Renovation of Idiomatic Crosscutting Concerns in Embedded Systems

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 17 maart 2008 om 10:00 uur

door

Magiel BRUNTINK

doctorandus informatica
geboren te Delfzijl

Dit proefschrift is goedgekeurd door de promotoren:
Prof. dr. A. van Deursen
Prof. dr. P. Klint

Samenstelling promotiecommissie:
Rector Magnificus, voorzitter
Prof. dr. A. van Deursen, Technische Universiteit Delft, promotor
Prof. dr. P. Klint, Universiteit van Amsterdam en CWI, promotor
Prof. dr. M. Harman, King's College, London
Prof. dr. ir. M. Akşit, Universiteit Twente
Prof. dr. ir. E. Brinksma, Universiteit Twente
Prof. dr. ir. I. Lagendijk, Technische Universiteit Delft
Prof. dr. ir. H. J. Sips, Technische Universiteit Delft

Dr. T. Tourwé heeft als begeleider in belangrijke mate aan de totstandkoming van het proefschrift bijgedragen.

The work in this thesis has been carried out at Centrum Wiskunde & Informatica (CWI) in Amsterdam as part of the Ideals project, which has been executed under the responsibility of the Embedded Systems Institute, and is partially supported by the Netherlands Ministry of Economic Affairs under the SenterNovem TS program (grant TSIT3003). Magiel Bruntink is a student of research school IPA (Institute for Programming Research and Algorithmics).

IPA Dissertation Series 2008-03

ISBN 90-6196-545-4

This work has been typeset using Leslie Lamport's LaTeX, in Times New Roman 11 pt.

Printed by Ponsen & Looijen b.v., Wageningen.

Preface

First a word about the cover. This is the second time I used one of Akiyoshi Kitaoka's enticing figures. It seems one can easily find connections between these optical illusions and software engineering research. This particular figure is interesting because of its inherent repetition. At first sight, the repetition seems perfect: it's just the same wheel appearing over and over again. Yet somehow the wheels do not seem to line up very well... It turns out there are slight changes, a notch here and there, that make all the difference. The same is true for software, and in particular for the software idioms that this thesis is about.

When Arie van Deursen invited me to write a PhD thesis with him, he told me that it would involve the amount of effort I spend on my master's thesis times eight. I can say now that he was not exaggerating. Also, I have learned that it would have been impossible to accomplish without being exposed to his natural enthusiasm and mild bravado. He was able to be very critical and precise, yet at the same time inspiring me to rise up to his challenges. Arie, I am very grateful for your trust in me, and the opportunity to work with you.

My work took place mostly in Paul Klint's research group (SEN1) at the Centrum Wiskunde & Informatica (CWI) in Amsterdam. Paul, thanks for providing me with a great amount of freedom to do my own research, while remaining interested in my work and kindly agreeing to be my second promotor. Hereby I also thank the members of my defense committee: prof. dr. ir. M. Akşit, prof. dr. ir. E. Brinksma, prof. dr. M. Harman, prof. dr. ir. I. Lagendijk, and prof. dr. ir. H. Sips.

The better part of my four years in SEN1 I worked together with Tom Tourwé. He and Arie taught me how to do research and write papers. Tom also joined me on many trips to conferences and weekly project meetings. Tom, thanks for being a great mentor and friend! Many thanks also to the other members of SEN1, in particular Jan Heering, Jurgen Vinju, Leon Moonen, Maja D'Hondt, Paul Klint, Taeke Kooiker, Tijs van der Storm, and Rob Economopoulos. They made working at CWI such a pleasant experience, both inside and outside of working hours.

Most of my research would have been impossible without the inspiring cooperation between ASML (Veldhoven) and Ideals, the research project in which I participated. There are many people to thank for this succesful project, in particular the Embedded Systems Institute and of course ASML. Special thanks go out to ASML's Remco van Engelen for being such a great bridge between the research project and the company. His enthusiasm and skill played an indispensable role. I also value the time spent with the project members from the University of Twente (UT) and Technical University Eindhoven (TU/e), in particular Gürcan Güleşir (UT) and Pascal Dürr (UT).

Finally, this thesis was only completed thanks to the support of my loving parents, Hiljan and Roel, my family and friends, and Imke, my love. I could not have done it without all of you.

Magiel Bruntink, January 2008.

Contents

List of Acronyms

ADSL	Aspect-oriented Domain-Specific Language
AOP	Aspect-Oriented Programming
AOSD	Aspect-Oriented Software Development
ASF	Algebraic Specification Formalism
AST	Abstract Syntax Tree
CCx	C Component x
EH	Exception Handling
EHM	Exception Handling Mechanism
FCA	Formal Concept Analysis
HSML	Hot Spot Markup Language
PDG	Program Dependence Graph
KLOC	Kilo (1,000) Lines Of Code
LOC	Lines Of Code
MLOC	Million Lines Of Code
NLOC	Normalized Lines Of Code
PCSL	Parameter Checking Specification Language
RCI	Return Code Idiom
SDF	Syntax Definition Formalism
SEH	Structured Exception Handling
SCATR	Source Code Analysis and TRansformation
SGLR	Scanner-less Generalized LR
SBM	Source-Based Mapping
SM	State Machine
SMELL	State Machine for Exception Linking and Logging

Chapter 1

Introduction

1.1 Software Renovation Research

Software systems are constantly evolving. Regular maintenance fixes defects, ongoing development adds features, and changing requirements require modification of existing features. Lehman and Belady's laws (Lehman and Belady, 1985) of evolution of large software systems state that, over time, large software systems necessarily change to remain useful. Furthermore, these changes cause a decline in software quality unless preventive measures are being taken. In particular, evolving the software system will itself become harder. This phenomenon has been dubbed the software evolution paradox (van Deursen, 2005) since the necessary evolution of a software system seems to be hindered by its own progress.

Software renovation (van Deursen et al., 1999) is meant to counter the problems introduced by software evolution, such that software systems can continue to evolve in the future. In that sense, renovation is similar to software re-engineering (Arnold, 1993), and both activities can be seen as special cases of preventive software maintenance (Kitchenham et al., 1999), which has the goal of making software more maintainable. Software renovation can consist of changes to the software that lie beyond maintenance. For instance, renovation can introduce a new programming language into the software system, upgrading (part of) the system to a better representation expressed in the new language. In one approach, objects are automatically identified in the source code of a legacy COBOL system (van Deursen and Kuipers, 1999b). Renovation can also consist of the extraction of documentation from the software, such that further evolution can be guided by the extracted documentation (van Deursen and Kuipers, 1999a), removal of GOTO statements from COBOL programs (Veerman, 2004) to make programs easier to understand, and the numerous adaptations that were made to software systems around the world to prevent –succesfully– most problems caused by inadequate date representations during the year 2000 transition (see for instance Smith et al. (1997)).

1.1.1 Legacy Software Systems

The software systems that are typically targetted for renovation are legacy software systems. These systems are large, consist of older software technology, and have been exposed to many years of maintenance and evolution. They are still relied on by their respective organizations, and are too valuable to be replaced completely by new, modern, systems. Legacy systems are hard to change, especially at the design level, as years of evolution have eroded the correspondences between the system's original requirements, design, and source code. In fact, Brody and Stonebraker define a legacy system as "Any information system that significantly resists modification and evolution to meet new and constantly changing business requirements." (Brodie and Stonebraker, 1995) Renovation is required to alleviate this resistance to evolution.

1.1.2 Reverse Engineering

Renovation of a legacy system starts with reverse engineering. This process is concerned with obtaining sufficient knowledge of the legacy system to perform the modifications necessary to renovate the system. Legacy systems may have a weak correspondence between all the artifacts that make up the system. Requirements, design documents, and source code may be out-of-sync. Sometimes source code is the only artifact remaining. Reverse engineering starts with the artifacts that represent the system at a low level of abstraction, typically the source code, and works upwards to obtain representations at a higher level of abstraction (Chikofsky and Cross, 1990). In software renovation settings, the reverse engineering process can result in diverse forms of information. Among many examples, one can search for code smells (van Emden and Moonen, 2002) if the intent is to improve the general code quality, or mine for aspects to obtain opportunities to use aspects in the legacy system (Ceccato et al., 2006), but even simple code metrics may be enough to guide renovation.

1.1.3 Program Transformation

The size of legacy systems typically calls for automated tools to support the phase in which modifications to the system are made. Renovations can require widespread and complex transformations, which are tedious and error prone to be performed manually. Renovation factories (van den Brand et al., 2000b) constitute a full line of tools to renovate legacy systems, from the reverse engineering to the transformation phase. A number of generic systems also support automatic program transformations: The ASF+SDF Meta-Environment (van den Brand et al., 2001) and Stratego/XT (Bravenboer et al., 2007) support transformations of context-free languages through term rewriting, TXL (Cordy, 2006) is a functional programming language specifically targetted at language processing, and DMS (Baxter, 1992) transforms the source code of a system based on changes to an accurate design description.

1.2 Idiomatic Crosscutting Concerns

This thesis studies the renovation of idiomatic crosscutting concerns. In particular, the focus will be on the special class of crosscutting concerns implemented by idioms (hence, id-

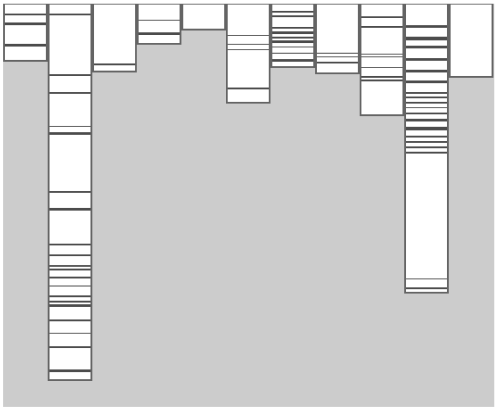

Figure 1.1: Scattering of a concern. Vertical bars represent modules. Within each vertical bar, horizontal lines of pixels correspond to lines of source code implementing the concern.

iomatic) within (legacy) embedded systems, and the benefits offered by AOP in that setting.

1.2.1 Crosscutting Concerns

Crosscutting concerns are phenomena that are present in almost any software system. They arise if the implementation of a concern –a requirement or design decision– does not fit neatly into the system's modular decomposition (Tarr et al., 1999). A crosscutting concern cannot be confined to a single modular unit in the implementation language without being tangled with the implementations of other concerns. The first symptom is referred to as *scattering*: The implementation of a concern is spread across multiple modules. The second symptom, *tangling*, refers to the implementations of two concerns being inseparable.[1] Figure 1.1 shows the scattering of a concern across a number of modules. The columns represent modules, while each row is a line of source code within a module. Dark rows are lines of source code belonging to the scattered concern. White rows are code lines of other concerns. Clearly, the concern is scattered, as its source code is present in multiple modules. Whether the scattered concern is also tangled is not apparent in Figure 1.1. It may be possible to rearrange the dark source code lines such that they all end up in a single modules, with no white lines present. For crosscutting concerns such a rearrangement is not possible.

The example of Figure 1.1 is actually a real crosscutting concern (it is discussed in detail in Chapter 3). It is responsible for the validation of pointer values in a C system. The concern must assure that every parameter of a C function that is of pointer type has been checked for the NULL value. If a NULL value is encountered, the remainder of the function must not be executed to prevent NULL pointer dereferences. In C, it is not possible to express this concern separately from the actual function definitions: resulting in an implementation that is scattered across all functions, and tangled with the other concern(s) implemented by each function.

[1] The terms scattering and tangling are due to the original paper on AOP by Kiczales et al. (1997).

1.2.2 Idioms

> *An idiom is an expression (i.e., term or phrase) whose meaning cannot be deduced from the literal definitions and the arrangement of its parts, but refers instead to a figurative meaning that is known only through common use.* (Wikipedia, November 2007)

The idioms studied in this thesis occur within source code. In programming jargon, the terms 'boilerplate code', 'template', 'pattern', or 'recipe' refer loosely to the same phenomenon. On the one hand, they are fragments of code that occur frequently and are repetitive, hence tedious, to reproduce. For example, even for the most simple C program that produces any output one has to explicitly include the standard input/output library. On the other hand, they represent common, well-tested and scrutinized solutions to frequently occurring programming problems. Design patterns (Gamma et al., 1995) are examples of the latter case. At both extremes, the use of idioms is a manual implementation technique, and hence can be considered to be a fault-prone and effort intensive practice compared to automatic code generation.

Figure 1.2 shows a realistic example (see Chapter 5) of the widespread C *return code idiom* that can be used to implement exception handling in C programs. Exceptions are represented by integers, and are passed on as return values of functions. The example shows how exceptions flow through the body of a function as prescribed by the idiom. The variable *ev* is used to keep track of the exception state. Exceptions are raised by assigning an integer value to the *ev* variable, either directly (lines 6 and 14), or by calling another function (line 10). The control flow is programmed explicitly to guard (line 9) certain statements from execution while an exception has been raised previously. Handling of exceptions is also explicitly programmed. At line 12 a condition checks whether any exception has occurred, and if true, directs control to handling code. Logging exceptional events (lines 5 and 13) is also part of the idiom.

1.2.3 Idiomatic Implementation of Crosscutting Concerns

Idiomatic implementation, the practice of applying idioms to implement something, is a common programming technique. Copy-paste-adapt programming (as studied by Kim et al. (2005), among many others) is an example of the practice. The use of design patterns (Gamma et al., 1995) is another prevalent example. Idiomatic implementation represents a style of reuse of programming solutions (i.e., the idioms) that is informal in the sense that it does not use programming language features to explicitly specify reuse. Instead, the programming solution itself is replicated, and possibly adapted slightly to fit its new context. Some forms of idiomatic implementation, in particular copy-paste-adapt programming, have debatable merit for software evolution (Koschke et al., 2007; Kapser and Godfrey, 2006; Aversano et al., 2007).

Crosscutting concerns are concerns that are scattered across the modules of a system, and tangled with other concerns. Some crosscutting concerns are scattered such that a relatively similar programming problem has to be solved in a large number of modules. These crosscutting concerns are termed *homogeneous* by Colyer et al. (2004). The relatively simple pointer validation concern discussed earlier is a homogeneous crosscutting concern, but also more

```
1  int f(int a, int b) {
2    int ev = OK;
3
4    if (a < 0) {
5      LOG(F_ERROR, OK, "a < 0");
6      ev = F_ERROR;
7    }
8
9    if (ev == OK) {
10     ev = g(a);
11
12     if (ev != OK) {
13       LOG(F_ERROR, ev, "error from g");
14       ev = F_ERROR;
15     }
16   }
17
18   return ev;
19 }
```

Figure 1.2: C return code idiom.

complex concerns like exception handling might be considered homogeneous (Lippert and Videira Lopes, 2000).

Since implementing a homogeneous crosscutting concern requires solving a relatively similar programming problem in a large number of modules throughout a system, an idiomatic approach can be used. That is, the system's programmers define an idiom that can be applied within the context of each module to implement the crosscutting concern. For instance, the idiom is included in the system's programming manual (this is the case for the return code idiom example in Figure 1.2).

We view the idiomatic implementation of crosscutting concerns as a way of *coping* with crosscutting concerns that is sometimes necessary. There may be no means of changing the system such that concerns are no longer crosscutting. In particular, the modular decomposition of a system may be biased towards a single concern (Tarr et al., 1999) and simply cannot be changed, or the system's programming language may be lacking in features to modularize certain crosscutting concerns (Kiczales et al., 1997).

The practice of idiomatic implementation of crosscutting concerns may be especially prevalent in legacy systems, since legacy systems can contain many crosscutting concerns. Due to their age, legacy systems are implemented in older programming languages. Such languages can lack the language features necessary to succinctly express certain concerns. For instance, the C language (by default) does not have the structured exception handling support of modern languages like Java and C#. Furthermore, the lack of aspects, or other modularization features (e.g., object-oriented features such as inheritance and polymorphism), can

increase the number of crosscutting concerns a legacy system has. Finally, legacy systems are not easily changed, especially at the design level (Brodie and Stonebraker, 1995). This is partly due to the *dominant* decomposition (Tarr et al., 1999) a system has, i.e, the system is decomposed according to a certain design, which is crystallized in the system's source code. The original design may not have foreseen the full evolution of the system during its lifetime. Concerns added to the system later on may therefore fit badly into the decomposition, and hence become crosscutting.

1.2.4 Aspect-Oriented Programming (AOP)

AOP is a relatively new addition to the spectrum of programming paradigms. It aims at providing programming language constructs that modularize crosscutting concerns. Since the original paper by Kiczales et al. (1997), these constructs are typically called aspects. Aspects are comparable to modules of a system, like objects, classes, headers, methods, functions, and so on. Compared to these traditional modules, aspects differ in the way they are composed with other modules. Traditional modules typically *import* functionality from other modules: C header files can be included, functions or methods call each other, objects refer to each other via field accesses, and so on. In contrast, aspects have the ability to *export* functionality specified in the aspect. The functionality in the aspect can interfere with the functionality specified in other (traditional) modules. Importantly, the interference can occur while the traditional modules remain *oblivious* (Filman and Friedman, 2001) to the fact, in the sense that there is no additional functionality required to facilitate the interference, *as far as the programmer of the receiving module is concerned.* Aspects can thus interfere with the functionality of another module without the other module being aware of the interference.

AOP is easily explained by an example. Consider the two C functions in the top part of Figure 1.3. Suppose that tracing functionality has to be added: i.e., on each entry of a function, print the value of the parameter `a`, and on each exit, print the return value. The traditional way to add this functionality is to insert print statements at the appropriate places. The bottom part of Figure 1.3 shows this solution: print statements have been added at lines 2, 4, 9, and 11.

AOP can solve this problem more elegantly, by specifying separately how the tracing functionality should be implemented. The original code will not need to be changed. Figure 1.4 shows an possible AOP solution.[2] The code is split into two parts: the base code consisting of the original code, and an aspect that describes the addition of the necessary print statements. The aspect consists of *pointcuts*, and *advice* associated with the pointcuts. A pointcut basically describes *where* or *when* something should happen. In AOP terminology, a pointcut specifies a number of *joinpoints*, which are events in the execution of the program.

In Figure 1.4, the pointcut at lines 1 and 2 captures two joinpoints: one for each function in the example. The signature `functions(int a)` on line 1 gives the pointcut a name (i.e., `functions`), and states that it has a parameter `a` of type `int`. On line 2 an expression specifies the joinpoints that pointcut `functions` captures. The first clause, i.e., `execution(int $ (int))` specifies all joinpoints that are executions of functions that have the `int $ (int)` signature. The `$` in this signature is a wildcard that matches any possible function name.

[2] The syntax of the example aspect in Figure 1.4 is that of Aspect-Oriented C, or ACC (Aspect-oriented C, 2007).

Base code

```
1  int f(int a) {
2    a = a + 1;
3    return a;
4  }
5
6  int g(int a) {
7    a = a + 2;
8    return a;
9  }
```

Base code with tracing code

```
1  int f(int a) {
2    printf("> f: a = %d", a);
3    a = a + 1;
4    printf("< f: %d", a);
5    return a;
6  }
7
8  int g(int a) {
9    printf("> f: a = %d", a);
10   a = a + 2;
11   printf("< f: %d", a);
12   return a;
13 }
```

Figure 1.3: Manually adding tracing code.

This clause thus specifies the execution of all functions that return an `int` and have one `int` parameter. Both functions in the original example match this signature. The second clause, `args(a)`, makes the run-time value of the `int` parameter available to users of the pointcut (as `a`).

Pointcuts are used by advice to specify when (or where) the advice should be applied. Lines 4–9 in Figure 1.4 describe advice that implements the tracing functionality. Line 4 specifies that the advice body should be applied `around` the joinpoints of the pointcut `functions`. Advice of type `around` is executed instead of the joinpoints it is applied to, so in our case it will replace the original function bodies. Other advice types are `before` and `after`, which execute advice before or after the original joinpoints, respectively. Lines 5–8 perform the actual print statements needed for tracing. At line 7, a special keyword, `proceed` is used to execute the original body of the function that the advice has been applied to. Its return value is the return value of the original function. The return value is stored in the `result variable`, such that it can be printed later at line 8. The print statements in the advice make use of the keyword `this` to obtain the name of the function to which the advice has been applied. This

Base code

```
1  int f(int a) {
2    a = a + 1;
3    return a;
4  }
5
6  int g(int a) {
7    a = a + 2;
8    return a;
9  }
```

Aspect

```
1  pointcut functions(int a):
2    execution(int $ (int)) && args(a);
3
4  int around (int a) : functions (a) {
5    int result;
6    printf("> %s: a = %d\n", this->funcName, a);
7    result = proceed();
8    printf("< %s: %d\n", this->funcName, result);
9  }
```

Figure 1.4: An aspect adding tracing code.

keyword is part of the aspect language and is used to expose run-time information about the joinpoints to which advice has been applied.

The base code and the tracing aspect are compiled together by an *aspect weaver* that performs the task of applying the advice to the joinpoints as specified by the aspects. Running the traditional and the AOP tracing implementations will now produce the same output (assuming a main function that first calls `f(1)`, then `g(2)`):

```
> f: a = 1
< f: 2
> g: a = 2
< g: 4
```

A broad range of aspect languages are currently being used in industrial and research settings. To name just a few, aspectJ (Kiczales et al., 1997) is an industrial-strength aspect weaver for Java, while AspectC++ (2007), and Aspect-oriented C (2007) are research prototypes for aspects in C++ and C, respectively.

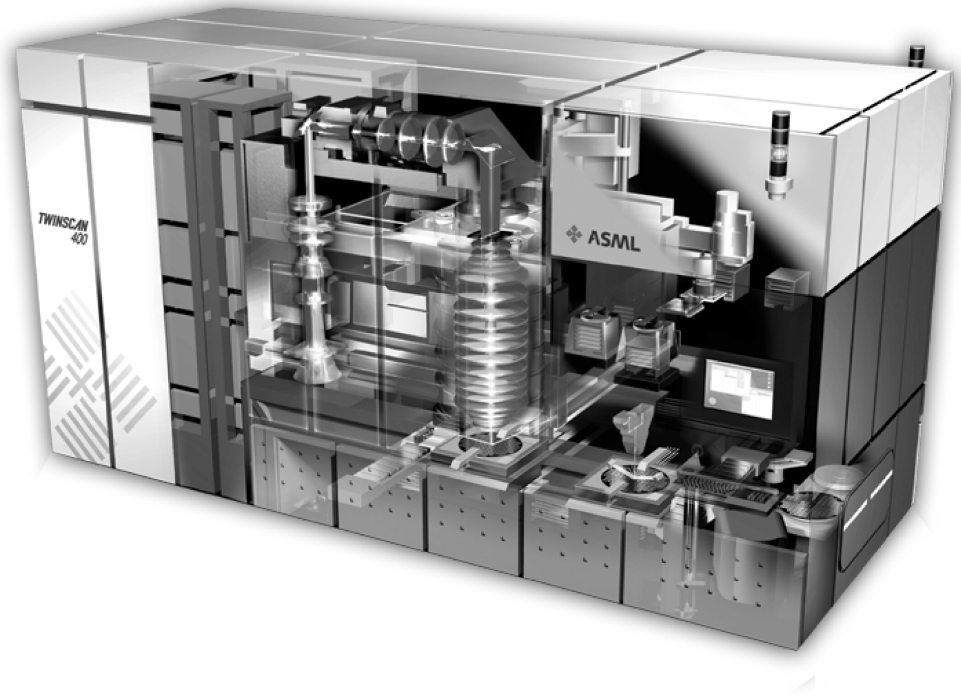

Figure 1.5: Cutout of an ASML TWINSCAN XT:400F wafer scanner (courtesy of ASML).

1.3 Industrial Context: ASML

All the research described in this thesis has been performed in the particular industrial context of ASML. ASML is a provider of lithography systems for the semiconductor industry. They are based in Veldhoven, the Netherlands, and various other locations throughout the world. Lithography systems are a key component of integrated circuit (IC) production, and are subject to a highly innovative and competitive market. The basic functionality provided by a lithography system consists of imaging extremely precise (nanometer scale) circuits onto silicon wafers. A current family of ASML lithography systems is TWINSCAN (see Figure 1.5). TWINSCAN systems are also known as wafer scanners.

The ASML wafer scanners are very complex systems. They consist of numerous hardware components that operate at dramatically different levels of precision. Silicon wafers are moved inside a wafer scanner at average speeds measured on a meters per second (m/s) scale, yet the wafers are still aligned such that a nanometer ($10^{-9}m$) scale precision is obtained during imaging. To accomplish this feat, the components of a wafer scanner are tightly integrated by means of a large embedded software system that is responsible for the proper cooperation of all the hardware components.

This embedded software system is the object of study in this thesis. Its estimated size is 15

million lines of source text[3] written mostly in the C programming language. This source text has been growing (linearly) with the consecutive generations of ASML lithography systems, in a timeframe of 20 years. It is being maintained, that is, developed, extended, fixed, and tested, by roughly 475 people (in 2002).

Idiomatic implementation is an integral part of software development at ASML. Key concerns, like exception handling, are implemented by applying an idiom, as well as some auxillary concerns. The idioms themselves are described in a manual that every programmer must adhere to. In the manual, code examples are an important means of conveying the form of the idioms. Anecdotal evidence suggests that many programmers use the code examples as templates, filling in the blanks with their own code.

The example return code idiom presented in Figure 1.2 is actually modelled after a real idiom that is being applied at ASML. The return code idiom is one of the idioms that is applied in almost every function in the ASML system, since the concern (exception handling) that the idiom is meant to implement is almost always applicable. The exception handling concern and the return code idiom are discussed in more detail in Chapters 5, and 7. Other concerns that are implemented by idioms are tracing of the in- and output values of a function (Chapter 6) and the validation of pointer values (Chapter 3). These concerns crosscut the ASML system in the sense that they are scattered, i.e., apply to almost all modules, and are tangled with the implementation of other concerns. Figure 1.2 shows a clear example of a tangled implementation: The exceptional control flow is tightly connected to the normal control flow of the function.

The industrial context of ASML provides many interesting and realistic problems such as the improvement of the return code idiom. Studying such realistic problems is not easy: Many factors influence the observations made, and many practical problems have to be solved before effective research can be done. To obtain some solid ground to build on, the research presented in this thesis uses the industry-as-laboratory research method as proposed by Potts (1993). Section 8.4 discusses the experiences with this method, any challenges that were encountered, and recommendations for future research.

1.4 Research Questions

Idiomatic crosscutting concerns such as the ASML exception handling concern are the objects of study of this thesis. The goal is to research such concerns in their legacy situation, quantify and qualify the problems currently being experienced, and identify possible benefits offered by renovating idiomatic crosscutting concerns using aspects. The research is performed in the industrial context of the ASML software system. Idioms are explicitly being used within this system to implement a number of crosscutting concerns. We will now discuss the four research questions that drive the research presented in this thesis. Table 1.1 provides an overview of the research questions and the crosscutting concerns that were investigated during the research.

[3] Including comments and formatting (whitespace).

Research Question	Chapter(s)
1. Can idiomatic crosscutting concerns be identified automatically? In particular, are clone detection tools suitable for this purpose?	2
2. Is it possible to renovate idiomatic crosscutting concerns? What are the challenges for an automatic approach?	3, 4, 6
3. Are idiomatic crosscutting concerns sources of implementation faults or inconsistencies?	5, 6
4. What are the benefits offered by renovating idiomatic crosscutting concerns using aspects?	3, 6, 7

Crosscutting Concern	Chapter(s)
Parameter checking	2, 3
Tracing	2, 6
Exception handling	2, 5, 7

Table 1.1: Overview of the main topics of this thesis.

Research Question 1

Can idiomatic crosscutting concerns be identified automatically? In particular, are clone detection tools suitable for this purpose?

Code duplication (or code cloning) is the phenomenon that source code contains multiple identical (or very similar) fragments of code. Such duplicated fragments can exist because programmers sometimes use a copy-paste-adapt style of programming, i.e., existing code is copied to another context, and, if necessary, slightly adapted (Kim et al., 2004). Chapter 2 investigates the correspondence between code duplication and idiomatic crosscutting concerns. The hypothesis for this investigation is that since the use of idioms will plausibly result in duplicated code, the source code of idiomatic crosscutting concerns will indeed exhibit duplication. An actual C component of ASML is analyzed by three different clone detection tools. Then, the clone detection results are compared to a reference body of idiomatic crosscutting concern code, resulting in quantitative data on the correspondence.

This investigation is interesting in the context of aspect mining (see Kellens et al. (2007) for a survey of aspect mining techniques). Idiomatic crosscutting concerns are believed to be good aspect candidates, and aspect mining aims at automatically finding aspect candidates in

existing software systems. In Chapter 2 the use of clone detection tools for the purpose of aspect mining is discussed in light of the actual correspondence found between clones and crosscutting concerns.

Research Question 2

> *Is it possible to renovate idiomatic crosscutting concerns? What are the challenges for an automatic approach?*

According to Baniassad et al. (2002) crosscutting concerns are detrimental to software evolution. Idiomatic crosscutting concerns are no exception, since the use of idioms does not address the problems of scattering and tangling. Modern programming techniques, in particular aspect-oriented programming, probably alleviate the evolution problems caused by crosscutting concerns. It is therefore interesting to consider the renovation of crosscutting concerns within legacy systems such that aspect-oriented programming can be used.

