




Centrum voor Wiskunde en Informatica

View metadata, citation and similar papers at core.ac.uk

brought to you by  CORE

provided by CWI's Instituut

REPORTRAPPORT

SEN

Software Engineering



Software ENgineering

Isolating crosscutting concerns in system software

M. Bruntink, A. van Deursen, T. Tourwé

REPORT SEN-R0504 FEBRUARY 2005

CWI is the National Research Institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organization for Scientific Research (NWO).

CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

Copyright © 2005, Stichting Centrum voor Wiskunde en Informatica

P.O. Box 94079, 1090 GB Amsterdam (NL)

Kruislaan 413, 1098 SJ Amsterdam (NL)

Telephone +31 20 592 9333

Telefax +31 20 592 4199

ISSN 1386-369X

Isolating crosscutting concerns in system software

ABSTRACT

This paper reports upon our experience in automatically migrating the crosscutting concerns of a large-scale software system, written in C, to an aspect-oriented implementation. We zoom in on one particular crosscutting concern, and show how detailed information about it is extracted from the source code, and how this information enables us to characterise this code and define an appropriate aspect automatically. 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 can lead to significant improvements in source code quality.

1998 ACM Computing Classification System: D.2.7 Distribution, Maintenance, and Enhancement
Keywords and Phrases: aspect-oriented software development; reverse engineering; refactoring

Isolating Crosscutting Concerns in System Software

Magiel Bruntink
Centrum voor Wiskunde en
Informatica
P.O. Box 94079
1090 GB Amsterdam, NL
Magiel.Bruntink@cwi.nl

Arie van Deursen^{*}
Centrum voor Wiskunde en
Informatica
P.O. Box 94079
1090 GB Amsterdam, NL
Arie.van.Deursen@cwi.nl

Tom Tourwé
Centrum voor Wiskunde en
Informatica
P.O. Box 94079
1090 GB Amsterdam, NL
Tom.Tourwe@cwi.nl

ABSTRACT

This paper reports upon our experience in automatically migrating the crosscutting concerns of a large-scale software system, written in C, to an aspect-oriented implementation. We zoom in on one particular crosscutting concern, and show how detailed information about it is extracted from the source code, and how this information enables us to characterise this code and define an appropriate aspect automatically. 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 can lead to significant improvements in source code quality.

1. INTRODUCTION

Aspect-oriented software development (AOSD) [18] aims at improving the modularity of software systems, by capturing crosscutting concerns in a well-modularised way. In order to achieve this, aspect-oriented programming languages add an extra abstraction mechanism, an *aspect*, on top of already existing modularisation mechanisms such as functions, classes and methods.

In the absence of such a mechanism, crosscutting concerns are implemented explicitly using more primitive means, such as naming conventions and coding idioms (an approach we will 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, are easy to use, and allow developers to readily recognise the concerns in the code. The downside however is that these techniques require a lot of discipline, are particularly prone to errors, make concern code evolution extremely time consuming and often lead to code size explosion.

In this paper, we report on an experiment involving a large-scale, embedded software system written in the C programming language,

^{*}Also affiliated with Delft University, Software Evolution Research Laboratory (SWERL), Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS), Mekelweg 4, 2628 CD Delft, The Netherlands.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

AOSD 2005, Chicago, USA. Submitted
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

that features a number of typical crosscutting concerns implemented using naming conventions and coding idioms. 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, implementing it in the appropriate aspect(s), and removing all its 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 the amount of tangling, scattering and code duplication, the lines of code devoted to the concern and the correctness and consistency of its implementation.

1.1 Approach

Our approach to achieving our goals is to zoom in on one particular crosscutting concern, the *parameter checking* concern. Based on the existing source code and the requirements extracted from the manuals, we implement a *concern verifier* for the parameter checking concern. Its primary task is to reason about the current implementation of the concern in order to “characterise” it: the verifier reports where the code deviates from the standard idioms, which allows developers to correct the code when necessary. Manual inspection may also reveal that a particular deviation is in fact on purpose, in which case it will be marked as *intended*. Additionally, the verifier also recovers the specific locations where particular parameters are checked.

The information recovered by the concern verifier is used by the *aspect extractor* and the *concern eliminator*. The former defines an appropriate aspect for the parameter checking concern. This aspect will add parameter checks to the source code wherever necessary, and will make sure this code is not added for the intended deviations. The latter will remove the parameter checking concern from the original source code.

The aspect extractor outputs the aspect in a special-purpose aspect language. This definition is then translated automatically by our *AspectC translator* to an already-existing, general-purpose aspect language, that can weave the parameter checking concern back into the source code.

Once the correct aspect has been constructed, we can assess the quality of the aspect-oriented solution and compare that to the idioms-based solution.

1.2 Contributions

The contributions of this paper are the following:

- we evaluate the benefits of an aspect-oriented approach over

an idioms-based approach, in terms of selected quality attributes;

- we show how to verify the idioms-based implementation of crosscutting concerns in the source code;
- we show how crosscutting concerns can be extracted from the source code automatically, based on the information gathered by the concern verifier;
- we define a special-purpose aspect language, that can be used to implement aspects involving function parameters.

1.3 Outline

The remainder of the paper is structured as follows. The next section introduces the parameter checking concern, its requirements and the idioms used to implement it. Section 3 discusses the concern verifier, its implementation, and the results of running it on our case study. Section 4 presents the domain-specific aspect language we implemented for the parameter checking concern, discusses its implementation in terms of an already-existing aspect weaver, and compares this solution to the idioms-based solution. Section 5 then discusses the (conservative) migration of the idioms-based approach to the aspect-oriented approach. Section 6 shows in detail how the evolvability of the system improves thanks to using aspects, while Section 7 considers additional quality attributes to compare the aspect-oriented solution to the idioms-based solution. Finally, Section 8 discusses related work, and Section 9 presents our conclusions and future work.

2. CURRENT PARAMETER CHECKING IDIOM

2.1 Background

The subject system upon which we perform our experiments is an embedded system developed at ASML, the world market leader in lithography systems. The entire system consists of more than 10 million lines of C code. Our experiment, however, is based on a relatively small, but representative, software component (which we will call the *CC* component in this paper), consisting of about 19.000 lines of code.

Because the C language lacks explicit support for crosscutting concerns, ASML uses an idiomatic approach for implementing such concerns, based on coding idioms. As a consequence, a large amount of the code of each component is “boiler plate” code. A code template is typically reused and adapted slightly to the context.

