Isolating Idiomatic Crosscutting Concerns

Magiel Bruntink, Arie van Deursen,* Tom Tourwé†
Centrum voor Wiskunde en Informatica
P.O. Box 94079, 1090 GB Amsterdam
The Netherlands
{Magiel.Bruntink,Arie.van.Deursen,Tom.Tourwe}@cwi.nl

Abstract

This paper reports on our experience in automatically migrating the crosscutting concerns of a large-scale software system, written in C, to an aspect-oriented implementation. We present a systematic approach for isolating crosscutting concerns, and illustrate this approach by zooming in on one particular crosscutting concern. Additionally, we compare the already existing solution to the aspect-oriented solution, and discuss advantages as well as disadvantages of both in terms of selected quality attributes. Our results show that automated migration is feasible, and that adopting an aspect-oriented approach can lead to significant improvements in source code quality, if carefully designed and managed.

1. Introduction

Aspect-oriented software development (AOSD) [14] aims at improving the modularity of software systems, by capturing inherently scattered functionality (often called *crosscutting concerns*) in a well-modularised way. In order to achieve this, aspect-oriented programming languages add an extra abstraction mechanism, called an *aspect*, on top of existing modularisation mechanisms such as functions, classes and methods.

In the absence of aspects, crosscutting concerns are implemented explicitly using more primitive means, such as naming conventions and coding idioms (an approach we refer to as the *idioms-based approach* throughout this paper). The primary advantage of such techniques is that they are lightweight, i.e., they do not require special-purpose tools or languages, are easy to use, and allow developers to recognise the concerns in the code readily. The downsides however are that these techniques require a lot of discipline, are particularly prone to errors, make concern code evolution time consuming and often lead to code duplication [3].

In this paper, we report on a case study involving a large-scale, embedded software system written in the C programming language, featuring a number of typical crosscutting concerns implemented using the idioms-based approach. Our first aim is to investigate how this idioms-based approach can be turned into a full-fledged aspect-oriented approach automatically. In other words, our goal is to provide tool support for identifying the concern in the code, for implementing it in the appropriate aspect(s), and for removing all idiom traces from the code. Our second aim is then to evaluate the benefits as well as the penalties of the aspect-oriented approach over the idioms-based approach. We do this by comparing the quality of both approaches in terms of scalability, code quality and maintainability.

This paper is laid out as follows. The next section presents our approach to the problem of isolating crosscutting concerns, together with three adoption strategies. This approach is illustrated by looking at one particular concern, parameter checking, explained in Section 3. Section 4 presents the domain-specific aspect language we implemented for the parameter checking concern, and Section 5 discusses the migration of the idioms-based approach to the aspect-oriented approach. Section 6 then shows the results of applying our approach to our case study, allowing us to evaluate the approach and to compare the resulting AOSD solution to the current solution in Section 7. Finally, Section 8 discusses related work, and Section 9 presents our conclusions and future work.

2. Approach

2.1. Overview

The systematic approach we propose for isolating crosscutting concerns is illustrated in Figure 1.

The right part of the figure contains the target solution, in which the crosscutting concern is defined in a well-modularised way by means of a specification in a aspect-oriented domain-specific language (ADSL). The ADSL

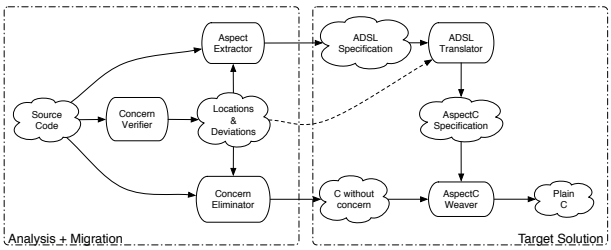

Figure 1. An overview of our isolation approach

code and the pure C are merged together automatically by means of a code weaver. In order to do this, the ADSL specification is translated to the general purpose AspectC language, which then allows us to reuse an existing AspectC weaver. Note that directly expressing the crosscutting concerns in AspectC is not always feasible or desirable, as we will see in Sections 4 and 6.

The left-hand side of Figure 1 describes the steps that make it possible to migrate existing C components to the ADSL solution.