The renovation (or re-engineering) of legacy systems is a challenging research area, where automation is one of the key challenges (Arnold, 1993; Brodie and Stonebraker, 1995). Automation is also increasingly important since the scale of legacy systems shows monotonous growth Lehman and Belady (1985). Chapter 3 proposes an approach that automates the process of renovating (idiomatic) crosscutting concerns. This approach focuses on the introduction of aspects as a replacement for the use of idioms.

An essential part of this approach is a tool that encodes an idiom such that the tool can find the exact locations in the source code where the idiom has been applied. Furthermore, the tool can find violations of the idiom, i.e., locations in source code where the idiom has not been implemented correctly. Chapter 5 discusses the use of such a tool, called SMELL, in the setting of finding implementation faults in idiomatic exception handling code.

Automatic renovation requires a source code analysis and transformation infrastructure to be carried out. In the case studies presented in this thesis different technologies are used to support various renovation tasks. The SMELL tool, for example, was implemented as a plug-in for Grammatech's CodeSurfer program analysis toolkit. However, various code transformations (e.g., removing legacy idiom occurrences, or generating aspect code) were build upon a different technology: the ASF+SDF Meta-Environment (van den Brand et al., 2001). A complicating factor of using different tools is that they can have different models of the source code that is processed. Since the results of a tool are typically expressed in terms of its own model, this can lead to incompatibilities in interpreting the results. Chapter 4 proposes SCATR, a framework that deals with the situation of tools with different source code models that are immutable, such that it may still be possible to transfer results between tools.

Another challenge for renovation consists of inconsistencies and faults within the source code of idiomatic crosscutting concerns.[4] Should variations among the scattered applications of an idiom be unified while renovating? Unification may be fine for small accidental variations, but essential variations (i.e., necessary deviations from the idiom) may be required to

[4]Note that the software evolution paradox predicted this challenge: renovation can be considered software evolution, and hence erosion caused by earlier evolution hinders its progress. The question remains whether renovation will break the paradox.

remain. How to handle implementation faults? Fixing faults may appear benign and beneficial, but a legacy system may depend on work-around's for those faults. Fixing the faults may break the work-around's, and thus imply serious risk. Chapter 6 discusses these problems in more detail. Whether idiomatic crosscutting concerns actually exhibit faulty or inconsistent implementations is the subject of the next research question.

Research Question 3

Are idiomatic crosscutting concerns sources of implementation faults or inconsistencies?

Idiomatic implementation is a manual and repetitive task. Crosscutting concerns, if implemented idiomatically, may therefore be particularly fault prone. Chapter 5 focuses on the return code idiom that was introduced in Figure 1.2, and that is used throughout the ASML system to implement exception handling. As argued before, the implementation of the exception handling concern is highly scattered and tangled as a result. Chapter 5 first defines a fault model for the faults that are expected to occur within the exception handling implementation. Then, an automatic fault finding tool, named SMELL, is constructed to find the faults defined in the fault model. This tool is used in a number of case studies to investigate the actual fault proneness of ASML software.

Chapter 2 shows that idiomatic crosscutting concern code is often duplicated or very similar. Unfortunately, the differences that do exist turn out to be far from consistent. Chapter 6 explores the variability that is present within the idiomatic code of a particular ASML concern: tracing. The ASML tracing concern requires two calls to the tracing library are made within the body of a function: One call at the start of the function, tracing the input parameters, and a second call at the end of the function that traces the output parameters. The form of the library call itself is dictated by the tracing idiom.

Variability occurs when the idiom is applied in the same context, but the resulting code differs unexpectedly. For instance, parameters need to be converted to a string as part of the tracing call. It is expected that parameters of the same type are converted consistently, but it turns out that these conversions are not always consistently implemented. Especially large structure types are converted differently depending on the function they are traced in. It can often be unclear whether such variability is essential, i.e., required for correct implementation (despite the idiom), or accidental, i.e., a mistake made by a programmer. Chapter 6 presents a method to explore the variability present in idiomatic implementations, and make a distinction between essential and accidental variability.

Research Question 4

What are the benefits offered by renovating idiomatic crosscutting concerns using aspects?

Aspect-oriented programming aims at preventing the problems caused by idiomatic crosscutting concerns that we observe in legacy systems. Therefore, renovating such concerns using aspects should yield benefits. The aspect-oriented programming community is not focused on the use of aspects within legacy systems, and hence evidence for the benefits of

the use of aspects in that context is currently lacking. This thesis presents a number of case studies of the renovation of idiomatic crosscutting concerns in a large-scale legacy system: the ASML software system, which consists of 15 million lines of C code. First, Chapter 3 reports on a case study of the ASML parameter checking[5] concern. Second, Chapter 6 provides an in-depth study of the ASML tracing concern, in particular the variability present within that concern, and the consequences of variability for the use of aspects. Finally, Chapter 7 describes the renovation of the ASML exception handling concern. The current exception handling idiom, which is similar to the C return code idiom, is first reengineered to a more structured idiom provided by a library for structured exception handling (Goodenough, 1975). Next, Chapter 7 analyzes the expected benefits of a re-engineering using aspects.

1.5 Software and Technology

This section describes the software that was developed to support the research performed for this thesis. The technologies used for implementation consisted of:

- The ASF+SDF Meta-Environment (van den Brand et al., 2001).
- The CodeSurfer program analysis toolkit, programmable edition (CodeSurfer, 2007). The functionality of CodeSurfer was extended using the Scheme programming language (STk dialect).
- Various scripting languages, such as Perl, Make, and Unix shell script.

Table 1.2 gives an overview of the developed software. The columns Scheme, ASF, and SDF show the non-blank line counts of the source texts in their respective languages. Scripting is the non-blank line count of scripts written in Perl, Make, or Unix shell script. Files is the number of files used, and Chapter refers to any chapter(s) in this thesis that describes(s) the software. From the top down, the software mentioned in Table 1.2 becomes more specific to a particular context: generic, specific to ASML, and specific to the three crosscutting concerns investigated (at ASML) in this thesis.

Significant effort has been invested in the development of a C grammar in SDF that is capable of dealing with most occurrences of preprocessor use within source text, without actually running the preprocessor. This grammar was developed initially by Jurgen Vinju (for a partial description, see Chapter 8 in Vinju (2005)), and was later adapted to the ASML context by the author of this thesis.

Table 1.2 includes the software developed to support the renovation steps described in Chapter 3. In the case of Exception Handling, transformation refers to the elimination of the RCI, and its replacement by SEH code (see Chapter 7). For the Tracing concern, transformation refers to elimination of tracing code, and a transformation step that isolates tracing code to facilitate the formal concept analysis experiment presented in Chapter 6. A large amount of code could be shared across the analyses for the different crosscutting concerns. Table 1.2 lists this code as analysis infrastructure. SCATR instantiation refers to code that is required to instantiate the SCATR framework for the AMSL context.

[5] Parameter checking consists of making sure the pointer-type parameters of a function are not NULL at runtime.

Software	Scheme	ASF	SDF	Scripting	Files	Chapter
Generic	80	440	920	395	47	
C grammar	-	-	544	-	11	-
SCATR	80	440	376	395	36	4
ASML-specific	3,206	39	578	-	61	
C grammar adaptation	-	-	279	-	15	-
Analysis infrastructure	3,206	-	-	-	39	-
SCATR instantiation	-	39	299	-	7	4
Parameter Checking	1,649	79	887	-	24	
Concern verifier	1,282	-	-	-	9	3
Aspect extractor	354	-	-	-	4	3
Concern eliminator	13	13	43	-	8	4
PCSL specification	-	66	484	-	3	3
Exception Handling	2,874	357	294	-	33	
SMELL	2,874	-	-	-	27	5
Aspect extractor	-	153	170	-	3	7
Transformation	-	204	124	-	3	7
Tracing	3,702	359	200	22	52	
Concern verifier	1,058	-	-	-	28	6
Aspect extractor	2,592	-	-	-	28	6
Transformation	52	359	200	22	14	4, 6
Totals	11,511	1,274	2,879	417	217	

Table 1.2: Overview of the developed software.

1.6 Origins of Chapters and Acknowledgements

Chapter 2. On the Use of Clone Detection for Identifying Crosscutting Concern Code. This chapter was published in the IEEE Transactions on Software Engineering in October 2005 (Bruntink et al., 2005b). It is co-authored by Arie van Deursen, Remco van Engelen and Tom Tourwé. An earlier version of this chapter as Bruntink et al. (2004a) won the best paper award at the 20st IEEE International Conference on Software Maintenance (ICSM 2004).

Chapter 3. Isolating Idiomatic Crosscutting Concerns. This chapter was published in the Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005) as Bruntink et al. (2005a). It is co-authored by Arie van Deursen and Tom Tourwé.

Chapter 4. Linking Analysis and Transformation Tools with Source-based Mappings. This chapter was published in the Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006) as Bruntink (2006). Thanks to Jurgen Vinju, Rob Economopoulos, Tijs van der Storm and Tom Tourwé for commenting on drafts of this chapter.

Chapter 5. Discovering Faults in Idiom-Based Exception Handling. This chapter was published in the Proceedings of the 28th International Conference on Software Engineering (ICSE 2006) as Bruntink et al. (2006). It is co-authored by Arie van Deursen and Tom Tourwé.

Chapter 6. Analysing Variability in Large-scale Idioms-based Implementations of Crosscutting Concerns. This chapter was published in the Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD 2007) as Bruntink et al. (2007). It is co-authored by Arie van Deursen, Maja D'Hondt and Tom Tourwé.

Chapter 7. Renovating Idiomatic Exception Handling. This chapter will appear in the Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008) in April 2008. Thanks to Tom Tourwé for commenting on a draft of this chapter.

Chapter 2

On the Use of Clone Detection for Identifying Crosscutting Concern Code*

In systems developed without aspect-oriented programming, code implementing a crosscutting concern may be spread over many different parts of a system. Identifying such code automatically could be of great help during maintenance of the system. First of all, it allows a developer to more easily find the places in the code that must be changed when the concern changes, and thus makes such changes less time consuming and less prone to errors. Second, it allows the code to be refactored to an aspect-oriented solution, thereby improving its modularity.

In this chapter, we evaluate the suitability of clone detection as a technique for the identification of crosscutting concerns. To that end, we manually identify five crosscutting concerns in the ASML C system, and analyze to what extent clone detection is capable of finding them.

2.1 Introduction

The tyranny of the dominant decomposition (Tarr et al., 1999) implies that no matter how well a software system is decomposed into modular units, some functionality (often called a *concern*) *crosscuts* the decomposition. In other words, such functionality cannot be captured cleanly inside one single module, and consequently its code will be spread throughout other modules.

From a maintenance point of view, such a crosscutting concern is problematic. Whenever this concern needs to be changed, a developer should identify the code that implements it. This may possibly require him to inspect many different modules, since the code may be

*This chapter was published in the IEEE Transactions on Software Engineering in October 2005 (Bruntink et al., 2005b). It is co-authored by Arie van Deursen, Remco van Engelen and Tom Tourwé.

scattered across several of them. Moreover, identifying the code specifically related to the relevant concern may be difficult. Apart from the fact that the developer may not be familiar with the source code, this code may also be tangled with code implementing other concerns, again due to crosscutting. It should thus come as no surprise that identifying crosscutting code may be a time-consuming and error-prone activity, as shown by Soloway et al. (1988) for delocalized plans.

Aspect-oriented software development (AOSD) has been proposed for solving the problem of crosscutting concerns. Aspect-oriented programming languages have an abstraction mechanism targetted specifically at crosscutting concerns, called an *aspect*. This mechanism allows a developer to capture crosscutting concerns in a localized way.

In order to use this new feature, and make the code more maintainable, existing applications written in ordinary programming languages could be evolved into aspect-oriented applications. Once again, this requires identifying the crosscutting concern code such that it can be refactored using aspects. The activity of finding opportunities for the use of aspects in existing systems is typically referred to as *aspect mining* (Hannemann and Kiczales, 2001).

To support developers in these tasks some form of automation is highly desirable. Clone detection techniques are promising in this respect, due to two likely causes of code cloning occurring within scattered crosscutting concern implementations. First, by definition scattered code is not well modularized. Several reasons can be identified for this lack of modularity, including missing features of the implementation language (exception handling, or aspects, for instance), or the way the system was designed. In both cases developers are unable to reuse concern implementations through the language module mechanism. Therefore, they are forced to write the same code over and over again, typically resulting in a practice of copying existing code and adapting it slightly to their needs (Kim et al., 2004).

Second, they may use particular coding conventions and idioms to implement *superimposed* functionality, i.e., functionality that should be implemented in the same way everywhere in the application. Logging and tracing are the prototypical examples of such superimposed functionality.

We hypothesize from these observations that clone detection techniques might be suitable for identifying some kinds of crosscutting concern code, since they automatically detect duplicated code in a system's source code. In this chapter we report on a case study in which we evaluate the suitability of three different clone detection techniques for identifying crosscutting concern code. We manually identify five crosscutting concerns and evaluate to what extent the crosscutting concern code is matched by three different clone detection techniques. The evaluation considers token, AST and PDG-based clone detection techniques (see Section 2.2), and provides a quantitative comparison of their suitability.

The case study considers crosscutting concerns prevalent in the source base of ASML, a producer of lithography systems based in Veldhoven, The Netherlands. A domain expert has manually annotated occurrences of five crosscutting concerns (Error Handling, Tracing, NULL-value Checking, Range Checking, and Memory Error Handling) in a component consisting of 16,406 lines of C code. The complete source base consists of roughly 15 million lines of C code, of which at least 25% is estimated to be dedicated to crosscutting concerns (based on the results found for the component considered in the case study).

The chapter is structured as follows. Section 2.2 discusses related work in the areas of clone detection and aspect mining. In Section 2.3 we describe the case study and the five

different crosscutting concerns. Subsequently, in Section 2.4 we detail the approach used to evaluate the capability of clone detection to find these crosscutting concerns. In Section 2.5 we present and explain the results obtained, followed by a discussion in Section 2.6. Finally, the chapter is concluded in Section 2.7.

2.2 Related Work

2.2.1 Clone Detection Techniques

Clone detection techniques aim at finding duplicated code that may have been adapted slightly from the original. Several clone detection techniques have been described and implemented:

Text-based techniques (Johnson, 1993; Ducasse et al., 1999) perform little or no transformation to the 'raw' source code before attempting to detect identical or similar (sequences of) lines of code. Typically, white space and comments are ignored.

Token-based techniques (Kamiya et al., 2002; Baker, 1995) apply a lexical analysis (tokenisation) to the source code, and subsequently use the tokens as a basis for clone detection.

AST-based techniques (Baxter et al., 1998) use parsers to first obtain a syntactical representation of the source code, typically an abstract syntax tree (AST). The clone detection algorithms then search for similar subtrees in this AST.

PDG-based approaches (Komondoor and Horwitz, 2001; Krinke, 2001) go one step further in obtaining a source code representation of high abstraction. Program dependence graphs (PDGs) contain information of a semantical nature, such as control- and data flow of the program. Komondoor and Horwitz (2001) look for similar subgraphs in PDGs in order to detect similar code. Krinke (2001) first augments a PDG with additional details on expressions and dependencies, and similarly applies an algorithm to look for similar subgraphs.

Metrics-based techniques (Mayrand et al., 1996) are related to hashing algorithms. For each fragment of a program the values of a number of metrics are calculated, which are subsequently used to find similar fragments.

Information Retrieval-based methods aim at discovering similar high level concepts by exploiting semantic similarities present in the source text itself (Marcus and Maletic, 2001; Mishne and de Rijke, 2004).

An important application of clone detection is the improvement of source code quality by refactoring duplicated code fragments (Rieger et al., 1999). Several authors have proposed to use clone detection in this setting. Baxter et al. (1998) search for opportunities for replacing clones with calls to a function that factors out the commonalities among the clones. Balazinska et al. (2000) focus on analyzing differences among clones, and their contextual dependencies, in order to determine suitable refactoring candidates. van Rysselberghe and Demeyer (2004) compare three classes of clone detection techniques, i.e., line matching, parametrized matching and metric fingerprinting with respect to refactoring the obtained clones. Removal of clones by refactoring is further studied by Fanta and Václav (1999).

Other applications exist as well. van Rysselberghe and Demeyer (2003), for example, use a clone detection algorithm to study the evolution of a software system. In particular, they try to distinguish *move method* refactorings that were applied when evolving one version of the software into another.

Following Walenstein (2003); Walenstein and Lakhotia (2003), clone detection adequacy depends on application and purpose. Finding crosscutting concerns is a completely new application area, potentially requiring specialized types of clone detection.

2.2.2 Aspect Mining

Although research on aspect mining is still in its infancy, several prototype tools have already been developed that support developers in identifying crosscutting code. These tools vary in accuracy and the level of automation that they offer.

The *Aspect Browser* (Griswold et al., 2001) is a programming environment that provides *text-based mining*, which means it relies on string pattern-matching techniques to identify aspects. A developer specifies a regular expression that describes the code belonging to the aspect of interest, and a color. The programming environment then identifies the code conforming to the regular expression, and highlights it using the associated color in the source code editor. Three concern elaboration tools, including the Aspect Browser, are compared in a recent study by Murphy et al. (2005). This study shows that the queries and patterns are mostly derived from application knowledge, code reading, words from task descriptions, or names of files. Prior knowledge of the system or known starting points strongly affect the usefulness of the outcomes of the analysis.

The *Aspect Mining Tool* (Hannemann and Kiczales, 2001) is an extension of the Aspect Browser that introduces a combination of text-based and *type-based mining*. Type-based mining considers the usage of types within an application to identify crosscutting code. It appears to be a good complement to simple text-based mining, and the combination of the two ensures that far less false positives and false negatives occur.

The *Prism* tool (Zhang and Jacobsen, 2004) (an earlier version is called AMTEX (Zhang and Jacobsen, 2003)) in its turn extends the Aspect Mining Tool, and additionally provides a *type ranking* feature and takes control flow information into account. The type ranking feature is based on the assumption that types that are used widely in the application are a good indicator of crosscutting code. Therefore, the tool ranks the types in the system according to their use. The tool also takes control-flow information into account to identify aspects: e.g. it considers the values involved in conditional branches and the code involved in accessing these values (assignments, method calls, etc). If such code is not well localized and appears in many places in the application, it may be a very good candidate for an aspect.

Ettinger *et al.* discuss a totally different approach to aspect mining that identifies entangled code based on input by the developer, and disentangles it using program slicing and aspect-oriented techniques (Ettinger and Verbaere, 2004). In other words, the developer points out a particular expression or statement and a tool automatically computes the corresponding slice. The code fragment computed in this way can then be extracted into an aspect.

Fully automated tools for aspect mining are also proposed in the literature. Breu and Krinke propose a tool that dynamically analyzes Java software to identify aspects (Breu and Krinke, 2004). To that end, program traces are generated and analyzed automatically. The idea is to detect particular patterns occurring in the trace, such as a call to a particular method `a` that is always followed by a call to method `b`, or a call to method `c` that always occurs inside a call to method `d`. Such patterns could point to before/after aspects.

Shepherd *et al.* present a tool that uses a clone detection algorithm based on a program dependence graph (Shepherd et al., 2004) representation of Java code. The tool identifies possible aspects fully automatically, focusing currently on a specific type of aspects that introduces code before function calls (i.e., *before* advices). Their approach seems capable of finding such aspects in Java code, though the authors report that evaluation of their findings has been difficult due to a lack of a reference set of desirable aspects. In our work such a reference set (consisting of manual annotations) is exploited in the evaluation.

Other techniques uses formal concept analysis (Tourwé and Mens, 2004) or metrics (Marin et al., 2006) to find crosscutting concern code, and combinations of these techniques are proposed to combine the respective advantages and counter the disadvantages (Ceccato et al., 2005).

Traditionally, AOP techniques have been applied to object-oriented applications. The idea of applying it for improving the modularity of large-scale C programs is not new, however. Most notably, Coady *et al.* report on an experiment using aspect-oriented techniques to modularize the implementation of prefetching within page fault handling in the FreeBSD operating system kernel (Coady et al., 2001). To that end, they make use of an aspect language tailored specifically to the C programming language called *AspectC*, which is currently under development. However, in their experiment, the crosscutting code is identified manually rather than automatically.

2.3 Case Study

2.3.1 Setup

In Section 2.1 we argued that the presence of crosscutting concerns in a system could be a cause for code duplication. The failure to properly modularize a crosscutting concern, due to missing language features (e.g. exception handling or aspects) or improper system design, leads to programmers being forced to reuse crosscutting concern code in an *ad hoc* fashion, i.e., by copy-paste-adapt. Over time, this practice may even become part of the development process of an organization, when common code fragments find their way into manuals as conventions or idioms.

The objective of this case study is to evaluate the hypothetical relation between five known (annotated) crosscutting concerns and the duplication of code in a component of a real system written in C. In particular, the case study focuses on the question how well the code of these crosscutting concerns is found by three clone detectors implementing different clone detection techniques.

Clone detectors are designed to find duplicated fragments of code, using a specific clone detection technique (see Section 2.2.1). However, for the purpose of this case study, a clone detector is regarded as a search algorithm for crosscutting concern code. Consequently, well-known performance measures can be used from the field of information retrieval (van Rijsbergen, 1979) (also suggested by Walenstein and Lakhotia (2003)). First, *recall* is used to evaluate how much of the code of a crosscutting concern is found by a clone detector. Second, *precision* gives the ratio of crosscutting concern code to unrelated code found by the clone detector. Finally, *average precision* provides an aggregate measure of the performance

Concern	Line Count (%)	Files (%)	Functions (%)
MEMORY	750 (4.6%)	8 (73%)	43 (27%)
NULL	617 (3.8%)	9 (82%)	67 (42%)
RANGE	387 (2.4%)	7 (64%)	38 (24%)
ERROR	927 (5.7%)	11 (100%)	128 (82%)
TRACING	1501 (9.1%)	10 (91%)	110 (70%)

Table 2.1: Line counts of the five concerns in the *CC* component. The *CC* component consists of 16,406 lines, in 11 files, and 157 functions.

of a clone detector over all recall and precision levels. These measures are defined in detail in Subsection 2.4.5.

2.3.2 Subject System

The software component selected for this case study is called *CC*, and consists of 16,406 lines of C code[1]. It is part of the larger code base (comprising over 10 million lines of code) of ASML. *CC* is responsible for the conversion of data between several data structures and other utilities used by communicating components.

Developers working on *CC* have expressed the feeling that a disproportional amount of effort is spent implementing 'boiler plate' code, i.e., code that is not directly related to the functionality the component is supposed to implement. Instead, much of their time is spent dealing with concerns like Error Handling and Tracing (explained below).

This problem is not limited to just the component we selected; it appears in nearly the entire code base. Since the developers at ASML use an idiomatic approach to implement various crosscutting concerns in applicable components, similar pieces of code are scattered throughout the system. Clearly, significant improvements in code size, quality and comprehensibility are to be expected if such concerns could be handled in a more systematic and controlled way.

Five crosscutting concerns within *CC* were considered in the case study:

- Memory Error Handling; dedicated handling of errors originating from memory management functions.
- NULL-value Checking; checking the values of pointer parameters of a function against NULL (indicative of a failed or missing memory allocation attempt).
- Range Checking; checking whether values of input parameters (other than pointers) are within acceptable ranges.
- Error Handling, which is responsible for roughly three tasks: the initialization of variables that will hold return values of function calls, the conditional execution of code

[1]The line count (LC) used throughout this chapter is defined as the number of lines, excluding completely blank lines.

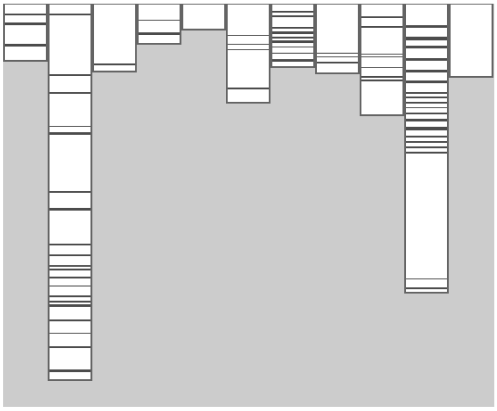

Figure 2.1: Scattering of NULL-value Checking in the *CC* component. Vertical bars represent the .c files of *CC* (header files are not included). Within each vertical bar, horizontal lines of pixels correspond to lines of source code implementing the NULL-value Checking concern.

depending on the occurrence of errors, and finally administration of error occurrences in a data structure.

- Tracing; logging the values of input and output parameters of C functions to facilitate debugging. Each C function is required to perform tracing at its entry and exit points.

All together, these concerns comprise 4182 lines (25.5%) of the 16,406 lines of *CC*. The details are shown in Table 2.1. In tables throughout the chapter, the concerns are referred to by the short-hands MEMORY, NULL, RANGE, ERROR and TRACING, respectively.

For each concern, Table 2.1 also shows the number of files and functions that includes at least one line of the concern. The *CC* component consists of 11 files, containing 157 functions. Fig. 2.1 illustrates the scattered nature of the NULL-value Checking concern by highlighting the code fragments that implement it. The vertical bars represent the files of *CC*, and within each vertical bar, horizontal lines of pixels correspond to lines of source code within the file. Colored lines are part of the NULL-value Checking concern. The other concerns exhibit a similarly scattered distribution.

2.4 Experimental Setup

2.4.1 Annotation

The first phase of the case study consisted of a manual annotation effort performed by one of the authors of the *CC* component. For each of the concerns described in Section 2.3 the author of the component marked those source code lines which belong to each concern. Each line in the component was annotated with at most one mark, and thus, each source code line belongs to at most one of the concerns, or to no concern. See Table 2.1 in Section 2.3 for an overview of the number of lines belonging to each of the concerns.

2.4.2 Selected Clone Detectors

During the second phase of the case study, we performed clone detection on the *CC* component. For this purpose we have used three different tools, which implement different clone detection techniques. First, we used the clone detection tool contained in Project Bauhaus (version 5.1.1) (Project Bauhaus, 2007), a tool set developed at the University of Stuttgart with the goal to support program understanding and reengineering. The clone detector, called 'ccdiml', is an implementation of a variation of Baxter's approach to clone detection (Baxter et al., 1998), and thus falls in the category of AST-based clone detectors. Second, we used CCFinder (version 7.1.1) (Kamiya et al., 2002), a clone detection tool based on tokenized representations of source code. Finally, the PDG-based clone detector developed by Raghavan Komondoor (Komondoor and Horwitz, 2001) was added to the study initially presented in Bruntink et al. (2004a). This clone detector will be referred to as PDG-DUP throughout the chapter.

A distinction between the clone detectors is due to the preprocessing required for Bauhaus' ccdiml and PDG-DUP before clone detection can commence. In effect, both of these tools detect clones in the preprocessed C code, instead of in the un-preprocessed code. In contrast, CCFinder is able to detect clones directly in the un-preprocessed code.

Consequently, Bauhaus' ccdiml and PDG-DUP have to process a larger amount of source code than the 16,406 lines mentioned earlier. The larger amount of source code may have performance implications. In total, the LC of the *CC* component after preprocessing is 40,005. On this particular component, the running time[2] of PDG-DUP is close to 4 hours of processor time, on a 2 GHz AMD Athlon processor. In comparison, Bauhaus' ccdiml requires 3 minutes of processor time to calculate its results. CCFinder needs less than 1 minute to compute its clone classes, running on a slower 1.4 GHz Intel processor. It is not the goal of this study to compare the running times of the clone detectors, but repetition of the study on larger components may require the performance of PDG-DUP to be improved.

The use of preprocessing by Bauhaus' ccdiml and PDG-DUP has no major implications for the case study. Both Bauhaus' ccdiml and PDG-DUP create a mapping for clones detected in the preprocessed code back to the original un-preprocessed code. As a consequence, the clone detection results can be interpreted based on the un-preprocessed code.

2.4.3 Clone Detector Configuration

The selected clone detection tools can be configured prior to execution which affects the types of clones detected.

Bauhaus' ccdiml A prevalent setting of clone detectors is the minimum size of the reported clones. For the purpose of this case study, the minimum size should be set such that the largest, still tractable (with respect to processing time and memory requirements) volume of results is obtained. In case of Bauhaus' ccdiml the minimum clone size was set to be 2 lines. For other (larger) components this value may have to be increased, such that a smaller number of (small) clones is obtained.

[2]Excluding parsing the C code and calculating the PDG.

Furthermore, Bauhaus' ccdiml is capable of detecting three types of clones. First, *exact* clones are simply verbatim copies, although white space and comments are ignored. Second, *parametrized* clones are like exact clones but the leaves of the AST's are ignored during comparison. The result is that variable and type names and literal values are not taken into account. Third, *near* clones are like parametrized clones but allow for insertion and deletion of code. For our experiment we consider only the first two types, i.e., exact and parametrized clones, because near clones cannot be abstracted into clone classes (see Subsection 2.4.4).