The implementation of the parameter checking concern in the *CC* component consists of 961 lines of code, scattered across the many different functions, which amounts to 5% of the total number of lines of code of the component.

2.2 Parameter Checking Requirements

ASML operates in the embedded systems domain, in which reliability and maintainability are key quality attributes. For that reason, ASML has adopted a number of coding conventions. One of them is the parameter checking concern, which is responsible for implementing pointer checks for function parameters and raising warnings whenever such a pointer contains a non-expected (*NULL*) value. The purpose of such checks is to improve the reliability and the error reporting of the software system. The requirement for the concern is that each parameter that has type pointer and is defined by a public (i.e. not *static*) function should be checked against

NULL values. If a *NULL* value occurs, an error variable should be assigned, and an error should be logged in the global log file. 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 the parameter is checked before its value is used, the requirement is met. Additionally, some exceptions to this requirement exist, as a limited number of functions can explicitly deal with *NULL* values, so the corresponding parameters should not be checked.

The implementation of a check depends on the *kind* of parameter. The ASML code distinguishes between three different kinds: *input*, *output* and the special case of *output pointer* 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. Note that the set of output pointer parameters is a subset of the set of output parameters. Since output and output pointer parameters are always of type pointer, they should always be checked, but only input parameters that are passed as pointers should be checked.

2.3 Idioms Used

Parameter checks occur at the beginning of a function and always look as follows:

```
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 of course depends on the type of the variable (*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 exactly the same, except for the string that is logged.

Since output pointer parameters are output parameters as well, they should also be checked for null values. Additionally, one extra check is required to prevent memory leaks. The requirement at ASML is that output pointer parameters may not point to a location that already contains a value, because the function will overwrite the pointer to that value. Since the original value is then never freed, a memory leak could occur. In order to avoid such leaks, the following test is added for each output pointer parameter:

```
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*, that now checks whether the dereferenced parameter already contains some data (*!= NULL*), and in the string that is written to the log file.

3. CONCERN VERIFIER

The concern verifier is an automated tool that reasons about the idioms-based implementation of the parameter checking concern. This section motivates why we need such an automated tool, explains the information that it recovers from the source code, the

coding idioms used, as well as the implementation of the algorithm that verifies these idioms, and the results of running this algorithm on our case study.

3.1 Motivation

The idioms-based approach requires much effort and strict discipline from the developers. Since the concern does not form part of the core functionality they need to implement, it is not their primary focus. Additionally, the idioms are often not documented explicitly, or only in an informal way, and every developer may thus have his own idea and interpretation of them. Consequently, if the idioms are not strictly enforced in some automated way, developers will inevitably write code that does not adhere to them. If we want to transform the idioms-based approach into an aspect-oriented one, we should thus first “characterise” the implementation. In other words, we should first locate places where parameters checks occur and mandatory parameter checks are missing, and identify parameters that do not need to be checked.

We achieve this characterisation by implementing a *concern verifier* which checks the implementation of the concern with respect to the coding idioms that hold for it. The verifier outputs a list of *locations*, i.e. functions where parameter checks occur, and a list of *deviations*, i.e. locations in the source code that lack a parameter check although it should be present according to the idioms. This latter list is inspected by a domain expert, who identifies the *intended* and *unintended* deviations. The intended deviations indicate exceptional cases (e.g. parameters that are allowed to be NULL), whereas unintended deviations indicate parameters for which a check was forgotten and should be implemented. As we will see later on, our concern verifier is able to identify some intended deviations automatically. In those cases, these deviations are not reported, but simply registered as exceptions.

Thus the following important information is recovered from this code:

- the list of intended deviations informs us which parameters form an exception to the rule. As such, this important information becomes explicitly available, whereas it was not before;
- the number of unintended deviations is a measure for the quality of the idioms-based approach. The smaller this number, the better the quality of the implementation. We expect this number to increase linearly with the size of the source code;
- the verifier identifies the specific location in the code where a particular parameter is checked. Remember that the requirement does not specify where the check should occur, as long as it occurs before the parameter is used.

As we will see in the next sections, this information is vital to our aspect extractor. The aspect it defines should add all necessary parameter checks to the code, but should not insert checks for exceptional parameters. Additionally, it should make sure that the aspect preserves the behaviour of the original idioms-based implementation, which it does by simply implementing the checks at the same locations.

We continue this section by explaining the implementation of the concern verifier in more detail. First, we formalise the coding idioms that need to be followed. Afterwards the algorithm that verifies these idioms is presented.

3.2 Formalising the Concern Idioms

In order to verify the proper use of the parameter checking idiom, we propose the following formalisation of the concern:

$$\begin{aligned} &\forall f \in \text{PublicFunctions}(CC) : \\ &\quad \forall p \in \text{parameters}(f) : \text{checksParameter}(f, p) \\ &\text{checksParameter}(f, p) \leftrightarrow \\ &\quad \text{implementsCheck}(f, p) \vee \\ &\quad \forall f_1 \in \text{CalledFunctions}(f) : \\ &\quad \quad \forall p_1 \in \text{parameters}(f_1) \wedge \text{dataDependent}(p_1, p) : \\ &\quad \quad \quad \text{checksParameter}(f_1, p_1) \end{aligned}$$

where *PublicFunctions* is the set of all functions that are globally visible in the component (e.g. the set of functions not defined `static`), *parameters* is the set of (input, output and output pointer) parameters defined by a function, *implementsCheck* is a predicate that returns true if its first argument checks the parameter denoted by its second argument in the appropriate way (depending on the kind of the parameter), *CalledFunctions* computes the set of all functions that can be called by a function *f*, and *dataDependent* verifies whether its first argument is data dependent on its second argument. Clearly, if this coding idiom is strictly adhered to, the parameter checking requirement will be met.

3.3 Verifier Implementation

The concern verifier has been developed as a *plugin* in the *CodeSurfer* source code analysis and navigation tool¹. This tool provides us with programmable access to data structures such as system- and program-dependence graphs, and defines advanced analysis techniques over these structures, such as control- and data-flow analysis and program slicing.

We used CodeSurfer to implement checks for the *checksParameter* predicate presented above. To that end, we need to consider each public function and see if the necessary parameter checks occur in it or in the functions it calls. This requires knowledge about the particular kind of a parameter: whether it is input, output or output pointer. Our verifier first extracts this knowledge from the source code in the following way:

1. for each function, it considers each formal parameter that has pointer type;
2. for each of these parameters, it computes the list of (transitively) called functions, and retains those functions from it that are passed the formal parameter as an actual;
3. for each of the remaining functions, it checks whether the functions assign some value to the parameter in some execution path.