The key step is the *concern verifier*, which is capable of checking the proper use of a coding idiom. It not only detects the locations where the idiom is actually used, but also identifies *deviations*, i.e., places where it thinks the idiom should have been used but for one reason or another was not. Note that such deviations may be on purpose in certain cases. Since most coding idioms do not provide mechanisms to explicitly indicate intended deviations, we must rely on a domain expert to separate deviations reported by the verifier into intended and unintended deviations. This is the only activity in our approach requiring human intervention.

Subsequently, the results of the verifier are used to generate the appropriate ADSL and C code. The *aspect extractor* uses the locations and deviations to come up automatically with an ADSL specification of the crosscutting concern at hand. The *concern eliminator* uses the locations to remove idiom code from the C code automatically. Clearly, the verifier, extractor, eliminator and translator need knowledge about the concern. Therefor, the approach is concern dependent, but can be instantiated for many different concerns.

In order to minimise the risk of introducing subtle errors in the course of the migration, the migrated code obtained through the ADSL should be as similar as possible to the original code. This is suggested by the dashed arrow in Figure 1, which indicates that locations found for idioms can be reused in the translator in order to put concern code back at the original position. In Section 5 we will return to this issue of *conservative migration*.

2.2. Adoption Strategies

Our tools (concern verifier, aspect extractor, concern eliminator and ADSL translator) can be adopted in several ways:

No automated weaving: The tools are used for analysis purposes only. They assist the developer in verifying that he used the idiom correctly and consistently. The tools may produce C code that the developer can copy-paste into the sources if he chooses so.

The benefit of this approach is that it is low risk: developers do what they always did, but are now supported by a concern verifier and an example code generator.

Full automation: All concern code is specified using the ADSL and is eliminated from the existing code base, automatically transformed into the DSL, and then woven back into the C code. New applications directly use the DSL.

This will generally be considered a high risk endeavour, since it implies making modifications to the full code base.

Hybrid: An aspect-oriented approach is adopted for some components, while an idiom-based approach is used for others.

This is possible since the code produced by the aspect-oriented weaver is fully compatible with the original idiom.

3. Parameter Checking

3.1. Industrial Context

The system on which we perform our case study is an embedded system developed at ASML, the world market leader in lithography systems, where reliability and maintainability are key quality attributes. For that reason, ASML adopted a number of coding conventions. One of them is the parameter checking concern.

The entire software system consists of more than 10 million lines of C code. Our case study, however, is based on five subsystems, comprising about 160,000 lines of code.

3.2. The Parameter Checking Concern

The requirement for the parameter checking concern is that each parameter that has type pointer and is defined by a public (i.e., not declared `static`) function should be

checked before it is dereferenced. The purpose of such checks it to improve the reliability and error reporting of the software system.

The ASML code distinguishes between four different kinds of parameters: *input, output* and the special case of *output pointer* parameters, and *in/out* parameters. Input parameters are used to pass a value to a function, and can be pointers or values. Output parameters are used to return values from a function, and are represented as pointers to locations that will contain the result value. The actual values returned can be references themselves, giving rise to a double pointer. The latter kind of output parameters are called *output pointer* parameters. In/out parameters are parameters that are used as both input and output parameters.

The implementation of a check depends on the kind of parameter. An input parameter should contain an actual value, i.e., should not be `NULL`. An output parameter should point to an existing variable or location, i.e., should not be `NULL`. An output pointer parameter may not point to a location that already contains a value (in order to reduce the chance of memory leaks). If these preconditions are not met, an error value is assigned to a dedicated error variable, and an appropriate error message is logged. Our tools do not currently deal with in/out parameters.

Note that the requirement does not specify where exactly a parameter should be checked. This can be done in the function itself, or alternatively anywhere in the call graph of the function, as long as a check occurs before a dereference.

3.3. Coding Idiom Used

Parameter checks occur at the beginning of a function and are similar to:

```
if(queue == (CC_queue *) NULL) {
   r = CC_PARAMETER_ERR;
   CC_LOG(r,0,("%s: Input parameter %s error (NULL)",
          "CC_queue_empty", "queue"));
}
```

where the type cast depends on the type of the variable (`CC_queue *` in this case). The second line sets the error that should be logged, and the third line reports that error in the global log file. It is not strictly specified which string should be passed to the `CC_LOG` function. Checks for output parameters look almost the same, except for the string that is logged. Checks for output pointer parameters look as follows:

```
if(*item_data != (void *) NULL) {
   r = CC_PARAMETER_ERR;
   CC_LOG(r,0,("%s: Output parameter %s may already "
          "contain data (!NULL). This data will "
          "be overwritten, which may lead to memory "
          "leaks.", "queue_extract", "item_data"));
}
```

The only difference with the previous test lies in the condition of the `if`, which now checks whether the dereferenced parameter already contains some data (`!= NULL`), and in the string that is written to the log file.