The exact command line (without input and output files) to execute ccdiml is given by:

```
ccdiml -all_statements -minlines 2
```

CCFinder For CCFinder, we left all settings at their defaults, except for the minimum length a clone must have in order to be included in the output: a clone must at least be 7 tokens long. A smaller minimum length resulted in more clones than could be handled by CCFinder, causing an abort.

CCFinder was executed using the following command line options:

```
ccfinder C -b 7,1.0
```

where `C` indicates that the tokeniser should expect C code, and `-b 7,1.0` sets the minimum clone size to 7 tokens.

PDG-DUP The size of clones detected by the PDG-based clone detector is expressed as the number of vertices in the PDG that are included in a clone. For the *CC* component 3 vertices is the smallest minimum size that could still be handled; using a minimum size of 2 vertices caused PDG-DUP itself to abort.

Furthermore, PDG-DUP requires the user to set a commonality threshold, which is used to remove clones that are overlapped too much by other clones. Per the recommendation of PDG-DUP's author, the `COMMON` threshold was set to 80%.

2.4.4 Abstracting Clone Detection Results

Some clone detectors produce output consisting of pairs of clones, i.e., they report which pairs of code fragments are similar enough to be called clones. However, for our purpose the pairs of clones are not very interesting. Instead, we investigate the groups of code fragments that are all clones of each other. These groups of code fragments are termed *clone classes* (Kamiya et al., 2002).

More formally, a clone detector defines a relation between code fragments and typically yields the tuples of this relation as its output. Instead of investigating these tuples on their own, we assume this *clone* relation to be an equivalence relation. It is clear that a clone fragment is always either an exact or parametrized clone of itself (reflexivity). Also, if code fragment A is an exact or parametrized clone of code fragment B, then it is clear that B is also an exact or parametrized clone of A (symmetry). Finally, if code fragment A is a clone of B and B is a clone of C, then A is also an exact or parametrized clone of C (transitivity). Subsequently clone classes are comprised of the equivalence classes of the *clone* relation.

The output of CCFinder and PDG-DUP indeed describe equivalence relations between code fragments, and thus obtaining the clone classes is a straightforward task. However, our version of Bauhaus' ccdiml does not produce an equivalence relation. Given the types of clones we include in the study, i.e., either exact or parametrized clones, it is justified to augment the output of ccdiml such that it does constitute an equivalence relation. For this purpose we use grok, a relational algebra program developed by Holt (1998). The equivalence classes were obtained by applying a simple union-find algorithm to the reflexive transitive closure of the clone relation.

2.4.5 Measurements

In the third phase of the case study we performed measurements to test the hypothesis that the clone classes detected by the three clone detectors, i.e., Bauhaus' ccdiml, CCFinder and PDG-DUP, match the annotated crosscutting concern code.

A clone class defines a (non-contiguous) region of source code that is related according to a clone detector. The manually annotated source code is also partitioned in several (non-contiguous) regions, namely those lines of source code that implement the Memory Error Handling, NULL-value Checking, Range Checking, Error Handling and Tracing concerns, and other code. With regard to the goal, an interesting criterion for evaluation is the extent to which the regions defined by the annotations are matched by the regions defined by the clone classes.

Subsection 2.3.1 proposed to use performance measures from the field of information retrieval to evaluate the match between crosscutting concern code and clone classes. We now define those measures in detail:

Definition 1 (Concern) *Each* concern *is represented by a set containing the source lines of the concern, as specified by the annotations.*

Definition 2 (Clone) *A* clone *is defined as a set of source code lines.*

Definition 3 (Clone Class) *A* clone class *is a set consisting of clones. For a clone class CC, we define*

$$lines(CC) = \bigcup_{c \in CC} c.$$

Definition 4 (Clone Class Collection) *A* clone class collection *is a set of clone classes. For a clone class collection D, we also define*

$$lines(D) = \bigcup_{d \in D} lines(d).$$

Definition 5 (Recall and Precision) *Let C be a concern, and D a clone class collection, then we define* recall *and* precision *(van Rijsbergen, 1979) as r and p, respectively:*

$$r(C,D) = \frac{|C \cap lines(D)|}{|C|},$$

$$p(C,D) = \frac{|C \cap lines(D)|}{|lines(D)|},$$

	Clones	Clone Classes	LC
Bauhaus	5,694	617	8,606
CCFinder	8,105	1,101	10,584
PDG-DUP	23,427	4,240	8,292

Table 2.2: Raw clone detection results.

where $|S|$ is the cardinality of a set S. Clearly, $0 \leq r \leq 1$ *and* $0 \leq p \leq 1$.

Originally, some blank lines and lines containing only opening and closing brackets, i.e., '{' and '}', were included in the annotations. Such lines will never be included in the results of the PDG-DUP clone detector, because such lines are not included in PDG-DUP's mapping from PDG vertices to source code. Therefore, such lines had their annotations removed.

Note that this raises an issue of fairness. While PDG-DUP does not include such lines in its results, Bauhaus' ccdiml and CCFinder possibly do. Since those lines no longer belong to a concern (have an annotation), the precision of Bauhaus' ccdiml and CCFinder may be adversely affected. Therefore, blank lines and lines containing only opening and closing brackets were also removed from the clones classes calculated by Bauhaus' ccdiml and CCFinder. The line counts presented in Table 2.1 and Table 2.2 were performed after the removal of these lines. We are not aware of other issues regarding fairness at the syntactical level.

The clone detectors produce a large number of results for *CC*. See Table 2.2 for an overview of these results. Given the raw clone detection results, many clone class collections are possible for each clone detector. For the comparion of the clone detectors we consider appropriate selections of clone classes instead. First, we define the notion of a clone class selection:

Definition 6 (Clone Class Selection) *Given a clone class collection D, a* clone class selection S_k *is a sequence of clone classes* $\langle x_1, x_2, \ldots, x_n \rangle$*, such that* $\{x_1, x_2, \ldots, x_k\} \subseteq D$.

For each clone detector and each concern, we consider a clone class selection of size k, i.e., we select k clone classes from the set of all clone classes found by a clone detector. Given such a selection, a recall-precision graph (van Rijsbergen, 1979) can be plotted to give an overview of the quality of the match between the selected clone classes and a concern. For example, Fig. 2.2 contains the recall-precision graphs for the match between the three clone detectors and the Memory Error Handling concern. A recall-precision graph shows the recall (x-axis) and precision (y-axis) levels obtained for a clone class selection S_k. Each point on a recall-precision graph shows the recall and precision of a clone class collection $\{x_1, x_2, \ldots, x_l\}$ consisting of the first l clone classes of S_k, and $1 \leq l \leq k$.

For each concern, we attempt to find a clone class selection such that the selected clone classes together provide the best possible match with a concern. Intuitively, good matches are provided by clone class selections that result in high recall while maintaining high precision. The *average precision* (*AP*) (van Rijsbergen, 1979) is a commonly used measure that captures this intuition. We define average precision (*AP*) for a clone class selection as follows:

Definition 7 (Average Precision) *Given a clone class selection $S_k = \langle x_1, x_2, \ldots, x_k \rangle$, let S'_l $(1 \leq l \leq k)$ be the clone class collection $\{x_1, x_2, \ldots, x_l\}$ consisting of the first l clone classes in the selection S_k. With C a concern, we now define:*

$$AP(C, S_k) \quad = \quad \sum_{i=1}^{k} p(C, S'_i) \Delta r(C, S'_i),$$

where $\Delta r(C, S'_i)$ is the difference between $r(C, S'_i)$ and $r(C, S'_{i-1})$ $(S'_0 = \emptyset)$.

Each clone detector yields clone classes containing the clones it found in *CC*. To compare the clone detectors at matching crosscutting concern code, for each concern and each clone detector a clone class selection is made such that the average precision for the selection is (approximately) optimal.

Additionally, the average precision measure is helpful for the comparison of the recall-precision graphs obtained for the clone detectors, since average precision maps each recall-precision graph to a single number. Therefore, average precision is used as the primary means of evaluation for the performance of the clone detectors at matching the various concerns.

2.4.6 Calculating Clone Class Selections

Unfortunately, finding a clone class selection S_k that has an optimal average precision is a computationally hard problem. Processing the results has therefore been done using an approximate (greedy) algorithm which iterates k times over all clone classes, and each iteration selects the clone class which adds the most average precision. Previously selected clone classes and lines of a concern are disregarded, such that each iteration considers those lines of a concern that are still remaining.

See Algorithm 1 for a pseudo-code specification of the algorithm. It computes a clone class selection of size k as follows: line 1 initializes `remainder` with all the lines of the specified concern. `remainder` will be used as the work list for the algorithm. The loop initiated in line 6 makes sure that k clone classes will be selected. In line 10 the iteration over all clone classes is started. Lines 11–12 calculate the concern lines included in the clone class under consideration (`hits`), and the other lines that the clone class includes (`misses`). Based on the hits and misses, the algorithm calculates the current precision (`P`), added recall (`dR`), and added average precision (`dAP`) in lines 14–18. Subsequently, the added average precision is then compared to the current maximum, and if greater, the current clone class is marked as the current best choice (lines 20–25). After the iteration over all clone classes is finished, the clone class that adds the most average precision is known. The hits and misses of this clone class are added to the totals (lines 28–29), and the hits are subtracted from the remainder of the concern (line 30). In line 31 the selection is extended with the best clone class, after which the algorithm iterates in order to determine the next best clone class for the ordered selection. Finally, the selected clone class is written to output at line 34.

The algorithm described above possibly calculates clone class selections that are not optimal with respect to their average precision. Small differences between average precision values are therefore required to be interpreted cautiously.

```
SELECT CLONE CLASSES(concern, cloneClasses, k)
(1)   remainder ← concern
(2)   totalHits ← ∅
(3)   totalMisses ← ∅
(4)   selection ← ⟨⟩

(6)   for i = 1 to k
(7)     bestCC ← ∅
(8)     bestdAP ← 0

(10)    for each CC ∈ cloneClasses
(11)      hits ← remainder ∩ CC
(12)      misses ← CC − hits − totalMisses

(14)      tHits ← totalHits ∪ hits
(15)      tMisses ← totalMisses ∪ misses
(16)      P ← ♯(tHits)/(♯(tHits) + ♯(tMisses))
(17)      dR ← ♯(hits)/♯(concern)
(18)      dAP ← P · dR

(20)      if dAP > bestdAP
(21)        bestCC ← CC
(22)        bestdAP ← dAP
(23)        bestHits ← hits
(24)        bestMisses ← misses
(25)      end if
(26)    end for

(28)    totalHits ← totalHits ∪ bestHits
(29)    totalMisses ← totalMisses ∪ bestMisses
(30)    remainder ← remainder − bestHits
(31)    selection ← APPEND(bestCC, selection)
(32)  end for

(34)  OUTPUT(selection)
```

Algorithm 1: Select Clone Classes

2.5 Results

For each of the five crosscutting concerns, we used Algorithm 1 to calculate a clone class selection for each of the clone class collections yielded by the clone detectors. The maximum number (k) of clone classes to select per clone class collection was set to 100. In case of the *CC* component this value is high enough to ensure that the point is reached where selecting

	MEMORY	NULL	RANGE	ERROR	TRACING
Bauhaus	.65	.99	.71	.38	.62
CCFinder	.63	.97	.59	.36	.57
PDG-DUP	.81	.80	.42	.35	.68

Table 2.3: Average Precision values.

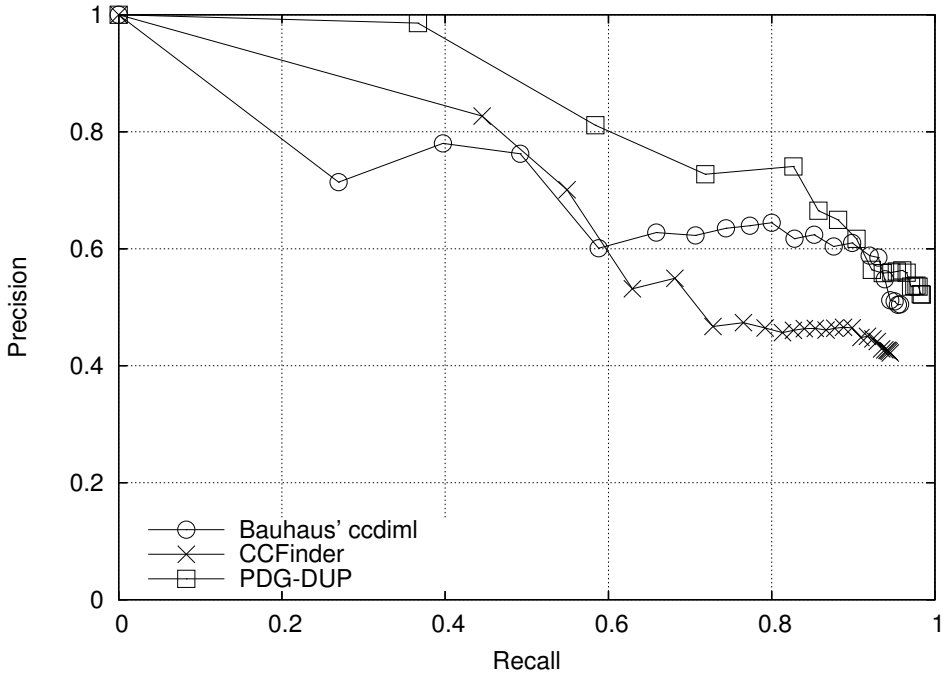

Figure 2.2: Recall-Precision graphs for Memory Error Handling. 21 clone classes shown for Bauhaus' ccdiml, 26 for CCFinder, and 20 for PDG-DUP.

an additional clone class no longer adds average precision. The recall-precision graphs presented for each concern are limited to the range where average precision is added by selecting each new clone class. The average precision levels reached by the clone class selections are presented in Table 2.3.

Figures 2.2, 2.4, 2.6, 2.8, and 2.10 depict the recall-precision graphs of the clone class selections made for the five crosscutting concerns. All graphs are rooted at 0.0 recall and 1.0 precision, which is the case where no clone classes are selected, i.e., S_0. The results for each of the concerns will now be discussed in detail.

2.5.1 Memory Error Handling

Based on the recall-precision graphs, and the resulting average precision values (see Table 2.3) for Memory Error Handling, the PDG-DUP clone detector clearly performs best. The recall-precision graph for PDG-DUP is significantly above those of Bauhaus' ccdiml and CCFinder for almost all *l*. Consequently, the final AP level reached by PDG-DUP is significantly higher as well.

The difference between Bauhaus' ccdiml and CCFinder is not so clear. Bauhaus' ccdiml does better in the high recall area (above .60 recall, in the right half of the figure), while CCFinder does better in the low recall area. Their respective AP values are quite close as well.

Observe in Fig. 2.2 that CCFinder reaches .45 recall using only 1 clone class (the first data point for CCFinder). This particular clone class contains 96 clones which are 6 lines in length. Fig. 2.3 shows an example clone from this class. While the lines marked with 'M' belong to the Memory Error Handling concern, only the lines marked with 'C' are included in the clones. Note that completely blank lines, and lines containing only brackets have no annotation in this example, as was discussed earlier in Subsection 2.4.1. Consequently, those lines were also removed from the clone classes to allow for a fair comparison. In Fig. 2.3 the lines which were removed from the clones are marked with '-'.

As can be seen from the line markers, the CCFinder clone captures the Memory Error Handling fragment only partly, stopping half way the parameter list of a function call. CCFinder allows clones to start and end with little regard to syntactic units. (see Fig. 2.12 for another example) In contrast, Bauhaus' ccdiml does not allow this, due to its AST-based clone detection algorithm. PDG-DUP is bound to conform to language syntax as well, since its PDGs are built on top of ASTs.

```
M C if (r != OK)
  - {
M C   ERXA_LOG(r, 0, ("PLXAmem_malloc failure."));
  -
M C   ERXA_LOG(CCXA_MEMORY_ERR, r,
M C          ("%s: failed to allocated %d bytes.",
M            func_name, toread));

M     r = CCXA_MEMORY_ERR;
    }
```

Figure 2.3: CCFinder clone ('C' lines) covering Memory Error Handling ('M' lines).

The first clone class of CCFinder does not cover Memory Error Handling code exclusively. In Fig. 2.2, note that the precision obtained for the first clone class is roughly .83. Through inspection of the code we found that some of the clones do not cover memory error handling code at all, but code that is similar at the lexical level, yet conceptually different. In other words, some clones capture entirely different functionality. The results show that PDG-DUP is better able to make this distinction, resulting in a higher level of precision.

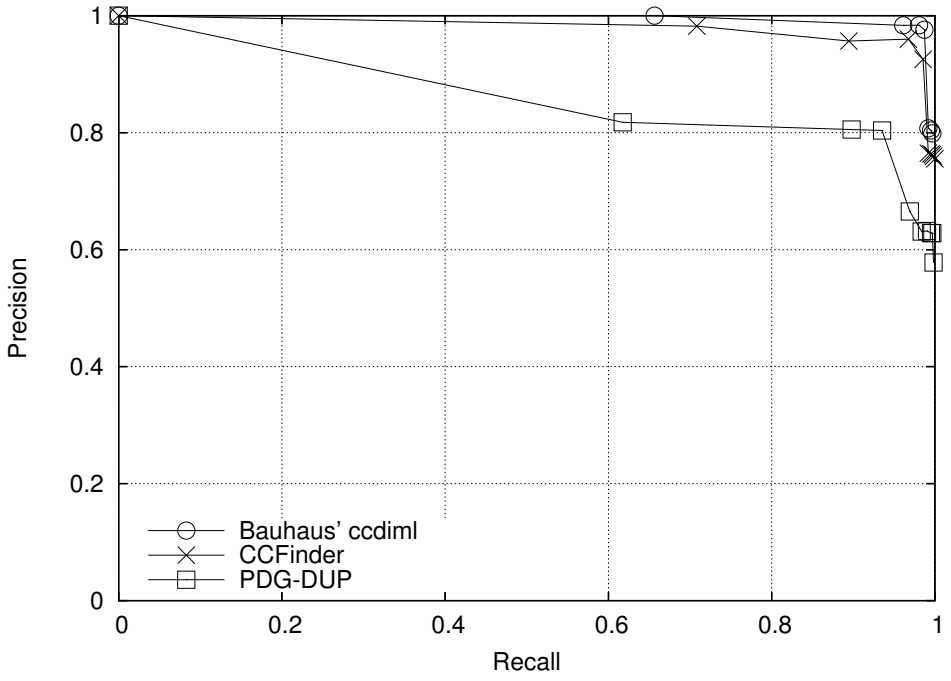

Figure 2.4: Recall-Precision graphs for NULL-value Checking. 8 clone classes shown for Bauhaus' ccdiml, 10 for CCFinder, and 10 for PDG-DUP.

2.5.2 NULL-value Checking

Fig. 2.4 shows the recall-precision graphs for the NULL-value Checking concern. All clone detectors obtain excellent results here, with Bauhaus' ccdiml and CCFinder even approaching 1 (perfect) average precision. PDG-DUP performs significantly worse than both Bauhaus' ccdiml and CCFinder, but still obtains a high average precision of .80.

The clone class that was selected first in the case of Bauhaus' ccdiml captures .66 of the concern, at a precision of 1. This class consists of 77 clones, spanning 405 lines of NULL-value Checking code. In Fig. 2.5 we show an example clone of this clone class. The lines marked with 'N' belong to the NULL-value Checking concern and those marked with 'C' are the lines included in the clone. Again, lines marked with '-' are included in the original clones, but were removed for the comparison.

The coding style adopted at ASML distinguishes three kinds of pointer parameters: input, output and output pointer parameters. All parameters of type pointer are checked against NULL by the NULL-value checking concern. Both input and output parameters are checked in the same way, whereas a check for an output pointer parameter differs slightly from the other checks as it requires an extra dereference. The clone detection results confirm this; the first two selected clone classes of Bauhaus' ccdiml cover input/output and output pointer parameter checks with high precision, respectively.

```
N C if ((r == OK) && (msg == (void *) NULL))
  - {
N C   r = CCXA_PARAMETER_ERR;
  -
N C   ERXA_LOG(r, 0,
N C     ("%s: input parameter '%s' is NULL.",
N C     func_name,
N C     "msg"));
  - }
```

Figure 2.5: Bauhaus' ccdiml clone ('C' lines) covering NULL-value Checking ('N' lines).

The first clone class selected for PDG-DUP results in far lower precision than Bauhaus' ccdiml or CCFinder. It turns out that, although the first clone classes are similar for all clone detectors, all the clones of PDG-DUP are extended with additional lines. For instance, the fragment in Fig. 2.5 is also found by PDG-DUP, but as part of a larger clone. The larger clone includes the declarations (and initializations) of the `r` and `func_name` variables, which are not considered to be part of the NULL-value Checking concern. The purpose of these variables is to facilitate error handling in general, and thus their declarations are part of the Error Handling concern.

PDG-DUP adds these declarations to its clones despite the fact that the declarations are not textually near the other code fragment in Fig. 2.5. This behavior is due to the existence of data dependency edges in the PDG between nodes representing uses of the variables and nodes representing their declarations. Since Bauhaus' ccdiml and CCFinder do not regard data dependencies, they do not extend their clones to include the declarations.

2.5.3 Range Checking

As indicated by the average precision values, Bauhaus' ccdiml (AP .71) outperforms the other clone detectors at finding Range Checking code. Especially PDG-DUP performs badly, resulting in only .42 average precision. CCFinder's performance is in-between Bauhaus' ccdiml and PDG-DUP, with .59.

The recall-precision graph in Fig. 2.6 shows some significant drops in precision. For example, the 4th clone class selected for PDG-DUP (recall .53, precision .28) causes a drop in precision of .36. Similar drops in precision happen for Bauhaus' ccdiml (clone classes 3 and 4) and CCFinder (clone class 5). These clone classes add code fragments which are similar to the one in Fig. 2.7, yet do not represent Range Checking code. In fact, these fragments perform checks on the return values of function calls, while Range Checking is concerned with checking the values of input parameters. The way invalid values are handled is identical in both cases, which explains why these fragments are included in clone classes together with Range Checking code.

2.5.4 Error Handling

Of the concerns we consider, error handling is clearly the worst in terms of the recall-precision graphs (Fig. 2.8) and average precision. Bauhaus' ccdiml has a slight advantage over CCFinder and PDG-DUP in the low recall area.

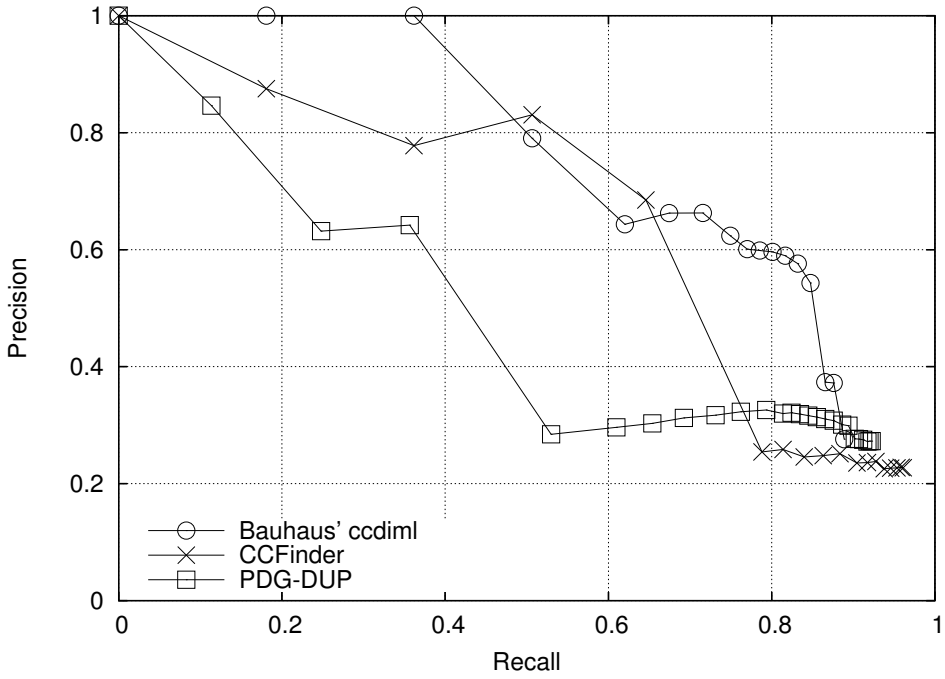

Figure 2.6: Recall-Precision graphs for Range Checking. 17 clone classes shown for Bauhaus' ccdiml, 18 for CCFinder, and 26 for PDG-DUP.

The error handling concern can be partitioned into three sub-concerns: *initialization*, *error linking* and *skipping*. First, the *initialization* sub-concern deals with the initialization of variables used to keep track of return values. Second, *error linking* handles the administration of error occurrences in a data structure. Third, *skipping* is concerned with making sure that specific parts of a function are not executed in case an error has occurred.

Further inspection has shown that the initialization and error linking sub-concerns are included almost entirely by the first and second clone class of Bauhaus' ccdiml, respectively. However, the skipping sub-concern is found very badly, which explains why the error handling concern in general is found badly.

Consider the code fragment in Fig. 2.9, a simple example of code belonging to the skip-

```
    default:
R C   r = CCXA_PARAMETER_ERR;

R C   ERXA_LOG(r, 0,
R C     ("%s: unknown type code encountered (%d).",
R C     func_name,
R C     desc_src->type_code));
```

Figure 2.7: PDG-DUP clone ('C' lines) covering Range Checking ('R' lines).

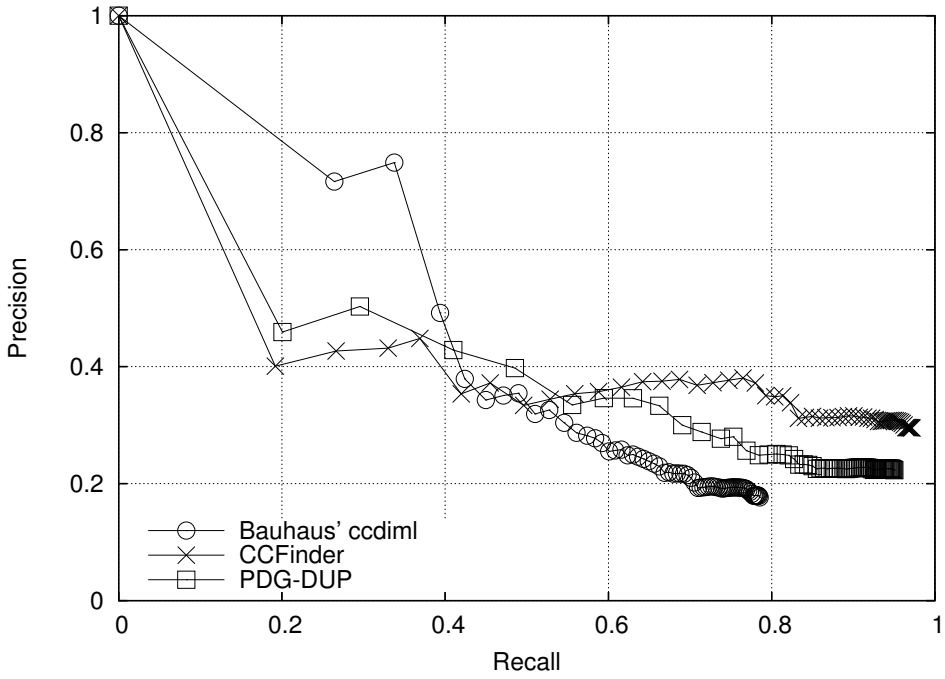

Figure 2.8: Recall-Precision graphs for Error Handling. 69 clone classes shown for Bauhaus' ccdiml, 62 for CCFinder, and 74 for PDG-DUP.

ping sub-concern. The line marked with 'S' belongs to the skipping (sub-)concern. Skipping accounts for 445 lines of code, i.e., 2.7% of the CC component, and furthermore it is present in the entire code base. In the example, the `r` variable is used to hold return values of previous function calls, and the if statement ensures the conditional execution of the remaining code.

```
S if (r == OK)
  {
    r = FD_read(read_fd,
                &msg_hdr,
                (int) sizeof(CCCN_msg_header));
  }
```

Figure 2.9: Instance of the skipping concern ('S' line).

The clone classes yielded by the three different clone detectors do not provide a good match with this concern for the same reason: the pieces of skipping code are simply too small to qualify for clones by themselves due to the limits we set in Section 2.4. Furthermore, the code that appears inside the `if` statements can differ greatly. As a result no clone classes are found that cover just the skipping concern. However, some clone classes cover the skipping sub-concern by accident, i.e., the clones cover a large number of non-concern lines compared to the number of skipping lines covered.