If a formal parameter is assigned to by some function, it is considered to be an output parameter, otherwise it is an input parameter. Output pointer parameters are then just output parameters that have double pointer type.

The algorithm checks for assignments to a parameter by looking for *kill* (or *def*) statements for that parameter inside a function’s body. Since we only consider a single component in our experiment, we do not have access to such information for external functions, however. In order to deal with this, the signature of external functions, together with an annotation of the kind of their parameters, is made available to the algorithm. Although this requires developer intervention, it only concerns a limited number of functions, and could be circumvented if we consider all source code.

¹www.grammatech.com

| | required | actually checked | deviations detected | unintended deviations | intended deviations |
|---------|----------|------------------|---------------------|-----------------------|---------------------|
| input | 57 | 40 | 26 | 17 | 9 |
| output | 143 | 94 | 49 | 49 | 0 |
| out ptr | 45 | 15 | 35 | 30 | 5 |
| total | 245 | 149 | 110 | 96 | 14 |

Figure 1: Number of top level parameter checks found for the CC component.

Once the particular kind of a parameter is determined, we can verify whether the necessary checks for it occur in the implementation:

- for each input parameter of pointer type of a public function, the function itself, or all its called functions that are passed the parameter as an actual, should implement an input parameter check;
- for each output parameter of a public function, the function itself, or all its called functions that are passed the parameter as an actual, should implement an output parameter check;
- for each output pointer parameter of a public function, the function itself or all its called functions that are passed the parameter as an actual, should implement an output and an output pointer parameter check.

If a parameter is not checked, the concern verifier tries to infer if the function is robust for exceptional values, before it registers an unintended deviation. For the parameter `At` at the moment, it uses a simple heuristic: if the function compares the value of a parameter to `NULL` each time before it uses that parameter, we assume it can deal with a `NULL` value. This heuristic does not suffice for identifying all exceptions, however. Distinguishing intended from unintended deviations thus still requires a manual effort. More elaborate heuristics are possible, but are considered future work.

As can be observed, the call graph of a function is used, both for determining the kind of a parameter and for verifying whether the parameter is checked in the appropriate way. Particular functions that are called by many other functions, will thus be considered multiple times by our algorithms. In order to improve performance, both algorithms employ caching techniques to store and reuse values already computed.

3.4 Verification Results

Applying the verifier to the case at hand yields the data displayed in Figure 1. The CC component implements 157 functions, with 386 parameters in total. 245 of these parameters must be checked, since they are defined by public functions and have pointer type. This is indicated in the first column of Figure 1, which also provides the distribution among the different kinds of parameters. The locations obtained from the verifier tell us which of these 245 parameters are actually checked, as displayed in the second column. It turns out that only 149 (i.e., 60%) of the parameters requiring a check are in fact checked.

The deviations obtained from the verifier aim to help in identifying the remaining 96 parameters that need to be checked. The verifier reports a total of 110 deviations (column 3). Manual inspection of these deviations eliminated 14 intended deviations (for 9 input parameters and 5 output pointer parameters, cfr. column 5).

4. A DOMAIN-SPECIFIC LANGUAGE FOR PARAMETER CHECKING

In order to arrive at a more rigorous treatment of parameter checks (avoiding the situation that as many as 40% of them deviated from

the specifications), we propose a domain-specific language (DSL) for representing the kind of parameter checks that are required. In this section we describe the language and corresponding tool support — in the next we explain how existing components can be migrated to this target solution.

4.1 Motivation

Different from general-purpose aspect languages, such as AspectJ or AspectC, our DSL is a special-purpose language, that can only be used to specify aspects involving parameters. The specific reason we decided to implement our own language, is that current general-purpose aspect languages lack the necessary means and flexibility to express the parameter checking concern in an adequate way. This has been reported in [5], and we will come back to this issue in Section 7. It is also important to stress that the current focus of our research is not language engineering. We thus implemented a language that fulfils our requirements only, and we did not consider other important issues such as interoperability with other languages, for example. However, the language is not tied to our specific case, and can be used in any other application where parameters need to be considered in an aspect.

4.2 Specification

The idea underlying the language is that a developer annotates a function’s signature, by documenting the specific kind of its parameters, i.e. either input or output. Output parameters that are of output pointer kind can also be specified. When a parameter does not require a check, for whatever reason, this can be annotated as well. Additionally, 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 output pointer parameters to be specified separately. Although in this paper we do not need it, the DSL also has provisions to express advice code for deviations.

As an example, consider the (partial) specification of the parameter checking aspect for the CC component as depicted in Figure 2. It states that the parameters `CC.queue *queue` and `void **queue_data` of the functions `CC.queue_peek.front` and `CC.queue_peek.back` 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. Additionally, the advice code implements the required checks for input, output and output pointer parameters. The special-purpose `thisParameter` variable 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.

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. The specific location where the checks should be implemented is not specified in the aspect, however. It abstracts from these implementation details, and as such becomes a declarative and intentional specification, that is as concise as possible.

4.3 Compilation and Weaving

The use of the DSL frees the developer of having to insert parameter checks manually. Instead, the checks specified in the DSL are


```

component CC {
  CC_queue_peek_front(output CC_queue *queue, output output_pointer void **queue_data);
  CC_queue_peek_back(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 parameter %s may already contain a value. This value will be"
                  "overwritten, which may lead to a memory leak",
                  thisParameter.function.name, thisParameter.name));
    }
  }
}
}

```

Figure 2: DSL specification of the parameter checking concern

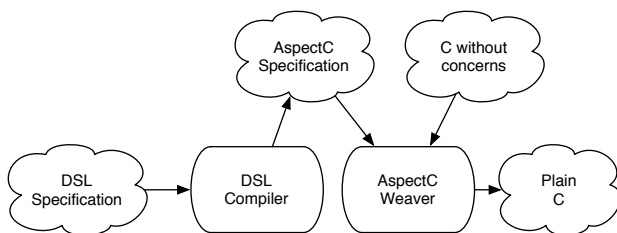


Figure 3: Merging C and DSL code

automatically woven into the pure C component that was written without checks.

Rather than implementing our own aspect weaver for the parameter checking DSL, we translate it into an already-existing general-purpose aspect language for the C programming language. As such, we get the benefits of both worlds: we can use a special-purpose, intuitive and concise DSL, for which we don not need to implement a sophisticated weaver ourselves. This process is illustrated in Figure 3.

