Simple Crosscutting Concerns Are Not So Simple

Analysing Variability in Large-Scale Idioms-Based Implementations

Magiel Bruntink
Centrum voor Wiskunde en Informatica
magiel.bruntink@cwi.nl

Arie van Deursen[1]
Delft University of Technology
arie.vandeursen@tudelft.nl

Maja D'Hondt *
Centrum voor Wiskunde en Informatica
maja.d-hondt@cwi.nl

Tom Tourwé
Eindhoven University of Technology
t.tourwe@tue.nl

Abstract

This paper describes a method for studying idioms-based implementations of crosscutting concerns, and our experiences with it in the context of a real-world, large-scale embedded software system. In particular, we analyse a seemingly simple concern, tracing, and show that it exhibits significant variability, despite the use of a prescribed idiom. We discuss the consequences of this variability in terms of how aspect-oriented software development techniques could help prevent it, how it paralyses (automated) migration efforts, and which aspect language features are required in order to obtain precise and concise aspects. Additionally, we elaborate on the representativeness of our results and on the usefulness of our proposed method.

Categories and Subject Descriptors D.2.7 [*Distribution, Maintenance, and Enhancement*]

General Terms Restructuring, reverse engineering, and reengineering

Keywords Aspect-oriented programming, variability, idioms, crosscutting concerns, formal concept analysis

1. Introduction

The lack of certain languages features, such as aspects or exception handling, can cause developers to resort to the use of idioms [2] for implementing crosscutting concerns. Idioms (informally) describe an implementation of required functionality, and can often be found in manuals, or reference code bodies. A well-known example is the *return-code idiom* we have studied in a realistic setting in [5]. It is used in languages such as C to implement exception handling. It advocates the use of error codes that are returned by functions when something irregular happens and caught whenever functions are invoked. Idioms are also used purposefully as a means of design reuse, for instance in the case of (design) patterns [7, 10].

Using idioms can result in various forms of code duplication [6]. Despite this duplication, idioms-based implementations are not guaranteed to be consistent across the software, however. Several factors may give rise to variability in the use of the idiom. Some variability, which is essential, occurs if there is a deliberate deviation from the idiom, for example in order to deal with specific needs of a subsystem, or to deal with special cases not foreseen in the idiom description. In addition to this, variability will occur accidentally due to the lack of automated enforcement (compilers, checking tools), programmer preference or skill, changing requirements and idiom descriptions, and implementation errors.

In this paper, we are interested in the answer to the following question:

> *Is the idioms-based implementation of a crosscutting concern sufficiently systematic such that it is suitable for an aspect-oriented solution (with appropriate pointcuts and advice)?*

While answering this question is an endeavour too ambitious for this paper, we do take an important step towards an answer by addressing the following sub questions: First, can we analyse the variability of the idioms-based implementation of a crosscutting concern? And secondly, can we determine the aspect language abstractions required for designing aspects that succintly express the common part and the variability of a crosscutting concern?

We have encountered a number of examples of idiomatically implemented crosscutting concerns [6, 4, 5]. Several more are mentioned in the literature [9, 8]. The questions we ask in this paper need to be answered in order to start migrating these crosscutting concerns to aspect-oriented solutions.

We present a generally-applicable method for analysing the occurrence of variability in the idioms-based implementation of crosscutting concerns, that will help us answer these questions. We show the results of applying this method in order to analyse the *tracing* idiom in four selected components (ranging from 5 to 31 KLOC) of a 15 million line C software system that is fully operational and under constant maintenance. Tracing is one of the ubiquitous examples from aspect-oriented software development (AOSD), and although it is a relatively simple idiom, we show that it exhibits significant and unexpected variability.

* This work was carried out during the tenure of an ERCIM fellowship.

[1] Also affiliated with CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

[2] Synonyms are code templates, coding conventions, patterns, etc.

AOSD 07 March 12-16, 2007, Vancouver, Canada

We also discuss the implications of this variability. We illustrate the limitations of idioms-based implementations and as such provide a solid motivation, based on our experiences with a large legacy system, for using aspect technology as a means to localise implementations and avoid accidental variability. This should interest the AOSD community as a whole. We also discuss how variability complicates and even paralyses efforts to migrate legacy code towards modern languages. Researchers investigating such (automated) migration of code can study our results and use them to improve their methods and techniques, such that they can deal with the significant variability we observed. Additionally, the results of our method's variability analysis can be used directly to determine the required aspect language features, capable of expressing the idioms with their essential variability. We discuss two such language requirements for the tracing idiom under investigation.

The structure of the paper is as follows. The next section briefly presents our method for analysing variability by describing each individual step. Sections 3–7 then describe how we applied each step on the selected components of our subject system in order to analyse the tracing idiom's variability. Section 8 then presents a discussion of the repercussions of these results and an evaluation of our method. Section 9 discusses related work and Section 10 presents our conclusions.

2. A Method for Analysing Idiom Variability

This section proposes the general approach we use to acquire a deep understanding of the variability in the idioms-based implementation of a crosscutting concern, and explains how to use this understanding in subsequent aspect specification and design phases.

2.1 Idiom Definition

The aim of this step is to provide a definition that is as clear and unambiguous as possible for the idiom that we want to study. The input for this (manual) step is typically found in the documentation accompanying the software, by means of code inspections, or by discussions with developers. In this respect, this step closely resembles the *Skim the Documentation*, *Read all the Code in One Hour* and *Chat with the Maintainers* patterns discussed in the *First Contact* cluster of [11].

While this step may seem simple, in our experience idiom descriptions in coding standard manuals often leave room for interpretation. When presenting our results, it happened more than once that developers started a heated debate on whether a particular use of the idiom was valid or not.

2.2 Idiom Extraction

In this step, the code implementing the idiom is automatically extracted from the source code. This requires that the idiom code is recognised, and hence the output of the previous step is used as input for this step. The result of this step is similar to a slice [30], albeit that the extracted code does not necessarily need to be executable. Nevertheless, the extracted code can be compiled and analysed by standard tools, and it is much smaller than the original code, allowing us to scale up to large systems.

Naturally, the complexity of this step is strongly dependent on the idiom: idioms that are relatively independent of the code surrounding them are easy to extract using simple program transformations, whereas idioms that are highly tangled with the other code require much more work.

2.3 Variability Modelling

In this step, we describe which properties of the idiom can vary and indicate which variability we will target in our analysis. It is important to note that we do not require a description of variabilities that actually occur in the source code. We only need to know where we can expect variabilities, given the definition of the idiom. For example, variability in the tracing idiom under investigation can occur in the specific macro that is used to invoke the tracing functionality. In practice, it might turn out that the same macro is used consistently throughout the source code, or it might not.

Additionally, it is preferable to model different levels of variability separately in order to understand them fully, and subsequently to consider combinations. For example, in the tracing idiom there is the aforementioned variability in the way the tracing functionality is invoked, but also variability in the way the function parameters are converted to strings before being traced.

Finally, we do not require all possible variability to be modelled. As we discuss later, we only study part of the variability of the tracing idiom, while other parts are not considered. This is no problem if this is taken into account when discussing the results of the analysis. In other words, these results can be seen as a lower bound of the amount of variability that occurs.

2.4 Variability Analysis

This step forms the core of our method, as it analyses the variabilities actually present in the source code. This is achieved by taking the extracted idiom code, and analysing it considering the variabilities that were modelled in the previous step. We are particularly interested in finding out how properties that can vary are typically related. For example, is it the case that tracing macro m is always invoked with either parameter c_1 or c_2, but never with c_3? Answering such questions can help us in designing the simplest aspect that captures all combinations as occurring in practice.