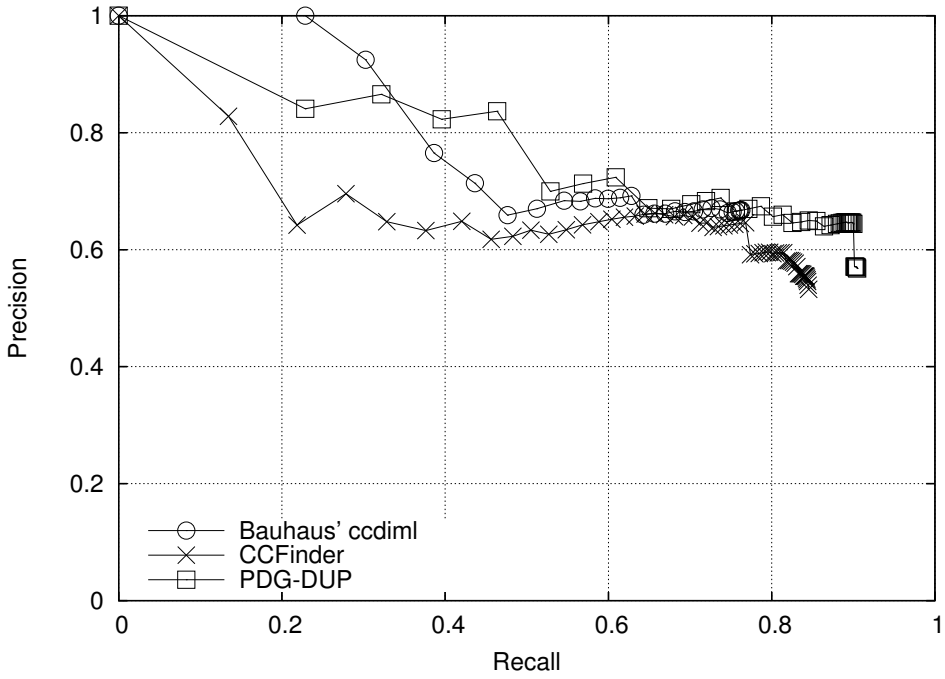

Figure 2.10: Recall-Precision graphs for Tracing. 28 clone classes shown for Bauhaus' ccdiml, 65 for CCFinder, and 41 for PDG-DUP.

2.5.5 Tracing

The average precision values show that PDG-DUP clone classes have the best match with Tracing code, although the difference with Bauhaus' ccdiml is not large. All three clone detectors show notable drops in precision; Bauhaus' ccdiml at clone class 3, CCFinder at clone class 2 and PDG-DUP at clone class 5.

It turns out that these drops in precision are caused by a small number of functions which are almost entirely cloned. The clone classes mentioned before contain clones which are almost as big as entire functions. As a result, the included Tracing code is accompanied by a large number of lines belonging to other concerns.

The first selected clone class for Bauhaus' ccdiml obtains .23 recall at 1.0 precision. An example clone from the first clone class is shown in Fig. 2.11. In total, 71 clones of this class are present in the CC component, spanning 343 lines. The lines belonging to the Tracing concern are marked with 'T', while lines marked with 'C' belong to the example clone.

The clones for PDG-DUP contain non-Tracing lines for the reason also observed for the NULL-value Checking concern. Due to a data dependency between the use of the `func_name` variable in the call to the tracing function and the declaration (and initialization) of `func_name`, the declaration is also included in the clone. Again, this declaration is not part of the Tracing concern, but instead it is included in the Error Handling concern.

```
T C THXAtrace(CC,
T C           THXA_TRACE_INT,
T C           func_name,
T C           "< () = %R",
T C           r);
```

Figure 2.11: Bauhaus' ccdiml clone ('C' lines) covering Tracing ('T' lines).

	MEM	NULL	RAN	ERR	TRA
Bauhaus ∪ CCFinder	.69	.98	.76	.52	.70
Bauhaus ∪ PDG-DUP	.83	.99	.72	.53	.78
CCFinder ∪ PDG-DUP	.79	.97	.62	.38	.73
Bauhaus ∪ CCFinder ∪ PDG-DUP	.77	.98	.79	.54	.81

Table 2.4: Average Precision values for combined clone detection results.

All the code belonging to the Tracing concern is very similar to the example in Fig. 2.11. The `THXAtrace(...)` function is always called, and its first three arguments are typically the same. However, a variable number of arguments can follow the first three. As a consequence, we also find clone classes which consist of calls to `THXAtrace(...)` with 5 arguments, 6 arguments, and so on. In fact, the second clone class selected for Bauhaus' ccdiml contains clones of the `THXAtrace(...)` function call with 6 arguments.

CCFinder does not yield clones of the `THXAtrace(...)` function call with less than 6 arguments, simply because we have limited the minimum size of a clone to 7 tokens (see Section 2.4). However, clone classes including `THXAtrace(...)` function calls with less than 6 arguments do turn up, but those also include a number of non-Tracing lines. A clone belonging to the first clone class selected for CCFinder is shown in Fig. 2.12. It does in fact include many of the same lines as the first clone class selected for Bauhaus' ccdiml, but as can be seen in Fig. 2.12, the clones are extended with non-Tracing lines.

```
T C THXAtrace(CC,
T C           THXA_TRACE_INT,
T C           func_name,
T C           "< () = %R",
T C           r);
  -
  C return r;
  - }
```

Figure 2.12: CCFinder clone ('C' lines) covering Tracing ('T' lines).

2.5.6 Combining Clone Detectors

Table 2.4 contains average precision values obtained for combinations of the three clone detectors. A combination of two (or more) clone detectors consists of the union of their respective clone class collections. The union is then subject to the same selection procedure as for the individual clone detectors, i.e., Algorithm 1. The resulting clone class selections then possibly consist of a mix of clone classes from the combined clone detectors.

Some combinations of clone detectors perform better than the individual clone detectors at matching some concerns. The Range Checking concern is matched better by the combination of Bauhaus' ccdiml and CCFinder than by any individual clone detector (see Tables 2.3 and 2.4). The combination of all clone detectors reaches the highest *AP* for the Range Checking concern. The same result is true for the Error Handling and Tracing concerns. Matching of Error Handling code especially seems to benefit from combining clone detectors.

Clearly, combinations of clone detectors allow for the balancing of the weaknesses and strengths of the individual clone detectors. Further research will be required to provide a qualitative explanation of these results.

It is expected that the combination of all clone detectors performs best at matching a concern. However, this is not the case for the Memory Error Handling and NULL-value Checking concerns. For Memory Error Handling the *AP* for Bauhaus ∪ PDG-DUP is .83 (see Table 2.4) while for Bauhaus ∪ CCFinder ∪ PDG-DUP the obtained *AP* is lower at .77. Similarly for NULL-value Checking, the *AP* for Bauhaus ∪ PDG-DUP is .99 while Bauhaus ∪ CCFinder ∪ PDG-DUP falls short at .98. These small anomalies can be explained by the fact that a non-optimal algorithm is used to calculate the clone class selections.

2.5.7 Summary

Based on the average precision values in Table 2.3, the clone class selections obtained for Bauhaus' ccdiml provide the best match with the Range Checking, NULL-value Checking and Error Handling concerns. However, CCFinder's clone class selections perform almost equally well for NULL-value Checking and Error Handling.

For the remaining concerns, i.e., Tracing and Memory Error Handling, the clone class selections for PDG-DUP perform best. Especially for NULL-value Checking high average precision values are obtained, which even approach the perfect score in case of Bauhaus' ccdiml and CCFinder. The Error Handling concern is matched badly across the board, however.

Combining clone detectors is expected to improve the matching of crosscutting concern code, and our results confirm this expectation (except for some small anomalies due to the algorithm used in the evaluation). Especially the Error Handling and Tracing concerns are matched better when a combination of clone detectors is considered.

2.6 Discussion

2.6.1 Limitations

Clone detection techniques identify the code of our crosscutting concerns because code duplication is the way programmers at ASML reuse (parts) of these concerns. Furthermore, the development process at ASML has a strong idiomatic character. First, it provides strict rules on the implementation of the concerns in question, in the sense that programmers are required to implement the concerns for (almost) every function. Second, programmers are inclined to implement the crosscutting concerns in a similar fashion each time, maybe even making verbatim copies of existing implementations. The programming manual used by each

programmer even provides templates for the implementation of some crosscutting concerns, such as Error Handling, Tracing and NULL-value Checking. As a result, a large number of similar implementations of the crosscutting concerns are scattered across the system. We view an idiomatic nature of the development process (such as the one at ASML) as a major condition to the applicability of our results.

The evaluation performed does not punish the clone detectors for yielding irrelevant clone classes. For example, clone classes that do not contain any lines of a concern are not considered. In general, the clone class selection algorithm only considers those clone classes that can add recall at a given point during the selection. If no such clone classes remain, then the average precision of the clone class selection is fixed. Clone classes which do not add recall do not influence the average precision, since their Δr is 0 by definition. For the evaluation of the case study hypothesis it does not matter that clone detectors also yield irrelevant clone classes: The case study addresses the question which clone detector is capable of providing the closest match between crosscutting concern code and the detected clones, not whether all detected clones match crosscutting concern code.

A limitation that surfaced mainly for the PDG-DUP clone detector is due to the line granularity of the case study. Each source code line can belong to at most one concern, while in some cases we could consider including a line in multiple concerns. An example was discussed in Subsection 2.5.2. In that case the first PDG-DUP clone class for the NULL-value Checking concern consists of clones covering mostly NULL-value Checking code. However, each clone is extended with the declarations of the variables used within the clones. In turn, these declarations are considered to be part of another concern, Error Handling in this case. The main use of these variables lies with the Error Handling concern, yet they are used in an auxiliary fashion by a couple of other concerns, e.g., NULL-value Checking, Tracing and Memory Error Handling. An appropriate solution could be to allow lines to belong to multiple concerns, however this option remains unexplored for now.

2.6.2 Oracle Reliability

A key element of the case study consists of source code annotations produced by a human oracle. The main author of the studied component marked those lines of source code that are to be considered part of one of five (crosscutting) concerns. Several measures were employed to assure the quality of these annotations.

First, the annotated source lines were manually inspected to identify any obvious mistakes made during annotation. As a result, annotations of blank lines and lines containing nothing but opening and closing brackets were removed (see Subsection 2.4.1).

Second, the crosscutting concerns considered in the case study are not specific to the selected component. In fact, all five concerns are present in a large number of other components of the ASML source base. Furthermore, concerns such as error handling, tracing and NULL-value checking are described in detail by the actual programming manuals. As a consequence, the nature of the crosscutting concerns was well-known, allowing the annotations to be checked for non-trivial mistakes.

Third, clone classes that exhibit high added recall fractions, yet cause significant drops in precision were inspected manually and discussed with the component's author. Clone classes such as these relate a number of clones that match part of a concern (explaining the added

recall) to a large number of clones that match other functionality (explaining the drops in precision). Especially when a number of clones matches a part of the concern precisely, and the other clones in the clone class match only other functionality, doubts about the completeness of the annotations could arise. An example of such a clone class was encountered for the Range Checking concern, as discussed in Subsection 2.5.3. The author of the component verified that the non-concern clones were in fact implementing different functionality. No missing annotations were discovered in this way.

2.6.3 Consequences for Aspect Mining

The results presented in this chapter have consequences for the extent to which the studied clone detectors can be used for the purpose of automatic aspect mining. In the case study we determined the (approximately) best match of (the clone classes of) each clone detector with five crosscutting concerns. We were able to determine those matches due to the availability of annotations that map each source line to either a crosscutting concern or to other functionality.

Automatic aspect mining is expected to work without manually obtained annotations. An aspect miner based on a clone detector typically fits the following framework. First, the clone detector calculates clone classes. Second, a clone class selection is made such that best aspect candidates are selected first. The absence of annotations requires that a different approach is used to perform the clone class selection.

It is reasonable to believe that an automatic aspect miner is not going to deliver a better match with crosscutting concerns than a manual annotation effort. In that sense, the clone class selections we obtained based on the annotations can be seen as the best result that can be expected of an automatic clone class selection approach. For example, the results show that the Error Handling concern is matched badly by the clone class selections that we calculated based on the annotations (see 2.5.4). Therefore, an automatic aspect miner that uses one of the three clone detectors studied here cannot be expected to provide a good match with the Error Handling concern. In general, the average precision values in Tables 2.3 and 2.4 give an indication of the suitability of the three clone detectors for the purpose of automatically mining for aspect candidates like the five crosscutting concerns considered in the case study.

An example aspect mining approach using AST-based clone detection is described in Bruntink (2005). Clone classes can be characterized by simple metrics like the number of clones contained in the class, or the number of lines covered by the class, but more complex metrics can be derived as well, such as the distribution of clones over different files. These metrics can subsequently be used to guide the clone class selection process.

2.6.4 Clone Extension

The results show many examples of clone classes that consist of clones which all include some lines of a particular concern, yet also some other lines. One example is the first selected PDG-DUP clone class for the NULL-value Checking concern (see Subsection 2.5.2). In that case, each clone has been extended to include the variable declarations of the variables used in the clone. These variable declarations are not considered to be part of the NULL-value Checking concern, and hence their inclusion results in a lower precision. Another example was discussed for the Tracing concern, where the clones of the first selected clone class for

CCFinder include the `return` statement which always follows the tracing code at the end of a function.

The clone detectors considered in this study are programmed such that (only) maximally large clones are presented to the user. Smaller clones which appear as intermediate results are removed when they are (partly) 'subsumed' by bigger clones, and hence do now show up in the final results. For the purpose of finding duplicated code this is desirable behavior. However, it is clear from the examples above that in case of matching crosscutting concerns, precision can be adversely affected. If the subsumed clone classes had not been discarded from the final results, those classes would have been selected instead of the current ones, resulting in higher precision.

It should be noted that failing to discard subsumed clones altogether will probably result in an intractable number of results. Instead, a better solution is to allow the user to control the extent to which the clone detector discards subsumed clones. PDG-DUP allows the user limited control over this behavior by means of the `COMMON` variable, but to our knowledge Bauhaus' ccdiml and CCFinder have no such controls.

2.7 Conclusions

2.7.1 Contributions

First, our results confirm the belief that some crosscutting concerns are implemented by similar pieces of code, which are scattered throughout a system. Our case study shows that these pieces of code can contribute up to 25% to the code size. Large gains in terms of maintainability and evolvability are thus to be expected from methods supporting the identification and refactoring of these crosscutting concerns.

Second, we have evaluated to what extent the code of five crosscutting concerns is identified by three clone detection techniques. To that end, we manually annotated the code of five specific concerns in an industrial C application, and analyzed the recall, precision and average precision obtained by clone classes yielded by the three clone detection tools. It turns out that the clone classes obtained by Bauhaus' ccdiml can provide the best match with the Range Checking, NULL-value Checking and Error Handling concerns. However, CCFinder's clone classes perform almost equally well for NULL-value Checking and Error Handling. The remaining concerns, i.e., Tracing and Memory Error Handling, can best be matched by clone classes of PDG-DUP.

Finally, we discussed how the results obtained in the case study pose an upper limit to the suitability of using the studied clone detectors for aspect mining purposes. In particular, since Error Handling code is matched badly by all three clone detectors, automatic aspect mining approaches using (one of) these clone detectors cannot be expected to adequately find code belonging to the Error Handling concern. On the other end of the spectrum, code of the NULL-value Checking concern could be found very well by using CCFinder or Bauhaus' ccdiml.

2.7.2 Future Work

The crosscutting concerns we considered in the case study also occur in a range of other ASML components. We will investigate how we can identify the code belonging to these concerns without manual annotations, using the clone classes found in the *CC* component as a starting point. In other words, the *CC* clone classes can be used as seeds for the crosscutting concern identification in other components of the ASML code base.

The code base studied in this chapter is confidential, which hinders exact reproduction of the experiment by other researchers. However, we believe that crosscutting concerns similar to the ones studied here are also present in publicly accessible source code bases. For instance, NULL-value checking, Error Handling and Tracing are common concerns for any reasonably large C system. Therefore, it would be worthwhile and feasible to reproduce the experiment on a publicly accessible source base.

Clone classes can be characterized by simple metrics like the number of clones contained in a clone class, or the number of lines covered by the class, but more complex metrics can be derived as well, such as the distribution of clones over different files. Metrics such as these could be used to study the nature of clone classes that capture crosscutting concerns (and those that do not), given that these clone classes are known. The case study presented in this chapter shows one way of identifying such clone classes, i.e., using manually obtained annotations. Consequently, relationships between clone class metrics and recall and precisions levels could be discovered based on the results of our case study. Such relations could then be tested on other systems, including those written in other languages.

Aspect mining techniques based on clone detection can possibly benefit from knowing which clone class characteristics relate to crosscutting concerns. For instance, in Bruntink (2005) we discuss how a number of clone class metrics can be used to filter clone detection results.

Finally, we are working toward the elimination of the NULL-value Checking and Tracing concerns from the original source code (see Chapter 3). The implementations of these concerns are replaced by domain-specific solutions, which subsequently generate aspect-oriented code. An interesting issue with respect to clone detection is the suitability of the detected clones for such (semi-automatic) refactorings. For instance, clones that do not consist of complete syntactical units are likely unsuitable for this purpose. Furthermore, clones that have context dependencies (data or control) require additional effort to be successfully extracted. The PDG-DUP clone detector appears to be most suitable for such refactoring activities, since it both respects syntactic integrity and includes context dependencies in its clones.

Chapter 3

Isolating Idiomatic Crosscutting Concerns*

This chapter reports on our experience in automatically renovating the crosscutting concerns of the ASML C system using aspect-oriented progrmaming. We present a systematic approach for isolating crosscutting concerns, and illustrate this approach by zooming in on one particular crosscutting concern: parameter checking. Additionally, we compare the legacy solution to the aspect-oriented solution, and discuss advantages as well as disadvantages of both in terms of selected quality attributes. Our results show that automated migration is feasible, and that adopting an aspect-oriented approach can lead to significant improvements in source code quality, if carefully designed and managed.

3.1 Introduction

Aspect-oriented software development (AOSD) (Kiczales et al., 1997) aims at improving the modularity of software systems, by capturing inherently scattered functionality (often called *crosscutting concerns*) in a well-modularised way. In order to achieve this, aspect-oriented programming languages add an extra abstraction mechanism, called an *aspect*, on top of existing modularisation mechanisms such as functions, classes and methods.

In the absence of aspects, crosscutting concerns are implemented explicitly using more primitive means, such as naming conventions and coding idioms (an approach we refer to as the *idioms-based approach* throughout this chapter). The primary advantage of such techniques is that they are lightweight, i.e., they do not require special-purpose tools or languages, are easy to use, and allow developers to recognise the concerns in the code readily. The downsides however are that these techniques require a lot of discipline, are particularly prone to errors, make concern code evolution time consuming and often lead to code duplication (see Chapter 2).

*This chapter was published in the Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005) (Bruntink et al., 2005a). It is co-authored by Arie van Deursen and Tom Tourwé.

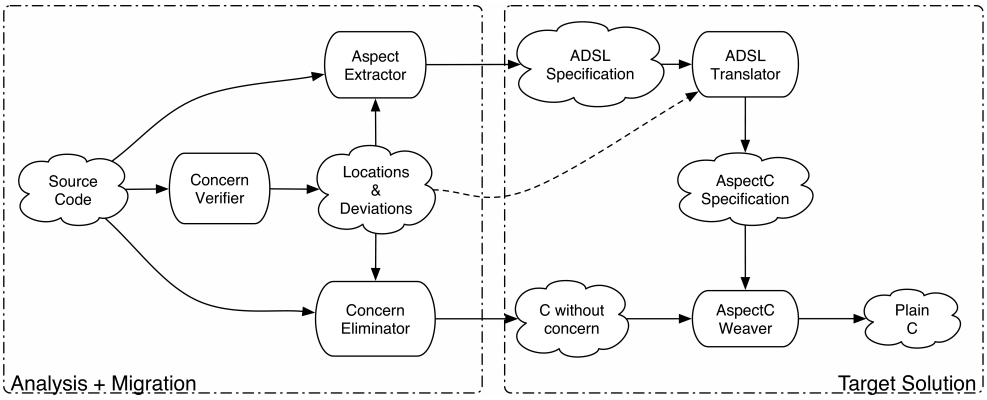

Figure 3.1: An overview of our isolation approach

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

This chapter is laid out as follows. The next section presents our approach to the problem of isolating crosscutting concerns, together with three adoption strategies. This approach is illustrated by looking at one particular concern, parameter checking, explained in Section 3.3. Section 3.4 presents the domain-specific aspect language we implemented for the parameter checking concern, and Section 3.5 discusses the migration of the idioms-based approach to the aspect-oriented approach. Section 3.6 then shows the results of applying our approach to our case study, allowing us to evaluate the approach and to compare the resulting AOSD solution to the current solution in Section 3.7. Finally, Section 3.8 discusses related work, and Section 3.9 presents our conclusions and future work.

3.2 Approach

3.2.1 Overview

The systematic approach we propose for isolating crosscutting concerns is illustrated in Figure 3.1.

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

specific language (ADSL). The ADSL code and the pure C are merged together automatically by means of a code weaver. In order to do this, the ADSL specification is translated to the general purpose AspectC language, which then allows us to reuse an existing AspectC weaver. Note that directly expressing the crosscutting concerns in AspectC is not always feasible or desirable, as we will see in Sections 3.4 and 3.6.

The left-hand side of Figure 3.1 describes the steps that make it possible to migrate existing C components to the ADSL solution.

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

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

In order to minimise the risk of introducing subtle errors in the course of the migration, the migrated code obtained through the ADSL should be as similar as possible to the original code. This is suggested by the dashed arrow in Figure 3.1, which indicates that locations found for idioms can be reused in the translator in order to put concern code back at the original position. In Section 3.5 we will return to this issue of *conservative migration*.

3.2.2 Adoption Strategies

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

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

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

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

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

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

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

3.3 Parameter Checking

3.3.1 Industrial Context

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

The entire software system consists of more than 15 million lines of C code. Our case study, however, is based on five subsystems, comprising about 160,000 lines of code.

3.3.2 The Parameter Checking Concern

The requirement for the parameter checking concern is that each parameter that has type pointer and is defined by a public (i.e., not declared `static`) function should be checked before it is dereferenced. The purpose of such checks it to improve the reliability and error reporting of the software system.

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

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

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

3.3.3 Coding Idiom Used

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

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

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

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

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

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

3.4 An ADSL for the Parameter Checking Concern

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

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

3.4.1 Specification

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

As an example, consider the (partial) specification of the parameter checking aspect for one of the considered components, as depicted in Figure 3.2. It states that the parameters `CC_queue *queue` and `void **queue_data` of the `CC_queue_peek_front` function 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.

[1] PCSL is most easily pronounced pixel.

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

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

Besides the signature specification, the developer can specify *advice code*, i.e., the code that will perform the actual check. Since this code can differ for the different kinds of parameters, we allow advice code for input, output and output pointer parameters to be specified separately. Although in this chapter we do not need it, PCSL also has provisions to express advice code for deviations.

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

```
1  component CC {
2       CC_queue_peek_front(output CC_queue *queue,
3                           output output-pointer void **queue_data);
4       CC_queue_empty(input CC_queue *queue, output bool *empty);
5       CC_queue_init(output CC_queue *queue, deviation void *queue_data);
6       ...
7       input advice {
8          if(thisParameter.name == (thisParameter.type) NULL) {
9             r = CC_PARAMETER_ERR;
10            CC_LOG(r,0,("%s: Input parameter %s error (NULL)",
11                   thisParameter.function.name, thisParameter.name));
12         }
13      }
14      output advice {
15         if(thisParameter.name == (thisParameter.type) NULL) {
16            r = CC_PARAMETER_ERR;
17            CC_LOG(r,0,("%s: Output parameter %s error (NULL)",
18                   thisParameter.function.name, thisParameter.name));
19         }
20      }
21      output-pointer advice {
22         if(*thisParameter.name != (thisParameter.type*) NULL) {
23            r = CC_PARAMETER_ERR;
24            CC_LOG(r,0,("%s: Output Pointer parameter %s error",
25                   thisParameter.function.name, thisParameter.name));
26         }
27      }
28 }
```

Figure 3.2: PCSL specification of the parameter checking concern

3.4.2 Translation to AspectC

PCSL code is automatically transformed into AspectC code, which in turn is woven with the C code containing the implementation of the primary functionality (as seen in Figure 3.1).

The AspectC weaver we use is a stripped-down variant of the AspectC language defined by Coady et al. (2001). It has only one kind of joinpoint, function execution, and allows us to specify around advice only. Of course, before and after advice can be simulated easily using such around advice. Figure 3.3 contains an example that shows how the `advice on` keyword is used to specify advice code for a particular function.

The translation is implemented in ASF+SDF (van den Brand et al., 2001), a general program transformation tool that includes a C grammar, and proceeds as follows. For each parameter that does not have the `deviation` annotation, the translator looks up the parameter's kind, retrieves the corresponding advice code, and expands that code into the actual check that should be performed.

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

An illustration of the translation of the specification of Figure 3.2 is given in Figure 3.3. An input and an output parameter check are added to the `CC_queue_empty` function for its `queue` and `empty` parameters, respectively.

```
1  int advice on (CC_queue_empty) {
2       int r = OK;
3       if(queue == (CC_queue *) NULL) {
4          r = CC_PARAMETER_ERR;
5          CC_LOG(r,0,("%s: Input parameter %s error (NULL)",
6                      "CC_queue_empty", "queue"));
7       }
8       if(empty == (bool *) NULL) {
9          r = CC_PARAMETER_ERR;
10         CC_LOG(r,0,("%s: Output parameter %s error (NULL)",
11                     "CC_queue_empty", "empty"));
12      }
13      if (r == OK)
14         r = proceed();
15      return r;
16 }
```

Figure 3.3: AspectC code generated for the PCSL specification

3.5 Migration Support

The aspect solution as described in the previous section can be used when new components are built, as shown in the right part of Figure 3.1. However, the majority of the software development activities at ASML (and, in fact, at most companies) is not devoted to greenfield

engineering, but to adjusting existing components. In order to achieve the same benefits for such existing components, we describe how these can be migrated to an ADSL solution. In particular, we discuss how concern code can be recognized in the original implementation, how an aspect can be created from it, and how the original concern code can be removed from the C code.

3.5.1 Concern Verification

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

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

The verifier for the parameter checking idiom has been developed as a plugin for the *CodeSurfer* source code analysis and navigation tool (CodeSurfer, 2007). CodeSurfer can construct program dependence graphs for systems written in C, and provides a programmable interface to navigate through such graphs.

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

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

3.5.2 Aspect Extraction

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

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

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

3.5.3 Concern Elimination

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

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

In order to solve this issue, we implemented a parameter checking eliminator using ASF+SDF (van den Brand et al., 2001). Transformations recognise parameter checking code that obeys the coding idiom explained in Section 3.3, and subsequently remove such code.

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

3.5.4 Conservative Translation

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

Conservative translation is based on information obtained from the verifier (as explained above and indicated in Figure 3.1). Four different situations occur for a parameter p of a function f:

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

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

3.6 Case Studies

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

	parameters to check	deviations detected	unintended deviations	intended deviations
CC3 (3 kLoC)	32	8	0	8
CC10 (19 kLoC)	238	65	58	7
CC8 (12 kLoC)	218 of 723	23	16	7
CC8 (98 kLoC)	505 of 723	53	41	12
CC2 (14.5 kLoC)	190	31	24	7
CC1 (15 kLoC)	67	5	4	1

Table 3.1: Parameter checking results

3.6.1 Intended and Unintended Deviations

An overview of the parameter checking verifier results is provided in Table 3.1. The top half of the table lists systems for which the results have been discussed with system developers, and for which we have accurate figures on the intended versus unintended deviations.

As can be seen, our verifier reports that 32 of the parameters of the CC3 component must be checked (recall that only pointers need to be checked). 8 deviations are reported, all of which are considered to be intended deviations. For the CC10 component, 238 parameters need to be checked, 65 of which are reported as deviations, and manual inspection eliminated 7 intended deviations. Due to timing constraints, results for the CC8 component are only confirmed for a 12 kLoC subset. 23 deviations are reported, 16 of which are unintended.

The bottom half of Table 3.1 lists systems to which we have applied our tools, but for which we do not have confirmed figures on the number of (un)intended deviations. The figures listed are estimates, obtained by extrapolating the results from the top half of the Table, which indicate that 74 out of 96 (77%) deviations are considered unintended. Thus, we predict that 69 unintended deviations will be present in the remainder of the CC8 component and in the CC2 and CC1 components.

3.6.2 Coding Idiom Conformance

The verifier recovers all parameter checks from the code, which allows us to assess how consistent the various developers have implemented the coding idiom explained in Section 3.3.3.

Most components implement the checks in the same way, but each component logs different strings, even for the same parameter error. The CC3 component logs two different strings, for instance, and the CC10 component logs 30 different strings. The CC8 component and the CC2 component log 15 and 5 strings, respectively, whereas the CC1 component does not log any string. This is due to the fact that it does not contain any parameter checks. The component defines few public functions and hence requires few checks, which do not seem to be implemented.

The reason CC10 uses many more strings than the other components is due to the fact that the logged errors in CC10 are specific to the kind of parameter, whereas other components use generic strings.

	CC3	CC10	CC8	CC2	CC1
Original C code	56	961	456	133	0
PCSL code	46	132	787	166	75
AspectC code	122	1200	1214	272	223

Table 3.2: Lines of code figures for various parameter checking representations

3.6.3 Code Size

Table 3.2 shows the difference measured in lines of concern code[3] between the idioms-based, PCSL and AspectC approaches. Surprisingly, we found that using an aspect-oriented approach, based on a domain-specific or on a general-purpose aspect language, does not necessarily reduce the code size of the components. This contrasts sharply with an often (informally) claimed benefit of aspect-oriented software development.