The general-purpose aspect language is a stripped-down variant of the AspectC language [6]. 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 6 contains some examples, which show how the `advice on` keyword is used to specify advice code for a particular function.

The translation process itself proceeds as follows: the translator considers each parameter of each function in the original DSL specification, looks at its kind(s), retrieves the corresponding advice code, expands that code into the actual check that should be

performed, and inserts the expanded code in the function where the parameter is defined. 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). Note that, two checks are implemented for a parameter of output and output pointer kind, since both the output and output pointer advices are substituted for such parameters.

4.4 Application in Case Study

The parameter checking concern in the original CC implementation required 961 lines of C code (see Figure 5). Using the parameter checking DSL, only 133 lines are needed instead: One line for each of the 109 functions that require one their parameters to be checked, $(2 * 7) + 8$ lines for the three different kinds of advice required, and a start and an end line.

It is noteworthy that the equivalent solution in the general aspect language consists of 1200 lines of code (this is the AspectC code generated by the DSL compiler). In particular, observe that the solution in the general aspect language requires more lines of code than the idioms-based approach. In other words, although the solution in the general aspect language removes the scattering, it requires an extra amount of effort to implement, and contains just as much duplication as the idioms-based approach (as shown in Figure 6). This observation is confirmed by [5], where we tried to extract the same parameter checking aspect into the AspectC language as defined in [6]. It illustrates that simply using an aspect-oriented solution does not necessarily improve the maintainability and reusability of the code. Additionally, it shows that both the AspectC language and the stripped-down variant we use in this paper

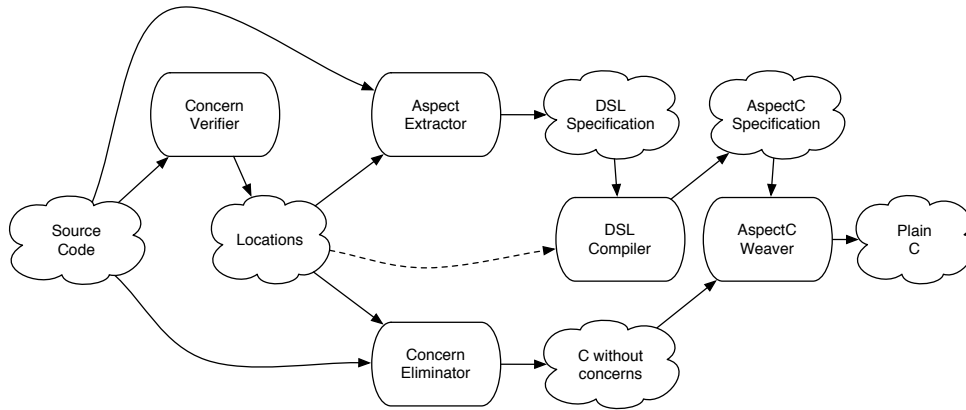


Figure 4: Migrating Existing Components to the DSL

| | Lines of code |
|--------------------|---------------|
| Original C code | 961 |
| DSL representation | 132 |
| AspectC code | 1200 |

Figure 5: Lines of code figures for various parameter checking representations

lack the necessary features and flexibility to define aspects involving function parameters in a concise way. This problem is apparent in most general-purpose aspect languages.

5. MIGRATION

5.1 Motivation

The DSL and compiler as described in the previous section can be used in a greenfield engineering setting when new components are built. However, the majority of the software development activities at ASML (and, in fact, at many companies) is devoted to evolution: adjusting existing components. In order to free the developer from having to deal with the parameter checking concern when evolving the component, we describe how this component can be migrated to a DSL solution automatically.

The steps involved in migration are depicted in Figure 4. The key steps involved are the extraction of aspect code from the source code, and the elimination of the parameter checking code from the original sources. As we will see, for both steps, the locations obtained by the verifier discussed in Section 3 provides essential information. Moreover, these locations will play a role in the DSL compiler, which can use them in order to regenerate code that is as close as possible to the original code.

5.2 Aspect Extraction

When developing new code, a developer can use the DSL to specify parameter checking aspects, instead of implementing the checks manually. In a migration setting, however, we don't want developers to wade through millions of lines of already existing source code to annotate function signatures and define an appropriate aspect. Rather, we want to extract such an aspect definition from the existing code automatically. The information required to perform this extraction consists of just (i) the *kind* of each parameter; (ii) whether it requires a check or not; and (iii) if so, the code that needs to be executed for such a check (i.e. the advice code). Apart from

this advice code, all this information has already been computed by the concern verifier. Recall from Section 3 that the verifier automatically identifies input, output and output pointer parameters, and that the list of deviations is split into intended and unintended deviations. Our aspect extractor thus merely reuses this information. The advice code, on the other hand, is not considered by our concern verifier. As explained in Section 2, the advice code for input, output and output pointer parameters always consists of an if-test, an assignment and a call to a log function. Our aspect extractor simply constructs this code as the advice code definition.

Note that the verifier cannot recover *intended deviations*, and hence the aspects for such parameter checks cannot be extracted either. Intended deviations should be recorded in a separate DSL specification, which is then taken into account in the subsequent code generation process as well.

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 checks occur, and can be used for these purposes.

The verifier obtains the locations through CodeSurfer, based on abstract syntax tree and program dependence graph information. Although line numbers for relevant statements can be obtained in this manner, the precise start and end points of the checking code is not always available (for example, brackets of compound statements are not visible in CodeSurfer). We currently use a fairly simple solution to deal with these issues, based on a prototype implementation in Perl. This is possible because the parameter checking concern is not tangled with the other code, and is easy to recognise and remove. This works well enough for the cases under study at the moment. On the long term, we expect to implement a more rigorous concern eliminator based on program transformation tools such as ASF+SDF [3].

5.4 Conservative Translation

The DSL code recovered can be used directly to generate intermediate AspectC code, which then in turn can be woven with the C code from which we eliminated the concern code.