To analyse such relations between variable properties we use formal concept analysis (FCA) [16]. FCA is a mathematical technique for analysing data which takes as input a so-called *context*. This context is basically a matrix containing a set of *objects* and a set of *attributes* belonging to these objects. The context specifies a binary relation that signals whether or not a particular attribute belongs to a particular object. Based on this relation, the technique finds maximal groups of objects and attributes — called a *concept* — such that

- each object of the concept shares the attributes of the concept;
- every attribute of the concept holds for all of the concept's objects;
- no other object outside the concept has those same attributes, nor does any attribute outside the concept hold for all objects in the concept.

Intuitively, a concept corresponds to a maximal "rectangle" in the context, after permutation of the relevant rows and columns.

The resulting concepts form a lattice and therefore we can use relations between concepts, as well as characteristics of the concepts themselves, to get statistics and interpret the results.

2.5 Aspect Design

If we assume that accidental variability in the implementation of an idiom is ultimately removed, the next step is to design aspects that replace the idiom implementation, taking into account its essential variability. However, aspect design is constrained by the choice of the target aspect-oriented programming language. Ideally the selected language should provide abstractions for representing the idiom's common pattern and its variations, as defined in [15]. If not, the common pattern has to be repeated for each variation, which results in code duplication *in the aspect*. Evidently, this partly undermines the expected usefulness of the aspect-oriented solution.

```
int f(chuck_id* a, scan_component b) {
    int result = OK;
    char* func_name = "f";
    ...
    trace(CC, TRACE_INT, func_name, "> (b = %s)", SCAN_COMPONENT2STR(b));
    ...
    trace(CC, TRACE_INT, func_name, "< (a = %s) = %d", CHUCK_ID_ENUM2STR(a), result);
    return result;
}
```

Figure 1. Code fragment illustrating the tracing idiom at ASML.

In this step, we determine the required abstractions in aspect languages, which can be nearly directly distilled from the results of the variability analysis in the previous step. We discuss two such requirements for the tracing idiom under investigation later on in the paper.

3. Defining the Tracing Idiom

The idiom we study in the paper is the tracing idiom, as adopted by ASML. ASML is the world market leader in lithography systems, and their software controls wafer scanner machines used to produce computer chips. It consists of 15 million lines of code, spread over approximately 200 components, implemented almost entirely in the C programming language.

As we have discussed in previous papers [4, 5, 6], the software implements a number of crosscutting concerns, such as tracing, parameter checking, memory handling and exception handling. ASML uses idioms to implement these concerns, and in this paper, we study one such idiom, tracing, and consider its implementation in 4 different components.

Tracing is a seemingly simple idiom, used at development-time to facilitate debugging or any other kind of analysis. The base code is augmented with tracing code that logs interesting events (such as function calls), such that a log file is generated at runtime. The simplicity of the idiom is reflected in its simple definition: "Each function should trace the values of its input parameters before executing its body, and should trace the values of its output parameters before returning"

The ASML documentation describes the basic implementation version of the idiom, which looks as in Figure 1. The `trace` function is used to implement tracing and is a variable-argument function. The first four arguments are mandatory, and specify the following information:

1. the component in which the function is defined;
2. whether the tracing is internal or external to that component;
3. the function for which the parameters are being traced;
4. a `printf`-like format string that specifies the format in which parameters should be traced.

The way in which each of these four parameters should be passed on to the `trace` function is described by the standard, but not enforced. For example, some components follow the standard and use the `CC` constant, which always holds the component's name, to specify the name, while others actually hardcode the name with a string representing the name (as in `"CC3"`). Similarly, the `func_name` variable should be used to specify the name of the function whose parameters are being traced. Since `func_name` is a local variable, however, different functions might use different names for that variable (`f_name`, for instance). The structure of the format string is also not fixed, and developers are thus free to construct strings as they like.

The optional arguments for `trace` are the input or output parameters that need to be traced. If these parameters are of a complex type (as opposed to a basic type like `int` or `char`), they need to be converted to a string representation first. Often, a dedicated function or macro is defined exactly for this purpose. In Figure 1, `SCAN_COMPONENT2STR` and `CHUCK_ID_ENUM2STR` are two such examples. Developers can choose to trace individual fields of struct instead of using a converter function, however.

Although the idiom described above is the standard idiom, some development teams define special-purpose tracing macro's, as a wrap around the basic idiom. These macro's try to avoid code duplication by filling in the parameters to `trace` in the standard way beforehand. Typically, tracing implementations by means of such macro's thus require fewer parameters, although sometimes extra parameters are added as well, for example to include the name of the file where tracing is happening.

It should be clear from this presentation that the tracing idiom precisely prescribes what information should be traced, but that the way in which this information is provided is not specified. Hence, we can expect a lot of variability, as we will discuss in Section 5.

4. Extracting the Tracing Idiom

Extraction of the tracing idiom out of the source code is achieved by using a combination of a code analysis tool, called CodeSurfer,[3] and a code transformation tool, called ASF+SDF [2]. The underlying idea is that the analysis tool is used to identify all idiom-related code in the considered components and that this information is passed on to the transformation tool that extracts the idiom code from the base code. The end result is a combination of the base code without the idiom-related code, and a representation of the idiom code by itself.

5. Modelling Variability in the Tracing Idiom

Tracing is generally considered as a very simple example of a crosscutting concern that can be captured in an aspect easily. This is confirmed by the fact that we can express the requirements for tracing in one single sentence, and hence we could expect an aspect to be simple as well. However, the tracing idiom we consider here is significantly more complex than the simple example often mentioned and than the requirement would reveal. Rather, it represents a good example of what such an at first sight simple idiom looks like in a real-world setting.

The following characteristics of the tracing idiom distinguish it from a simple logging concern:

[3] www.grammatech.com

	CC1	CC2	CC3	CC4	global
LOC	29,339	17,848	31,165	4,985	83,337
functions	328	134	174	68	704
parameter types	108	71	65	49	249
tracing macro's	1	1	2	1	2
component names	2	3	1	2	6
function names	3	1	1	1	3

Table 1. Basic statistics of the analysed components.

- A simple logging aspect typically weaves in log calls at the beginning and end of a function, and often only logs the fact that the function has been entered and has been exited. The tracing idiom described above also logs the values of actual parameters and the module in which the function is defined. Moreover, it differentiates between input and output parameters, which have to be traced differently.
- Tracing the values of actual parameters passed to a C function is a quite complex matter. Basic types such as `int` or `bool` can be printed easily, but more complex types, such as `structs` and `enums`, are a different story. These should be converted to a string-based representation first, which differs for different `structs` and `enums`. Moreover, certain fields of a struct may be relevant in the context of a particular function, but may not be relevant elsewhere. Hence, the printed value depends on the context in which the type is used, and not only on the type itself.
- The conversion of complex types to a string representation is quite different in C than in Java, or any other modern programming language. C does not provide a default `toString` function, as do all Java classes, for example. Consequently, a special-purpose converter method for complex types needs to be provided explicitly. Additionally, since C does not support overloading of function names, each converter function needs to have a unique name.

These issues, together with the way tracing is invoked as explained in Section 3, show that variability can occur at many different levels. In the remainder of this paper, however, we will focus on *function-level* and *parameter-level* variability. The variability present on those levels possibly has the biggest impact on the definition of aspects for the tracing concern.

