

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) [5] 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, 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

^{*}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.

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 *DSL compiler* 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 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 considers various quality attributes to compare the aspect-oriented solution to the

idioms-based solution. Finally, Section 7 presents our conclusions and future work.

2. CURRENT PARAMETER CHECKING IDIOM

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.

2.1 Parameter Checking Requirements

The parameter checking concern is responsible for implementing pointer checks for function parameters and raising warnings whenever such a pointer contains a non-expected (NULL) value. 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. 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.

2.2 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

If we want to transform the idioms-based approach into an aspect-oriented one, we should 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.

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

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

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.

3.2 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.

Our verifier needs to consider each public function and verify 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 by simply checking for assignments to a parameter, looking at *kill* (or *def*) statements for that parameter inside a function’s body.

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

3.3 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).

¹www.grammatech.com

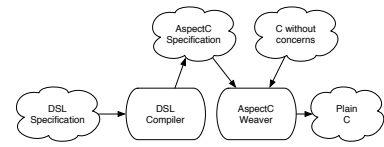


Figure 2: Merging C and DSL code

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 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 1. 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.

4.2 Compilation and Weaving

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 do not need to implement a sophisticated weaver ourselves. This process is illustrated in Figure 2.

The general-purpose aspect language is a stripped-down variant of the AspectC language [2]. 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 4 contains an example, which shows how

```

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 1: DSL specification of the parameter checking concern

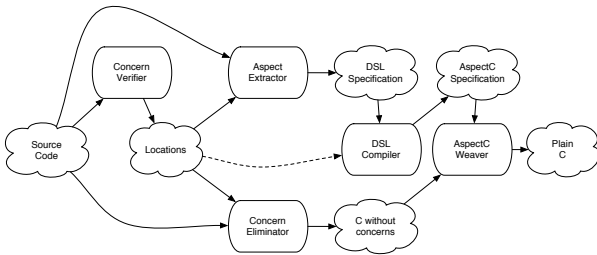


Figure 3: Migrating Existing Components to the DSL

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.

4.3 Application in Case Study

The parameter checking concern in the original CC implementation required 961 lines of C code (see Figure 2). 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

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

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

be checked, $(2 * 7) + 8$ lines for the three different kinds of advice required, and a start and an end line.

5. MIGRATION

5.1 Motivation

The steps involved in migration of the idioms-based approach to the DSL approach are depicted in Figure 3. 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

```

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 parameter %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 4: AspectC specification of the parameter checking concern

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.

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. We currently use a fairly simple solution to eliminate the concern code, 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.

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 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 3). 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 1 is given in Figure 4. The 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

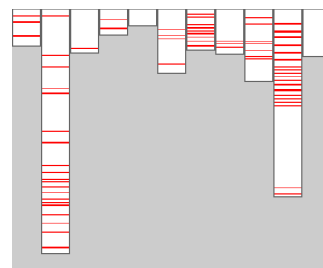


Figure 5: Parameter checking code in the CC component

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. DISCUSSION

In this section we discuss the pros and cons of the DSL approach for the parameter checking concern.

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 [1], lower maintenance costs.

Scattering and Tangling Figure 5 (generated using the Aspect-Browser [4]) shows how the parameter checking concern, implemented using the idioms-based approach, is distributed over the code of the CC component.

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

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.

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.

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 [3]. 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.

Scalability Although our tools and approach show promising results 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.

7. CONCLUDING REMARKS

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.

8. REFERENCES

- [1] B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.
- [2] 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.
- [3] A. van Deursen and P. Klint. Little languages: Little maintenance? *Journal of Software Maintenance*, 10:75–92, 1998.
- [4] 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.
- [5] 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.