However, when adopting the generated C code in a production environment, we would like to eliminate as many risks as possible. In other words, it is preferable to make the compiler as conservative as possible, trying to stay very close to the original C code. For that reason, the DSL compiler offers the possibility to re-introduce

```

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
}
int advice on (queue_extract) {
    int r = OK;
    if(queue == (CC_queue *) NULL) {
        r = CC_PARAMETER_ERR;
        CC_LOG(r,0,("%s: " Output parameter %s error (NULL)", "queue_extract", "queue"));
    }
    if(item_data == (void **) NULL) {
        r = CC_PARAMETER_ERR;
        CC_LOG(r,0,("%s:" Output parameter %s error (NULL)", "queue_extract", "item_data"));
    }
    if(item_data != (void **) NULL) {
        r = CC_PARAMETER_ERR;
        CC_LOG(r,0,("%s:"Output paramater %s may already contain data (!NULL). This data will be"
                    "overwritten which may lead to memory leaks", "queue_extract", "item_data"));
    }
    if (r == OK)
        r = proceed();
    return r;
}

```

Figure 6: AspectC specification of the parameter checking concern

the parameter checks at exactly the same locations as where they were found originally. To that end, it uses information obtained from the verifier (as indicated by the dashed arrow in Figure 4). Naturally, this is only possible for parameters that were already checked correctly, and not for newly introduced checks.

An illustration of the translation of the specification of Figure 2 is given in Figure 6. The first example adds an input and an output parameter check to the `CC_queue_empty` function for its `queue` and `empty` parameters, respectively. Since this function originally returned some value, the advice code returns the value obtained by calling the `proceed` function. The second example states that the `queue_extract` function should implement two output parameter checks and one output pointer parameter checks. This function is a non-public function, and a specification for it did thus not appear in the DSL specification. The reason it is included in the AspectC specification is that both the `CC_queue_peek_front` and `CC_queue_peek_back` functions call the `queue_extract` function, and both parameters of those former functions are checked in the `queue_extract` function in the original code. When translating the specification of the `CC_queue_peek_front` and `CC_queue_peek_back` functions, the translator consults the verifier to see where their parameters are checked, and generates advice code correspondingly.

6. UNIFYING EVOLVABILITY & PARAMETER CHECK CONSISTENCY

Before comparing the idioms-based solution to the aspect-oriented solution in the next section, we first illustrate how the latter solution improves the evolvability of the system. In particular, we

show how the aspect-oriented solution allows us to determine the ideal locations for parameter checks automatically. Consequently, a developer can evolve the system, without worrying about the parameter checking concern at all. Additionally, we will show that automatically determining the locations will reduce the total number of implemented checks, reducing the run-time overhead of parameter checking.

6.1 Motivation

Recall from Section 2 that the requirements for the parameter checking concern do not specifically state where a particular parameter check should be implemented. A developer will try to insert the check at the best possible location, by taking into account two important considerations. First of all, he will try to reduce the amount of implementation effort required, by reusing checks as much as possible. To this extent, it is desirable to implement these checks in functions that are called by many other functions. Second, he will try to reduce the number of redundant checks, i.e. a single parameter that is checked multiple times by different functions that are called from one single function.

Determining the best location for a check requires good knowledge of the implementation of the system. The developer needs to know the internal details of the functions, in particular their call graph and the dependencies between formal and actual parameters. Such information is rather volatile, like all global system information, since it changes every time the system evolves. Consequently, the concern's implementation needs to be revised after every evolution, in order to ensure the checks are still at the best possible locations. Given the fact that the concern code is scattered all over the system, this is problematic.

According to the parameter checking requirement, for all *public* functions, the value of a formal parameter of type pointer should be checked against NULL before it is referenced, but the specific location of the check is not specified. As long as a check occurs before the first reference, the requirement is met. It may therefore be possible for our AspectC translator to improve upon the simple solution of implementing a check for every pointer formal in every public function.

The aspect-oriented solution allows us to automate this approach, because it makes the concern explicitly available and implements it in a modular way. As such, we can reason about the concern in order to determine the best possible location for a particular check automatically. In other words, we will improve upon the simple solution of our AspectC translator that implements a check for every pointer formal in every public function. As we will see, this only requires information about the call graph of the current implementation, the parameters that need to be checked, and where these parameters are used. For example, consider two public functions $f(*a)$ and $g(*a)$ that both do not use their parameter a , but pass it on to a function $h(*a)$, which does reference its parameter a . In this case it is safe to implement a check just for the formal a in h , since the values of the formals of both f and g will be checked before reference. A real-world example in our case study was shown at the end of Section 5.4.

As an additional benefit, the algorithm can also be used by the concern verifier, when dealing with unintended deviations. Instead of providing the developer with a list of all parameters that are not checked but should be, a list of locations is returned that identifies only those places where checks need to be inserted, sufficient to cover all deviations.

6.2 Algorithms

In the general case, it is required that each (interprocedural) execution path that leads to a pointer reference that is (indirectly) data dependent on a formal of a public function should encounter an appropriate NULL check prior to the reference. Given that all such paths are known, we can determine a minimal set of locations where NULL checks should be implemented such that the requirement is met. The algorithm described in the remainder of this section calculates such a minimal set of locations. NULL check elimination has been researched previously in the context of optimizing (Java) compilers [17].

The set of locations where checks can be implemented is given by the set of functions in the system; it is assumed that we can insert checks in each function for all its pointer formals, prior to any pointer reference occurring in the function.

For each formal x_1 of each public function we find those (interprocedural) execution paths that lead to a pointer reference that is (indirectly) data dependent on x_1 . Given such an execution path, we derive a *chain*. A chain is a sequence of formals that is terminated by a pointer reference node. The first formal in the chain is given by x_1 . Formals are then added to the chain if the execution path enters other functions, and data dependences exist between the last formal in the chain and formals of the called functions. Finally, a node which represents the pointer reference that is data dependent on the last formal in the chain is added to terminate the chain. The pointer reference is thus data dependent on x_1 via the other formals in the chain. The following definitions initialize the sets which are used in Algorithm 1:

Chains := $\{C_1, C_2, \dots, C_n\}$,
Formals := $\{f \mid f \text{ is a formal of type pointer in any function}\}$,
Checks := \emptyset

| | # of checks + # extra | # total reduced | % reduced |
|----------------|--------------------------|--------------------|--------------|
| input | 34 + 16 = 50 | 44 | 12 |
| output | 93 + 45 = 138 | 110 | 20 |
| output pointer | 12 + 24 = 36 | 32 | 11 |
| total | 139 + 85 = 224 | 186 | 17 |

Table 1: Reduced number of checks

where each $C = \langle x_1, x_2, \dots, x_n \rangle \in \text{Chains}$ adheres to the following properties:

- x_1 is a formal that represents a parameter of type pointer of a *public* function,
- x_n is a node that is data dependent on x_1 and that represents a pointer reference,
- x_2, \dots, x_{n-1} are formals which are data dependent on x_1 and which belong to a execution path from x_1 to x_n in the interprocedural CFG,
- $C = \langle x_1, x_2, \dots, x_n \rangle$ contains no duplicates, i.e. the chain is acyclic.