At the function-level, the variability occurs in the specific way the tracing functionality is invoked. This depends on four different properties: the name of the tracing function that is used (for example `trace`), the way the component name and the function name are specified (by using `CC` and `func_name`, for example), and whether internal or external tracing is used. More properties are considered when a different tracing idiom requires more parameters when it is called, for example the name of the file in which the traced function is defined.

At the parameter-level, the variability involves the different ways in which a parameter of a particular kind is traced. As explained in Section 3, a parameter of a complex type can be traced by first invoking a converter function that converts the complex type to a string representation, or by tracing the fields of the complex type individually. In this case, we are interested in verifying whether a particular type of parameter is traced in a systematic and uniform manner across the considered components, and if not, how much variability occurs.

6. Analysing the Tracing Idiom's Variability

As shown in Table 1, our experiments involve 4 different components, comprising 83,000 lines of non-white lines of C code. These components define 704 functions in total, which in turn define 249 different parameter types.[4]

The table also lists the different number of ways in which tracing is invoked, i.e., the different tracing macros that are used, as well as the different component names and function names that are specified [5] The numbers clearly show the variability present in the idiom at the function level, since globally 2 different tracing macro's, 6 different ways to specify the component name and 3 different ways for specifying the function name are used.

The goal of our analysis is to identify, at the function level, which functions invoke tracing in the same way, and at the parameter level, which parameter types are converted consistently. Analysing this allows us to make headway into answering our key question, since it shows us where the implementation is systematic and what is variable. Since FCA, introduced in Section 2.4, is capable of identifying meaningful groupings of elements, we use it in our variability analysis.

The FCA algorithm needs to be set up before it can be applied, i.e., we need to define the objects and attributes of the input context. The next subsection explains how this is achieved for our experiment. Subsequent subsections then describe, for function-level and parameter-level variability, the results of running FCA on each of the components separately, as well as on all components together. This will allow us to discuss the variability within a single component, as well as the between different components.

6.1 Setting up FCA for Analysing Tracing

We first explain how objects and attributes are chosen for our experiment, and how we run the FCA algorithm. Afterwards, we explain how we interpret the results.

6.1.1 Objects and Attributes

For studying function-level variability, the objects and attributes are chosen such that all functions that invoke tracing in the same way are grouped. Hence, the objects we use in the FCA context are the names of all functions defined in the components we consider. The attributes are different instantiations of the four properties used to invoke tracing, as discussed in Section 3. A sample context is shown in the upper part of Table 2.

For the analysis at the parameter level, the objects are slightly less obvious to choose. Our goal is to let the FCA algorithm group functions that have a parameter of a certain type and convert that parameter in the same way. The objects thus have to be unique for a particular function that uses a particular parameter type. This means that functions cannot serve as objects, since they may have different parameters. Similarly, parameter types cannot serve as objects, since they can be used by many different functions. Hence, we form a combination of the parameter type and the function that uses it.

The attributes we consider are, on the one hand, the types used in the considered components, and on the other hand, the particular converter functions that are used (if any) or the constant `no_tracing` when the parameter is not traced by that particular function.

A sample of a corresponding context can be found in the lower part of Table 2. The functions `f` and `h` both define a formal parameter of type `CC_scan_component` and both use the `CC_SCAN_COMPONENT2STR` converter function. Similarly, the func-

[4] Note that types may be shared across components, hence the total number of types is smaller than the sum of the numbers of types per component.

[5] Due to space restrictions, we do not provide equivalent numbers for the parameter-level variability. Such numbers would have to be specified for each type defined by the four components, and the table would hence contain more then 249 rows.

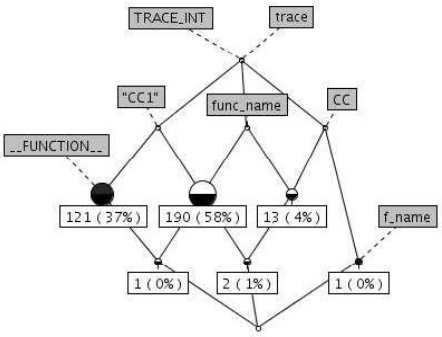

Figure 2. Function-level variability in the CC1 component

tions `f, g` and `i` define a formal of type `CC_chuck_id`, but only function `f` uses a converter function, the other two functions do not trace their parameter of that type.

6.1.2 Applying FCA

Once the context is set up, the algorithm can be applied. We use Lindig's Concepts tool to compute the actual concepts [21]. The context is specified in a file in a specific format, which we generate using ASF+SDF and the extracted tracing representation files. The tool can output the resulting concepts in a user-defined way, and we tune the results so that they can be read into a Scheme environment. This allows us to reason about the results using Scheme scripts.

An alternative is to use the ConExp tool[6], which requires a slightly different input format, but that can visualise the concepts (and the resulting lattice) so that it can be inspected easily. The graphical representations of lattices in this paper are obtained by this tool.

6.1.3 Interpreting the Results

From running the FCA algorithm, we obtain a concept lattice that shows the different concepts identified and the relation between them. An example lattice appears in Figure 2. Each dot in the lattice represents a concept, and the lines connecting the dots represent concepts that are related because they share objects and/or attributes.

While traversing a lattice from top to bottom, following the edges that connect concepts, attributes are gradually added to the concepts, and objects are removed from them. The top concept contains all objects and all attributes shared by all objects (if any), whereas the bottom concept contains all attributes and all objects shared by all attributes (if any). At some point in the lattice, a concept contains objects that are not contained within any of its sub-concepts. Those objects are the concept's *own objects*. The attributes associated with the own objects of a concept are always "complete", in the sense that in the input context passed to the FCA algorithm, the own objects precisely are related to precisely those attributes.

A concept with own objects represents a single variant for invoking tracing, or a single variant for converting a particular type. In the first case, for example, the own objects are functions, all these functions share the same (complete) set of attributes, an no other attribute is shared by these functions. In Figure 2, the concepts with own objects are denoted by nodes whose bottom half is coloured black and whose size is proportional to the number of own objects they contain. They also have white labels indicating the number of own objects and the percentage of own objects with respect to the total number of objects of the concept. The largest concepts contains 190 own object, which are functions in this case.

We observe that a particular kind of variability occurs when either input and output tracing in the same function are invoked in a different way, or a single type is converted using two different converter functions. Such situations, which are in most cases clearly examples of accidental variability, immediately show up in the concept lattice. They are embodied by concepts with own objects that have at least one parent concept with own objects. Indeed, such concepts have more attributes than is necessary, hence some of these attributes are different variations for the same property. As an example, consider again Figure 2 and observe the two concepts in the lower left part that contain 1 and 2 own objects, respectively. From their positions in the lattice, it can be derived that the leftmost concept uses both `__FUNCTION__` and `func_name` for specifying the function name when tracing, and the other concept `"CC1"` and `CC` for specifying the component name.

6.2 Function-level Variability

The upper half of Table 3 presents the results of analysing the function-level variability in the four components we consider. The first row of data contains the total number of concepts that are found by the FCA algorithm. The second row lists the number of different tracing invocations that are found (i.e., the total number of concepts containing own objects). The third row then lists the number of functions that implement the standard tracing idiom as described in ASML's coding standards (i.e., the number of own objects found in the concept with attributes `trace, CC, TRACE_INT` or `TRACE_EXT` and `func_name`), and the last row presents the percentage of those functions with respect to the total number of functions in the component.

The most striking observation revealed by these results is that only 5.7% (40 out of 704) of all functions invoke tracing in the standard way, as described in Section 3. This immediately raises the questions why developers do not adhere to the standard, and if a new standard should perhaps be considered, more specifically the way most functions invoke tracing. Whereas we cannot currently answer the first question, we can provide an answer to the second.