For the PCSL approach, we see that the code size of the CC10 component is reduced significantly, but that this is not the case for the other components. The CC8 specification has 72% more lines of code than the original C code. The reason is that the number of parameter checks in CC8 is relatively low compared to the number of parameters that need to be checked, but that in PCSL all (public) function signatures should be annotated. This situation is even more apparent for the CC1 component, which does not implement any checks, whereas the PCSL specification consists of 75 lines of code. We will further discuss this issue in Section 3.7.3.

With respect to the AspectC solution, we observe that all aspects, except the CC1 aspect, require more lines of code than the idioms-based approach. In other words, it requires an extra amount of code to implement, and contains just as much duplication as the idioms-based approach (as shown in Figure 3.3). Admittedly, our AspectC code is generated and does not try to factor out commonalities using generic pointcuts. This is very difficult to achieve with generic languages such as AspectJ or AspectC, because they currently lack the necessary features and flexibility for implementing concerns such as parameter checking (Adams and Tourwé, 2005).

3.7 Evaluation

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

3.7.1 Scalability

The idioms-based approach does not scale.

The results of running the parameter checking verifier show that most of the reported deviations are unintended deviations. These results prove that our verifier is reliable and

[3] Including whitespaces.

worthwhile to consider in a code reviewing activity. Additionally, this shows that the idioms-based approach is error-prone and does not scale, even for simple concerns, and that a more rigorous treatment for parameter checking is needed.

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

- The size of the component. The smallest component (CC3) contains no deviations, whereas the largest component (CC8) contains the most deviations;
- The age of the component and the amount of evolution it has undergone. The CC3 component was redeveloped completely recently, whereas CC8 is already quite old and has undergone many changes;
- The number of programmers working on the component. Components developed by a small team of developers do not exhibit many unintended deviations, whereas components developed by many different developers show many more deviations.
- The cost of adding a check manually compared to the benefits it may provide to the developer. The potential benefits of the check are not experienced by the developers themselves, but by potential clients of the component who would be helped with a parameter warning upon improper use of the function.

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

3.7.2 Code Quality

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

Minimising Code Duplication

In Chapter 2 we evaluated the amount of code duplication in a number of crosscutting concerns in the CC10 component, among which the parameter checking concern. The results confirm the common belief that the idioms-based approach leads to a large amount of duplication. The specific reason for the duplication is that, due to the crosscutting nature of the code, reuse of that code is not possible in ordinary programming languages, since it does not fit in a module.

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

Note that we specifically devised PCSL with maximal reuse of advice code in mind. Using a current-day general-purpose aspect language would make it much harder to reuse the code and avoid duplication, however. This is discussed in Bruntink et al. (2004b) and shows that simply using AOSD techniques will not necessary reduce (crosscutting) code duplication.

Improving Uniformity

The results in Section 3.6.2 show that the coding idiom for parameter checking is not strictly adhered to. Although uniform checks are not that important, uniform error strings are. Those strings are used by automated tools that reason about the logged errors in order to identify and correct the primary cause of a particular error. The idioms-based approach clearly aggravates this task.

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

Improving Understandability

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

Reducing Scattering and Tangling

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

3.7.3 Maintainability

An AOSD approach introduces additional maintainability risks.

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

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

A complementary solution came up when talking to ASML developers. They suggested that the function signature annotation and advice code specification can best be separated. The rationale is that the advice code almost never changes, but the annotations will change

more frequently. Additionally, they suggested to allow annotations only for parameters that do not need to be checked. The majority of the functions behaves normally with respect to the idiom, and does not need annotation, since we can infer parameter kinds automatically from the source code. As such, the required lines of code for the PCSL specification will be reduced significantly.

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

3.7.4 Change Management

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

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

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

3.8 Related Work

Our parameter checking verifier resembles tools that verify the quality of the source code. A number of tools for this purpose have been developed over the years (Johnson, 1977; Paul and Prakash, 1994). Most of them are only able to detect basic coding errors, such as using = instead of ==, and are incapable of enforcing domain-specific coding rules. More advanced tools exist (van Emden and Moonen, 2002; Kataoka et al., 2001; Marinescu, 2002; Tourwé and Mens, 2003), but these are restricted to detecting higher-level design flaws in object-oriented code. Tools which are capable of checking custom (domain-specific) coding rules are described by Eichberg et al. (2004) and Engler et al. (2000).

There is also some similarity with tools from the area of *plan recognition* (Wills, 1992; van Deursen et al., 1997). Such tools are parameterised with a library of "program plans", typical ways of solving known programming problems, which bear some resemblance with

our idioms. Plans are typically described by data or control dependencies, as done, for example in van Deursen et al. (1997) to characterise leap year computations in Cobol code. The recogniser can then search for plan instances in a code base. In our case we decided to characterise the idioms in Scheme as a CodeSurfer plugin, which provided the necessary flexibility and was readily available.

The work described in this chapter has some similarities with work done by Coady et al. (2001), who describe how they identified *prefetching code* in the FreeBSD OS kernel, and propose a new solution in terms of AspectC. Their work does not focus on a general approach for isolating crosscutting concerns, since they restructured the code manually in an ad-hoc way.

A number of other studies have investigated the applicability of aspect-oriented techniques to various (domain specific) crosscutting concerns. Murphy et al. (2001) proposes guidelines for preparing the code for isolating concerns and performing the necessary restructurings. Lippert and Videira Lopes (2000) targets exception detection and handling code in a large Java framework. Both works discuss advantages of using AOSD, such as reduced code duplication and improved cohesion, and discuss some particular limitations of using AspectJ. In Lippert and Videira Lopes (2000), the aspect solution does reduce the code size, contrary to our findings.

An approach to refactoring which specifically deals with tangling is presented by Ettinger and Verbaere (2004). Their work shows how slicing techniques can help automate restructuring of tangled (Java) code, and could be a good candidate to include in our approach, when we are dealing with more complex concerns. Hanenberg et al. (2003) provides a more general discussion of both refactoring in the presence of aspects, and refactoring of object-oriented systems toward aspect-oriented systems.

3.9 Concluding Remarks

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

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

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

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

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

- We are in the process of applying this approach to a larger number of components within ASML;
- Parameter checking is a concern that is interesting outside ASML as well. Our approach is mostly generally applicable. The only ASML-specific elements are localised in (1) the places in the verifier where existing checks are recognised; and (2) the specific advice specified in the ADSL. Both can be easily changed, making the approach applicable to, for example, open source systems in which parameter checking advice should consist of C `assert` statements.
- The next concern on our list is *exception handling*. This concern is significantly more complicated than parameter checking (its implementation being much more tangled). Chapter 5 describes SMELL, a concern verifier for ASML exception handling. SMELL is used later in Chapter 7 to facilitate the migration approach outlined in this chapter. First, SMELL is used to discover idiom violations (faults), which are subsequently fixed. Second, the encoding of the exception handling idiom within SMELL is exploited to guide manual reengineering steps.

Chapter 4

Linking Analysis and Transformation Tools with Source-based Mappings*

This chapter discusses an approach to linking separate analysis and transformation tools, such that analysis results can be used to guide transformations. The approach consists of two phases. First, the analysis tool maps its results to relevant locations in the source code. Second, a mapping in the reverse direction is performed: the analysis results expressed as source positions and data are mapped to the abstractions used in the transformation tool. We discuss a prototype implementation of this approach in detail, and present the results of two applications within the context of the ASML C system.

4.1 Introduction

There exists a vast collection of source code analysis and transformation tools. Most of these tools specialize in either analysis or transformation, and rarely a tool is suitable for both tasks. Ironically, most non-trivial transformation tasks require deep analysis. Migrating legacy software to recent technology, such as AOP (See Chapter 3), is but one example.

Combining tools is the obvious solution to this functionality schism. However, tool combination introduces the issue of tool interoperability, and despite the ample attention it has received from the research community, it still remains a largely open problem (Cordy and Vinju, 2006). Previous work in this area has focused on solving low-level compatibility issues, resulting in many proposals for generic data formats, and communication protocols (Holt et al., 2000b; van den Brand et al., 2000a; Bergstra and Klint, 1998; Ebert et al., 2001; Jin and Cordy, 2005). These technologies have proved to be useful in several successful tool col-

*This chapter was published in the Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006) (Bruntink, 2006).

laborations such as the ASF+SDF Meta-Environment (van den Brand et al., 2001). Another key feature of tools like the ASF+SDF Meta-Environment is that they operate on an abstract representation (i.e., ASTs) that is shared by all of their components. A common term for such an abstract representation is *schema*, which we will use throughout this chapter.

A remaining challenge consists of coping with differences between schemas (of the source code) employed by the various tools. For instance, an analysis tool such as Grammatech's CodeSurfer (Anderson et al., 2003) revolves around program dependence graphs (PDGs). In contrast, transformation tools such as ASF+SDF (van den Brand et al., 2001) operate primairily on abstract syntax trees (ASTs). To leverage analysis results expressed in the PDG domain (i.e., CodeSurfer), it is first required to map the analysis results to the AST domain (i.e., ASF+SDF). Clearly, creating such a mapping (or *bridge* (Cordy and Vinju, 2006)) is a non-trivial task, requiring deep understanding of the schemas used at both ends. Even if both tools are targeted at the same language, the use of different grammars, language dialects, and source correspondences complicate this task enormously, especially since it is often hard to change those features of a tool.

In this chapter we discuss an approach to create *source-based mappings* between tools using different schemas. A source-based mapping consists of pairs of a source code area and facts relevant at that area. The perspective taken for this discussion consists of two tools working together on the same source code; one tool performs the analysis required for the transformations performed by the other tool. We will define when a source-based mapping is *strict* and *safe*, given the relevant abstractions in the analysis and transformation tools, and a body of source code. A strictly safe source-based mapping guarantees that analysis results are mapped to the desired abstraction in the transformation tool. Furthermore, we show how a source-based mapping compares to mappings created using higher-level schemas.

The chapter is organized as follows. Section 4.2 presents source-based mappings in detail. The idea of source-based mappings has been implemented as a framework called SCATR, which is described in Section 4.3. SCATR has been applied to a number of cases, which we report upon in Section 4.4. In Section 4.5 we compare source-based mappings with schema-based mappings, and propose a way to automatically check the safeness and strictness properties. Section 4.6 discusses related work.

4.2 Source-based Mappings

Figure 4.1 shows the general idea of source-based mappings. The left hand side is the domain of an analysis tool, while the transformation domain resides on the right hand side. Both operate on the same body of source code. As is suggested by the figure, the tools work with different schemas.

The dashed circles and arrows show how a source-based mapping operates. First, an element of the schema used by the analysis tool is selected. We will refer to such an element as an *instance* of the schema used by the tool. An instance can be of a certain *type*, for example a PDG or AST node.

Subsequently, the selected instance is mapped to an appropriate area of the source code, along with the facts of interest associated with the instance. Next, the process is reversed in the transformation domain. The source code area obtained in the previous step is used to map

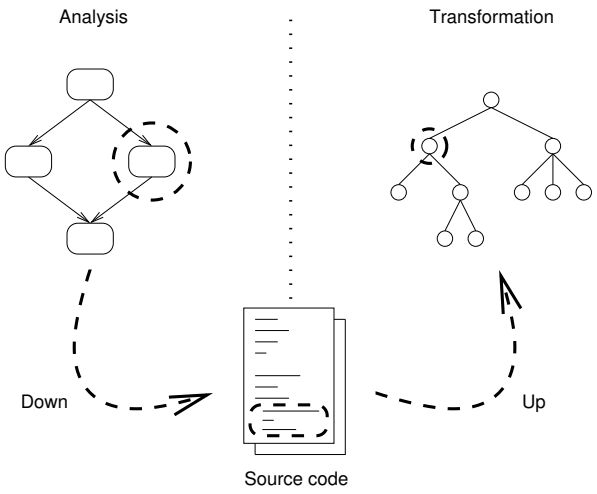

Figure 4.1: Source-based mappings.

the facts of interest to an appropriate instance of the schema used by the transformation tool.

If analysis and transformation operate on the same schema, and their mapping to and from the source code is identical, it is clear that a source-based mapping will allow facts about arbitrary instances to be exchanged. In practice, this situation is a rare exception, unless analysis and transformation are performed by the same tool. We are interested in the case where analysis and transformation are done by different tools, and possibly using different schemas, and therefore with a different source code correspondence.

A source-based mapping can be split into two functions, *down* and *up*. *Down* represents the arrow on the left hand side of Figure 4.1, while *up* represents the right hand side arrow. Given that we have fixed types S and T of instances in the analysis and transformation tools respectively, *down* and *up* have the following signatures:

$$\begin{aligned} down(S) &\longmapsto Area, \\ up(Area) &\longmapsto T, \end{aligned}$$

where *Area* refers to a source code area, e.g., a start line and column, paired with an end line and column. A source-based mapping then consists of the composition $up(down(s))$, where s is an instance of type S, and the result is an instance of type T.

Note that with the current definition, $down(s)$ yields a single area corresponding to s. However, elements of some representations may not be mappable to a single area of source code. For example, a usage dependency (e.g., an edge in a call graph) between modules may map to any of the call sites or variable accesses giving rise to the dependency. In these cases it may be desirable to allow $down(s)$ to yield a list of areas, and apply up to each area separately. We plan to investigate this matter in the future.

Consider the example source code in Figure 4.2, and its abridged AST representation. The Stat nodes in the AST are annotated with the line numbers that they correspond to. Suppose

```
1  for (i = 0; i < length; i++)
2  {
3    if (array[i] > max)
4    {
5      max = array[i];
6    }
7  }
```

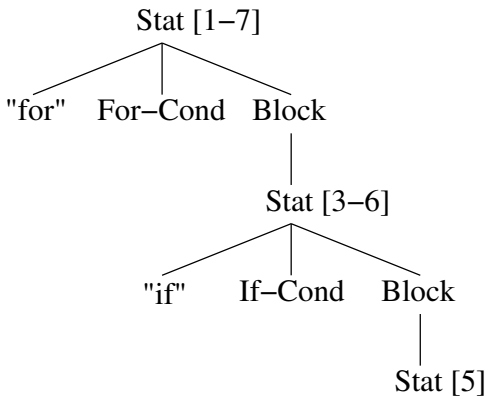

Figure 4.2: Source code example.

analysis results defined for PDG nodes are to be mapped to Stat nodes in the AST (i.e., S is fixed to PDG nodes, and T is fixed to Stat nodes). Let s be the PDG node representing the condition of the for loop, and let $down(s)$ yield the area a which spans line 1. Now $up(a)$ should yield the Stat node corresponding to the for loop, since a is included (only) in the area ([1–7]) of that node.

There may be some situations in which a source-based mapping is more problematic. First, the area yielded by $down(s)$ might correspond to more than one instance of type T. A common cause of this problem is recursion in grammars. For instance, expressions are typically defined recursively, and as a result, more than one expression may be defined at a source code area. In Figure 4.2, the statement `max = array[i];` at line 5 is nested within both the for and if statements. Consider that s is a PDG node representing the statement at line 5, and $down(s)$ is the area spanning line 5. Now there are 3 Stat nodes t to which $up(down(s))$ could map, because line 5 is included within the area of any of the for, if, and assignment statements.

A partial solution to this problem would be to have more fine-grained source code correspondence within both analysis and transformation tools. For example, if the source correspondence of the AST node for the if statement in Figure 4.2 would consist of the lines 3, 4, and 6, instead of the entire range 3–6, then the if statement does not need to be considered as a target of the up of line 5.

Note that the $down$ function also needs to accommodate the finer-grained source correspondence. Since the up of line 5 no longer yields the AST node of the if statement, $down(s)$ has to output any of lines 3, 4 or 6 if s is a PDG node representing the if statement. Clearly,

whether a finer-grained source correspondence can be used on one end is dependent on the other end. *Down* and *up* have to be implemented in a compatible way, and therefore the implementor has to be aware of the source correspondences of both S and T.

Tools with an inaccurate source correspondence are therefore particularly problematic. However, some means are needed to cope with these inaccuracies in practice, since source correspondence within existing tools cannot always be easily improved. If the source correspondence at either end is not accurate enough to obtain a unique target for $up(down(s))$, a strategy has to be defined which implements a choice. For example, up could select the instance that is most specific to the area generated by $down(s)$. In Figure 4.2, up would then map line 5 to the assignment statement without a problem. Our SCATR framework (see Section 4.3) implements this strategy.

Another problem that may occur due to an inaccurate source correspondence is that the source code area $down(s)$ can not be associated with any instance of type T, because no such instance is defined at $down(s)$. This problem may also be caused by a bad choice of instance types, e.g., trying to map PDG nodes representing assignments to AST nodes representing if statements. The implementor of a source-based mapping has to make sure the instance types are chosen such that this problem cannot occur.

If the second problem (i.e., no instances of type T at $down(s)$) is not present, or in other words, if $up(down(s))$ is defined for all s from the domain, we call a source-based mapping *safe*. Furthermore, a source-based mapping that yields exactly one t for each s is called a *strictly safe* mapping. Both properties can be checked to hold given the instance types S and T, implementations of $down$ and up, and a body of source code. We discuss this further in Section 4.5.

4.3 SCATR

SCATR (short for Scaffolding And TRansformation, and pronounced as 'scatter') is a framework supporting the use of source-based mappings in the context of linking analysis and transformation tools. Scaffolding is a technique proposed by Sellink and Verhoef (2000), which constitutes the foundation of SCATR.

SCATR is not completely generic, in the sense that the target transformation tool is fixed; it is aimed at transformations expressed in ASF+SDF (van den Brand et al., 2001) only. Nevertheless, SCATR is not tied to a particular analysis tool. Furthermore, SCATR is independent of the language used in the source code, provided an SDF grammar for that language is available.

In terms of Figure 4.1, SCATR operates within the "Transformation" domain. Its purpose consists of inserting analysis results expressed as scaffolding specifications into the AST used by the transformation tool (i.e., ASF+SDF). Figure 4.3 gives an overview of SCATR. Three steps are performed to decorate an AST with analysis results.

1. Parsing with a grammar extended with support for *scaffolds*, resulting in an AST corresponding to the source code. Scaffolds are akin to parse tree annotations (Purtilo and Callahan, 1989), and allow analysis results to be attached to nodes in the AST. The precise definition of scaffolds is discussed below.

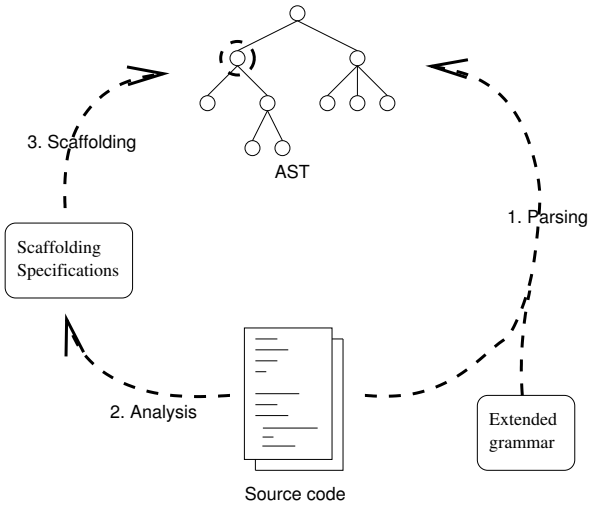

Figure 4.3: SCATR overview.

2. Analysis results are generated by an appropriate analysis tool. The results of the tool are expressed as *scaffolding specifications*, which steer the process of inserting scaffolds in the AST. Scaffolding specifications are also discussed below.

3. Scaffolding is the final step in which the analysis results are inserted into the AST based on the scaffolding specifications.

In the remainder of this section we will first discuss the implementation of SCATR, followed by a discussion of design decisions underlying SCATR's architecture.

4.3.1 Implementation

Figure 4.4 presents the core modules of the ASF+SDF implementation of SCATR. The format used is SDF, which is similar to EBNF, except that the right and left hand sides of the grammar rules are swapped. Furthermore, non-terminals are referred to as *sorts* in SDF. Any parameters of a module are listed between square brackets next to the module's name. A parameter of an SDF module allows the user to specify a sort for which the module should be instantiated. The result of supplying an argument to a parameter is essentially a textual replacement of the occurrences of the parameter by its argument.

Extended-Language. The module ExtendedLanguage allows a grammar to be extended to facilitate scaffolding. It defines grammar productions that allow (one or more) Extensions before or after the Element of interest. The user can specify two parameters when using this module. Element is the sort the user intends to extend. For instance, Statement would be specified if the user wishes to extend statements. Extension would normally be specified as Scaffold, but additional uses (e.g., comments, annotations) justify an additional layer of abstraction, as proposed by Sellink and Verhoef (2000).

```
module Extended-Language [ Element Extension ]

Extension+ Element                    →  Element
Element Extension+                    →  Element
```

```
module Scaffolder [ Program Element ]

scaffolder ( Program, ScS* )          →  Program
```

```
module Scaffolding-Spec

"begin" Scaffold Type Position "end"  →  Scaffolding-Spec
"before" | "after"                    →  Type
"(" Natural Natural ")"               →  Position
```

```
module Scaffold [ Ext-Scaffold-Data ]

"SCAFFOLD" "[" Scaffold-Data* "]"     →  Scaffold
Data-Name "[" Scaffold-Data* "]"      →  Scaffold-Data
[A-Z_]+                               →  Data-Name
String                                →  Scaffold-Data
Ext-Scaffold-Data                     →  Scaffold-Data
```

Figure 4.4: SDF excerpts of the core modules of SCATR.

Scaffolder. The main module of SCATR defines the scaffolder function. The scaffolder function traverses its Program argument and inserts scaffolds to nodes of type Element according to a list of ScaffoldingSpecs (ScS*) supplied as the second argument.

The user of this module has to make sure the Program and Element parameters are set correctly. Program is to be instantiated as the top-level sort of the source code grammar, while the Element parameter should be set to the sort the user wishes to add scaffolds to.

Scaffolding-Spec. A ScaffoldingSpec specifies the insertion of a scaffold at a certain node in the AST. To select the target node the user specifies a Position, that is, line and column number, in the source file from which the AST was derived. The scaffolder will attach the scaffold to the lowest node in the AST that includes the position in its source range. Whether the scaffold is added to the left or to the right of the selected node is determined by the Type, i.e., respectively `before` or `after`.

Scaffold. The syntactical definition of a scaffold resides in this module. This definition is loosely based on the definition in Sellink and Verhoef (2000). Scaffolds can contain nested lists of named data, which can be of various sorts. By default, Strings are allowed as Scaffold-Data, but the user can add custom sorts by instantiating the Ext-Scaffold-Data parameter.

4.3.2 Architecture

The source-based mapping for which SCATR was designed consists of source code positions, that is, pairs of line and column numbers. The analysis tool is expected to map an instance of its schema (e.g., a PDG node in CodeSurfer) to a single source code position (*down* function in Section 4.2). SCATR will attempt to map this source code position to an appropriate node in an AST maintained by ASF+SDF (*up* function in Section 4.2). The analysis tool is burdened with making sure that the source-based mapping obtained is safe, i.e., it must ensure that an AST node of the selected sort is defined at the source code positions it exports.

Scaffolding. Determining which nodes are extended with a scaffold depends on two sources of information. First, the user of the SCATR framework specifies the type[1] of AST nodes that can receive a scaffold. For instance, the user may choose to add scaffolds to Statement nodes, if Statement is a sort defined by the grammar. In Subsection 4.3.1 we discuss how the user achieves the sort selection.

Second, the scaffolding specification lists source code positions paired with scaffolds containing data. The scaffolder function adds a scaffold to a node if and only if the node is of the selected sort, and the source code position specified with the scaffold is included in the source code area spanned by the node.

Note that this process requires that the target AST is fully decorated with source position information, that is, each node in the AST can be mapped to its corresponding area within the source code.

One intricacy of the scaffolding process remains to be explained. A source code position can point to more than one node of the user selected sort. In C, for example, nested statements, or expressions, can cause this effect. SCATR ensures that a *strict* (see Section 4.2) mapping is obtained through two design decisions. The AST is traversed in a bottom-up fashion, and a scaffold is inserted at most once. In effect, a scaffold is added to the most specific (or lowest) AST node of the user selected sort that includes the specified source code position in its area. This behavior implemented by SCATR may not always be desirable (though it has been for our purposes).

The number of AST nodes that are pointed to by the source-based mapping could possibly be reduced by improving upon the accuracy of source code positions. Source code areas could alternatively be used to create a source based mapping. A source code area more accurately describes the source representation of an instance by specifying the line and column numbers of the start and end of the instance. We discussed this solution in Section 4.2, and have shown that the use of a finer grained source correspondence on one end (here in the analysis tool) requires changes in the other end (here the transformation tool). For flexibility SCATR uses the relatively inaccurate source code positions, and deals with multiple matching nodes by picking the most specific node.

Grammar Extension. In order for scaffolds to be added to AST nodes, the grammar used to parse the source code needs to be extended such that nodes of interest (i.e., of type T in terms of Section 4.2) can be preceded or followed by nodes representing scaffolds. [2] SCATR provides for a flexible way of extending grammars. The module Extended-Language

[1] The type of an AST node corresponds to a sort in the SDF grammar.

[2] In systems which do not require AST transformations to be syntax preserving such grammar modification may be unnecessary.

adds two grammar productions to extend the sort of interest such that it can be preceded and followed by scaffolds. The details of this module are presented in Subsection 4.3.1.

Lexical Scaffolding. Scaffolding as defined by Sellink and Verhoef (2000) operates slightly differently. Their approach extends the target grammar much more extensively, by allowing scaffolds in front of each terminal (occurrence of a lexical sort). This has the advantage that scaffolds can also be added directly to the source code itself, followed by an invocation to the parser to obtain a scaffolded AST. In our simple approach to grammar extension this results in many ambiguities during parsing.

An advantage of our approach is that the scaffolding process inserts scaffolds at exactly the nodes of interest in the AST. This is beneficial for the purpose of specifying transformations based on the scaffolds, as no extra work has to be done to locate the scaffolds (if any) associated with the node. Sellink and Verhoef's approach causes the scaffolds to be added as leaves in the AST, possibly a long way from the nodes of interest. Without additional support for locating scaffolds in the AST, this is an unpractical situation for the specification of transformations. Kort and Lämmel (2003) provide methods that are capable of locating scaffolds, and dealing with them in transformations.

Finally, one could argue that simple grammar extension could suffice if one would lexically insert *bracketed* scaffolds. A bracketed scaffold surrounds the source region it applies to with brackets, so that no ambiguity arises during parsing. A similar approach is taken by source code factors (Malton et al., 2001). As it turns out, lexically inserting bracketed scaffolds is not practical. The analysis tool exporting its results would then need to generate the positions of the brackets in a way that is lexically compatible with the grammar used by the transformation tool. In our approach, the analysis tool can suffice by generating a position that it knows to lie within the source area of an AST node of the desired sort (safeness).

4.4 Applications

The SCATR framework is currently being used in several real transformation tasks. These tasks consider components of a 15 million line C system, developed and maintained by ASML, a Dutch manufacturer of lithography solutions. The tasks are related to our earlier work on (crosscutting) concern isolation (see Chapter 3), and consist of elimination of concern code, and insertion of annotations (among others). These transformations are required in a larger migration effort toward aspect-oriented technology.

Source-to-source transformations are desired in these cases, since developers have to be able to work with the transformed code. Specifically this requires the abilities to parse code in the presence of C preprocessor directives (including macros), and to preserve comments and white-space. Due to the availability of an SDF grammar for ANSI C extended with preprocessing directives and rewriting with layout capability (Vinju, 2005), the ASF+SDF Meta-Environment (van den Brand et al., 2001) is used to implement the transformation tasks. The C grammar was modeled strictly after the ANSI C specification, and extended with support for the specific preprocessor use within ASML.

Several analyses required to identify concern code have previously been implemented (see Chapter 3) as plugins to GrammaTech's CodeSurfer (Anderson et al., 2003). Since these analyses are not trivial, and significant effort would be needed to re-implement them in

```
868  THXAtrace(CC,
869    THXA_TRACE_INT,
870    func_name,
871    "> (read_fd=%d, timeout=%d)",
872    read_fd,
873    timeout);
```

Figure 4.5: Example tracing call.

ASF+SDF, the choice was made to reuse the CodeSurfer plugins. SCATR was developed to solve the problem of leveraging CodeSurfer's analysis results in transformations expressed in ASF+SDF.

SCATR has been used for the transformation of two components, CC1 and CC2, consisting of 32,402 and 17,716 non-blank lines of code, respectively. Efforts are currently ongoing to apply SCATR to 10 components, totalling approximately 2 million lines of code.

4.4.1 Concern Code Elimination

The first transformation task we consider consists of the elimination of code belonging to a number of concerns:

- **Tracing**. Dynamic execution tracing of each function such that the values of input and output parameters can be inspected.
- **Timing**. Collection of timings for each function execution.
- **Function Naming**. Each function has a local variable which holds a string representing the function's name. These strings are used within tracing and logging calls.
- **Parameter Checking**. Pointer parameters of functions should not be NULL before they are referenced, each function therefore has to implement checks. The parameter checking concern is discussed in detail in Chapter 3.

