.0

in *aspicere*. Instead of ignoring the whole base program, we chose to ”skip” (as yet) unsupported 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 only 2362 filtered out, or a minor 4 percent. This is likely due to the fact that in a particular file lots of invocations of the same method have been skipped during weaving, because it was called multiple times with the same or similar variables. This was confirmed by several random screenings of the code.

Another fact to note is that we constantly opened, flushed and closed the tracefile, certainly a non-optimal solution from a performance point of view. Normally, *aspicere*'s weaver transforms aspects into plain compilation modules and advice into ordinary methods of those modules. So, we could get hold of a static file pointer and use this throughout the whole program. However, this would have meant that 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.

7.4 Scaleability issues

Running the program Not only the compilation was influenced by our aspect weaving process. Also the running of the application itself. The scenario we used (see Section 4), normally runs in about 1.5 hours. When adding our tracing advice, it took 7 hours due to the frequent file I/O.

Tracefile volume The size of the logfile also proved problematic. The total size is around 90GB, however, the linux 2.4 kernel Kava is using was not compiled with large file support. We also hesitated from doing this afterwards because of the numerous libraries used throughout the various applications and fear for nasty pointer arithmetic waiting to grab us. As a consequence, only files up to 2GB could be produced. So, we had to make sure that we split up the logfiles in smaller files. Furthermore, we compressed these smaller logfiles, to conserve some diskspace.

Effort analysis Table 3 gives an overview of the time-effort of performing each of the analyses. As you can see, even a trouble-free run (i.e. no manual adaptation of makefiles necessary) would at least take 29 hours.

Task	Time	Previously
Makefile adaptations	10 s	–
Compilation	17h 38min	15min
Running	7h	1h 30min
Code coverage	5h	–
Frequency analysis	5h	–
Webmining	10h	–
Total	44h 38min 10s	1h 45min

Table 3. Overview of the time-effort of the analyses.

8 Conclusion and future work

This paper describes our experiences with applying dynamic analysis in an industrial legacy C context. We used two dynamic analysis techniques that we had previously developed and validated for Object Oriented software and added a simple test coverage calculation. Furthermore, this paper describes how we used *aspicere*, our “AspectC” implementation for collecting the traces we needed for performing the dynamic analyses.

From the resultsets we obtained from our dynamic analysis experiments, we can conclude that:

- The webmining approach results in a ranking of modules according to their importance from a program comprehension point of view. Interviews with the developers fully confirm the results that our heuristic delivered. The only false negative we could note, was a container class that the developers deemed important, but was judged as being unimportant by our technique. This is due to the low to non-existent level of export coupling from this particular module.
- The frequency analysis approach allowed to easily audit the system's internal structure. We found that most of the modules are (strongly) cohesive, which indicates that the structure is well balanced and reuse is a definite possibility. The developers agreed with our views and told us that many modules are frequently reused.
- The test coverage itself wasn't surprising: most results were well in line with the developers' expectations. However, perhaps it still was the most interesting technique for Kava, our industrial partner. It allowed them to remember that they still had significant portions of dead code in their system.

As a vehicle to perform our dynamic analysis, we used *aspicere*, which allowed us to use the clean and non-intrusive, yet powerful mechanism of Aspect Orientation to

trace the entire application.

As a clear downside of our approach, we should note the effort it takes to perform the entire analysis. If no problems are encountered, the entire analysis we described takes just under 45 hours, for a system that should be considered as medium-scale. As such, we acknowledge that we should improve the efficiency of our tools.

In the future we will work on techniques that allow for more fine-grained knowledge mining. One of the paths we are currently pursuing is one where we start from the most important classes as identified by our webmining technique and then use Aspect Orientation to gather more fine-grained information such as changes to parameter values, return values, etc. Another path we are examining is to perform similar experiments to the ones we described in this paper, but then using *cobble* — sibling to *aspicere* — to extract knowledge from Cobol systems.

9 Acknowledgements

The authors would like to dedicate this paper, in memoriam, to professor dr. Ghislain Hoffman, who passed away on August 25th, 2005.

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.

References

[1] B. Adams, K. De Schutter, and A. Zaidman. AOP for Legacy Environments, a Case Study. In *Proceedings of the 2nd European Interactive Workshop on Aspects in Software*, 2005.

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

[3] T. Ball. The concept of dynamic analysis. In *ESEC / SIGSOFT FSE*, pages 216–234, 1999.

[4] K. Bennett. Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23, 1995.

[5] M. Brodie and M. Stonebreaker. *Migrating Legacy Systems: Gateways, Interfaces & The Incremental Approach*. Morgan Kaufmann, 1995.

[6] E. J. Chikofsky and J. H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, pages 13–17, Jan. 1990.

[7] C. M. de Oca and D. L. Carver. Identification of data cohesive subsystems using data mining techniques. In *ICSM*, pages 16–23. IEEE, 1998.

[8] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

[9] O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR*, pages 314–323. IEEE, 2005.

[10] A. Hamou-Lhadj, E. Braun, D. Amyot, and T. Lethbridge. Recovering behavioral design models from execution traces. In *CSMR*, pages 112–121. IEEE, 2005.

[11] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings ECOOP*, volume 1241, pages 220–242. Springer-Verlag, 1997.

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

[13] R. Lämmel and K. D. Schutter. What does Aspect-Oriented Programming mean to Cobol? In *AOSD '05*, pages 99–110, New York, NY, USA, 2005. ACM Press.

[14] K. Mens, B. Poll, and S. Gonzalez. Using intentional source-code views to aid software maintenance. In *ICSM*, pages 169–178. IEEE, 2003.

[15] I. Michiels, D. Deridder, H. Tromp, and A. Zaidman. Identifying problems in legacy software: Preliminary findings of the ARRIBA project. In *Proceedings of the ELISA workshop at ICSM*, 2003.

[16] I. Michiels, K. D. Schutter, T. D'Hondt, and G. Hoffman. Using dynamic aspect to extract business rules from legacy code. In *Online proceedings of Dynamic Aspects Workshop at AOSD*, 2004. http://aosd.net/2005/workshops/daw/.

[17] D. L. Moise and K. Wong. An industrial experience in reverse engineering. In *WCRE*, pages 275–284, Washington, DC, USA, 2003. IEEE.

[18] H. M. Sneed. Measuring the effectiveness of software testing. In S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, editors, *SOQUA/TECOS*, volume 58 of *LNI*, page 109. GI, 2004.

[19] H. M. Sneed. Program comprehension for the purpose of testing. In *IWPC*, pages 162–171. IEEE, 2004.

[20] H. M. Sneed. An incremental approach to system replacement and integration. In *CSMR*, pages 196–206. IEEE, 2005.

[21] A. Zaidman, T. Calders, S. Demeyer, and J. Paredaens. Applying webmining techniques to execution traces to support the program comprehension process. In *CSMR*, pages 134–142. IEEE, 2005.

[22] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *CSMR*, pages 329–338. IEEE, 2004.