Looking at the second row in the upper half of Table 3, we see that 29 different tracing variants are used in the four components. If we consider each component separately, we find 31 variants in total. This difference can be explained by the fact that 3 components invoke tracing according to the standard idiom, and that the functions of these components doing so are all grouped in one single concept when considering the components together. This results in one concept replacing three other concepts, hence the reduction with two concepts. Reversing this reasoning also means that there is no other way of invoking tracing that is shared by different components, or in other words, all components invoke tracing by using their own variant(s). Consequently, we can not select one single variant that can be considered as the standard among these 29 variants, with the other variants being simple exceptions to the general rule. This is confirmed by looking at the lattices.

Looking at Figures 2 and 3, it is clear that both components use a similar tracing variant implemented by most functions (190 or 58% functions in the case of CC1, 123 functions or 92% in the case of CC2). Additionally, CC1 has yet another "big" variant that uses the `__FUNCTION__` preprocessor token instead of the variable `func_name`. This variant is used in 121 functions (37%).

[6] `http://conexp.sf.net`

	trace	CC_TRACE	TRACE_INT	TRACE_EXT	CC	func_name	f_name
f	√	-	√	-	√	√	-
g	-	√	-	-	-	√	-
h	√	-	-	√	√	-	√
i	-	√	-	-	-	√	-
j	√	-	√	-	√	√	-

	CC_SCAN_COMPONENT2STR	CC_CHUCK_ID_ENUM2STR	no_tracing	CC_chuck_id	CC_scan_component
f_CC_scan_component	√	-	-	-	√
f_CC_chuck_id_enum	-	√	-	√	-
g_CC_chuck_id_enum	-	-	√	√	-
h_CC_scan_component	√	-	-	-	√
i_CC_chuck_id_enum	-	-	√	√	-

Table 2. A sample input context.

	CC1	CC2	CC3	CC4	total	global
Function-level variability						
#concepts	11	6	24	2	43	47
#tracing variants	6	4	19	2	31	29
#functions w. std. tracing	13	1	26	0	40	40
% of total functions	4	0.7	15	0		7.5
Parameter-level variability						
#concepts	191	120	194	84	589	517
#not traced	61	49	4	16	130	115
#consistently traced	15	5	16	19	55	40
#inconsistently traced	32	17	45	14	108	94
#w.o. not traced	11	6	39	8	64	57

Table 3. Function-level and parameter-level variability results

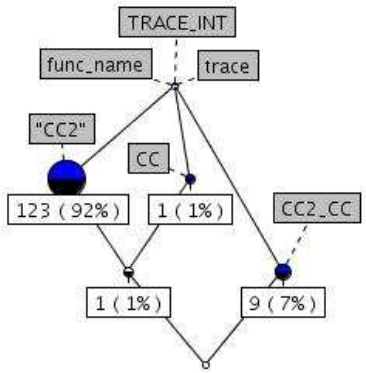

Figure 3. Function-level variability in the CC2 component

Figures 5 and 4 show significantly different results. The CC4 component implements only two tracing variants, implemented by 31 and 37 functions respectively. The difference between the two variants is that one is an extension of the other: one variant uses `CC4_LINE` to denote the component name, whereas the other uses both `CC4_LINE` and `CC4_CC`. The CC3 component implements 19 different variants, and none can be selected as the most representative or resembles the variants of another component. The variability in this case stems from the fact that the CC3 component defines its own macro for invoking tracing, and that this macro requires one extra argument, namely the name of the file in which is defined the function that is being traced. This is clearly visible in the lattice: each concept corresponding to a specific tracing variant that corresponds to a specific file in the source code, contains an extra attribute that denotes the constant used in the trace call corresponding with the file. Interestingly, although CC3 defines its own macro, it is also the component that uses the standard idiom the most. Whether the mixing of the standard idiom with the dedicated macro is a deliberate choice or not is an issue that remains to be discussed with the developers.

Summarising, we can state that very few functions implement the standard tracing variant, that no other standard variant can be identified that holds for all components, but that within one single component a more common variant can sometimes be detected.

The previous subsection discussed an example of accidental variability in the CC1 component. A similar situation occurs in the CC2 component, as can be seen in Figure 3, where one function uses `CC` and `"CC2"`. The CC3 component contains one variant that is accidental, as confirmed by the ASML developers, consisting of a copy/paste error when passing a constant representing the file name in invoking the `CC3_trace` macro.

6.3 Parameter-level Variability

The parameter-level variability involves the way a parameter of a specific kind is traced, i.e., whether it is converted to a string representation by means of a converter function, whether it is traced in a different way or whether it is not traced at all. Note that we do not show lattices for this part of experiment since the large number of parameter types generates too many concepts. Instead

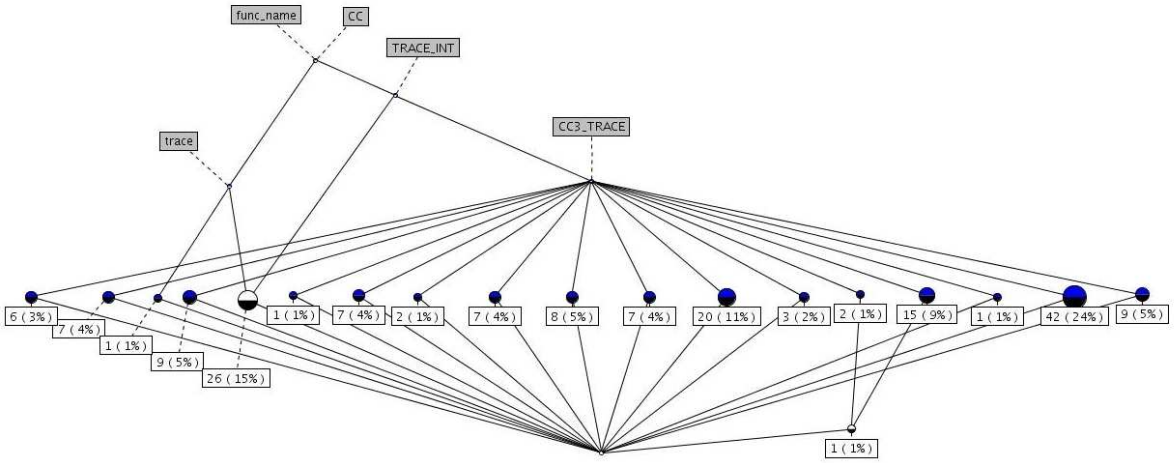

Figure 4. Function-level variability in the CC3 component

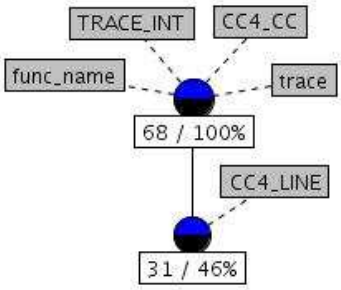

Figure 5. Function-level variability in the CC4 component

we produce statistics from the results of the FCA algorithm with our Scheme scripts.

The lower half of Table 3 summarises the results for this experiment on our four components. The first row describes the total number of concepts found for each component. The second row shows the number of types that are never traced, while the third and fourth depict the number of types that are used consistently (i.e., that are always converted in the same way) and the number of types that are not used consistently.