The instantiation of SCATR for the elimination of these concerns is very similar in all cases, therefore we suffice with a discussion of the elimination of the tracing concern in this chapter. The tracing concern consists of calls to a tracing function, where the arguments are the values of either input or output parameters. An example is shown in Figure 4.5. Chapter 6 discusses the ASML tracing concern is more detail.

A CodeSurfer plugin was previously developed to identify all the tracing calls for all functions. The result consists of a set of PDG nodes representing the calls to the tracing function. Furthermore, a utility script was developed to export these PDG nodes along with the fact that they belong to the tracing concern, into SCATR's scaffolding specification format (see Section 4.3). An example scaffolding specification is shown in Figure 4.6. It states that a scaffold of the form `SCAFFOLD["TRACING"]` should be added before the instance at source code line 868, column 0. This source code position corresponds to the first character of the tracing call.

Effectively this export script implements the *down* function that was described in Section 4.2. The other part of the source-based mapping, the *up* function, is implemented by

```
begin
 SCAFFOLD["TRACING"]
 before
 (868 0)
end
```

Figure 4.6: Scaffolding specification for a single tracing call.

```
SCAFFOLD["TRACING"]<<THXAtrace(CC,
          THXA_TRACE_INT,
          func_name,
          "> (read_fd=%d, timeout=%d)",
          read_fd,
          timeout);>>
```

Figure 4.7: Tracing call decorated with a scaffold.

SCATR. Function calls are parsed as Statements in our SDF C grammar, thus instantiating SCATR for this task starts by extending the C grammar such that scaffolds can be added to Statement nodes. As was explained in Section 4.3, this is done through the parameters of the module ExtendedLanguage (see Figure 4.4).

The next step consists of the invocation of the scaffolder function, with the (parsed) source code and all generated scaffolding specifications as arguments. The result is an AST in which all the Statement nodes pointed to by the scaffolding specifications are decorated with a scaffold. Figure 4.7 shows how this would look if a decorated node was pretty printed.

Finally, the AST is traversed one more time by a function that removes all nodes decorated with a specific scaffold. In this case the traversal would look for tracing scaffolds, but for the other concerns the scaffolds contain the respective names of the concerns.

The source-based mapping we defined in this case works because it is strict and safe, as defined in Section 4.2. First, safeness holds because in the ANSI C grammar the first character of the name of the called function is guaranteed to point to an AST node of type Statement. Second, the mapping is also strict, due to SCATR's strategy of selecting the most specific node of type Statement. The Statement node representing the tracing call will always be lower in the AST than Statement nodes representing any surrounding statements. For now these properties follow from the structure of the grammar, and SCATR's selection strategy. In the future we would like to implement a tool to check whether these properties hold given a body of source code, and a defined source-based mapping.

4.4.2 Insertion of Annotations

Annotations can be used in source code representations to store data at the locations that the data are relevant. SCATR can be used for the purpose of inserting such annotations. We will discuss an example application of using SCATR in this way within the context of the tracing concern.

After all tracing code has been eliminated (see previous Subsection), a compile-time weaver is responsible for regenerating the tracing functionality. As it turns out, the tracing

```
549 int CCCN_Wait(int read_fd,
550                int timeout)
551 __trace__(in (read_fd timeout) out ())
552 {
553   . . .
554 }
```

Figure 4.8: Tracing annotation.

```
1 begin
2  SCAFFOLD [IN [read_fd timeout] OUT []]
3  after
4  (550 25)
5 end
```

Figure 4.9: Scaffolding specification for temporary scaffolds.

concern requires some non-trivial analysis to figure out which function parameters are used as input and which are used as output parameters. Since it would be costly to integrate this analysis into the build process, it was decided to perform a one-time analysis of the source code, and insert the results (i.e., input / output characteristics) into the code as annotations. In our case, the annotations are specifically targeted at a compile-time weaver for C, WeaveC[3]. Here we will discuss the use of SCATR in the annotation process.

Again, the analysis is performed by a CodeSurfer plugin, and results in a list of input and output parameters for each function (PDG). Figure 4.8 gives an example of how a annotation should be inserted in the source code at line 551. The annotation has been inserted after the function signature, and before the defining block of the function. It shows that this function (`CCCN_Wait`) has two input parameters, `read_fd` and `timeout`, and no output parameters.

The insertion process works by first inserting temporary scaffolds containing the analysis results, and then translating the scaffolds into the desired annotation format. SCATR again requires the selection of the sort to which scaffolds need to be added. Since the annotation has to be inserted after the function signature, the appropriate sort is Declarator, which spans the area from the start of the function name (`CCCN_Wait` in Figure 4.8), up to and including the closing parenthesis of the signature. A CodeSurfer script is used to implement *down* by generating a scaffolding specification with the input/output information wrapped in a scaffold, and the source code position corresponding to the closing parenthesis of the function signature. An example is shown in Figure 4.9.

Similar to the result in Figure 4.7, running the scaffolder function on the source code with the annotation scaffolding specifications results in function signatures with scaffolds appended to them. The final step then consists of a traversal which trivially translates the present scaffolds into the tracing annotations shown in Figure 4.8.

Safeness and strictness of the mapping used in this case follows by the same argument we used before. According to the ANSI C grammar, the closing parenthesis of a function signature matches exactly one node of sort Declarator.

[3] `http://weavec.sourceforge.net/`

4.5 Discussion

Source-based mappings versus schema-based mappings. The defining feature of source-based mappings is that they locate the instances of interest through pointers in the source code itself. Other approaches exist to solve the location problem that do not utilize source correspondences at all (or as much). These *schema-based* approaches locate instances of interest in the target (transformation) tool through queries expressed using the target schema. HSML is, in essence, such an approach (Cordy et al., 2001), since it allows maintenance hot spots to be identified through complex queries expressed using the grammar of the target language. Other examples are the various query languages for XML documents, e.g. XPath or XQuery.

We now consider how a schema-based mapping could be used in the context of linking an analysis and a transformation tool. The setting is the same as defined in Section 4.2, i.e., we want to map analysis results for instances of type S to instances of type T in a transformation tool. The analysis and transformation tools have different schemas of the source code. What alternatives to the $down$ and up functions would need to be implemented?

To start with $down$, recall that its purpose is to map an instance s to an appropriate area of source code, such that up can map that area to an instance of type T. The analogue of $down$ in a schema-based mapping then consists of a function that maps s to an appropriate expression in the schema used by the transformation tool. Subsequently, the analogue of up is tasked with interpreting this expression and applying the results to the matching instances. For instance, an expression pointing out the assignment statement (at line 5) in Figure 4.1 could consist of *Stat – Block – Stat [0] – Block – Stat [0]*, where *Stat [i]* would refer to the i-th statement within a block.

The question that arises is what knowledge is required for the implementation of a schema-based mapping, and how does this compare to a source-based mapping? Creating an expression that points out an instance of interest requires knowledge of the target schema. For example, the expression above can only be created if it is known that the Stat node of interest is the 0-th child of its parent Block node containing it, which is a child of a Stat node itself, and so forth, all the way up to the top Stat node. As a result, creating such an expression requires knowledge of the target schema from the top down to at least the type of the instances of interest. Possibly this knowledge requirement can be mitigated by designing a query language that allows abstraction, but we conjecture that at least all the containment relations must still be known in order to accurately point out the instances of interest.

Source-based mappings also require knowledge of the target schema. $Down(s)$ has to yield a source code area that corresponds to an instance of type T at the transformation end (safeness). Therefore, implementing $down$ cannot be done without knowledge of the source correspondence of instance type T. Recall the example explained in Section 4.2, where a finer-grained source correspondence within the transformation tool required $down$ to be changed accordingly. However, as long as the safeness property can be guaranteed, $down$ does not have to re-generate the exact source correspondence of the transformation tool.

Additional awareness is needed for guaranteeing the strictness property, i.e., making sure that at most one instance of type T is the target of $up(down(s))$. This problem can occur if $down(s)$ is included in the source correspondence of more than one instance of type T. Therefore, implementing a source-based mapping requires knowledge of overlapping source cor-

respondences of instances of type T. If the problem cannot be evaded by changing $down(s)$ to generate a more specific source code area, a strategy will have to be implemented in up to make a choice, as was discussed in Section 4.2.

In summary, a schema-based mapping requires detailed knowledge of the target schema. It may be required to know the definition of other types of instances than the type of interest, because containment relations need to be traversed from the top down leading to the instance of interest. In contrast, the knowledge a source-based mapping needs of the target schema is limited to the type of interest only. However, it is required to be aware of its source corresponce, and possible overlapping source correspondences of instances of the type of interest.

Automatically checking safeness and strictness. The use of source-based mappings in real transformation tasks may benefit from some form of automated verification. In particular, checking the safeness and strictness properties could be a good starting point. Fortunately, these properties can be checked automatically for a fixed body of source code by the following process. First, the domain of the source-based mapping is established. All instances s of type S belong to the domain. For each s then $up(down(s))$ is performed, yielding a list of instances of type T. If this list is empty, no instance of type T was found defined at $down(s)$, the source-based mapping is not safe, and an error must be reported. If the list contains exactly one instance t, the mapping is safe and strict for s. Finally, a list of length 2 and more indicates that multiple instances are defined at $down(s)$, and the mapping is not strict, resulting in an error. After all s have been checked, the complete mapping is only safe and strict if no errors have been reported.

For some schemas, the safeness and strictness properties could even be determined for all possible bodies of source code. For now this remains the area of future research.

4.6 Related Work

Tool interoperability The topic of tool interoperability has been widely discussed in the community (Cordy and Vinju, 2006; Ebert et al., 2001). A large number of proposals exist in the literature that contribute a solution to interoperability issues. Among others, technologies like the ToolBus (Bergstra and Klint, 1998), the Ontological Adaptive Service-Sharing Integration System (OASIS) (Jin and Cordy, 2005), and the Interface Description Language (IDL) (Snodgrass and Shannon, 1986) provide architectures for integration of tools. Communication is an essential part of tool interoperability, and as such a number of data interchange formats have been defined. Examples are the Graph eXchange Language (GXL) (Holt et al., 2000b), ATerms (van den Brand et al., 2000a), and Rigi Standard Format (RSF) (Müller et al., December 1993). Technologies like these provide tool interoperability solutions at a different level than source-based mappings. In terms of Cordy and Vinju (2006), these technologies provide protocols, (data) marshalling, or representations. Source-based mappings are aimed at solving the identification (of source elements) problem.

Markup and annotations Source code markup is a technique that has been used in many different contexts. Here we focus only on those approaches that are closely related to source-

based mappings. Scaffolding by Sellink and Verhoef (2000) is proposed to be used to store intermediate results of transformations, and share results between tools via markup in the source code. However, they do not explicitly focus on the issue of tools using different schemas. Source code factors by Malton et al. (2001) is an approach that is very similar to scaffolding, as it also marks up the source code with intermediate analysis and transformation results.

HSML by Cordy et al. (2001) is a markup approach that allows maintenance hot spots to be defined as queries expressed in the target schema. In that sense it is a schema-based mapping as defined in Section 4.5, except that the results of the mapping are also made visible in the source code through markup. The difference with source-based mappings consists of the extensive use of the target schema by HSML. A source-based mapping is less dependent on the target schema, but more dependent on the source correspondence.

XML is a popular means to marking up source code. Cordy (2003) proposes a method to markup source code with task-specific XML. By employing agile parsing (possibly combined with island grammars (Moonen, 2001)), the source code grammar can be adjusted to focus the source markup to those pieces of source code that are interesting to the task at hand. In our SCATR framework, pretty printing the scaffolded AST has the same result, since scaffolds are only added to those nodes that are interesting within a transformation task. Power and Malloy (2002) instead proposes to markup all the source code with XML corresponding to its AST, resulting in a verbose representation.

Tool interoperability schemas A number of schemas have been proposed specifically for the purpose of tool interoperability. The Dagstuhl Middle Metamodel (DMM) is aimed at object-oriented and procedural languages, and can further be extended by the user of the schema (Lethbridge et al., 2004). Columbus is a specific schema for C++ programs (Ferenc and Beszédes, 2002). Both Columbus and DMM are expressed as UML diagrams. Holt et al. (2000a) instead use an E/R diagram to define a schema for Datrix, a software exchange format for C, C++ and Java programs.

4.7 Conclusion

In this chapter we discussed source-based mappings, a technique to link analysis and transformation tools. The setting used for this discussion consisted of analysis and transformation tools that do not share a schema of the source code, and therefore reuse of analysis results by the transformation tool is not trivial. We defined two properties, safeness and strictness that constitute a base of confidence in the mapping between two tools. These properties can be checked automatically for a given body of source code, allowing for a practical way of verification of a source-based mapping.

The idea of source-based mappings was implemented in the SCATR prototype tool, which allows analysis results to be mapped into ASTs produced by ASF+SDF. Two applications showed how this technology could be used in practice to implement transformation tasks such as concern code elimination or insertion of annotations.

An interesting link may exist between this work and island grammars (Moonen, 2001), or agile parsing (Cordy, 2003). These technologies allow the easy adaptation of grammars

to specific tasks. For instance, the grammar could be limited to defining only the statements that need to be removed, or the program elements that analysis results must be attached to.

Chapter 5

Discovering Faults in Idiom-Based Exception Handling*

In this chapter we analyse the exception handling mechanism of the ASML C system. Like many systems implemented in classic programming languages (e.g, C), the ASML system uses the popular return-code idiom for dealing with exceptions. Our goal is to evaluate the fault-proneness of this idiom, and we therefore present a characterisation of the idiom, a fault model accompanied by an analysis tool, and empirical data. Our findings show that the idiom is indeed fault prone, but that a simple solution can lead to significant improvements.

5.1 Introduction

A key component of any reliable software system is its exception handling. This allows the system to detect errors, and react to them correspondingly, for example by recovering the error or by signalling an appropriate error message. As such, exception handling is not an optional add-on, but a sine qua non: a system without proper exception handling is likely to crash continuously, which renders it useless for practical purposes.

Despite its importance, several studies have shown that exception handling is often the least well understood, documented and tested part of a system. For example, Toy (1982) states that more than 50% of all system failures in a telephone switching application are due to faults in exception handling algorithms, and Lions (1996) explains that the Ariane 5 launch vehicle was lost due to an unhandled exception.

Various explanations for this phenomenon have been given.

First of all, since exception handling is not the primary *concern* to be implemented, it does not receive as much attention in requirements, design and testing. Robillard and Murphy (1999) explains that exception handling design degrades (in part) because less attention

*This chapter was published in the Proceedings of the 28th International Conference on Software Engineering (ICSE 2006) (Bruntink et al., 2006). It is co-authored by Arie van Deursen and Tom Tourwé.

is paid to it, while Christian (1995) explains that testing is often most thorough for the ordinary application functionality, and least thorough for the exception handling functionality. Granted, exception handling behaviour is hard to test, as the root causes that invoke the exception handling mechanism are often difficult to generate, and a combinatorial explosion of test cases is to be expected. Moreover, it is very hard to prepare a system for all possible errors that might occur at runtime. The environment in which the system will run is often unpredictable, and errors may thus occur for which a system was not prepared for.

Second, exception handling functionality is crosscutting in the meanest sense of the word. Lippert and Videira Lopes (2000) shows that even the simplest exception handling strategy takes up 11% of an application's implementation that it is scattered over many different files and functions and that it is tangled with the application's main functionality. This has a severe impact on understandability and maintainability of the code in general and the exception handling code in particular, and makes it hard to ensure correctness and consistency of the latter code.

Last, older programming languages, such as C or Cobol, that do not explicitly support exception handling, are still widely used to develop new software systems, or to maintain existing ones. Such explicit support makes exception handling design easier, by providing language constructs and accompanying static compiler checks. In the absence of such support, systems typically resort to systematic coding idioms for implementing exception handling, as advocated by the well-known *return code* technique, used in many C programs and operating systems. As shown in Chapter 3, such idioms are not scalable and compromise correctness.

In this chapter, we focus on the exception handling mechanism of a 15 year-old, real-time embedded system, developed by ASML, a Dutch company. The system consists of approximately 15 million lines of C code, and is developed and maintained using a state-of-the-art development process. It applies (a variant of) the return code idiom consistently throughout the implementation. The central question we seek to address is the following: "how can we reduce the number of implementation faults related to exception handling implemented by means of the return code idiom?" In order to answer this general question, a number of more specific questions needs to be answered:

1. What kinds of faults can occur? Answering this question requires an in-depth analysis of the return code idiom, and a fault model that covers the possible faults to which the idiom can lead.

2. Which of these faults do actually occur in the code? A fault model only predicts which faults can occur, but does not say which faults actually occur in the code. By carefully analysing the subject system (automatically) an estimate of the probability of a particular fault can be given.

3. What are the primary causes of these faults? The fault model explains *when* a fault occurs, but does not explicitly state *why* it occurs. Because we need to analyse the source code in detail for detecting faults, we can also study the causes of these faults, as we will see.

4. Can we eliminate these causes, and if so, how? Once we know why these faults occur and how often, we can come up with alternative solutions for implementing exception

handling that help developers in avoiding such faults. An alternative solution is only a first step, (automated) migration can then follow.

We believe that answers to these questions are of interest to a broader audience than the original developers of our subject system. Any software system that is developed in a language without exception handling support will suffer the same problems, and guidelines for avoiding such problems are more than welcome. In this chapter we offer experience, an analysis approach, tool support, empirical data, and alternative solutions to such projects.

5.2 Related Work

Fault (Bug) Finding Recently a lot of techniques and tools have been developed that aim at either static fault finding or program verification. However, we are not aware of fault finding approaches specifically targeting exception handling faults.

Fault finding and program verification have different goals. On the one hand, fault finding's main concern is finding as many (potentially) harmful faults as possible. Therefore fault finding techniques usually sacrifice formal soundness in order to gain performance and thus the ability to analyse larger systems. Specifically, Metal (Engler et al., 2000), PREfix (Bush et al., 2000), and ESC (Flanagan et al., 2002) follow this approach. We were inspired by many of the ideas underlying Metal for the implementation of our tool (see Section 5.5).

Model checking is also used as a basis for fault finding techniques. CMC (Musuvathi et al., 2002) is a model checker that does not require the construction of a separate model for the system to be checked. Instead, the implementation (code) itself is checked directly, allowing for effective fault finding. In Yang et al. (2004) the authors show how CMC can be used to find faults in file system implementations.

On the other hand, program verification is focused on proving specified properties of a system. For instance, MOPS (Chen and Wagner, 2002) is capable of proving the absence of certain security vulnerabilities. More general approaches are SLAM (Ball and Rajamani, 2002) and ESP (Das et al., 2002). While ESP is burdened by the formal soundness requirement, it has nevertheless been used to analyse programs of up to 140 KLOC.

Idiom Checkers A number of general-purpose tools have been developed that can find basic coding errors (Johnson, 1977; Paul and Prakash, 1994). These tools are however incapable of verifying domain-specific coding idioms, such as the return code idiom. More advanced tools (van Emden and Moonen, 2002; Tourwé and Mens, 2003) are restricted to detecting higher-level design flaws but are not applicable at the implementation level.

In Chapter 3, we present an idiom checker for the parameter checking idiom, also in use at ASML. This idiom, although much simpler, resembles the exception handling idiom, and the verifier is based on similar techniques as presented in this chapter.

Exception Handling Several proposals exist for extending the C language with an exception handling mechanism. Lee (1983); Roberts (1989) and Winroth (1993) all define exception handling macro's that mimic a Java/C++ exception-handling mechanism. Although slightly varying in syntax and semantics, these proposals are all based around an idiom using the C `setjmp/longjmp` facility.

Exceptional C (Gehani, 1992) is a more drastic, and as such more powerful, extension of C with exception handling constructs as present in modern programming languages. It

allows developers to declare and raise exceptions and define appropriate handlers. A function's signature should specify the exceptions that the function can raise, which allows the preprocessor to check correctness. Standard C code is generated as a result.

All these proposals differ from our proposal (Section 5.7) in that our proposal still uses the return-code idiom, but makes it more robust by hiding (some of) the implementation details. This makes migration of the old mechanism to the new one easier, an important concern considering ASML's 10 MLoC code base.

Several papers describe exception handling analyses for Java-like exception handling mechanisms. Robillard and Murphy (2003, 1999) show how exception structure can degrade and present a technique based on software compartmenting to counter this phenomenon. Their work differs from ours in that they reason about the application-specific design of exception handling, whereas we focus on the (implementation of) the exception handling mechanism itself. Fu et al. (2005) present an exception-flow analysis that enables improving the test coverage of Java applications. Similarly, Sinha and Harrold (1999) present a class of adequacy criteria that can be used for testing exception handling behaviour. Although this work could lead to better tests, and hence well-tested exception handling code, a test-based approach remains necessarily partial. Hence, such techniques should be considered complementary to our technique. Additionally, both techniques are targeted toward Java-like exception handling mechanisms, and it is not clear how they would work for systems using the return-code idiom.

5.3 Characterising the Return Code Idiom

The central question we seek to answer is how we can reduce the number of faults related to exception handling implemented by means of the return code idiom. To arrive at the answer, we first of all need a clear description of the workings of (the particular variant of) the return code idiom at ASML. We use an existing model for exception handling mechanisms (EHM) (Lang and Stewart, 1998) to distinguish the different components of the idiom. This allows us to identify and focus on the most error-prone components in the next sections. Furthermore, expressing our problem in terms of this general EHM model makes it easier to apply our results to other systems using similar approaches.

5.3.1 Terminology

An exception at ASML is "any abnormal situation found by the equipment that hampers or could hamper the production". Exceptions are logged in an *event log* that provides information on the machine history to different stakeholders (such as service engineers, quality assurance department, etc).

The EHM itself is based on two requirements:

1. a function that detects an error should log that error in the event log, and recover it or pass it on to its caller.

2. a function that receives an error from a called function must provide useful context information (if possible) by *linking* an error to the received error, and recover the error

```
1  int f(int a, int* b) {
2     int r = OK;
3     bool allocated = FALSE;
4     r = mem_alloc(10, (int *)b);
5     allocated = (r == OK);
6     if((r == OK) && ((a < 0) || (a > 10))) {
7        r = PARAM_ERROR;
8        LOG(r,OK);
9     }
10    if(r == OK) {
11       r = g(a);
12       if(r != OK) {
13          LOG(LINKED_ERROR,r);
14          r = LINKED_ERROR;
15       }
16    }
17    if(r == OK)
18       r = h(b);
19    if((r != OK) && allocated)
20       mem_free(b);
21    return r;
22 }
```

Figure 5.1: Exception handling idiom at ASML.

or pass it on to the calling function.

An error that is detected by a function is called a *root* error, while an error that is linked to an error received from a function is called a *linked* error.

If correctly implemented, the EHM produces a chain of related consecutive errors in the event log. This chain is commonly referred to as the *error link tree*, and resembles a stack trace as output by the Java virtual machine, for example.

Because ASML uses the C programming language, and C does not have explicit support for exception handling, each function in the ASML source code follows the *return code* idiom. Figure 5.1 shows an example of such a function. We will now discuss this approach in more detail.

5.3.2 Exception Representation

An exception representation *defines what an exception is and how it is represented.* At ASML, a *singular* representation is used, in the form of an *error variable* of type `int`. Line 2 in Figure 5.1 shows a typical example of such an error variable, that is initialised to the `OK` constant. This variable is used throughout the function to hold an *error value*, i.e., either `OK` or any other constant to signal an error. The variable can be assigned a constant, as in lines 7 and 14, or can be assigned the result of a function call, as in lines 4, 11 and 18. If the function does not recover from an error itself, the value of the error should be propagated through the caller by the `return` statement (line 21).

5.3.3 Exception Raising

Exception raising is *the notification of an exception occurrence*. Different mechanisms exist, of which ASML uses the *explicit control-flow transfer* variant: if a root error is encountered, the error variable is assigned a constant (see lines $6-9$), the function logs the error, stops executing its normal behaviour, and notifies its caller of the error.

Logging occurs by means of the `LOG` function (line 8), where the first argument is the new error encountered, which is linked to the second argument that represents the previous error value. The function treats root errors as a special case of linked errors, and therefore the root error detected at line 8 is linked to the previous error value, `OK` in this case.

Explicit guards are used to skip the normal behaviour of the function, as in lines 10 and 17. These guards check if the error variable still contains the `OK` value, and if so, execute the behaviour, otherwise skip it. Note that such guards are also needed in loops containing function calls.

If the error variable contains an error value, this value propagates to the `return` statement, which notifies the callers of the function.

5.3.4 Handler Determination

Handler determination is *the process of receiving the notification, identifying the exception and determining the associated handler*. The notification of an exception occurs through the use of the `return` statement and catching the returned value in the error variable when invoking a function (lines 4, 11 and 18). This approach is referred to as *explicit stack unwinding*.

The particular exception that occurs is not identified explicitly most of the time, rather a *catch-all* handler is provided. Such handlers are mere guards that check whether the error value is not equal to `OK`. Typically, such handlers are used to link extra context information to the encountered error (lines $12-15$), or to clean up allocated resources (lines $19-20$).

5.3.5 Resource Cleanup

Resource cleanup is *a mechanism to clean up resources, to keep the integrity, correctness and consistency of the program*.

ASML has no automatic cleanup facilities, although specific handlers typically occur at the end of a function if cleaning up of allocated resources is necessary (lines $19-20$).

5.3.6 Exception Interface & Reliability Checks

The exception interface *represents the part in a module interface that explicitly specifies the exceptions that might be raised by the module*. ASML developers sometimes specify (parts of) this interface in informal comments, but this is not strictly required.

Consequently, reliability checks that *test for possible faults introduced by the EHM itself* are currently not possible. The focus of this chapter is to analyse which faults can be introduced and to show how they can be detected and prevented.

5.3.7 Other Components

An EHM consists of several other components than the ones mentioned above. Although these are less important for our purposes, we shortly describe them here for completeness.

Handler scope is *the entity to which an exception handler is attached*. At ASML, handlers have *local* scope: handlers are associated to function calls (lines $12 - 15$), where they log extra information, or can be found at the end of a function (lines $19 - 20$), where they clean up allocated resources.

Handler binding *attaches handlers to certain exceptions to catch their occurrences in the whole program or part of the program.* ASML uses *semi-dynamic* binding, which means that the same exception can be handled differently in different locations in the source code, by associating a different handler with each exception occurrence.

Information passing is defined as *transfer of information useful to the treatment of an exception from its raising context to its handler*. At ASML there is no information passing except for the integer value that is passed to a caller. Although an error log is constructed, the entries are used only for offline analysis.

Criticality management represents *the ability to dynamically change the priority of an exception handler, so that it an be changed based on the importance of the exception, or the importance of the process in which the error occurred*. This is not considered at ASML.

5.4 A Fault Model for Exception Handling

Based on the characterisation presented in the previous section, we establish a fault model for exception handling by means of the return code idiom in this section. The fault model defines when a fault occurs, and includes failure scenarios which explain what happens when a fault occurs.

5.4.1 General Overview

Our fault model specifies possible faults occurring in a function's implementation of the *exception raising* and *handler determination* components. Those components are clearly the most prone to errors, because their implementation requires a lot of programming work, a good understandability of the idiom, and strict discipline. Although this also holds for the *resource cleanup* component, at ASML this component primarily deals with memory (de)allocation, and we therefore consider it to belong to a *memory handling* concern, for which a different fault model should be established.

The return code idiom at ASML relies on the fact that when an error is received, the corresponding error value should be logged and should propagate to the `return` statement. The programming language constructs that are used to implement this behaviour are function calls, return statements and log calls. The fault model includes a predicate for each of these

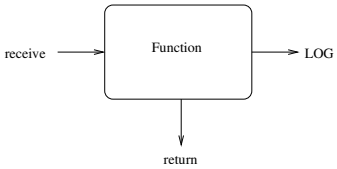

Figure 5.2: Inputs and outputs of a function with respect to exception handling.

Category 1		**Category 2**		**Category 3**	
receive(x)	$\wedge$	receive(x)	$\wedge$	receive(x)	$\wedge$
LOG(y,z)	$\wedge$	LOG(y,x)	$\wedge$	LOG($void, void$)	$\wedge$
$x \neq z$		return(z)	$\wedge$	return(y)	$\wedge$
		$y \neq z$		$x \neq y$	

Figure 5.3: Predicates for the three fault categories.

constructs, and consists of three formulas that specify a faulty combination of these constructs. If one of the formulas is valid for the execution of a function, the EH implementation of the function necessarily contains a fault.

A function is regarded as a black box, i.e., only its input–output behaviour is considered. This perspective allows easy mapping of faults to failure scenarios, at the cost of losing some details due to simplification. Figure 5.2 gives an overview of the relevant input and outputs of a function. Any error values received from called functions (*receive* predicate) are regarded as input. Outputs are comprised of the error value that is returned by a function (*return*), and values written to the error log (*LOG*). We map the input and outputs to logical predicates as follows.