Parameter checking is representative for several other idioms in use at ASML. Since it is one of the simplest, it is suitable for conducting first experiments with our approach.

4. A Domain-Specific Language for the Parameter Checking Concern

In order to arrive at a more rigorous treatment of parameter checking, we propose an ADSL which we have baptised PCSL,[1] the *Parameter Checking Specification Language*. In this section we describe the language and corresponding tool support[2] — in the next we explain how existing components can be migrated to this target solution.

Observe that language engineering issues are not the focus of the present paper. Thus, issues such as interoperability with other languages are not considered. Also note that the language (and the corresponding translator) is not specific to our case nor to code from ASML: it can be used in any other application involving aspects working on parameters (such as pre- and postcondition checking, for example).

4.1. Specification

The idea underlying PCSL is that a developer annotates a function's signature, by documenting the specific kind of its parameters, i.e., either input, output or output pointer. When a parameter does not require a check, for whatever reason, this can be indicated as well.

As an example, consider the (partial) specification of the parameter checking aspect for one of the considered components, as depicted in Figure 2. It states that the parameters `CC_queue *queue` and `void **queue_data` of the function `CC_queue_peek_front` are output and output pointer parameters, respectively, and that parameter `CC_queue *queue` of function `CC_queue_init` is an output parameter, whereas parameter `void *queue_data` does not need to be checked.

The aspect only documents the public functions of the component, since these are the only ones that need to have their parameters checked, according to the requirement. Of course, the actual checks themselves can occur in non-public functions.

Besides the signature specification, the developer can specify *advice code*, i.e., the code that will perform the actual check. Since this code can differ for the different kinds of parameters, we allow advice code for input, output and

[1] PCSL is most easily pronounced pixel.

[2] The PCSL tools can be obtained by contacting the authors.

output pointer parameters to be specified separately. Although in this paper we do not need it, PCSL also has provisions to express advice code for deviations.

The special-purpose `thisParameter` variable used in the advice denotes the parameter currently being considered by the aspect, and exposes some context information, such as the name and the type of the parameter and the function defining it. In this respect, it is similar to the `thisJoinPoint` construct in AspectJ. Due to the generality introduced by this variable, we only need to provide *three* advice definitions in order to cover the implementation of the concern in the complete ASML source code. As a comparison, using the general-purpose AspectC language directly instead of PCSL would require providing different advice code for each parameter. In other words, the `thisParameter` variable is one of the main reasons why we need a domain-specific aspect-oriented language as opposed to a general-purpose one.

4.2. Translation to AspectC

PCSL code is automatically transformed into AspectC code, which in turn is woven with the C code containing the implementation of the primary functionality (as seen in Figure 1).

The AspectC weaver we use is a stripped-down variant of the AspectC language defined by [5]. It has only one kind of joinpoint, function execution, and allows us to specify around advice only. Of course, before and after advice can be simulated easily using such around advice. Figure 3 contains an example that shows how the `advice on` keyword is used to specify advice code for a particular function.

The translation is implemented in ASF+SDF [2], a general program transformation tool that includes a C grammar, and proceeds as follows. For each parameter that does not have the `deviation` annotation, the translator looks up the parameter's kind, retrieves the corresponding advice code, and expands that code into the actual check that should be performed.

The expansion phase is responsible for assembling and retrieving the necessary context information (i.e. setting up the `thisParameter` variable), and substituting it in the advice code where appropriate. At the end, this advice code will call the original function by calling the special `proceed` function, but only if none of the parameters contain an illegal value (i.e. the error variable is still equal to the `OK` constant).

An illustration of the translation of the specification of Figure 2 is given in Figure 3. An input and an output parameter check are added to the `CC_queue_empty` function for its `queue` and `empty` parameters, respectively.

5. Migration Support

The aspect solution as described in the previous section can be used when new components are built, as shown in the right part of Figure 1. However, the majority of the software development activities at ASML (and, in fact, at most companies) is not devoted to greenfield engineering, but to adjusting existing components. In order to achieve the same benefits for such existing components, we describe how these can be migrated to an ADSL solution. In particular, we discuss how concern code can be recognized in the original implementation, how an aspect can be created from it, and how the original concern code can be removed from the C code.

