Applying AOP in an Industrial Context

An Experience Paper

Pascal Durr, Lodewijk Bergmans,
Gurcan Gulesir, Mehmet Aksit
University of Twente, The Netherlands
{durr,bergmans,gulesirg,aksit}
@ewi.utwente.nl

Remco van Engelen
ASML, The Netherlands
remco.van.engelen@asml.com

1. INTRODUCTION

This position paper presents a case study we carried out in the Ideals project. The Ideals project is a collaboration between four major research institutions and an industrial partner. The research institutions are: the University of Twente[5], the Centrum voor Wiskunde en Informatica[7], the Technical University Eindhoven[2] and the Embedded System Institute[3]. The industrial partner in this project is ASML[1], from ASMLs website: "*ASML is the world's leading provider of lithography systems for the semiconductor industry, manufacturing complex machines that are critical to the production of integrated circuits or chips.*".

The lithography machines that ASML develops, are large embedded systems. The size of the source code of this system is more than 10 million lines of code(LOC), written in the C programming language. The system is divided into several architectural layers and consists of more than 200 components. Within this context the Ideals projects aims at identifying crosscutting concerns at both implementation and architecture level, and to modularize these crosscutting concerns using (new) AOP techniques.

This paper presents the preliminary results of the case study we carried out at ASML. We took a representative part of the code base, identified the crosscutting concerns, migrated the source code to remove the crosscutting concerns from the base code, and add aspects to incorporate the crosscutting concerns in a modular way.

2. CASE STUDY SETUP

For our case study, we selected a subsystem driver component, that is composed of 30458 lines of source code, containing 6389 control and executable statements. We selected this, because it is representative in terms of crosscutting concerns. In addition, it was recently maintained for better modularity, which would increase our chances to identify the "real" crosscutting concerns, i.e. not the ones due to the "aging" of the component.

To identify crosscutting concerns, we "manually" investigated source code, and used AspectBrowser [4]. As a result, we identified six crosscutting concerns in the component: **reflective information** (e.g. listing 1, line 2), **exception handling** (e.g. lines 1, 3, 7, 10, 11, and 14), **profiling** (e.g. lines 4, 5, and 13), **parameter tracing** (e.g. lines 6 and 12), **parameter checking** (not shown), **memory handling** (not shown). Note that most of the functions contain more core functionality than this one. Also, the crosscutting concern overhead is not as extreme as in this example.

```
1  int get_roi(ROI_struct* ROI_ptr) {
2     const char* func_name = "get_roi";
3     int result = OK;
4     timing_handle timing_hdl = NULL;
5     TIMING_IN;
6     trace_in(mod_data.tr_handle, func_name);
7     if (result == OK){
8        /* Retrieve current ROI */
9        *ROI_ptr = mod_data.ROI;
10    }
11    LC(result, GET_ROI_FAILED_obj);
12    trace_out(mod_data.tr_handle, func_name, result);
13    TIMING_OUT;
14    return result;
15 }
```

Listing 1: A simple getter function that exemplifies some of the identified crosscutting concerns in a nutshell.

Out of the identified crosscutting concerns, we addressed only parameter tracing, profiling, and reflective information, because these crosscutting concerns made up for 28% of the code size, and realizing them using aspects was feasible within our limited time frame.

To apply AOP, we first removed the selected crosscutting concerns using *sed*, which is a regular expression engine. Next, we specified the aspects for the three crosscutting concerns. Finally, we used our weaver, namely WeaveC [6], to weave the aspects.

3. AOP TECHNOLOGY

WeaveC is a source-to-source weaver for the C programming language, and it was developed at an earlier time in the Ideals project. It uses an XML input format and supports the following join points and introductions:

- Function call
- Function execution
- Local variable introduction in a function
- Global variable introduction in a file
- Field introduction in a global structure

The first two elements are join points, where one can apply before and after advice. These join points and introductions were sufficient to address tracing, profiling and reflective information. The current implementation uses a grammar which can parse only preprocessed C code. There is also limited support for annotations and plug-ins.

This section describes some of the issues which should be addressed for applying an AOP approach. Although these might seem ASML specific. We do feel that these issues are shared by a large number,

if not all, companies. Therefore we consider this set as a collection of common or fundamental issues that must be addressed by any AOP mechanism, which is used in an industrial context.

3.1 Migration

The concerns we handled in this case were mostly implemented on separate lines in the code and were easy to identify with "simple" tooling. However, migrating the tracing concern was more difficult, because determining whether to trace a variable at the start or end of a function relies on the parameter's input and output specification. We implemented this by annotating which parameters are input, output or both. Automatically generating these annotations requires control flow analysis and read-write analysis of the parameters. The CWI is currently developing tooling to automate the migration process, i.e. to remove the crosscutting concern code, to insert the tracing annotations and to indicate the deviations which must be handled manually.