First, *receive* is a unary predicate that is true for an error value that is received by the function during its execution. For instance, if a function receives an error `PARAM_ERROR` somewhere during its execution, then *receive*(`PARAM_ERROR`) holds true. If a function does not receive an error value during its execution (either because it does not call any functions, or no exception is raised by a called function), then *receive*(OK) holds. Likewise, *return* is a unary predicate that holds true for the error value returned by a function at the end of its execution. Finally, *LOG* is a binary predicate that is true for those two error values (if any) that are written to the error log. The first position of the *LOG* predicate signifies the new error value, while the second position signifies the (old) error to which a link should be established. If and only if nothing is written to the error log during execution, *LOG*(void, void) holds.

The fault model makes two simplifications: it assumes a function receives and logs at most one error during its execution. This is reasonable, because if implemented correctly, no other function should be called once an error value is received. Additionally, if only one error value can be received, it makes little sense to link more than one other error value to it.

5.4.2 Fault Categories

The fault model consists of three categories, each including a failure scenario, which are explained next. The predicates capturing the faults in each category are displayed in Figure 5.3. Example code fragments corresponding to Categories 1–3 are displayed in Figures 5.4–5.6, respectively.

Category 1 The first category captures those faults where a function raises a new error (y), but fails to perform appropriate logging. There are two cases to distinguish. First, y is considered a root error, i.e., no error has been received from any called function, and therefore $receive(OK)$ holds. The function is thus expected to perform $LOG(y, OK)$. However, a category 1 fault causes the function to perform $LOG(y, z)$ with $z \neq OK$.

Second, y is considered a linked error, i.e., it must be linked to a previously received error x. So, $receive(x)$ holds with $x \neq OK$, and the function is expected to perform $LOG(y, x)$. A category 1 fault in its implementation results in the function performing $LOG(y, z)$ with $x \neq z$.

Category 1 faults have the potential to break error link trees in the error log. The first case causes an error link tree to be improperly initiated, i.e., it does not have the *OK* value at its root. The second case will break (a branch of) an existing link tree, by failing to properly link to the received error value. Furthermore, the faulty *LOG* call will start a new error link tree which has again been improperly rooted. Especially in the latter case it will be hard to recover the chain of errors that occurred, making it impossible to find the root cause of an error.

Category 2 Here the function properly links a new error value y to the received error value x, but then fails to return the new error value (and instead returns z). The calling function will therefore be unaware of the actual exceptional condition, and could therefore have problems determining the appropriate handler. In the special case of $receive(OK)$, the function properly logs a root error y by performing $LOG(y, OK)$, but subsequently returns an error z different from the logged root error y.

Possible problems include corruption of the error log, due to linking to the erroneously returned error value z. Calling functions have no way of knowing the actual value to link to in the error log, because they receive a different error value. Even more seriously, calling functions have no knowledge of the actual error condition and might therefore invoke functionality that may compromise further operation of software or hardware. This problem is most apparent if *OK* is returned while an error has been detected (and logged).

Category 3 The last category consists of function executions that receive an error value x, do not link a new error value to x in the log, but return an error value y that is different from x. The failure scenario is identical to category 2.

5.5 SMELL: Statically Detecting Error Handling Faults

Based on the fault model we developed SMELL, the *State Machine for Error Linking and Logging*, which is capable of statically detecting violations to the return code idiom in the

source code, and is implemented as a plugin for CodeSurfer (2007). We want to detect faults statically, instead of through testing as is usual for fault models, because early detection and prevention of faults is less costly (Boehm, 1981; Bush, 1990), and because testing exception handling is inherently difficult.

5.5.1 Implementation

SMELL statically analyses executions of a function in order to prove the truth of any one of the logic formulas of our fault model. The analysis is static in the sense that no assumptions are made about the inputs of a function. Inputs consist of formal or global variables, or values returned by called functions.

We represent an execution of a function by a finite path through its control-flow graph. Possibly infinite paths due to recursion or iteration statements are dealt with as follows. First, SMELL performs intra-procedural analysis, i.e., the analysis stays within a function and does not consider functions it may call. Therefore recursion is not a problem during analysis. Intra-procedural analysis only does not impact SMELL's usefulness, as it closely resembles the way ASML developers work with the code: they should not make specific assumptions about possible return values, but should instead write appropriate checks after the call. Second, loops created by iteration statements are dealt with by caching analysis results at each node of the control-flow graph. We discuss this mechanism later.

The analysis performed by SMELL is based on the evaluation of a deterministic (finite) state machine (SM) during the traversal of a path through the control-flow graph. The SM inspects the properties of each node it reaches, and then changes state accordingly. A fault is detected if the SM reaches the *reject* state. Conversely, a path is free of faults if the SM reaches the *accept* state.

The error variable is a central notion in the current implementation of SMELL. An error variable, such as the `r` variable in Figure 5.1, is used by a programmer to keep track of previously raised errors. SMELL attempts to identify such variables automatically based on a number of properties. Unfortunately, the idiom used for exception handling does not specify a naming convention for error variables. Hence, each programmer picks his or her favourite variable name, ruling out a simple lexical identification of these variables. Instead, a variable qualifies as an error variable in case it satisfies the following properties:

- it is a local variable of type `int`,
- it is assigned only constant (integer) values or function call results,
- it is not passed to a function as an actual, unless in a log call,
- no arithmetic is performed using the variable.

Note that this characterisation does not include the fact that an error variable should be returned or that it should be logged. We deliberately do not want the error variable identification to depend on the correct use of the idiom, as this would create a paradox: in order to verify adherence to the idiom, the error variable needs to be identified, which would need strict adherence to the idiom to start with.

Most functions in the ASML source base use at most one error variable, but in case multiple are used, SMELL considers each control-flow path separately for each error variable. Functions for which no error variable can be identified are not considered for further analysis. We discuss the limitations of this approach at the end of this section.

The definition of the SM was established manually, by translating the informal rules in the manuals to appropriate states and transitions. Describing the complete SM would require too much space. Therefore we limit our description to the states defined in the SM, and show a subset of the transitions by means of example runs.

The following states are defined in the SM:

Accept and **Reject** represent the absence and presence of a fault on the current control-flow path, respectively.

Entry is the start state, i.e., the state of the SM before the evaluation of the first node. A transition from this state only occurs when an initialisation of the considered error variable is encountered.

OK reflects that the current value of the error variable is the OK constant. Conceptually this state represents the absence of an exceptional condition.

Not-OK is the converse, i.e., the error variable is known to be anything but OK, though the exact value is not known. This state can be reached when a path has taken the true branch of a guard like `if(r != OK)`.

Unknown is the state reached if the result of a function call is assigned to the error variable. Due to our limitation to intra-procedural analysis, we conservatively assume function call results to be unknown.

Constant is a parametrised state that contains the constant value assigned to the error variable. This state can be reached after the assignment of a literal constant value to the error variable.

All states also track the error value that was last written to the log file. This information is needed to detect faults in the logging of errors. Since we only perform intra-procedural analysis, we set the last logged v value to *unknown* in the case of an Unknown state (i.e., when a function was called). We thus assume that the called function adheres to the idiom, which allows us to verify each function in isolation. Faults in these called functions will still be detected when they are being checked.

While traversing paths of the control-flow graph of a function, the analysis caches results in order to prevent infinite traversals of loops and to improve efficiency by eliminating redundant computations. In particular, the state (including associated values of parameters) in which the SM reaches each node is stored. The analysis then makes sure that each node is visited at most once given a particular state. The same technique is used by Engler et al. (2000).

5.5.2 Example Faults

The following three examples show how the SM detects faults from each of the categories in the fault model. States reached by the SM are included in the examples as comments, and where appropriate the last logged error value is mentioned in parentheses. First, consider the code snippet in Figure 5.4.

```
1  int calibrate(int a) {      // Entry
2    int r = OK;               // OK
3    r = initialise();         // Unknown
4    if(a == 1)
5      LOG(RANGE_ERROR, OK);   // Reject
6    ...
7  }
```

Figure 5.4: Example of fault category 1.

A fault of category 1 possibly occurs on the path that takes the true branch of the `if` statement on line 4. If the function call at line 3 returns with an error value, say `INIT_ERROR` then receive(`INIT_ERROR`) holds. The call to the LOG function on line 5 makes LOG(`RANGE_ERROR`, `OK`) true, and since OK is different from `INIT_ERROR`, all clauses of the predicate for category 1 are true, resulting in a fault of category 1.

SMELL detects this fault as follows, starting in the Entry state on line 1. The initialisation of `r`, which has been identified as an error variable, causes a transition to the OK state on line 2. The assignment to `r` of the function call result on line 3 results in the Unknown state. On the true branch of the `if` statement on line 4, a (new) root error is raised. The cause of the fault lies here. SMELL reaches the Reject state at line 5 because if an error value (say `INIT_ERROR`) would have been returned from the call to the `initialise` function, it is required to link the `RANGE_ERROR` to the `INIT_ERROR`, instead of linking to OK.

```
1  int align() {             // Entry
2    int r = OK;             // OK
3    r = initialise();       // Unknown
4    if(r != OK)             // Not-OK
5      LOG(ALIGN_ERROR, r); // Not-OK (ALIGN_ERROR)
6    return r;               // Reject
7  }
```

Figure 5.5: Example of fault category 2.

The function in Figure 5.5 exhibits a fault of category 2 on the path that takes the true branch of the `if` statement. Again, suppose receive(`INIT_ERROR`) holds, then the function correctly performs LOG(`ALIGN_ERROR`, `INIT_ERROR`). The fault consists of the function not returning `ALIGN_ERROR`, but `INIT_ERROR`, because after linking to the received error, the new error value is not assigned to the error variable.

Again SMELL starts in the Entry state, and subsequently reaches the OK state after the initialisation of the error variable `r`. The `initialise` function is called at line 3, and causes SMELL to enter the Unknown state. Taking the true branch at line 4 implies that the value of `r`

must be different from OK, and SMELL records this by changing to the Not-OK state. At line 5 an ALIGN_ERROR is linked to the error value currently stored in the r variable. SMELL then reaches the return statement, which causes the error value to be returned that was returned from the initialise function call at line 3. Since the returned value differs from the logged value at this point, SMELL transits to the Reject state.

```
1  int process(int a) {    // Entry
2    int r = OK;           // OK
3    r = initialise();     // Unknown
4    if(a == 2) {
5      r = PROCESS_ERROR; // Reject
6    }
7    ...
8    return r;
9  }
```

Figure 5.6: Example of fault category 3.

Category 3 faults are similar to category 2, but without any logging taking place. Suppose again that for the function in Figure 5.6 receive(INIT_ERROR) holds. For the path taking the true branch of the if statement a value different from INIT_ERROR will be returned, i.e., PROCESS_ERROR.

Until the assignment at line 5 the SM traverses through the same sequence of states as for the previous examples. However, the assignment at line 5 puts SMELL in the Reject state, because the previously received error value has been overwritten. A category 3 fault is therefore manifest.

5.5.3 Fault Reporting

SMELL reports the presence of faults using a more fine-grained view of the source code than the fault model. While the fault model takes a black box perspective, i.e., regarding behaviour only at the interface level, SMELL reports detected faults using a white box perspective, i.e., considering the implementation level details of a function. The white box perspective is considered to be more useful when interpreting actual fault reports, which developers may have to process.

In the following we present a list of "low-level faults", or programmer mistakes that SMELL reports to its users. For each programmer mistake we mention here the associated fault categories from the fault model. SMELL itself does not report these categories to the user. To help users interpreting the reported faults, SMELL prints the control-flow path leading up to the fault, and the associated state transitions of the SM.

function does not return occurs when a function declares and uses an error variable (i.e., assigns a value to it), but does not return its value. If present, SMELL detects this fault at the return statement of the function under consideration. This can cause category 2 or 3 faults.

wrong error variable returned occurs when a function declares and uses an error variable but returns another variable, or when it defines multiple error variables, but only returns one of them and does not link the others to the returned one in the appropriate way. This can cause category 2 or 3 faults.

assigned and logged value mismatch occurs when the error value that is returned by a function is not equal to the value last logged by that function. This can cause category 2 faults.

not linked to previous value occurs when a `LOG` call is used to link an error value to a previous value, but this latter value was not the one that was previously logged. If present, SMELL detects this fault at the call site of the log function. This causes category 1 faults.

unsafe assignment occurs when an assignment to an error variable overwrites a previously received error value, while the previous error value has not yet been logged. Clearly, if present SMELL detects this fault at the assignment that overwrites the previous error value.

5.5.4 Limitations

Our approach is both formally unsound and incomplete, which is to say that our analysis proves neither the absence nor the presence of 'true' faults. In other words, both false negatives (missed faults) or false positives (false alarms) are possible. False negatives for example occur when SMELL detects a fault on a particular control-flow path, and stops traversing that path. Consequently, faults occurring later in the path will go unnoticed. The unsoundness property and incompleteness properties do not necessarily harm the usefulness of our tool, given that the tool still allows us to detect a large number of faults that may cause much machine down-time, and that the number of false positives remains manageable. The experimental results (see Section 5.6) show that we are currently within acceptable margins.

SMELL also exhibits a number of other limitations:

Meta assignments Meta assignments are assignments involving two different error variables, such as `r = r2;`. SMELL does not know how to deal with such statements, since it traverses the control-flow paths for each error variable separately. As a result, when considering the `r` variable, SMELL does not know what the current value of `r2` is, and vice versa.

For the moment, SMELL recognises such statements and simply stops traversing the current control-flow path.

Variableless log calls Variableless log calls are calls to the `LOG` function that do not use an error variable as one of their actual arguments, but instead only use constants, such as in the following example:

```
r = PARAM_ERROR;
LOG(PARAM_ERROR,OK);
```

The problem appears when a function defines more than one error variable. Although a developer is able to tell which error variable is considered from the context of the call, SMELL has trouble associating the call to a specific error variable.

	reported	false positives	limitations	validated
CC3 (3 kLoC)	32	2	4	26 (13)
CC10 (19 kLoC)	72	20	22	30
CC11 (15 kLoC)	16	0	3	13
CC2 (14.5 kLoC)	107	14	13	80
CC1 (15 kLoC)	9	1	3	5
total (66.5 kLoC)	236	37	45	154 (141)

Table 5.1: Reported number of faults by SMELL for five components.

Whenever possible, SMELL tries to recover from such calls intelligently. In the above case, SMELL is able to infer that the log call belongs to the `r` variable, because it logs the constant that is assigned to that variable. However, the problem reappears when a second error variable is considered. When checking that variable and encountering the `LOG` call, SMELL will report an error if the error value contained in the second error variable differs from the logged value, because it does not know the `LOG` call belongs to a different error variable.

Infeasible Paths Infeasible paths are paths through the control-flow graph that can never occur at runtime, but that are considered as valid paths by SMELL. SMELL only considers the values for error variables, and smartly handles guards involving those variables. But it does not consider any other variables, and as such cannot infer, for example, that certain conditions using other variables are in fact mutually exclusive.

Wrong Error Variable Identification The heuristic SMELL uses to identify error variables is not perfect. False positives occur when integer values are used to catch return values from library functions, for example, such as `puts` or `printf`. Additionally, false negatives occur when developers pass the error variable as an actual or perform some arithmetic operations on it. This is not allowed by the ASML coding standard, however. Currently, false positives are easily identified manually, since SMELL's output reports which error variable was considered. If this error variable is meaningless, inspection of the fault can safely be skipped.

In order to overcome these limitations, we are currently reimplementing SMELL, using *path-identification* and *constant propagation* algorithms involving all (local) variables of a function. This will solve the problems of variableless log calls and meta assignments, because we will approximate the values of all variables, and reduce the problem of infeasible paths. We believe this new implementation will also eliminate the need for explicitly identifying the error variables, but this remains to be investigated.

5.6 Experimental Results

5.6.1 General Remarks

Table 5.1 presents the results of applying SMELL on 5 relatively small ASML components. The first column lists the component that was considered together with its size, column 2 lists the number of faults reported by SMELL, column 3 contains the number of false positives we manually identified among the reported faults, column 4 shows the number of SMELL limita-

tions (as discussed in the previous section) that are encountered and automatically recognised, and finally column 5 contains the number of validated faults, or 'true' faults.

Four of the five components are approximately of the same size, but there is a striking difference between the numbers of *reported* faults. The number of reported faults for the CC11 and CC1 components is much smaller than those reported for the CC10 and CC2 components. When comparing the number of *validated* faults, the CC2 component clearly stands out, whereas the number for the other three components is approximately within the same range.

Although the CC3 component is the smallest one, its number of validated faults is large compared to the larger components. This is due to the fact that a heavily-used macro in the CC3 component contains a fault. Since SMELL is run after macro expansion, a fault in a single macro is reported at every location where that macro is used. In this case, only 13 faults need to be corrected (as indicated between parenthesis), since the macro with the fault is used in 14 places.

The number of validated faults reported for the CC1 component is also interestingly low. This component is developed by the same people responsible for the EHM implementation. As it turns out, even these people violate the idiom from time to time, which shows that the idiom approach is difficult to adhere to. However, it is clear that the CC1 code is of better quality than the other code.

Overall, we get 236 reported faults, of which 45 (19 %) are reported by SMELL as a limitation. The remaining 191 faults were inspected manually, and we identified 37 false positives (16 % of reported faults). Of the remaining 154 faults, 141 are unique, and so in other words, we found 2.1 true faults per thousand lines of code.

5.6.2 Fault Distribution

A closer look at the 141 validated faults shows that 13 faults are due to a function not returning, 28 due to the wrong error variable being returned, 54 due to unsafe assignments, 10 due to incorrect logging, and 36 due to an assigned and logged value mismatch.

The *unsafe assignment* fault occurs when the error variable contains an error value that is subsequently overwritten. This kind of fault is by far the one that occurs the most (54 out of 141 = 38%), followed by the *assigned and logged value mismatch* (36 out of 141 = 26%). If we want to minimise the exception handling faults, we should develop an alternative solution that deals with these two kinds of faults.

Accidental overwriting of the error value typically occurs because the control flow transfer when the exception is raised is not implemented correctly. This is mostly due to a forgotten guard that involves the error variable ensuring that normal operation only continues when no exception has been reported previously. An example of such a fault is found in Figure 5.6.

The second kind of fault occurs in two different situations. First, as exemplified in Figure 5.5, when a function is called and an exception is received, a developer might link an exception to the received one, but forgets to assign the linked exception to the error variable. Second, when a root error is detected and a developer assigns the appropriate error value to the error variable, he might forget to log that value.

5.6.3 False positives

The number of false positives is sufficiently low to make SMELL useful in practice. A detailed look at these false positives reveals the reasons why they occur and allows us to identify where we can improve SMELL.

Of the 37 false positives identified, 23 are due to an incorrect identification of the error variable, 7 are due to SMELL getting confused when multiple error variables are used, 4 occur because an infeasible path has been followed, and 3 false positives occur due to some other (mostly domain-specific) reason.

These numbers indicate that the largest gain can be obtained by improving the error variable identification algorithm, for example by trying to distinguish ASML error variables from "ordinary" error variables. Additionally, they show that the issue of infeasible paths is not really a large problem in practice.

5.7 An Alternative Exception Handling Approach

In order to reduce the number of faults in exception handling code, alternative approaches to exception handling should be studied. A solution which introduces a number of simple macros has been proposed by ASML, and we will discuss it here. We thereby keep in mind that we know that the two most frequently occurring faults are overwriting of the error value and the mismatch between the value assigned to the error variable and the value actually logged.

The solution is based on two observations.

First, it encourages developers to no longer write assignments to the error variable explicitly, and it manages them automatically inside the macros. Such assignments can either be constant assignments, when declaring a root error, or function-call assignments, when calling a function. By embedding such assignments inside specific macros and surrounding them with appropriate guards, we can prevent accidental overriding of error values.

Second, the macros ensure that assignments are accompanied by the appropriate `LOG` calls, in order to avoid a mismatch between logged and assigned values. As explained in the previous section, such a mismatch occurs when declaring a root error or when linking to a received error. Consequently, we introduce a `ROOT_LOG` and `LINK_LOG` macro that should be used in those situations and that take care of all the work.

The proposed macro's are defined in Figure 5.7. The `ROOT_LOG` macro should be used whenever a root error is detected, while the `LINK_LOG` macro is used when calling a function and additional information can be provided when an error is detected. Additionally, a `NO_LOG` macro is introduced that should be used when calling a function and not linking extra information if something goes wrong.

Using these macros, the example code from Section 5.3 is changed into the code that can be seen in Figure 5.8.

It is interesting to observe that using these macros drastically reduces the number of (programmer visible) control-flow branches. This not only improves the function's understandability and maintainability, but also causes a significant drop in code size, if we consider that the return code idiom is omnipresent in the ASML code base. Moreover, the exception han-

```
#define ROOT_LOG(error_value, error_var)\
      error_var = error_value;\
      LOG(error_value, OK);\

#define LINK_LOG(function_call, error_value, error_var)\
   if(error_var == OK) {\
      int _internal_error_var = function_call;\
      if(_internal_error_var != OK) {\
         LOG(error_value, _internal_error_var);\
         error_var = error_value;\
      }\
   }

#define NO_LOG(function_call, error_var)\
   if(error_var == OK) \
      error_var = function_call;
```

Figure 5.7: Definitions of proposed exception handling macro's.

dling code is separated from the ordinary code, which allows the two to evolve separately. More research is needed to study these advantages in detail.

The solution still exhibits a number of drawbacks.

First of all, the code that cleans up memory resources remains as is. This is partly due to the fact that we did not focus on such code, since we postulate that it belongs to a different concern. However, such code also differs significantly between different functions and source components, which makes it harder to capture it into a set of appropriate macros.

Second, reliability checks are still not available. It remains the developer's responsibility to use the macros and use them correctly. Developers can still explicitly assign something to the error variable, without using the correct macro, for example. Or, they can still raise exceptions that should not be raised by a particular function.

Last, the macros do not tackle faults that concern the returning of the appropriate error value. Since this was a deliberate choice, because such errors are rather scarce and can be easily found, this comes as no surprise.

```
int f(int a, int* b) {
   int r = OK;
   bool allocated = FALSE;
   r = mem_alloc(10, (int *)b);
   allocated = (r == OK);
   if((a < 0) || (a > 10))
       ROOT_LOG(PARAM_ERROR,r);
   LINK_LOG(g(a),LINKED_ERROR,r);
   NO_LOG(h(b), r);
   if((r != OK) && allocated)
      mem_free(b);
   return r;
}
```

Figure 5.8: Function `f` implemented by means of the alternative macros.

5.8 Discussion

In our examples, we found 2.1 deviations from the return code idiom per 1000 lines of code. In this section, we discuss some of the implications of this figure, looking at questions such as the following: How does the figure relate to reported defect densities in other systems? What, if anything, does the figure imply for system reliability? What does the figure teach us on idiom and coding standard design?

5.8.1 Representativeness

A first question to be asked is to what extent our findings are representative for other systems.

The software under study has the following characteristics:

- It is part of an embedded system in which proper exception handling is essential.
- Exception handling is implemented using the return code idiom, which is common for C applications.
- Before release, the software components in question are subjected to a thorough code review.
- The software is subjected to rigorous unit, integration, and system tests.

In other words, we believe our findings hold for software that is the result of a state-of-the-art development process and that uses an exception handling mechanism similar to the one we considered.

The reason so many exception handling faults occur is that current ways of working are not effective in finding such faults: tool support is inadequate, regular reviews tend to be focused on "good weather behaviour" — and even if they are aimed at exception handling faults these are too hard to find, and testing exception handling is notoriously hard.

5.8.2 Defect Density

What meaning should we assign to the value of 2.1 exception handling faults per 1000 lines of code (kLoC) we detected?

It is tempting to compare the figure to reported defect densities. For example, an often cited paper reports a defect density between 5 and 10 per kLoC for software developed in the USA and Europe (Dyer, 1992). More recently, in his ICSE 2005 state-of-the-art report, Littlewood states that studies show around 30 faults per kLoC for commercial systems Littlewood (2005).

There are, however, several reasons why making such comparisons is questionable, as argued, for example, by Fenton and Pfleeger (1997). First, there is neither consensus on what constitutes a defect, nor on the best way to measure software size in a consistent and comparable way. In addition to that, defect density is a product measure that is derived from the process of finding defects. Thus, "defect density may tell us more about the quality of the defect finding and reporting process than about the quality of the product itself" (Fenton and

Pfleeger, 1997, p.346). This particularly applies to our setting, in which we have adopted a new way to search for faults.

The consequence of this is that no conclusive statement on the relative defect density of the system under study can be made. We cannot even say that our system is of poorer quality than another with a lower reported density, as long as we do not know whether the search for defects included a hunt for idiom errors similar to our approach.

What we can say, however, is that a serious attempt to determine defect densities should include an analysis of the faults that may arise from idioms used for dealing with crosscutting concerns. Such an analysis may also help when attempting to explain observed defect densities for particular systems.

5.8.3 Reliability

We presently do not know what the likelihood is that an exception handling fault actually leads to a failure, such as an unnecessary halt, an erroneously logged error value, or the activation of the wrong exception handler. As already observed by Adams (1984), more faults need not lead to more failures. We are presently investigating historical system data to clarify the relation between exception handling faults and their corresponding failures. This, however, is a time consuming analysis requiring substantial domain knowledge in order to understand a problem report, the fault identified for it (which may have to be derived from the fix applied) and to see their relation to the exception handling idiom.

5.8.4 Idiom design

The research we are presenting is part of a larger, ongoing effort in which we are investigating the impact of crosscutting concerns on embedded C code (see Chapters 2 and 3). The traditional way of dealing with such concerns is by devising an appropriate coding idiom. What implications do our findings have on the way we actually design such coding idioms?

One finding is that an idiom making it too easy to make small mistakes can lead to many faults spread all over the system. For that reason, idiom design should include the step of constructing an explicit fault model, describing what can go wrong when using the idiom. This will not only help in avoiding such errors, but may also lead to a revised design in which the likelihood of certain types of errors is reduced.

A second lesson to be drawn is that the possibility to check idiom usage automatically should be taken into account: static checking should be designed into the idiom. As we have seen, this may require complex analysis at the level of the program dependence graph as opposed to the (elementary) abstract syntax tree.

5.9 Concluding Remarks

Contributions

Our contributions are summarised as follows. First, we provided empirical data about the use of an exception handling mechanism based on the return code idiom in an industrial setting. This data shows that the idiom is particularly error prone, due to the fact that it is omnipresent

as well as highly tangled, and requires focused and well-thought programming. Second, we defined a series of steps to regain control over this situation, and answer the specific questions we raised in the introduction. These steps consist of the characterisation of the return code idiom in terms of an existing model for exception handling mechanisms, the construction of a fault model which explains when a fault occurs in the most error prone components of the characterisation, the implementation of a static checker tool which detects faults as predicted by the fault model, and the introduction of an alternative solution, based on experimental findings, which is believed to remove the faults most occurring.

We feel these contributions are not only a first step toward a reliability check component for the return code idiom, but also provide a good basis for (re)considering exception handling approaches when working with programming languages without proper exception handling support. We showed that when designing such idiom-based solutions, a corresponding fault model is a necessity to assess the fault-proneness, and the possibility of static checking should be seriously considered.

Future work

There are several ways in which our work can be continued:

- apply SMELL to more ASML components, in order to perform more extensive validation. Additionally, some components already use the macros presented in Section 5.7, which allows us to compare the general approach to the alternative approach, and assess benefits and possible pitfalls in more detail. We initiated such efforts, and are currently analysing approximately two million lines of C code for this.
- apply SMELL to non-ASML systems, such as open-source systems, in order to generalise it and to present the results openly.
- apply SMELL to other exception handling mechanisms for C, such as those based on the `setjmp/longjmp` idiom, to analyse which approach is most suited.
- investigate aspect-oriented opportunities for exception handling, since benefits in terms of code quality can be expected if exception handling behaviour is completely separated from ordinary behaviour. Chapter 7 discusses the renovation of idiomatic expection handling using structured exception handling and aspects.

Chapter 6

Analysing Variability in Large-scale Idioms-based Implementations of Crosscutting Concerns*

This chapter describes a method for studying idioms-based implementations of crosscutting concerns, and our experiences with it in the context of the ASML C 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.

6.1 Introduction

The lack of certain languages features, such as aspects or exception handling, can cause developers to resort to the use of idioms[1] 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 Chapter 5. It is used in languages such as C to imple-

*This chapter was published, as "Simple Crosscutting Concerns Are Not So Simple – Analysing Variability in Large-Scale Idioms-Based Implementations", in the Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD 2007) (Bruntink et al., 2007). It is co-authored by Arie van Deursen, Maja D'Hondt and Tom Tourwé.

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

ment 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 (Buschmann et al., 1996; Coplien, 1991).

Using idioms can result in various forms of code duplication (see Chapter 2). 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 skills, changing requirements and idiom descriptions, and implementation errors.

In this chapter, 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 chapter, 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 in Chapters 2, 3 and 5. Several more are mentioned in the literature (Colyer and Clement, 2004; Coady et al., 2001). The questions we ask in this chapter 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.