5.1. Concern Verification

The *parameter checking* concern verifier is used to "characterise" the source code: it infers where parameter checks should be present, according to the requirements, and verifies whether these checks are there. If a check is present, its location is reported, and if it is not, a deviation is reported.

The resulting list of deviations is inspected by a domain expert, who classifies a deviation as either *intended* or *unintended*. Intended deviations signal parameters that do not need a check, for example because the function has been designed such that the value null for the given parameter is meaningful, or because the check is considered too expensive in a performance-critical function. Unintended deviations signal a violation to the parameter checking requirement. Note that this manual step is required because even complex program analysis techniques do not suffice to recognise an intended from an unintended deviation.

The verifier for the parameter checking idiom has been developed as a plugin for the *CodeSurfer* source code analysis and navigation tool [3]. CodeSurfer can construct program dependence graphs for systems written in C, and provides a programmable interface to navigate through such graphs.

The verifier first extracts the parameter kinds from the source code. All formal parameters of all user-defined functions are considered, and checked whether they are assigned somewhere in the function's call graph. If they are, they are output parameters, otherwise they are input parameters. The functionality required for implementing this algorithm, such as computing the *killed set* of a parameter (i.e. the location in the code where a parameter's value is overwritten, if that exists) and the call graph of a function is provided by CodeSurfer library functions.

Once the parameter kinds are known, the plugin considers each parameter of a function. It traverses the control

[3] www.grammatech.com

```
component CC {
     CC_queue_peek_front(output CC_queue *queue, output output-pointer void **queue_data);
     CC_queue_empty(input CC_queue *queue, output bool *empty);
     CC_queue_init(output CC_queue *queue, deviation void *queue_data);
     ...
     input advice {
        if(thisParameter.name == (thisParameter.type) NULL) {
           r = CC_PARAMETER_ERR;
           CC_LOG(r,0,("%s: Input parameter %s error (NULL)", thisParameter.function.name, thisParameter.name));
        }
     }
     output advice {
        if(thisParameter.name == (thisParameter.type) NULL) {
           r = CC_PARAMETER_ERR;
           CC_LOG(r,0,("%s: Output parameter %s error (NULL)",  thisParameter.function.name, thisParameter.name));
        }
     }
     output-pointer advice {
        if(*thisParameter.name != (thisParameter.type*) NULL) {
           r = CC_PARAMETER_ERR;
           CC_LOG(r,0,("%s: Output Pointer parameter %s error",  thisParameter.function.name, thisParameter.name));
        }
     }
}
```

Figure 2. PCSL specification of the parameter checking concern

```
int advice on (CC_queue_empty) {
     int r = OK;
     if(queue == (CC_queue *) NULL) {
        r = CC_PARAMETER_ERR;
        CC_LOG(r,0,("%s: Input parameter %s error (NULL)", "CC_queue_empty", "queue"));
     }
     if(empty == (bool *) NULL) {
        r = CC_PARAMETER_ERR;
        CC_LOG(r,0,("%s: Output parameter %s error (NULL)", "CC_queue_empty", "empty"));
     }
     if (r == OK)
        r = proceed();
     return r;
}
```

Figure 3. AspectC code generated for the PCSL specification

flow graph of the function in order to verify whether a parameter check is encountered before the parameter is dereferenced. When parameters are passed on to other functions, the plugin considers the control-flow graph of those other functions as well. Since following such inter-procedural paths is time consuming, we employ caching to ensure each path is followed only once.

5.2. Aspect Extraction

Since the verifier exactly finds out which parameters need to be checked and what their kind is, we can use its output to generate a PCSL specification automatically for the verified component.

The PCSL code to be generated consists primarily of the signature declarations indicating the kind of each parameter. The actual advice code for the three kinds does not differ per parameter, and can be simply appended to the generated signatures.

For convenience, we implemented the aspect extractor as an extension to the concern verifier, i.e., also as a CodeSurfer plugin.

5.3. Concern Elimination

Besides extracting the aspect specification, the code originally implementing the concern has to be removed from the source code as well.