3.2 Availability and maturity of tooling and process

The current implementation and process has been developed as a proof of concept. These tools have to become industry strength, and this requires substantial investments. Using AOP for the selected concerns has clear benefits, but a large impact, too. The use of AOP requires a seamless integration into the development and build process. The tools used in these processes should be robust and mature. In an industrial development process there are usually specific roles, with different responsibilities. These roles should remain clearly separated even in the presence of aspects. If one uses AOP, there will be additional roles in the development life-cycle. Most notably, an aspect developer, a weave tool maintainer, and an integrator. The latter is responsible for the sanity of the entire system.

3.3 Ability to switch off aspects

It should be possible to switch on or off aspects for two reasons. Firstly, for tracking down errors, if the aspect code contains a bug the system would still be able to compile without a specific aspect. This can only be done if the impact of the aspects is not too severe. For example, turning off an exception handling aspect is not feasible as it would compromise the entire system. Secondly, for more control over the aspect application. In development versions of the software some aspects are turned on, while in the releases these aspects are turned off. Or, if some component is called often, Tracing might need to be turned off for runtime efficiency. As most identified aspects are cleanly separated from the base code, these can be turned off, if required. However, if there are base-aspect or aspect-aspect dependencies this is not trivial.

3.4 Understandability (mental model)

ASML uses the C programming language, hence switching to AOP is considered to be a real issue. There is, in AOP, a clear separation between aspect developers and core developers. Core developers still want to understand what goes on behind the scenes, as they can be a bit wary of new techniques, and want to verify the woven code.

3.5 Compile-time & Run-time performance

In large systems with many developers and many dependencies between components, the build performance can be quite substantial. As the throughput at runtime must be guaranteed, a runtime performance hit is usually not acceptable. A static weaving approach is thus more attractive. A compile-time performance hit is usually acceptable as long as the performance hit is not several factors higher than a regular build or the hit is exponential with the number of join points.

3.6 Ability to Debug

Debugging a program with aspects is also considered to be an issue. Especially, if an error occurs who is responsible for fixing this error? First of all this requires determining whether the error is in the aspect code, in the core code or in the combination. As the core developers may not be aware of aspects, they may not want to see the code with the aspects applied. However, when debugging the total system, one needs to be aware of the aspects. We came up with the following options: debug with the inserted code, debug with inserted code hidden, and debug normally and while executing the inserted code jump to the aspect definition. In our case study we implemented the second option.

4. EVALUATION AND BENEFITS

This section will briefly explain the results we achieved with our case study. Most of these are considered to be the key motivations for using AOP. We provide these in the context of the case study.

4.1 Statement reduction

As stated previously, our experiment was carried out on a component of about 6K LLOC. We use the term LLOC to indicate all logical lines, thus excluding white space and comments. In our experiment, we achieved a reduction of 26% of LLOC, compared to the orignal code. In our case study, we only implemented three aspects. If we would also address parameter checking and error handling, the reduction would be up to 50%. The migration path, especially for the error handling concern, is more difficult as the error handling code is tangled with the base code.

4.2 Local deviations

The solution we proposed still allows for local deviations. For example, there are functions which are called often and for these functions tracing is turned off. There are also low level components which do not trace, as these are active before the logging and tracing component is initiated. Although we do not endorse these deviations, one can implement these deviations with the use of annotations and a more powerful pointcut description language. Most aspects discussed here implement coding guidelines. These should therefore be enforced and applied uniformly. For this reason, the usage of local deviations has to be constrained. However, there are situations where local deviations are necessary from a business point of view.

4.3 Better software structure

This benefit has been stated as one of the corner stone benefits of AOP and separation of concerns in general. The pointcut and advice mechanisms provides an easy way to deal with changing coding guidelines. The benefit of AOP was clearly visible when we applied the profiling aspect on a different component (without profiling) within 10 minutes. Although this flexibility, in advice, can partly be achieved by using macros in C. Macros do not offer a pointcut like mechanism, one always has to import or include the macros explicitly in each file and place the macro calls in the correct locations.

5. CONCLUSION

In this paper we presented the preliminary results of a case study which we carried out with ASML. Our goal with the case study was to show how AOP can increase the software structure, reduce the code size and thus the development and maintenance effort. We took a small but representative component and used AOP techniques to address crosscutting concerns. We clearly showed that for a few concerns the benefits are quite substantial. ASML is currently investigating the use of AOP within their development process.

This work has been carried out as part of the Ideals project under the responsibility of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program.

6. REFERENCES

[1] ASML. ASML. http://www.asml.com/.

[2] T. U. Eindhoven. TUE. http://www.tue.nl/.

[3] E. S. Institute. ESI. http://www.esi.nl/.

[4] U. of California. ASPECTBROWSER. http://www-cse.ucsd.edu/users/wgg/Software/AB/.

[5] U. of Twente. UNIVERSITY OF TWENTE. http://www.utwente.nl/.

[6] U. of Twente. WEAVEC. http://weavec.sourceforge.net.

[7] C. voor Wiskunde en Informatica. CWI. http://www.cwi.nl/.