The main idea of Algorithm 1 is to place a minimal number of checks such that each chain has a check for at least one of its formals. The procedure FINDCHECKLOCATIONS first iterates over the set of chains (lines 1–3), and calls IMPLEMENTCHECK to put a check for each formal that is used as a pointer reference in its declaring function. In these cases it is not possible to delay the checking of the formal’s value to a function down the call chain, since its value will be referenced in the current function. IMPLEMENTCHECK stores the formal in the *Checks* set (line 1), which indicates that a check has to be implemented for the specified formal in its declaring function. In lines 2–4 any chains that include the now-checked formal are eliminated, in particular the chains that start at the now-checked formal. It is safe to consider these chains to be accounted for, since it is now guaranteed that for each chain the value of its first formal is checked against NULL before it is referenced. Furthermore, since it makes little sense to check the same formal twice in the same function, the now-checked formal is not considered during the remainder of the algorithm (line 5).

The algorithm will continue to eliminate chains by implementing checks for the remaining formal by a simple greedy strategy (lines 4–6 in FINDCHECKLOCATIONS). The formal that is included in the largest number of chains that are still remaining is assigned a check, and all those chains are then eliminated. This process is guaranteed to terminate, since each chain can always be eliminated by placing a check for the first formal in the chain. After termination, the result of the algorithm is represented by the *Checks* set.

6.3 Results

Table 1 shows the results of applying the algorithm on the CC component. As can be seen, 186 checks in total suffice for checking all parameters. If we compare that to the number of checks that are present in the current implementation, augmented with the number of checks that should be inserted in order to reduce the unintended deviations (second column in the table), we observe a reduction of 17%.

Two conclusions can be drawn from these results. First of all, they clearly show that an automated approach is to be preferred over a manual approach when determining the best possible location for a check. Clearly, a developer is not able to adequately study the call graph and the dependencies between functions. Moreover,

Algorithm 1: Find optimal parameter check locations.

```

FINDCHECKLOCATIONS()
(1)  foreach  $\langle x_1, x_2, \dots, x_n \rangle \in Chains$ 
(2)    if  $Function(x_1) = Function(x_n)$ 
(3)      ImplementCheck( $x_1$ )
(4)  while  $Chains \neq \emptyset$ 
(5)     $f := FindCandidateFormal(Chains, Formals)$ 
(6)    ImplementCheck( $f$ )

```

```

IMPLEMENTCHECK( $f$  : formal)
(1)   $Checks := Checks + f$ 
(2)  foreach  $c = \langle x_1, x_2, \dots, x_n \rangle \in Chains$ 
(3)    if  $f \in c$ 
(4)       $Chains := Chains - c$ 
(5)   $Formals := Formals - f$ 

```

```

FUNCTION( $n$  : CFG node)
Description: Returns the function that contains the specified CFG node.

```

```

FINDCANDIDATEFORMAL( $Chains$  : set,  $Formals$  : set)
Description: Returns a formal from  $Formals$  that is included in the most chains from  $Chains$ . A formal  $x$  is included in a chain  $C = \langle x_1, x_2, \dots, x_n \rangle$  iff  $x \in \langle x_1, x_2, \dots, x_n \rangle$ .

```

if the system evolves and the dependencies change, the developer will need to reconsider the situation. Second, it is only because we first transformed the original solution into an aspect-oriented solution that we were able to perform this optimisation. If we did not have information about the parameters that need to be checked, the coding idioms followed, how the checks should be implemented, etc. we would not have been able to perform this evolution. In our view, this strengthens the common belief that AOSD improves the evolvability of a system.

7. DISCUSSION

What do we gain from actually adopting the parameter checking DSL as proposed in the previous sections? What adoption scenarios exist? In this section we discuss the pros and cons of the DSL approach for the parameter checking concern.

7.1 Adoption Strategies

The different kind of tools discussed in the preceding sections can be adopted in several ways:

No automated weaving: The tools are used for analysis purposes only. The tools assist the developer in verifying that his parameter checks are consistent with the idiom. The tools may produce C code that the developer can copy-paste into the sources if he chooses so.

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

Full automation: All parameter checking is specified using the DSL. All parameter checking code 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 endeavor, since it implies making modifications to the full code base.

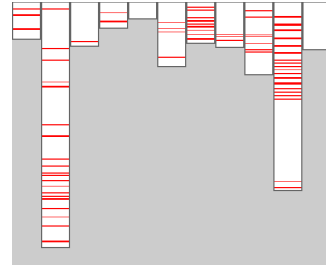


Figure 7: Parameter checking code in the CC component

Hybrid: Aspect-oriented parameter checking is adopted for some components, and hand-coded checking for others.

This is possible since the code produced by the aspect-oriented weaver is 100% compatible with the original idiom.

We expect that the adoption of our techniques will start with a non-weaving approach. Once developers and managers get familiar with the use of the DSL and the concern 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.

7.2 Code Quality Considerations

Code Size The aspect-oriented solution reduces the code size of the component by 7%, since the DSL allows us to specify the parameter checking concern in a concise way. The complete aspect definition is specified in only 132 lines, whereas the parameter checking concern in the original component comprised a total of 961 lines.

Naturally, reduced code size alone is an insufficient indicator for increased code quality. However, less code does give the benefits of fewer chances of error, fewer lines to write or understand, and, following Boehm's maintenance cost prediction model [2], lower maintenance costs.

Scattering and Tangling Figure 7 (generated using the Aspect-Browser [13]) shows how the parameter checking concern, implemented using the idioms-based approach, is distributed over the code of the CC component. Each column in the figure represents a particular source file of the component, and each horizontal line shows the occurrence of a parameter check. Thicker lines denote locations where multiple checks occur. As is clear, the code is scattered over many different functions in all except two source files.

The aspect-oriented solution cleanly captures the concern in a modular and centralised way, and thus removes the scattering all together. This does not only make the concern more explicit and tangible in the source code, but also improves its reusability, understandability and maintainability. The code is easier to understand, possible errors will be easier to spot and correct and the same checking code does not need to be implemented over and over again.

Removing the scattering also eliminates the *tangling* that is present in the C solution: without the DSL, many functions start with approximately seven lines per parameter. This code is unrelated to the key concern to be handled by the function, and causes unnecessary distraction for the developer.