The locations obtained by the verifier indicate where the parameter checks occur, and could be used for this purpose. Although line numbers for relevant statements are thus available, the precise start and end points of the checking code are not always known. For example, brackets of compound statements are not included in CodeSurfer's code representation.

In order to solve this issue, we implemented a parameter checking eliminator in ASF+SDF [2]. Transformations recognise parameter checking code that obeys the coding idiom explained in Section 3, and subsequently remove such code.

Although the concern eliminator obtained in this way works perfectly well, it is somewhat unsatisfactory as it effectively re-implements (a small) part of the verifier, namely the part recognising existing checks. We are currently looking at techniques to integrate the program analysis capabilities of CodeSurfer with the program transformation capabilities of ASF+SDF.

5.4. Conservative Translation

Besides straightforward translation (i.e., adding a check for each input, output and output pointer parameter in each public function), the PCSL translator also implements conservative translation: it can define an AspectC aspect that reintroduces the parameter checks at exactly the same locations as where they were found originally. Naturally, this is only possible for parameters that were already checked originally.

Conservative translation is based on information obtained from the verifier (as explained above and indicated in Figure 1). Four different situations occur for a parameter p of a function f:

1. the parameter p was checked in function f;
2. the parameter p was checked, but not in function f;
3. the parameter p was not checked, but was registered as an unintended deviation;
4. the parameter p was not checked, but was registered as an intended deviation.

In the last case, nothing needs to be done as no check is needed. In the second case, the function where p was checked is fetched, and we end up in case 1. In the first and third case, a check for p is added to f.

6. Case Studies

In this section, we present the results of running our tools on selected ASML components. The next section then discusses the lessons learned based on these results.

6.1. Intended and Unintended Deviations

An overview of the parameter checking verifier results is provided in Table 1. The top half of the table lists systems for which the results have been discussed with system developers, and for which we have accurate figures on the intended versus unintended deviations.

	deviations detected	unintended deviations	intended deviations
CC1 (3 kLoC)	8	0	8
CC2 (19 kLoC)	65	58	7
CC3 (12 kLoC)	23	16	7
CC3 (98 kLoC)	53	41	12
CC4 (14.5 kLoC)	31	24	7
CC5 (15 kLoC)	5	4	1

Table 1. Parameter checking results

As can be seen, our verifier reports that 32 of the parameters of the CC1 component must be checked (recall that only pointers need to be checked). 8 deviations are reported, all of which are considered to be intended deviations. For the CC2 component, 238 parameters need to be checked, 65 of which are reported as deviations, and manual inspection eliminated 7 intended deviations. Due to timing constraints, results for the CC3 component are only confirmed for a 12 kLoC subset. 23 deviations are reported, 16 of which are unintended.

The bottom half of Table 1 lists systems to which we have applied our tools, but for which we do not have confirmed figures on the number of (un)intended deviations. The figures listed are estimates, obtained by extrapolating the results from the top half of the Table, which indicate that 74 out of 96 (77%) deviations are considered unintended. Thus, we predict that 69 unintended deviations will be present in the remainder of the CC3 component and in the CC4 and CC5 components.

6.2. Coding Idiom Conformance

The verifier recovers all parameter checks from the code, which allows us to assess how consistent the various developers have implemented the coding idiom explained in Section 3.3.

Most components implement the checks in the same way, but each component logs different strings, even for the same parameter error. The CC1 component logs two different strings, for instance, and the CC2 component logs 30 different strings. The CC3 component and the CC4 component log 15 and 5 strings, respectively, whereas the CC5 component does not log any string. This is due to the fact that it does not contain any parameter checks. The component defines few public functions and hence requires few checks, which do not seem to be implemented.

The reason CC2 uses many more strings than the other components is due to the fact that the logged errors in CC2 are specific to the kind of parameter, whereas other components use generic strings.

	CC1	CC2	CC3	CC4	CC5
Original C code	56	961	456	133	0
PCSL code	46	132	787	166	75
AspectC code	122	1200	1214	272	223

Table 2. Lines of code figures for various parameter checking representations

6.3. Code Size

Table 2 shows the difference measured in lines of concern code [4] between the idioms-based, PCSL and AspectC approaches. Surprisingly, we found that using an aspect-oriented approach, based on a domain-specific or on a general-purpose aspect language, does not necessarily reduce the code size of the components. This contrasts sharply with an often (informally) claimed benefit of aspect-oriented software development.