The fact that the global number of consistently-used types is lower than the sum of the numbers of consistently-used types per component shows that there is variability between different components: one type can be converted consistently per component, but if these components each convert it in their specific way, the type becomes inconsistently-used at the global level. The opposite is of course not possible: if a type is used inconsistently within a single component, it can never become consistently used at the global level. The fact that the number of inconsistently-traced types drops as well on a global level, is due to the fact that the different components share types, and that the different ways in which these types are traced are combined into a single inconsistency.

One immediate conclusion that we can draw from these results is that 37.7% of the types (94 out of 249) are traced in an inconsistent way, and only 16% (40 out of 249) is traced consistently. If we consider that 115 types are not traced at all, we can even say that, of all types that are traced, 70.1% (94 out of 134) is traced inconsistently and 29.8% is traced consistently. However, we should take into account one particularity of the tracing idiom. Although its definition states that each function should trace all of its parameters, in practice this does not happen. Helper functions, in particular, often do not trace all of their parameters, since these are passed in from the calling function, and are traced there. In order to take this into account, we exclude from the number of inconsistently-traced types those types that are traced using one single converter function or are not traced at all. Hence, the fifth row in the table shows the types that are converted using more than one converter function, and thus we can conclude that 42.5% (57 out of 134) of all types are not traced consistently, and 57.5% (77 out of 134) are traced consistently.

In contrast to the situation at the function-level, closer analysis of the results at the parameter-level reveals that most of the types that are traced inconsistently are converted in two or three different ways only. This result is found by counting the number of unique conversion attributes that are included in concepts that a type appears in (except for the bottom concept, which includes all attributes but does not represent a meaningful grouping). The median and mode of the number of conversion functions for an inconsistently traced type are both 2.

There are two interesting outliers in this result. The basic type `double`, and simple derived type `bool`, are traced respectively in 13 and 11 different ways. What appears strange is that these basic types are sometimes converted with a converter function defined for another type. This might be explained by C's weak typing mechanism: the other types are basically defined in order to prevent overloading of the basic type and to make the code more readable, but are not always used consistently by the developers.

Studying the results at the level of individual components reveals interesting issues as well.

As can be seen in the table, the tracing implementation in the CC1 and CC3 components appears to be less consistent than in the other two components. If we take into account the basic statistics from Table 1, this seems logical: CC1 and CC3 are by far the largest components. However, taking into account size does not explain everything: the CC2 component defines more types than CC3, but is more consistent.

Even when excluding types that are either traced consistently or not traced at all, the CC3 component still traces a lot of its types in many different ways. When we take a detailed look at the way in which the 39 parameter types are traced inconsistently, we can observe a clear pattern however. It turns out that 28 of these types are traced using a slightly different variant of the standard tracing idiom: the value of the parameter is traced, as always, but this value is accompanied by the memory address of the parameter, as follows:

```
CC3_TRACE( CC, TRACE_INT, func_name,
    FILE_CONSTANT,
    "< (p = [%p], *p = [%s])",
          p,    p_store_values_ptr_args(p));
```

Our variability model does not take into account this slight variation in the idiom, and hence reports a lot of variability. A refinement of the variability model could prevent this.

Some cases very clearly show that the variability is not intended. For example, the CC2 component uses a type `chuck_enum` and a type `chuck_id_enum`. Each of these types has its own converter function, but the converter function for the `chuck_id_enum` type is used twice for converting a parameter of type `chuck_enum`. The CC3 component also uses the `chuck_id_enum` type, and converts it in three different ways, using converter functions defined in different components. It is not clear why this happens, and presumably this is undesirable behaviour.

7. Aspect Design

This section considers how the results of the variability analysis can be used in aspect design, more specifically to determine the required language abstractions for representing the particular concern. The purpose here is not to come up with new language features that should be provided by every aspiring aspect language, nor to conduct a study of existing aspect languages for determining the degree in which they can implement the required variability. In our concrete case, this exercise would be incomplete anyway, since we analysed part of the variability of the ASML tracing concern — enough to demonstrate the relevance of our method. Instead, we attempt to point out that from our method it is straightforward to determine the required (aspect) language abstractions for capturing the variability. Note that even if the target aspect language does not provide the required abstractions, it can probably still express the concern in one way or another. However, the resulting aspect implementation will be long and complicated, which we attempt to demonstrate later on in this section.

Typically, not all the discovered variability will be represented in an aspect-oriented solution, since a substantial amount of it is undoubtedly accidental. With our proposed method for analysing the variability we are able to make some educated guesses as to what is essential variability and what is accidental. However, the process of confirming these findings is one that requires feedback from the software developers, which is outside the scope of this paper but is discussed briefly in Section 8.3. In the context of aspect design, we assume that only the confirmed, essential variability will be considered.

In the next subsection we describe how the results of our method's variability analysis can be used directly to determine required aspect language abstractions, capable of expressing the concern with its essential variability. In the two following subsections, we discuss two such concrete language requirements for the tracing idiom under investigation.

7.1 From Variability Analysis to Language Abstractions

The results of the analysis described in the previous section summarise the tracing idioms and their variability. For example, the most common variant of the tracing idiom for component CC4 can be described quite succinctly as follows: *all functions invoke tracing with the function* `trace` *and values* `CC4_CC`, `TRACE_INT` *and* `func_name`, *except for functions* $\texttt{f}_1$, ..., $\texttt{f}_n$, *which invoke this trace function with* `CC4_LINE` *in addition to* `CC4_CC`. Similarly, the most common variant of the tracing idiom for component CC3 can be expressed as: *all functions invoke tracing with the function* `CC3_TRACE`, *values* `CC`, `TRACE_INT` *and* `func_name` , *and a variable that varies according to the name of the file*.

We observe that such statements can serve as a concise specification for a future aspect implementation per component. Indeed, they clearly specify what the common part of the aspect is, as well as its variation points. We argue briefly in Section 2.5 that the choice of the target aspect language should be such that it provides abstractions for capturing the specified variability. If not, the amount of duplication in the aspect implementation will increase with a factor that is equal to the number of variations. We attempt to express this in a more systematic way below. Consider the following representation of an aspect:

$$\forall x_1, ..., x_n : f(x_1, ..., x_n)$$

The variables $x_1, ..., x_n$ each correspond to different join points or values from join points. The types of these variables are elements of the program definition or execution[7]. The predicate f expresses that the functionality of the crosscutting concern will be woven for all $x_1, ..., x_n$. When employing an aspect language that supports quantification of all the types of elements to which $x_1, ..., x_n$ are bound, the aspect specification and implementation are structurally similar. However, if the aspect language does not support quantification of the type of element to which x_i is bound for example, the aspect implementation becomes:

$$(\forall x_1, ..., x_n : x_i = a_1 \wedge f(x_1, ..., x_{i-1}, b_1, x_{i+1}, ..., x_n)) \wedge$$
$$\ldots \wedge$$
$$(\forall x_1, ..., x_n : x_i = a_m \wedge f(x_1, ..., x_{i-1}, b_m, x_{i+1}, ..., x_n))$$

where m is the number of ways in which x_i can vary. As a result, the aspect implementation contains code that is duplicated m times. Additional limitations in quantification will again result in the code duplication increasing with a factor, and so on.

Based on the results of our analysis of the tracing idiom, we identified two aspect language abstractions that are required to capture the discovered variability and thus avoid duplication in the aspect implementation as described above. In the worst case, when employing an aspect language that is not able to meet these requirements, the aspect implementation converges to a state where there is one aspect per function. These aspects duplicate the entire tracing idiom but differ in the essential variability, which does not offer substantial advantages over an idioms-based implementation of the tracing concern.