Code Duplication A consequence of code scattering can be code duplication. Duplication is generally considered undesirable, not

| | # of deviations | reduced # of deviations | % reduced |
|----------------|-----------------|-------------------------|-----------|
| input | 26 | 19 | 27 |
| output | 49 | 45 | 8 |
| output pointer | 35 | 30 | 14 |
| total | 110 | 94 | 15 |

Table 2: Reduced deviations

only because it increases the code size, but also because it increases the chance of inconsistencies.

In a separate experiment [4], we evaluated the amount of code duplication in a number of crosscutting concerns present in the CC component, among which the parameter checking concern. As was observed, seven *clone classes* (groups of code fragments that are all clones of each other) cover over 80% of the parameter checking code. 523 lines of concern code are captured by the largest clone class of the concern alone. Clearly, 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. By using aspect-oriented techniques, however, reuse becomes possible again. This is reflected by the fact that in our DSL specification of the parameter checking concern, the advice code for each kind of parameter is specified only once and can be reused.

Evolvability Due to the scattering and the code duplication, evolving the system and thereby ensuring the parameter checking concern’s consistency is notoriously hard. As explained in the previous section, developers take into account the call-graph of the component in order to determine where checks need to be implemented. As the call graph will most probably change when the component changes, the implementation of the parameter checking concern has to be revised every time as well. The aspect-oriented solution on the other hand can determine the best possible location without developer intervention. As such, when the system evolves and the aspect is rewoven, it will make sure all necessary parameter checks are present. Of course, this requires the aspect specification to evolve when the system evolves, for example to add descriptions for new functions, or change existing descriptions. Such evolution is of course significantly easier than reconsidering the parameter checking implementation as a whole.

7.3 Concern Quality Considerations

Apart from system-wide benefits, the adoption of the DSL has consequences for the quality of the parameter checking concern implementation as well.

Unintended Deviations In Section 3 we have seen that as many as 40% of the parameters that ought to be checked are in fact never checked.

It is not immediately clear why so many parameters are left unchecked. One reason is probably that the punishment or reward for the developer is uncertain, and much later in time, happening only when another developer starts using the component in a wrong way that could have been prevented by a proper null pointer warning. Moreover, this figure seems to indicate that developers consider implementing this concern for each parameter too much effort.

With the tools we propose, this effort will be reduced. This will even be the case in the adoption scenario in which no automated weaving is used: our tools will just report the locations where checks should be inserted in order to cover all parameter checks.

As can be seen from the second column in Table 2, the number of reported deviations is reduced by 15% in this way.

Intended Deviations 13% of the reported deviations are intended deviations, i.e. parameters that need not be checked. Although we are presently investigating this issue, we do not see many opportunities to further refine our verifier in order to reduce this figure. These checks are simply “exceptions to the rule” to which the code should adhere. Note however, that it is important to identify these exceptions, because the aspect extractor relies on this information. Moreover, it can improve the understandability of the code. 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 its value is overwritten, and hence, no output pointer check is needed.

DSL/C Code Consistency A potential risk of separating the parameter checking code from the C code is that the two get out of sync. A remedy against this is to include sanity checks in the DSL compiler, which can warn about non-existing functions, parameters, or non-matching signatures. The result will certainly be more consistent than the current practice, which is to include parameter kind declarations in comments that are not automatically processed. In addition to that, we do believe developers will be motivated to keep the aspect specification in sync with the current implementation, as this helps them to achieve a correct parameter checking concern in a rather easy way.

Uniform Parameter Checking The advice code specifies how a parameter should be checked, and this code is specified only once and reused afterwards. Consequently, all parameters are checked and logged in the same way. This was not the case for the idioms-based implementation, where the logged strings often differ, or checks are implemented in slightly different ways. For example, all functions except one implement the checks according to the format explained in Section 2. When logging a possible error, 7 different strings are logged for an input parameter error, 4 different strings for an output parameter error and 4 for an output pointer parameter error.

The uniformity of the log file is important for automated tools that reason about the logged errors in order to identify and correct the primary cause of a particular error.

Documentation One of the benefits of using a declarative DSL, is that it can be used for additional purposes than compilation to C [7]. In particular, the parameter checking aspect acts as documentation of the component’s functions, or it can be used as input to a documentation generator. In the current implementation of the component, the kind of the parameter is documented inside comments. These comments are often not consistent with the source code however, and are sometimes outdated (e.g. a function defines new parameters that are not document, or vice versa). Moreover, such documentation does not include information about the exceptional parameters that do not need to be checked. The aspect however, makes all this information explicit, and thus improves the understandability of the concern. Additionally, since the aspect is extracted from the source code automatically, it is up to date, and as already explained, we believe it will remain so because developers profit from it.

Note that information about the exceptional parameters is not even present in the general-purpose aspect language specification of the concern. This shows one more limitation of general-purpose aspect languages as opposed to domain-specific aspect languages.

Scalability Although our tools and approach show promising re-

sults when applied on the CC component, it remains to be investigated whether they scale up to other components of the ASML code base. In particular, the question can be raised whether our results can be generalised to larger components, developed by other developers. This may have an effect on the way the parameter checking concern is implemented, for example.

In order to investigate this issue, we are currently experimenting with another component, consisting of 110,000 lines of code. The component consists of 1516 functions, with 3132 parameters in total. 705 of these parameters need to be checked (505 input, 188 output and 12 output pointer parameters). When running our verifier over this component, 218 deviations are reported: 98 input pointer parameter deviations, 108 output parameter deviations, and 12 output pointer parameter deviations. We currently have not yet investigated which of these deviations are actually intended. With respect to the number of lines of code, we observe that expressing the concern in the parameter checking DSL involves 774 lines, whereas the number of lines in the component devoted to parameter checking amounts to approximately 2500 lines of code.

Thus, compared to the CC component, we see that percentage of parameters requiring a check is smaller (705 out of 3132, (22%) versus 245 out of 386 (64%) for CC). This is explained by the fact that parameter checking is required for the *interfaces* only: large components will hide more behind their interface. As another comparison, we see that there are 218 deviations on 705 parameters (30%) whereas we had 110 out of 245 (44%) deviations for CC.

8. RELATED WORK

Our concern verifier resembles tools that verify the quality of the source code. A number of tools for this purpose have been developed over the years, [15, 1, 22]. 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 [16, 10, 20, 26], 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 [9] and [11].

The work described in this paper has some similarities with work done by Coady et. al. [6]. They consider a single concern (prefetching) within a large C program, the FreeBSD OS kernel. The code implementing the concern is shown to be scattered across the layers that make up the architecture of FreeBSD. Furthermore, the prefetching code is heterogeneous and tangled with other code, which makes it both hard to identify and change. Coady et. al. describe how they (manually) identified the prefetching code, and propose a new (manually obtained) solution in terms of AspectC.