For the PCSL approach, we see that the code size of the CC2 component is reduced significantly, but that this is not the case for the other components. The CC3 specification has 72% more lines of code than the original C code. The reason is that the number of parameter checks in CC3 is relatively low compared to the number of parameters that need to be checked, but that in PCSL all (public) function signatures should be annotated. This situation is even more apparent for the CC5 component, which does not implement any checks, whereas the PCSL specification consists of 75 lines of code. We will further discuss this issue in Section 7.3.

With respect to the AspectC solution, we observe that all aspects, except the CC5 aspect, require more lines of code than the idioms-based approach. In other words, it requires an extra amount of code to implement, and contains just as much duplication as the idioms-based approach (as shown in Figure 3). Admittedly, our AspectC code is generated and does not try to factor out commonalities using generic pointcuts. This is very difficult to achieve with generic languages such as AspectJ or AspectC, because they currently lack the necessary features and flexibility for implementing concerns such as parameter checking [1].

7. Evaluation

This section reflects on the lessons learned when applying our approach to the ASML components and when comparing the idioms-based approach with the aspect-oriented approach.

[4] Including whitespaces.

7.1. Scalability

The idioms-based approach does not scale.

The results of running the parameter checking verifier show that most of the reported deviations are unintended deviations. These results prove that our verifier is reliable and worthwhile to consider in a code reviewing activity. Additionally, this shows that the idioms-based approach is error-prone and does not scale, even for simple concerns, and that a more rigorous treatment for parameter checking is needed.

We can only speculate about the reasons why so many unintended deviations are present. The following factors seem to be important:

- The size of the component. The smallest component (CC1) contains no deviations, whereas the largest component (CC3) contains the most deviations;
- The age of the component and the amount of evolution it has undergone. The CC1 component was redeveloped completely recently, whereas CC3 is already quite old and has undergone many changes;
- The number of programmers working on the component. Components developed by a small team of developers do not exhibit many unintended deviations, whereas components developed by many different developers show many more deviations.
- The cost of adding a check manually compared to the benefits it may provide to the developer. The potential benefits of the check are not experienced by the developers themselves, but by potential clients of the component who would be helped with a parameter warning upon improper use of the function.

More work is required to pinpoint which of these factors influences the results the most.

7.2. Code Quality

An AOSD approach improves code quality by minimising code duplication, improving uniformity and understandability and reducing scattering and tangling.

Minimising Code Duplication In a separate experiment [3], we evaluated the amount of code duplication in a number of crosscutting concerns in the CC2 component, among which the parameter checking concern. The results confirm the common belief that the idioms-based approach leads to a large amount of duplication. The specific reason for the duplication is that, due to the crosscutting nature of

the code, reuse of that code is not possible in ordinary programming languages, since it doesn't fit in a module.

By using aspect-oriented techniques, however, reuse becomes possible again. This is reflected by the fact that in our PCSL specification of the parameter checking concern, the advice code for each kind of parameter is specified only once and can be reused.

Note that we specifically devised PCSL with maximal reuse of advice code in mind. Using a current-day general-purpose aspect language would make it much harder to reuse the code and avoid duplication, however. This is discussed in [4] and shows that simply using AOSD techniques will not necessary reduce (crosscutting) code duplication.

Improving Uniformity The results in Section 6.2 show that the coding idiom for parameter checking is not strictly adhered to. Although uniform checks are not that important, uniform error strings are. Those strings are used by automated tools, that reason about the logged errors in order to identify and correct the primary cause of a particular error. The idioms-based approach clearly aggravates this task.

In the AOSD solution, the advice code specifies how a parameter should be checked, and this code is specified only once for each component, and reused afterwards. Consequently, all parameters of a component are checked and logged in the same way.

Improving Understandability As the number of intended deviations is limited, documenting such exceptions is important. For example, we observed that most intended deviations for output pointer parameters are due to the parameter being used as a *cursor* when iterating over a composite data structure. Since the parameter points to an item in the list, it doesn't matter that it's value is overwritten, and hence, no output pointer check is needed. With the idioms-based approach, such information is only implicitly present in the source code, and is thus easily overlooked. Explicitly capturing such information in an aspect improves the understandability of the code.