7.2 Quantification of Parameters

Our experiments with respect to parameter-level variability show that complex parameter types require a converter function and that

[7] Depending on the join point model being static or dynamic, respectively.

each type requires a different function. If we look at mainstream aspect languages, the join point model does not explicitly include parameters or parameter types as quantifiable elements, however. We can only select functions based on their number of parameters, or based on specific types occurring in the function's signature. Few languages, such as AspectJ [18] , provide extensive reflective access to the current join point, however, and as such the actual and formal parameters can be retrieved. For expressing the tracing idiom in an aspect language for C with such capabilities, which to the best of our knowledge does not exist, the per-function advice needs to be parameterised with the list of parameters, and we need to be able to refer to the individual elements of this list in order to determine their converter method.

Let us consider an aspect language that is able to represent parameters as quantifiable elements directly. In pseudo-code and using a logic-based pointcut language, the pointcut expression could look as follows

```
execution( * *(?params))
```

which selects all functions regardless of their name and return type, and binding their parameter list to the `?params` variable. The advice code corresponding to the pointcut should then be able to refer to the individual parameters contained within the `?params` variable, and retrieve their corresponding converter functions. This requires meta-programming facilities to be present in the aspect language, not only to iterate over all parameters, but also to construct the actual trace call that will be woven into the code out of the different parameters that it requires.

7.3 Specifying Default Functionality and Exceptions

Another requirement is an aspect-language mechanism that allows us to specify default functionality as well as some exceptions to that general rule. As we have seen in our analysis, the implementation of the idiom is never consistent, not in a single component and not even when we only consider essential variability. For example, a parameter type might always be converted with the same converter function, except in one particular case when the developer is actually interested in the address of the parameter instead of in its value. Another example occurs in component CC3 where the use of a special-purpose tracing macro is mixed with the use of the default `trace` function.

One obvious solution for dealing with this kind of behaviour is to define separate aspects for these special cases. However, each exception then requires its proper aspect, and one single component might need many different aspects that have a lot of commonality. This is undesirable, since it can (and probably will) again lead to accidental variability and code duplication. Indeed, for a particular function, one single parameter might need to be converted differently, but the other parts of the tracing implementation remain standard, but need to be specified as well.

We argue that a mechanism for specifying default functionality together with its exceptions should be incorporated into the aspect language. This allows us to define one main aspect for a single component, that specifies what the default tracing implementation for that component looks like. Additionally, it allows for denoting those few cases where variability occurs. For example, a particular function that traces a particular parameter type in a different way, or that uses a different tracing macro.

The addition of annotations to the Java language and the way these annotations can be addressed in the AspectJ language are a good starting point for experimenting with such a feature. A default aspect is defined and used for all elements that have no annotation attached to them. When such an annotation is present, it should specify what it denotes and the weaver should then know how to handle the situation.

8. Discussion and Evaluation

This section discusses the implications of variability caused by idioms-based development from the perspectives of software development and legacy system migration. Whereas the discussion in the previous section concerned the essential variability, the discussion here is based on the occurrence of accidental variability. First, however, we discuss the consequences of taking into account additional variability that was not considered in our analysis.

8.1 Further Variability

It is important to note that we have only considered function-level and parameter-level variability in our experiments, and in our discussion above. However, the tracing idiom has other characteristics that we did not analyse in depth, and these characteristics make the idiom richer. Hence, more features might be needed in an aspect language then the ones we described above if we wish to express ASML's tracing idiom in an aspect.

For example, ASML code distinguishes between input and output parameters. Our analysis did not make that distinction and considered input and output tracing together. Although this allowed us to detect accidental variabilities that we would not have discovered otherwise, it also prevented us from considering the impact on an aspect implementation. An aspect needs to know which parameters are input and which are output in order to construct the appropriate input and output trace statements. An aspect weaver could extract such information from the source code using data-flow analysis, and could make it available in the aspect language, for example.

Other characteristics that we did not consider but that are relevant for such a discussion include the position of the input and output trace statements in the original code (do they always occur right at the beginning and at the end of a function's execution?), the tracing of other variables besides parameters (such as local and/or global variables), the order in which the parameters are traced, and the format string that is used, together with the format types for parameters contained within that string.

Clearly, the results we obtained can thus be seen as a lower bound of the real amount of variability present in the tracing idiom's implementation. Since the variability we found was considerable already, we arrive at our claim that simple crosscutting concerns do not exist, at least not for software systems of industrial size.

8.2 The Limitations of Idioms

A first point in the discussion of variability is more concerned with its cause than its implications. As is expected, shown in other work [4] [9], and again confirmed by the results in this paper, idioms-based development as opposed to the use of (aspect) language abstractions introduces accidental variability in the implementation of (crosscutting) concerns. Aspect-oriented languages typically provide abstractions for implementing crosscutting concerns in a localised way, thus avoiding code duplication and, more importantly, accidental variability in this duplicated code.

Consider for example the results of the analysis of variability in trace calls in the component CC2: 123 functions call the trace function using the same idiom. However, 11 functions introduce a variation in this idiom: nine functions use one variation, whereas two remaining functions each implement yet another variation. Based on these quantitative results and on an inspection of the source code, we conclude that 123 functions implement the default tracing idiom, whereas the other 11 functions exhibit accidental variability. This is confirmed informally by several ASML developers, (although we did not investigate systematically why ASML developers introduced this variability in the idiom).

Assuming our interpretation is correct, and the aforementioned variability is indeed accidental, the question is raised whether an

aspect-oriented solution for tracing would have prevented the accidental variability. Ignoring for now the parameter values that need to be traced, it is easy to imagine an aspect that captures all function executions, specifies that input and output tracing should be invoked around those executions, and provides the appropriate actual parameters for the trace invocation (`"CC2"`, `trace`, `TRACE_INT` and `func_name` in this case). Such an aspect would be preferred over an idioms-based implementation, since it specifies once in a single place how tracing should be invoked, and hence prevents the accidental variation exhibited when using idioms.

If an aspect-oriented solution can prevent accidental variability, the question remains whether all tracing idioms identified by our analysis can be expressed in a certain aspect language, such that the accidental variabilities are avoided, but the essential variabilities can be expressed. We believe the answer is yes, although the conciseness and declarativeness of the solution highly depends on the presence of certain aspect language characteristics or features, as discussed in Section 7.

It is important to note that the above does *not* show that over the course of many years, by large teams of changing developers, the aspect-oriented solution would not have introduced other accidental variabilities, ones that we cannot even envision currently because of the lack of legacy aspect-oriented systems.

The work presented in this work should therefore be complemented by a study of the 'human' causes behind the variability we observed in the code. A study of that kind would focus on the reasons for the use of a particular deviant idiom, and may reveal additional opportunities for useful abstraction within an aspect language. An example of such a study in the context of clone detection is presented by Kim *et al.* [19].

8.3 Migration of Idioms to Aspects

Given that an aspect-oriented solution has benefits over an idioms-based solution, it is relevant to study the risks involved in migrating the idioms-based implementation to an aspect-oriented implementation.

In general, migrating code of an operational software system is a high-risk effort. Although one of the biggest contributors to this risk is the scale of the software system, in our case this can be dealt with by approaching the migration of tracing incrementally [3], for instance on a component-per-component basis. However, other sources of risk need to be accounted for: the migrated code is of course expected to be functionally equivalent to the original code.