A number of other studies have investigated the applicability of aspect-oriented techniques to various (domain specific) crosscutting concerns. [21] shows the impact of refactoring several small crosscutting concerns on the code structure of two Java applications. [19] targets exception detection and handling code in a large Java framework, and shows how AspectJ can be used to improve its implementation.

An approach to refactoring which specifically deals with tangling is presented by Ettinger and Verbaere [12]. Their work shows how slicing techniques can help automate refactorings of tangled (Java) code. [14] provides a more general discussion of both refactoring in the presence of aspects, and refactoring of object-oriented systems toward aspect-oriented systems.

Another area of related work is the field of *plan recognition*, which originated from the work on the Programmer's Apprentice conducted at MIT [23]. This line of research aims at codifying programming knowledge in a library of *plans*. In a forward engi-

neering setting, the developer can use the plans, whereas in a reverse engineering setting code can be analyzed in order to discover known plans [27]. As an example, we have applied this plan recognition approach to the detection of leap year computations in Cobol code [8]. Future research is needed in order to analyze the applicability of plan recognition techniques for the purpose of identifying dedicated crosscutting concerns.

There has been some work in the area of representing crosscutting concerns in existing software systems, for example [24] describes how concern graphs can be used for this purpose in a Java setting. Similar to plans, concern graphs can be used to codify programmer knowledge. In [25], the authors show how code relevant to a concern can be identified by means of processing transcripts of program investigation activities. Subsequently the authors detail how to represent the identified code using concern graphs.

9. CONCLUDING REMARKS

Contributions

In this paper, we have shown how a idioms-based solution to crosscutting concerns as occurring in systems software can be migrated automatically into a domain-specific aspect-oriented solution. The approach is illustrated by zooming in on a particular concern, namely parameter checking. Our approach includes a number of different elements:

- Characterization of the idioms-based approach, resulting in a *concern verifier* that can check the way the concern is coded;
- Representation of the concern in an aspect-oriented domain-specific language, which can be mapped to a dialect of the general purpose AspectC language;
- A migration strategy for existing components, including an aspect extractor and a conservative translator.

We also discussed the advantages of the aspect-oriented solution compared to the idioms-based solution. Our results indicate that introducing aspects significantly reduces the code size, removes the scattering and code duplication, and improves the correctness and consistency of the concern implementation as well as the understandability of the application. Additionally, we showed how the explicit aspect-oriented solution ensures the evolvability of the application, while maintaining the consistency of the concern.

As main causes for the success of aspects, we identified that implementing the concern via coding idioms requires effort and strict discipline, and that aspects make explicit the important information otherwise only implicitly present in the code.

Future Work

The focus of this paper is on a specific concern (parameter checking) in two components from ASML. 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. The results so far show similar benefits in terms of maintainability and reliability improvements.
- Parameter checking is a concern that is interesting outside ASML as well. Our approach is mostly generally applicable. The only ASML-specific elements are in localized (1) the places in the verifier where existing checks are recognized; and (2) the specific advice specified in the DSL. 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 Remco van Engelen from ASML for discussing the results of the case study and for proofreading drafts of this paper, and all members of the Ideals project team, for input about the topic. 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.

10. REFERENCES

- [1] Alfred V. Aho, Brian W. Kernigan, and Peter J. Weinberger. Awk - A Pattern Scanning and Processing Language, 1980.
- [2] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- [3] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de 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.
- [4] Magiel Bruntink, Arie van Deursen, Remco van Engelen, and Tom Tourwé. An Evaluation of Clone Detection Techniques for Identifying Crosscutting Concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM)*. IEEE Computer Society, 2004.
- [5] Magiel Bruntink, Arie van Deursen, and Tom Tourwé. An Initial Experiment in Reverse Engineering Aspects. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society, 2004.
- [6] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg 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.
- [7] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.
- [8] 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.
- [9] Michael Eichberg, Mira Mezini, Thorsten Schfer, Claus Beringer, and Karl Matthias Hamel. Enforcing system-wide properties. In *Proceedings of the 2004 Australian Software Engineering Conference (ASWEC'04)*. IEEE Society Press, April 2004.
- [10] Eva van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*. IEEE Computer Society Press, 2002.
- [11] 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*, San Diego, CA, October 2000.
- [12] Ran Ettinger and Mathieu Verbaere. Untangling: A Slice Extraction Refactoring. In *Proceedings of the Aspect-Oriented Software Development Conference (AOSD)*, pages 93–101. ACM Press, 2004.
- [13] William G. Griswold, Yoshikiyo Kato, and Jimmy J. Yuan. Aspect-Browser: Tool Support for Managing Dispersed Aspects. Technical Report CS1999-0640, University Of California, San Diego, 3, 2000.
- [14] Stefan Hanenberg, Christian Oberschulte, and Rainer 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.
- [15] Stephen Johnson. Lint, a C Program Checker, 1978.
- [16] Yoshio Kataoka, Michael D. Ernst, William G. Griswold, and David 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.
- [17] M. Kawahito, H. Komatsu, and T. Nakatani. Effective null pointer check elimination utilizing hardware trap. In *Proceedings of the 9th international conference on Architectural support for programming languages and operating systems (ASPLOS-IX)*, pages 118–127, New York, NY, USA, November 2000. ACM Press.
- [18] Gregor Kiczales, John Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.
- [19] Martin Lippert and Christina 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 Press, 2000.
- [20] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, 2002.
- [21] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin 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 Press, 2001.
- [22] Santanu Paul and Atul Prakash. A Framework for Source Code Search using Program Patterns. *IEEE Transactions on Software Engineering*, 20(6), 1994.
- [23] C. Rich and R. Waters. *The Programmer's Apprentice*. Frontier Series. ACM Press, Addison-Wesley, 1990.
- [24] M.P. Robillard and G.C. Murphy. Concern graphs: Finding and describing concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, IEEE Computer Society Press, May 2002.
- [25] M.P. Robillard and G.C. Murphy. Automatically inferring concern code from program investigation activities. In *Proceedings of the 18th International Conference on Automated Software Engineering*, pages 225–234. IEEE, IEEE Computer Society Press, October 2003.
- [26] Tom Tourwé and Tom 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.
- [27] L. M. Wills. *Automated Program Recognition by Graph Parsing*. PhD thesis, MIT, 1992.