Reducing Scattering and Tangling

The parameter checking concern is clearly scattered over many different functions and files, since many functions implement the idiom. The aspect-oriented solution cleanly captures the concern in a modular and centralised way in an aspect, and thus removes the scattering all together. This also eliminates the *tangling* that is present in the C solution: without PCSL, many functions start with approximately five lines per parameter. This code is unrelated to the key concern to be handled by the function, and causes unnecessary distraction for the developer.

7.3. Maintainability

An AOSD approach introduces additional maintainability risks.

A potential risk when separating the aspect code from the base code is that the two get out of sync: when a component evolves, its associated aspects do not. This is an important issue, as our experience suggests that developers are reluctant to adopt a new technology when it introduces additional risks of inconsistency.

The simple remedy we adopted is to include sanity checks in the ADSL translator, which can warn about non-existing functions and parameters, or non-matching signatures. The result is certainly more consistent than the current practice, which is to include parameter kind declarations in comments that are not automatically processed.

A complementary solution came up when talking to ASML developers. They suggested that the function signature annotation and advice code specification can best be separated. The rationale is that the advice code almost never changes, but the annotations will change more frequently. Additionally, they suggested to allow annotations only for parameters that do not need to be checked. The majority of the functions behaves normally with respect to the idiom, and does not need annotation, since we can infer parameter kinds automatically from the source code. As such, the required lines of code for the PCSL specification will be reduced significantly.

In a future version of PCSL, we will thus allow developers to specify intended deviations inside structured comments, near the function definition itself, whereas the advice code will still be defined in a separate file. This might still introduce consistency problems: a parameter that deviates from the idiom in one version of the software might adhere to the idiom in the next, and vice versa, which requires the developer to update the annotation. This situation will not occur frequently, however, because deviations are scarce and occur only in very specific circumstances (such as for example, a cursor in a list). Additionally, we believe developers will be motivated to keep the annotations in sync with the current implementation, as this helps them to achieve a correct parameter checking concern in a rather easy way.

7.4. Change Management

Adoption of AOSD requires different adoption scenario's and change management.

We have observed that (ASML) developers are reluctant toward adopting a new and (for them) largely unknown solution. The same situation probably occurs in other software companies, and with other tools.

The different adoption strategies incorporated in our approach are expected to alleviate this problem. We expect that the adoption of our techniques will start with a non-weaving approach only. Once developers and managers get familiar with the use of PCSL and the parameter checking verifier, we expect that they will get interested in using automated weaving for certain components, thus adopting the hybrid approach. After this has been successful for a significant number of components, we anticipate a migration effort to the fully automated approach, thus eliminating the need for any hand-written parameter checks.

8. Related Work

Our parameter checking verifier resembles tools that verify the quality of the source code. A number of tools for this purpose have been developed over the years, [12, 18]. Most of them are only able to detect basic coding errors, such as using = instead of ==, and are incapable of enforcing domain-specific coding rules. More advanced tools exist [8, 13, 16, 19], but these are restricted to detecting higher-level design flaws in object-oriented code. Tools which are capable of checking custom (domain-specific) coding rules are described by [7] and [9].

There is also some similarity with tools from the area of *plan recognition* [20, 6]. Such tools are parameterised with a library of "program plans", typical ways of solving known programming problems, which bear some resemblance with our idioms. Plans are typically described by data or control dependencies, as done, for example in [6] to characterise leap year computations in Cobol code. The recogniser can then search for plan instances in a code base. In our case we decided to characterise the idioms in Scheme as a CodeSurfer plugin, which provided the necessary flexibility and was readily available.

The work described in this paper has some similarities with work done by Coady *et. al.* [5], who describe how they identified *prefetching code* in the FreeBSD OS kernel, and propose a new solution in terms of AspectC. Their work does not focus on a general approach for isolating crosscutting concerns, since they restructured the code manually in an ad-hoc way.

A number of other studies have investigated the applicability of aspect-oriented techniques to various (domain specific) crosscutting concerns. [17] proposes guidelines for preparing the code for isolating concerns and performing the necessary restructurings. [15] targets exception detection and handling code in a large Java framework. Both works discuss advantages of using AOSD, such as reduced code duplication and improved cohesion, and discuss some particular limitations of using AspectJ. In [15], the aspect solution does reduce the code size, contrary to our findings.