Our findings concerning variability of idioms-based concern implementations introduce an additional risk dimension. In particular, accidental variability is a complicating factor. Ignoring such variabilities by defining an aspect that only implements the essential variability means we would be changing the functionality of the system. A particular function that does not execute tracing as its first statement but only as its second or third statement, might fail once an aspect changes that behaviour, for example, when originally a check on null pointers preceded the tracing of a pointer value. So this risk is real even with functionality that is seemingly side-effect free, as is the tracing concern, and will become higher when the functionality does involve side-effects.

On the other hand, migrating the idiom including its accidental variability is undesirable as well: aspect-oriented languages are not well-equipped for expressing accidental variability and the resulting aspect-oriented solution quickly converges to a *one-aspect-per-function* solution. So the issue boils down to a trade-off between minimising the risk on the one hand, and on the other hand reducing the variability in favour of uniformity, in order to reach a reasonable aspect-oriented solution.

At the moment, we do not have an answer to the question how to migrate idioms of legacy systems with a high degree of accidental variability — at this point we do not even know what a *high degree* of accidental variability is, nor do we know whether automated migration towards aspects is feasible at all in practice, if a simple aspect such as tracing already exposes difficult problems. This discussion only serves to point out that the risk is present and that there are currently no processes or tools available for minimising the risks. Nevertheless, we can say that in the particular context of ASML, the initial proposal for dealing with the migration risk is to (1) confirm or refute the detected accidental variability, (2) eliminate the confirmed accidental variability in the idioms-based implementation of the legacy system incrementally and test if the resulting implementation is behaviour-preserving by comparing the compiled code, (3) remove the remaining idioms-based implementation of the crosscutting concern, and (4) represent the idiom and its essential variability as aspects.

8.4 Variability Findings

Our results indicate that only 7.5% of the functions implement tracing according to the standard predefined idiom, that no other standard idiom can be identified in the source code, and that 42.5% of the types defined by those functions is not traced consistently.

An important question is to what extent the figures we obtained for ASML's tracing idiom are representative. Assessing the representativeness of our findings allows us to answer the question whether we can expect similar figures for (1) other ASML components than the ones we studied; (2) other idioms in use at ASML; or (3) idioms-based software not developed by ASML.

The four components represent systems of different size, age, and maintenance history. The components we studied were selected by ASML developers because these components are currently being reworked, and they wanted an initial assessment of the variability present in the tracing implementation. They did not expect that variability to be significant. In other words, the components were chosen fairly randomly, and not with high or low variability of the tracing concern in mind.

We believe the amount of variability we observed for the tracing idiom will not be significantly lower for other idioms as applied by ASML. In another study [5], we have shown that the exception handling idiom they use is responsible for approximately 2 faults per 1000 lines of code, because the idiom is not applied consistently. Additionally, when studying the parameter checking idiom [4], we observed that 1 out of 4 parameters was not correctly checked, and that the implementation of the idiom was not at all uniform. Moreover, tracing is regarded as a very simple concern, since it is not a core functionality of the ASML software, and it is not tightly tangled with this core functionality, as opposed to exception handling and parameter checking. Hence, analysing such more complex idioms might result in significant more variability.

The question whether the ASML software is representative for software developed through idioms-based development is harder to answer. We can state that the software is developed using a state-of-the-art development process, that includes analysis, design, implementation, testing and code reviewing. The reasons for the observed variability can however be manifold: inadequate and imprecise documentation, many different developers working on a very large code base, no adequate automated verification, developers not understanding the relevance of tracing and hence paying less attention to it, etc. This situation is probably not that much different for software developed in other organisations, or even open source software. Hence, we are inclined to believe that a variability analysis for other software would show similar results. However, once again this is only speculation, and remains to be investigated further.

	Fact extraction	Lattice creation
Function	96.39 s	0.03 s
Parameter	140.8 s	0.91 s

Table 4. CPU times for tool execution on an AMD Athlon 64 3500+ with 1 GB RAM. Input consists of all components.

	O	A	Relation	Fill ratio	Concepts
Function	573	29	2331	0.14	47
Parameter	2219	385	4592	0.005	517

Table 5. Context and relation sizes for all components considered together.

8.5 Genericity of the Method

Another question concerns the genericity of the variability analysis method. ASML has expressed interest in conducting the method themselves, in order to assess the variability of tracing in other components. Furthermore, they would like to analyze the variability of other idioms. Likewise, we are interested in using the approach on non-ASML systems as well.

Several of the steps of our approach are largely manual. These include the idiom definition and variability modeling steps, as well as the aspect design step. These steps will be very similar independent of the idiom or component analyzed, and hence are sufficiently generic.

The idiom extraction and variability analysis steps require tool support. For the idiom extraction, the tools have to be configured so that they recognize the idiom at hand. Given our ASF+SDF and CodeSurfer infrastructure, this is a fairly simple step. It does, however, require knowledge of these tools, which for ASML may not be readily available. The formal concept analysis tools do not have to be adjusted: all that is needed is creating the ⟨object, attribute⟩ pairs in a simple textual format.

Based on these observations, we believe that the approach is applicable to different idioms and systems.

8.6 Scalability

The scalability of our approach is determined by two factors, tool execution time and the size of the resulting lattices. These lattices have to be processed by a human.

First, fact extraction is performed using the ASF+SDF Meta environment. The tracing code is parsed using a generalized LR parser (SGLR) [29], followed by a single traversal of the parse tree to extract the relevant facts (see Section 3). Subsequently, the FCA tool concepts is used to produce the concept lattices. Table 4 contains timing results for both the function-level and parameter level studies. In both cases, the timing results apply to the execution of the tools on all components together.

Second, concept lattices can grow exponentially with the size of the object–attribute relation. However, if a relation is sparsely filled, quadratic growth is observed in practice [20]. Table 2 shows the context and relation sizes for our studies. The *fill ratio* is defined by the actual relation size, i.e., the number of object–attribute tuples in the relation, divided by the maximum relation size, i.e., $O \cdot A$ where O is the number of objects, and A the number of attributes.

A sparsely filled relation (i.e., fill ratio below 0.1) appears to be no guarantee for a small enough number of concepts, as is shown by the parameter-level study. Inspecting 517 concepts is too big a task to be performed by a human. Fortunately, such a manual inspection is not required in our approach. The concepts of interest, i.e., those that contain own objects (see Section 6), are found automatically. The number of concepts containing own objects can be significantly lower, as can be seen in Table 3. The 'tracing variants' there correspond to concepts containing own objects.

Furthermore, the number of concepts containing own objects is a valuable indicator by itself. It tells us the number of variations on the idiom. Based on this number alone one could conclude that too much variability will prevent automatic transformation of the idiom. In that case the actual concepts do not have to be inspected by hand.

9. Related Work

The work presented in this paper can be situated between the work on *aspect mining* and that on *aspect refactoring*.

Aspect mining is the activity of (automatically) identifying crosscutting concerns in source code. Several techniques have been investigated, among which techniques based on formal concept analysis [28, 27]. An overview of these techniques can be found in [17, 23].

Once identified, the crosscutting concerns can be refactored into an aspect. Several authors have proposed a process for such migration [1, 24, 13]. All authors note that after such (semi-)automatic migration, the aspects should be "tidied up" in order to make them more general, for example by generalising advice code and creating sensible pointcuts. [13] even includes extra steps in the process to test whether the migration preserved the behaviour of the software as a whole. Both Binkley *et al* [1] and Marin *et al* [13] present results of applying their process to JHotDraw, a medium-sized object-oriented software system, while Monteiro and Fernandez [24] illustrate their approach on simple examples only.

