Applying Dynamic Analysis in a Legacy Context: An Industrial Experience Report

Andy Zaidman[1], Bram Adams[2], and Kris De Schutter[2]

[1]LORE, Department of Mathematics and Computer Science, University of Antwerp, Belgium,
Andy.Zaidman@ua.ac.be
[2]SEL, Department of Information Technology (INTEC), University of Ghent, Belgium
{Bram.Adams, Kris.DeSchutter}@ugent.be

Abstract

This paper describes our experiences with applying dynamic analysis solutions with the help of Aspect Orientation (AO) on an industrial legacy application written in C. The purpose of this position paper is two-fold: (1) we want to show that the use of Aspect Orientation to perform dynamic analysis is particularly suited for legacy environments and (2) we want to share our experiences concerning some typical pitfalls when applying any reverse engineering technique on a legacy codebase.

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 [10, 5, 8, 6].

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 scenarios 2, 3, 4 and 5, the software engineer would ideally want 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)[9]
- a well-covering (set of) regression test(s) to check whether the adaptations that are made, are behavior-preserving[6]

However, in practice, legacy applications seldom have up to date documentation available [8], nor do they have a well-covering set of tests.

The actual goals of this experiment are to (1) regain lost knowledge, (2) determine test coverage and (3) identify problematic structures in the source code. For this, we build upon a number of recently developed dynamic analysis techniques that were developed for object-oriented software [12, 11]. The emphasize for this paper however, is more on the pitfalls we encountered along the way when applying the different techniques on a legacy system.

This paper is organized as follows: Section 2 starts with a description of the case study. Section 3 introduces our AOP implementation, while Section 4 briefly discusses the dynamic analysis solutions we used. Section 5 mentions some typical legacy environment pitfalls we stumbled across. Section 6 concludes and points to future work.

2 Case study

The industrial partner that we cooperated within the context of this research experiment is *Koninklijke Apothekersvereniging Van Antwerpen* (KAVA)[1]. 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.

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

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.

3. AOP for legacy environments

We recently developed a framework for introducing AOP in legacy languages like Cobol [7] and C [2, 1]. The latter is called *aspicere*[2]. This paper applies aspicere 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].

Our industrial partner has a large codebase, mainly written in C, that's why we used aspicere for our experiments.

4. Dynamic analysis solutions

In total we applied 3 dynamic analysis solutions. This section will briefly introduce each of them.

Webmining This solution identifies the most important classes in a system with the help of a heuristic that uses dynamic coupling measures. The idea is based on the fact that tightly coupled classes, can heavily influence the control flow. To add a transitive measure to the binary relation of coupling, webmining principles are used. For a more detailed description of this technique, we refer you to a previous work [11].

Frequency analysis This idea is based on the concept of *Frequency Spectrum Analysis*, first introduced by Thomas Ball [3]. It is centered around the idea that the relative execution frequency of methods or procedures can tell something about which methods or procedures are working together to reach a common goal. For more details we refer to [12].

Test coverage When refactoring or reengineering a system, certain functionality often has to be preserved. Having a well-covering set of tests can be very helpful for determining whether the adaptations to the code are indeed behavior preserving. By establishing the test coverage of modules and procedures, we are able to have a clear view of which parts of the system are tested.

[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

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

Figure 1. 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 2. Adapted makefile.

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

Figure 3. Original makefile with esql preprocessing.

5. Pitfalls of dynamic analysis in a legacy environment

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.

5.1 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*[4]. 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 1 and 2.) 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 3 and 4.

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.

[4] `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) -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 4. Adapted makefile with esql preprocessing.

5.2 Compilation

A typical compile cycle of the application consisting of 407 C modules takes around 15 minutes[5]. 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).

5.3 Legacy issues

Even though Kava recently migrated from UnixWare to Linux, some remnants 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 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

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

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.

5.4 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 2), normally runs in about 1.5 hours. When adding our tracing advice, it took 7 hours due to the frequent file IO.

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 1 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 1. Overview of the time-effort of the analyses.

6. 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.

This paper focusses on some common problems we came across when trying to collect an event trace from a legacy C application using aspicere. Some of these problems can be catalogued as being technical, e.g. adapting heterogeneously structured makefiles or overcoming the maximum file size limit of the operating systems.

Some other problems are perhaps more fundamental:

- Performing an effort analysis shows that collecting the trace of the system takes more than 24 hours.
- Subsequently, any dynamic analysis solution, has to cope with analyzing an event trace of around 90 GB. Scaleability of the dynamic analysis solution is thus of the utmost importance.

As such, we can conclude that for what should be considered a medium-scale application, we are already having scaleability issues with our tools. As such, improving the efficiency of our tools is one of our immediate concerns.

7. 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.

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

[7] 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.

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

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

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

[11] 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 Computer Society, 2005.

[12] 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 Computer Society, 2004.