An approach to refactoring which specifically deals with tangling is presented by Ettinger and Verbaere [10]. Their work shows how slicing techniques can help automate restructuring of tangled (Java) code, and could be a good candidate to include in our approach, when we are dealing with more complex concerns. [11] provides a more general discussion of both refactoring in the presence of aspects, and refactoring of object-oriented systems toward aspect-oriented systems.

9. Concluding Remarks

Contributions The key contribution of this paper is a systematic approach for isolating crosscutting concerns from existing source code. We illustrated the effectiveness of this approach by considering the parameter checking concern, which resulted in:

1. a proposal for PCSL, a domain-specific language for implementing concerns dealing with parameters;

2. insight into the use of the idioms-based approach in an industrial setting, which showed its shortcomings: the approach does not scale and leads to inconsistencies;

3. insight into the benefits and pitfalls of the AOSD approach: improved source code quality at the cost of additional maintainability issues and required change management and adoption strategies.

Future Work The focus of this paper is on a specific concern (parameter checking) in five components from the ASML source code. We are presently extending the scope of our work in various directions:

- We are in the process of applying this approach to a larger number of components within ASML;

- Parameter checking is a concern that is interesting outside ASML as well. Our approach is mostly generally applicable. The only ASML-specific elements are localised in (1) the places in the verifier where existing checks are recognised; and (2) the specific advice specified in the ADSL. Both can be easily changed, making the approach applicable to, for example, open source systems in which parameter checking advice should consist of C `assert` statements.

- The next concern on our list is *error handling*. This concern is significantly more complicated than parameter checking (its implementation being much more tangled). However, it turns out that exactly the same approach (albeit it with different underlying algorithms and analysis techniques) can be applied, including a verifier, DSL, and migration tools.

Acknowledgements We would like to thank the ASML developers for discussing the results of the case study, and all members of the Ideals project team for input about the topic and proofreading drafts of this paper. Thanks to Kris De Schutter for providing us with his yerna-lindale aspect weaver. This work has been carried out as part of the Ideals project under the auspices of the Embedded Systems Institute. This project is partially supported by the Netherlands Ministry of Economic Affairs under the Senter program.

References

[1] B. Adams and T. Tourwé. Aspect-Orientation in C: Express Yourself. In *Proceedings of the AOSD Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT)*. Aarhus University, March 2005.

[2] M. van den Brand, A. van Deursen, J. Heering, H. Jong, M. Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The ASF+SDF Meta-Environment: a Component-Based Language Development Environment. In R. Wilhelm, editor, *Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

[3] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 200–209. IEEE Computer Society, 2004.

[4] M. Bruntink, A. van Deursen, and T. Tourwé. An Initial Experiment in Reverse Engineering Aspects. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 306–307. IEEE Computer Society, 2004.

[5] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to Improve the Modularity of Path-Specific Customization in Operating System Code. In *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-9)*, pages 88–98. ACM Press, 2001.

[6] A. van Deursen, S. Woods, and A. Quilici. Program plan recognition for year 2000 tools. In *Proceedings 4th Working Conference on Reverse Engineering; WCRE'97*, pages 124–133. IEEE Computer Society, 1997.

[7] M. Eichberg, M. Mezini, T. Schfer, C. Beringer, and K. M. Hamel. Enforcing system–wide properties. In *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*, pages 158–167. IEEE Computer Society, April 2004.

[8] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pages 97–106. IEEE Computer Society, 2002.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, San Diego, CA, October 2000. Usenix.

[10] R. Ettinger and M. Verbaere. Untangling: A Slice Extraction Refactoring. In *Proceedings of the Aspect-Oriented Software Development Conference (AOSD)*, pages 93–101. ACM Press, 2004.

[11] S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of Aspect-Oriented Software. In *4th Annual International Conference on Object-Oriented and Internet-based Technologies,Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 19–35. Springer Verlag, 2003.

[12] S. Johnson. Lint, a C Program Checker. Technical Report 65, Bell Laboratories, Dec. 1977.

[13] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated Support for Program Refactoring using Invariants. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 736–743. IEEE Computer Society, 2001.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the Europeen Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[15] M. Lippert and C. Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 418–427. IEEE Computer Society, 2000.

[16] R. Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, 2002.

[17] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 275–284. IEEE Computer Society, 2001.

[18] S. Paul and A. Prakash. A Framework for Source Code Search using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

[19] T. Tourwé and T. Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 91 – 100. IEEE Computer Society, 2003.

[20] L. M. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, 1992.