Our work is situated in between these activities, since we know what the crosscutting concern code looks like a-priori, and our analysis can provide hints about the difficulties we can encounter when refactoring it. Given our analysis of a simple concern and our conclusions about the difficulties with automated migration, it is worthwhile to study the behaviour of all these different approaches for ASML's tracing concern. Additionally, it would be interesting to study how the results of our analysis could be fed into these approaches in order to determine automatically the refactorings that should be applied, for example.

Lippert and Lopez [22] present a study in which they (manually) extracted the design-by-contract and exception handling behaviour from a software system into aspects. Just as in our case, they found that some of the variability present in the original implementation could not be expressed easily in the (early) version of AspectJ they were using. Interestingly, this variability also involved formal parameters. Another study, by Coady *et al* [8], describes how the prefetching concern of the FreeBSD operating system can be migrated into an aspect. Both lines of work are closely related to ours, but have a different focus: they are meant as a study into the benefits of AOSD technology. Hence, the focus of both papers is on the potential gains when using aspects, and little or no discussion is present on how the aspects were extracted from the source code, and what the difficulties are when doing so.

Coyler *et al.* also observe that variability is present in the idiomatic implementation of a tracing policy of a product line [9]. Their work is focused on refactoring the tracing concern (among others), and in their case studies the variability is (partly) eliminated by the use of aspects. In comparison, our work takes a more cautious approach by visualizing any variability that we detect, and facilitating the process of distinguishing between accidental and essential variability.

Our use of formal concept analysis and the results it provides can be seen as a means to identify appropriate aspects, given all concern-related code. Many researchers, Siff and Reps [25], Lindig and Snelting [21], and van Deursen and Kuipers [12] have been using formal concept analysis for exactly that purpose, albeit in

a procedural versus object-oriented context. The idea is to let the FCA algorithm group functions that use data structures in the same way, and that the concepts found in this way correspond naturally to classes. Interestingly, both [25] and [12] mention a problem that resembles the tangling of concerns and a solution to that problem. [25] refers to it as "tangled" code that uses multiple data structures at the same time, whereas [12] considers the problem of large data structures that are actually a combination of many smaller, and largely unrelated, data structures.

The work on aspect languages, and in particular on which features should be added in order to improve the expressiveness and conciseness of aspects is of course relevant for our research as well. Several aspect languages for C have been proposed [8, 14, 26, 31], and some of them could even express the variabilities we encountered. Most of these languages are experimental in nature, however, and it remains an open question whether they scale to the size of industrial systems. On the other hand, mature aspect languages, such as AspectJ and JBoss AOP, seem to lack most of the required features for expressing the variabilities we found in the tracing idiom.

10. Concluding Remarks

In this paper, we have studied "tracing in the wild" using idioms-based development. It turns out that for systems of industrial size, tracing is not as simple as one might think: in the code we analysed, the idiom used for implementing the tracing concern exhibits remarkable variability. Part of this variability is accidental and due to typing errors or improper use of idioms, which could be seen as a plea for using aspect-oriented techniques. A significant part of the variability, however, turns out to be essential: aspects must be able to express this variability in pointcuts or advice. Even with our partial analysis of the variability of the so-called "trivial" tracing concern, we discover the need for quite general language abstractions that probably no aspect language today can provide entirely, and certainly not in the context of an industrial system. This will only worsen when more variability is considered or more complex concerns are investigated.

In summary, this paper makes the following contributions:

1. We proposed a method to assess the variability in idioms-based implementation of (crosscutting) concerns.
2. We have shown how existing tools for source code analysis and transformation, and for formal concept analysis can be combined and refined to support the variability analysis process.
3. We presented the results of applying the method on selected components of a large-scale software system, showing that significant variability is present.
4. We show how the results of the variability analysis can be used almost directly to determine the appropriate language abstractions for expressing the concern and its essential variability.
5. We discussed the implications of the accidental variability caused by idioms-based development in the context of crosscutting concerns from the perspectives of software development and legacy system migration.

The most important direction for further research is strengthening the empirical basis of our work. This involves both extending the code base to which we have applied our variability analysis techniques, and involving more concerns, such as parameter checking or exception handling, in our case studies.

Acknowledgements 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. The authors would like to thank the Ideals and Weave/C team members for their feedback and fruitful discussions.

References

[1] D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In *21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 27–36. IEEE Computer Society, 2005.

[2] 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.

[3] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann, 1995.

[4] M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the International Conference on Software Maintenance (ICSM'05)*, pages 37–46. IEEE Computer Society, 2005.

[5] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *Proceedings of the International Conference on Software Engineering (ICSE'06)*. ACM Press, 2006.

[6] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the use of clone detection for identifying cross cutting concern code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley series in Software design patterns. John Wiley & Sons, 1996.

[8] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint European Software Engineering Conference (ESEC'01) and 9th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'01)*, pages 88–98. ACM Press, June 2001.

[9] A. Colyer and A. Clement. Large-scale AOSD for middleware. In *Proceedings of the 3rd international conference on Aspect-oriented software development (AOSD'04)*, pages 56–65, New York, NY, USA, 2004. ACM Press.

[10] J. Coplien. *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1991.

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

[12] A. van Deursen and T. Kuipers. Identifying objects using cluster and concept analysis. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE 1999)*, pages 246–255. ACM Press, 1999.

[13] A. van Deursen, M. Marin, and L. Moonen. A systematic aspect-oriented refactoring and testing strategy, and its application to jhotdraw. Technical Report SEN-R0507, Centrum voor Wiskunde en Informatica, 2005.

[14] P. Durr, G. Gulesir, L. Bergmans, M. Aksit, and R. van Engelen. Applying aop in an industrial context. In *Workshop on Best Practices in Applying Aspect-Oriented Software Development*, 2006.

[15] R. P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1996.

[16] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

[17] A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Submitted to "Transactions on AOSD"*, 2006.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M.

Loingtier, and J. Irwin. Aspect-oriented programming. In *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[19] M. Kim, L. Bergman, T. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in oopl. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering (ISESE'04)*, pages 83–92, Washington, DC, USA, 2004. IEEE Computer Society.

[20] C. Lindig. Fast concept analysis. In *Working with Conceptual Structures - Contributions to ICCS 2000*, pages 152–161. Shaker Verlag, August 2000.

[21] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proceedings of the 19th International Conference on Software Engineering*, pages 349–359. ACM Press, 1997.

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

[23] M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis ˙ *ACM Transactions on Softare Engineering and Methodology*, 2006. Submitted. Appeared as Technical Report TUD-SERG-2006-013, Delft University of Technology.

[24] M. P. Monteiro and J. M. Fernandes. Refactoring a Java code base to AspectJ: An illustrative example. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005)*, pages 17–26. IEEE Computer Society, 2005.

[25] M. Siff and T. W. Reps. Identifying modules via concept analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM 1997)*, pages 170–179. IEEE Computer Society, 1997.

[26] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. Aspectc++: an aspect-oriented extension to the c++ programming language. In *Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[27] P. Tonella and M. Ceccato. Aspect mining through the formal concept analysis of execution traces. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE 2004)*, pages 112–121. IEEE Computer Society, 2004.

[28] T. Tourwé and K. Mens. Mining Aspectual Views using Formal Concept Analysis. In *Proceedings of the 4th International Workshop on Source Code Analysis and Manipulation (SCAM'04)*, pages 97 – 106. IEEE Computer Society, September 2004.

[29] E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[30] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[31] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining lost knowledge through dynamic analysis and aspect orientation - an industrial experience report. In *Proceedings of the 10th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 91–102. IEEE Computer Society, 2006.