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 chapter is as follows. The next section briefly presents our method for analysing variability by describing each individual step. Sections 6.3–6.7 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 6.8 then presents a discussion of the repercussions of these results and an evaluation of our method. Section 6.9 discusses related work and Section 6.10 presents our conclusions.

6.2 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.

6.2.1 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 Demeyer et al. (2003).

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.

6.2.2 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 (Weiser, 1984), 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.

6.2.3 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.

6.2.4 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) (Ganter and Wille, 1999). 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.

```
1  int f(chuck_id* a, scan_component b) {
2      int result = OK;
3      char* func_name = "f";
4      ...
5      trace(CC, TRACE_INT, func_name, "> (b = %s)",
6                                      SCAN_COMPONENT2STR(b));
7      ...
8      trace(CC, TRACE_INT, func_name, "< (a = %s) = %d",
9                                      CHUCK_ID_ENUM2STR(a), result);
10
11     return result;
12 }
```

Figure 6.1: Code fragment illustrating the tracing idiom at ASML.

6.2.5 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 Gabriel (1996). 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.

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 chapter.

6.3 Defining the Tracing Idiom

The idiom we study in this chapter 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 Chapters 2, 3 and 5, 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 chapter, 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 6.1. 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 6.1, `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 6.5.

6.4 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 (2007) and a code transformation tool, called the ASF+SDF Meta-Environment (van den Brand et al., 2001). 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.

6.5 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:

- 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 6.3, show that variability can occur at many different levels. In the remainder of this chapter, 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.

	CC4	CC5	CC6	CC7	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 6.1: Basic statistics of the analysed components.

At the parameter-level, the variability involves the different ways in which a parameter of a particular kind is traced. As explained in Section 6.3, 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.

6.6 Analysing the Tracing Idiom's Variability

As shown in Table 6.1, 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.[2]

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 [3] 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 6.2.4, 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

[2]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.

[3]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 than 249 rows.

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

Table 6.2: Example FCA context for function-level variability

as on all components together. This will allow us to discuss the variability within a single component, as well as the between different components.

6.6.1 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.

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 6.3. A sample context is shown in the upper part of Table 6.2. In that table, 'trace' and 'CC_TRACE' represent names for the tracing function, 'TRACE_INT' and 'TRACE_EXT' are possible boolean values that select internal or external tracing behavior, 'CC' is a name for the component in which tracing occurs, and 'func_name' and 'f_name' are names for the functions which invoke tracing.

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 6.3. 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 functions `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.

	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 6.3: Example FCA context for parameter-level variability

Applying FCA

Once the context is set up, the algorithm can be applied. We use Lindig's Concepts tool to compute the actual concepts (Lindig and Snelting, 1997). 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[4], 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 chapter are obtained by this tool.

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 6.2. 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

[4] http://conexp.sf.net

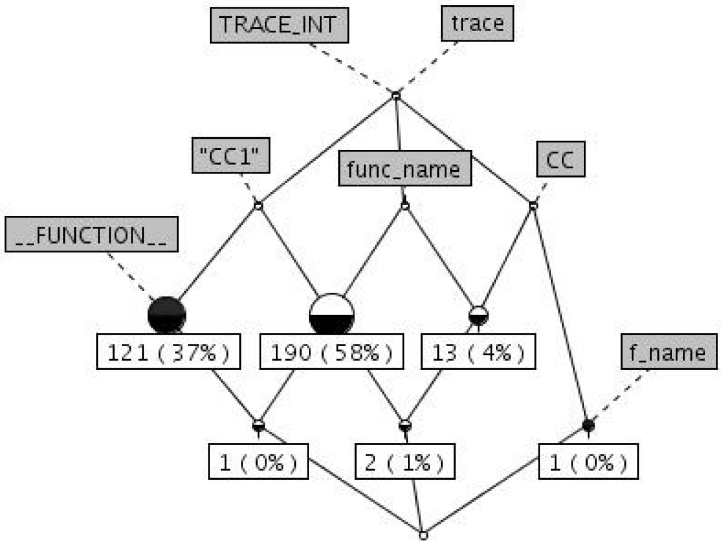

Figure 6.2: Function-level variability in the CC4 component

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 6.2, 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 6.2 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.

	CC4	CC5	CC6	CC7	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		5.7
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 6.4: Function-level and parameter-level variability results for 704 functions.

6.6.2 Function-level Variability

The upper half of Table 6.4 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 6.3. This immediately raises the question why developers do not adhere to the standard. Maybe a new standard for invoking tracing should be considered? Can we observe candidate standards in our results?

Looking at the second row in the upper half of Table 6.4, 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 6.2 and 6.3, it is clear that both components use a similar tracing variant implemented by most functions (190 or 58% functions in the case of CC4, 123 functions or 92% in the case of CC5). Additionally, CC4 has yet another "big" variant that uses the `__FUNCTION__` preprocessor token instead of the variable `func_name`. This variant is used in

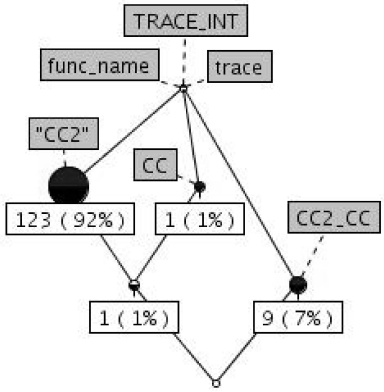

Figure 6.3: Function-level variability in the CC5 component

121 functions (37%).

Figures 6.5 and 6.4 show significantly different results. The CC7 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 CC6 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 CC6 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 CC6 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 CC4 component. A similar situation occurs in the CC5 component, as can be seen in Figure 6.3, where one function uses `CC` and `"CC2"`. The CC6 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.

6.6.3 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

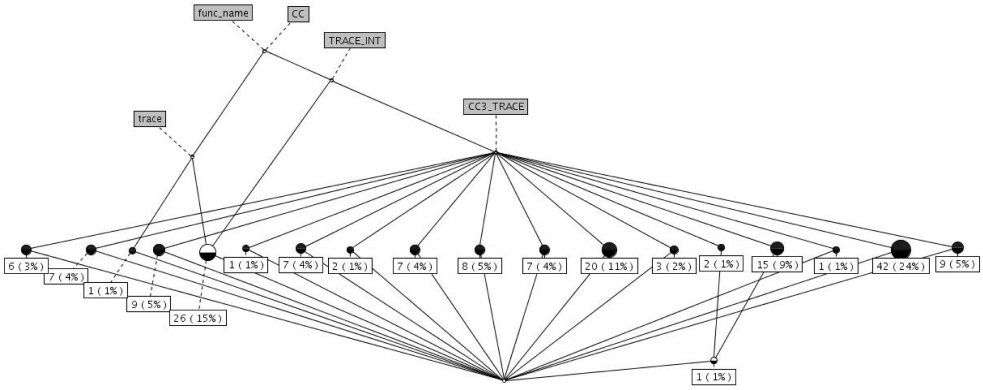

Figure 6.4: Function-level variability in the CC6 component

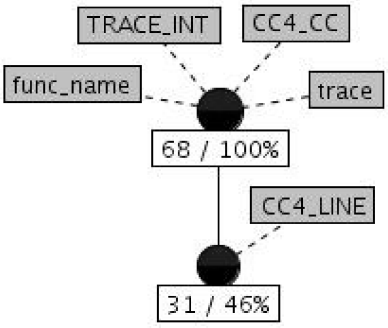

Figure 6.5: Function-level variability in the CC7 component

lattices for this part of experiment since the large number of parameter types generates too many concepts. Instead we produce statistics from the results of the FCA algorithm with our Scheme scripts.

The lower half of Table 6.4 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., they 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 CC4 and CC6 components appears to be less consistent than in the other two components. If we take into account the basic statistics from Table 6.1, this seems logical: CC4 and CC6 are by far the largest components. However, taking into account size does not explain everything: the CC5 component defines more types than CC6, but is more consistent.

Even when excluding types that are either traced consistently or not traced at all, the CC6 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:

```
1 CC3_TRACE( CC, TRACE_INT, func_name,
2    FILE_CONSTANT,
3    "< (p = [%p], *p = [%s])",
4         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 CC5 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 CC6 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.

6.7 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

chapter but is discussed briefly in Section 6.8.3. 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.

6.7.1 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* `f1, ..., fn`*, 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 6.2.5 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[5]. 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:

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

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

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

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.

6.7.2 Quantification of Parameters

Our experiments with respect to parameter-level variability show that complex parameter types require a converter function and that 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 (Kiczales et al., 1997), 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.

6.7.3 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 CC6 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.

6.8 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.

6.8.1 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.

6.8.2 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 Chapter 3, other work (Colyer and Clement, 2004), and again confirmed by the results in this chapter, 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 CC5: 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 6.7.

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 results 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. (2004).

6.8.3 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 (Brodie and Stonebraker, 1995), 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.

6.8.4 Variability Findings

Our results indicate that only 5.7% 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 Chapter 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 in Chapter 3, 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.

6.8.5 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 defini-

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

Table 6.5: 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 6.6: Context and relation sizes for all components considered together.

tion 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.

6.8.6 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) (Visser, 1997), followed by a single traversal of the parse tree to extract the relevant facts (see Section 6.3). Subsequently, Lindig's FCA tool *concepts*[6] is used to produce the concept lattices. Table 6.5 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 (Lindig, 2000). Table 6.6 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

[6] This tool is based on Lindig (2000).

is not required in our approach. The concepts of interest, i.e., those that contain own objects (see Section 6.6), are found automatically. The number of concepts containing own objects can be significantly lower, as can be seen in Table 6.4. 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.

6.9 Related Work

The work presented in this chapter 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 (Tourwé and Mens, 2004; Tonella and Ceccato, 2004). An overview of these techniques can be found in (Kellens et al., 2007; Marin et al., 2006).

Once identified, the crosscutting concerns can be refactored into an aspect. Several authors have proposed a process for such migration (Binkley et al., 2005; Monteiro and Fernandes, 2005; van Deursen et al., 2005). 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. van Deursen et al. (2005) 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. (2005) and van Deursen et al. (2005) present results of applying their process to JHotDraw, a medium-sized object-oriented software system, while Monteiro and Fernandes (2005) 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 Videira Lopes (2000) 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. (2001), 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.

Colyer and Clement (2004) also observe that variability is present in the idiomatic implementation of a tracing policy of a product line. 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 (1997), Lindig and Snelting (1997), van Deursen and Kuipers (1999b) 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 Siff and Reps (1997) and van Deursen and Kuipers (1999b) mention a problem that resembles the tangling of concerns and a solution to that problem. Siff and Reps (1997) refers to it as "tangled" code that uses multiple data structures at the same time, whereas van Deursen and Kuipers (1999b) 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 (Coady et al., 2001; Durr et al., 2006; AspectC++, 2007; Aspect-oriented C, 2007; Zaidman et al., 2006), 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.

6.10 Concluding Remarks

In this chapter, 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 chapter 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.

Chapter 7

Renovating Idiomatic Exception Handling*

Some legacy programming languages, e.g., C, do not provide adequate support for exception handling. As a result, users of these legacy programming languages often implement exception handling by applying an idiom. An idiomatic style of implementation has a number of drawbacks: applying idioms can be fault prone and requires significant effort. Modern programming languages provide support for Structured Exception Handling (SEH) that makes idioms largely obsolete. Additionally, Aspect-Oriented Programming (AOP) is believed to further reduce the effort of implementing exception handling. This chapter investigates the gains that can be achieved by reengineering the idiomatic exception handling of an ASML C component to these modern techniques. First, we will reengineer the legacy component such that its exception handling idioms are almost completely replaced by SEH constructs. Second, we will show that the use of AOP for exception handling can be beneficial, even though the benefits are limited by inconsistencies in the legacy implementation.

7.1 Introduction

Exception handling is a concern in any software system. Every (interesting) software system is dependent on a number of external inputs that it needs to respond to, and thus it needs to respond appropriately when those inputs do not fall within the expected range. Most modern programming languages provide facilities that support the handling of exceptional situations. For instance, Java, C++ and C# (among others) provide the common *try* and *catch* constructs that enable a programmer to attach handling code to blocks of code that potentially give rise to exceptional situations. The runtime systems of these languages also take care of

*This chapter will appear in the Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR 2008) in April 2008.

the propagation of exceptions (representations of the occurrence of an exceptional situation) throughout the software system.

However, some older programming languages, such as C, do not provide such support by default. Instead, the programmer has to use the general purpose constructs of the language to program the, often complicated, control flow required to handle exceptions. Furthermore, the programmer must make sure all relevant components of the software system become aware of the exception.

An *idiom* is sometimes used to guide programmers during this task. Idioms are common solutions (like code templates or examples) to commonly occurring programming problems, and are often found in manuals, text books or training documents. In Chapter 5 we studied the so-called return-code idiom that is used to implement exception handling within a large C software system. We found that such an idiomatic style of implementation can result in a faulty exception handling system.

This chapter reports on work following up on those findings. We aim at investigating what can be done to improve legacy software that implements its exception handling using a return code idiom. First, we are interested to know what improvements can be offered by the Structured Exception Handling (SEH) (Goodenough, 1975) (e.g., *try* and *catch*) constructs that are available in many modern languages. Second, the use of *Aspect-Oriented Programming* (AOP) (Kiczales et al., 1997) may bring some additional benefits, as was shown by several researchers in the AOP community (Lippert and Videira Lopes, 2000; Filho et al., 2007, 2006). In summary, the research questions this chapter attempts to answer are the following:

1. Is it technically feasible to replace a return code idiom by SEH constructs?

2. What are the advantages of using exception handling compared to using a return code idiom?

3. What gains can be expected from reengineering SEH code to an aspect-oriented solution?

The exception handling idiom we will study in this chapter is in use at ASML, a manufacturer of lithography solutions based in Veldhoven, the Netherlands. It occurs throughout a control system that is estimated to consist of around 15 million lines of C code. Given this context, we approach our research questions by analyzing the benefits of a reengineering of an ASML component. The component CC10 has been studied before in Chapter 5, and was shown to have a faulty exception handling implementation. It consists of 11,134 NLOC.[1] The analysis will consist of two steps. First, we semi-automatically reengineer the idiomatic exception handling code in the CC10 component to obtain a new implementation using SEH constructs. Second, we perform an analysis of the benefits offered by reengineering the SEH implementation using aspect-oriented programming.

Whether the ASML return code idiom can be replaced by SEH constructs is not immediately clear. This depends on the idiom definition and the actual applications of the idiom in the code. In Section 7.2 we will present the ASML return code idiom in detail. A proposal to map the idiom to SEH constructs, and the actual reengineering based on this mapping will

[1] Normalized Lines Of Code (NLOC) is defined as non-comment, non-blank, `indent` K&R style line count.

be discussed in Section 7.3. Issues that surfaced during reengineering will also be discussed in that section. At this point we will present data that enables a comparison between the idiom-based implementation and SEH constructs. The final step will consist of analyzing the benefits that can be expected if the SEH-based solution would be reengineered using AOP. The results of this analysis will be presented in Section 7.4. Further analysis and related work will be discussed in Sections 7.5 and 7.6, respectively.

7.2 Idiomatic Exception Handling

7.2.1 Context

We will study an exception handling idiom in the context of ASML, a manufacturer of lithography solutions in Veldhoven, the Netherlands. The exception handling idiom occurs in a control system of wafer scanner machines. The estimated size of this control system is around 15 million lines of C code.

Before we turn to the idiom itself, we detail the requirements that the idiom is meant to implement. The ASML manual prescribes the following requirements for its exception handling implementation:

1. Every function that detects an exception must
   (a) log the exception, and
   (b) recover the exception or pass it to the calling function.
2. Every function that receives an exception from a called function must
   (a) add any potentially useful context information by logging an exception linked to the received exception, and
   (b) recover the exception or pass it to the calling function.

Requirement 1 specifies the desired behavior when an exception is *detected* (and no exceptional situation existed before). Logging an exception consists of calling a predefined log macro that writes the exception (and a message string) to the exception log file. We will refer to this kind of exceptions as *root* exceptions.

Requirement 2 specifies the desired behavior in case an exception is *received* from a called function. If some useful context information is available when the exception is received, programmers can provide this by calling the log macro with an appropriate message string. Furthermore, they must *link* exceptions using the log macro. The first two parameters of the log macro are the received and a *linked* exception. The linked exception can be used to indicate that the scope of the exception situation has expanded. For instance, a 'file open failure' exception is received, and by calling the log macro an 'I/O error' is linked to it. Ultimately, this mechanism should result in a *chain* of exceptions that starts at the root exception and ends at whatever exception was linked to the chain last. This chain of exceptions should provide enough information about the problem that caused the exception(s) such that problem diagnosis is possible.[2]

[2]Java provides a similar facility by chaining exceptions.

```
1  int f(int a, int b) {
2    int ev = OK;
3
4    if (a < 0) {
5      LOG(F_ERROR, OK, "a < 0");
6      ev = F_ERROR;
7    }
8
9    if (ev == OK) {
10     ev = g(a);
11
12     if (ev != OK) {
13       LOG(F_ERROR, ev, "error from g");
14       ev = F_ERROR;
15     }
16   }
17
18   return ev;
19 }
```

Figure 7.1: Return code idiom used at ASML.

7.2.2 Return Code Idiom (RCI)

The exception handling requirements are implemented using an idiom, that is, programmers are trained to implement exception handling by learning code templates or examples described in the system manuals. The exception handling idiom in use at ASML is an instance of the return code idiom (RCI) as described in Chapter 5. In short, each C function uses an integer return code to report whether or not an error occurred during its execution. Received return codes are propagated manually along the control flow of a function, and are typically held in one or more local (integer) variables. We will refer to such variables as *exception variables*. Explicit guards check the exception variables to make sure that no functional code is executed in case an exception has occurred. At appropriate times the log macro is used to log the current exception, and link to any exception that occurred previously. An example of the use of this idiom is given in Figure 7.1.

The function `f` defines a variable `ev` at line 2 to hold return codes. At line 4, a condition checks whether something exceptional has occurred, and if so, a call to the log macro is done at line 5, and a return code is saved in the `ev` variable. The control flow is such that this value is returned at line 18 (assume no recovery can be done at this point), satisfying requirements 1 and 2 (the exception is logged and returned to the caller). If `ev` is still OK at line 9, function `g` is called and its return code is saved in `ev`. In case the call to `g` returned an exception, lines 12–15 link an exception to the one received from `g`. This is done by passing in the old exception (stored in `ev`) as the second argument of the `LOG` call at line 13, while the new exception (`F_ERROR`) is the first argument. The new exception is stored in `ev` at line 14 and will be returned at line 18.

It is clear that the consistent application of this idiom results in a large amount of 'boilerplate' code. Furthermore, the manual application of this idiom may be fault prone. In Chapter 5 we showed –using a static analysis tool– that per 1,000 lines of ASML code, 2 faults occur in the exception handling implementation that might cause a violation of the requirements.

7.2.3 Tool support for the RCI

The SMELL tool described in detail in Chapter 5 can be used to detect faults in the RCI exception handling implementation of ASML. SMELL consists of an encoding of the RCI as a state machine and can detect when the RCI is being violated by the code. As an extension, SMELL can visualize those pieces of code that are executed on exceptional paths, that is, paths through the code which are taken in case of an exception. During its execution, SMELL keeps track of the nodes (in a program dependence graph) that it visited in each of its states. Subsequently, it delivers these nodes at the end of its execution, and the user can choose to highlight those nodes in a code browser (Grammatech's CodeSurfer).

The analysis used in this tool is imprecise, i.e., some of the reported faults may be false alarms (about 20% in the case study presented in Chapter 5). We checked the reported faults with an ASML developer to obtain more precise results, fixed the real faults in the code, and improved the implementation of SMELL based on this feedback.

7.2.4 Renovation of Exception Handling

Chapter 5 showed that the use of the RCI for the implementation of the exception handling concern is effort intensive and can result in implementation faults. In this chapter we therefore study the renovation of the RCI using modern language technology, in particular Structured Exception Handling (SEH) and Aspect-Oriented Programming (AOP). The renovation process we will consider consists of two reengineering steps. These steps implement part of the approach that was described in Chapter 3 (see Figure 3.1).

First, Section 7.3 will describe the reengineering of the RCI code of an ASML component. We will semi-automatically *eliminate* the RCI code from the ASML component using the ASF+SDF Meta-Environment (van den Brand et al., 2001), and replace it by SEH constructs. In this step, the SMELL tool will perform the role of *concern verifier*. Second, Section 7.4 will analyze the further reengineering of the ASML component using aspects. We will analyze how aspects can be *extracted*, and the benefits that can be expected from the extraced aspects. We will not consider the *aspect weaving* and *translation* phases in this chapter.

7.3 Reengineering to Structured Exception Handling

Is it technically feasible to replace a return code idiom by structured exception handling?

We consider the reengineering of a single ASML C component that we will refer to as CC10. Table 7.1 shows the basic size measures of this component. Table 7.2 shows the

NLOC	11,134
files	12
functions	172

Table 7.1: Key figures for the CC10 component.

`ev == OK`	765	`ev != OK`	195
`ev = ERR;`	485	`ev = func();`	386
`int ev = OK`	157		

Table 7.2: Number of constructs in the RCI implementation.

numbers of RCI constructs that are present in the CC10 component. The identifier `ev` in this table refers to all possible exception variables that occur, i.e., all exception variables that are named according to a (manually determined) set of names for exception variables. `ev == OK` then represents the conditions that guard code based on the exception state. `ev != OK` are conditions that essentially guard handling code. Conditions that check for a specific exception state, e.g., `ev == MEM_ERR` do not occur in the CC10 component, and thus we do not consider those kinds of checks. `ev = ERR;` are assignment statements that record the occurrence of an exception. `ev = func();` are assignment statements that save exception states as they are returned by functions. `int ev = OK` represents declarations of exception variables (these occur one-to-one with return statements that use only an exception variable). `ev` is the count of all other uses (i.e., reads) of an variable.

The objective of the reengineering is to replace the RCI implementation of the exception handling in the CC10 component by a functionally identical SEH implementation. After the reengineering we will compare the resulting SEH implementation to the figures in Tables 7.1 and 7.2.

7.3.1 Structured Exception Handling (SEH)

SEH constructs were proposed to alleviate the programmer from the discipline required to apply idioms like the return code idiom (Goodenough, 1975). Figure 7.2 shows how the RCI example of Figure 7.1 could be expressed using the SEH constructs provided by the XXL library (Messier and Viega, 2007). The SEH model described in Lang and Stewart (1998) categorizes the XXL library as follows:

- **Semantics:** Termination semantics. The remainder of a scope is skipped after an exception is raised. The next statement executed is the one syntactically following the active handling block (if present).
- **Representation:** An exception is represented by an integer constant.
- **Handler determination:** Stack unwinding is performed until a matching handler is found (the C library calls `setjmp` and `longjmp` are used).
- **Handler scope:** Block scope. A handler can be attached to a block of code.

The following constructs (macros) are provided (we present only a subset here, using simplified syntax):

`TRY { $Block } $Handlers`
: Specifies a block of code to which `$Handlers` are bound. Handlers are specified by `CATCH`, `EXCEPT` or `FINALLY` constructs.

`CATCH($Int) { $Block }`
: Attaches a handler to the occurrence of the exception specified by integer constant `$Int`.

`EXCEPT { $Block }`
: Attaches a handler to the occurrence of any exception.

`FINALLY { $Block }`
: Specifies that `$Block` will be executed whether or not an exception has occurred.

`EXCEPTION_CODE()`
: Returns the current exception code (an integer).

```
1  void f(int a, int b) {
2
3    if (a < 0) {
4      LOG(F_ERROR, OK, "a < 0");
5      THROW(F_ERROR);
6    }
7
8    TRY {
9      g(a);
10   }
11   EXCEPT {
12     LOG(F_ERROR, EXCEPTION_CODE(), ...
13     THROW(F_ERROR);
14   }
15 }
```

Figure 7.2: Structured exception handling example.

Note that the `ev` variable that was used in Figure 7.1 has completely disappeared. Instead, in Figure 7.2 the exception state is managed by the SEH system. At line 5, a `THROW` statement is used to signal the occurrence of an exception. This exception will automatically propagate up the call stack until it reaches an appropriate handler block. Lines 8–14 show the definition of a handler block for (all) exceptions that are thrown within the call to `g`. To mimic the behavior implemented in Figure 7.1, the exception received from `g` is retrieved by the `EXCEPTION_CODE()` macro and linked to a new exception (`F_ERROR`) at line 12. This new exception is thrown at line 13.

Note that explicit condition statements such as the one at line 9 in Figure 7.1 are no longer necessary. The SEH system makes sure that no code is executed except for the handlers that have been defined for that particular exception. In general, the exceptional control flow is now handled by the SEH system, relieving the programmer from this task. Therefore, we expect a small reduction in code size when refactoring the RCI idiom into SEH constructs. More importantly, the SEH solution will likely have significantly reduced fault proneness compared to the RCI.

7.3.2 Code Transformations

Implementing the ASML exception handling requirements (see Section 7.2) using SEH constructs is straightforward. Requirements 1 and 2 (root exceptions) are dealt with by calling the logging facility and subsequently throwing an exception every time an exceptional situation is detected. Requirements 3 and 4 (linked exceptions) are satisfied by wrapping a function call in a `TRY` block, and calling the logging facility in an attached `EXCEPT` block, and `THROW`-ing a new exception if appropriate. This solution is shown in Figure 7.2.

The next question is: how do we map the legacy code that uses the RCI to the SEH solution? The RCI revolves around the use of one (or more) exception variables. The objective of the reengineering is to completely remove all uses of these variables, and replace them with SEH constructs where appropriate.

The following five steps are applied to transform legacy RCI code into SEH:

1. Transform assignments,
2. Transform conditions,
3. Transform other uses of error variables,
4. Remove exception variable returns,
5. Remove exception variable declarations.

Transform assignments. Assignments of the results of a function call to an exception variable are no longer necessary since the SEH library will keep track of exceptions.

```
ev = $FunctionCall;   ⟹   $FunctionCall;
```

The assignment of an (integer) constant to an exception variable is transformed into a `THROW` of the same constant.

```
ev = $Int;   ⟹   THROW ($Int);
```

This step can result in functionally different code in case in the RCI implementation functional code (in contrast with exceptional code) is executed after an assignment to an exception variable. Such assignments will be transformed into either `THROW` statements or `TRY` and `EXCEPT` blocks, changing control flow. Compare the examples in Figures 7.1 and 7.2. If line 6 is executed in the RCI case, the `if` statement at line 9 makes sure that the remainder of the function is not executed. In the SEH case, this happens automatically.

Now consider the case that lines 5 and 6 were swapped in Figure 7.1. Simply transforming an assignment to an exception variable into a `THROW` statement results in a problem. Since

lines 4 and 5 in Figure 7.2 would also be swapped, the THROW would happen before the logging call. In effect, the required logging call would never be executed in that case because the THROW statement skips the remainder of the function.

We now consider the general case. The RCI raises exceptions by assigning an integer constant to an exception variable. An exception is handled by a block of code that has a condition which checks for the occurrence of the exception. Consider all the execution paths that exist between an assignment that raises an exception and its possible handlers. If any function code code (i.e., code that is not part of the RCI) is executed on any of these paths, then a problem may occur if we translate the assignment into a THROW statement. The THROW statement will cause the immediate transfer of control to a handler, skipping any code that would have been executed in the RCI case.

In case of the example it is easy to fix the problem: simply move the logging call before the THROW statement. Unfortunately this fix does not work in general since code can be moved only if no data or control depedences will be changed. While reengineering the CC10 component we encountered only cases that were easy to fix, in particular instances of the logging example.

During the reengineering process, the SMELL tool (Subsection 7.2.3) was used to help ensure that no functional code is executed on exceptional control-flow paths, which would give rise to problems if a THROW statement is used to pass control to an exception handler.

Transform conditions. Conditions that check whether an exception has occurred come in two flavors. First, conditions like `ev == OK` are meant to prevent the execution of its 'then' block in case an exception has occurred. These conditions are no longer needed since this behavior is automatically provided by the SEH library.

```
ev == OK  ⟹
```

Second, conditions like `ev != OK` indicate the start of a handler block, i.e., code that must be executed in case an exception has occurred. The code in the 'then' block of an `ev != OK` condition is put into an EXCEPT statement. A preceding TRY statement then needs to be wrapped around all the code that can throw exceptions that need to be handled by the handler. All execution paths that assign a value to the exception variable that reaches the condition must be included in the TRY statement.

```
$Code
if (ev != OK) {
  $Handler
}

⟹

TRY    { $Code }
EXCEPT { $Handler }
```

In the CC10 component we did not encounter conditions that check for a specific exception, i.e., conditions of the form `ev == EXCEPTION_CONSTANT`.

All handlers are bound in a 'catch-all' style, and hence are transformed into EXCEPT blocks.

Remove exception variable returns. Exceptions are no longer passed to the calling function through the return value, hence return statements returning the value of an exception variable can be removed.

```
return ev;  ⟹
```

Transform other uses of error variables. Exception variables are used (i.e., read) typically in calls to the logging function, see for example Figure 7.1 line 13. Within a handling block (e.g., `EXCEPT`) these uses can be replaced by the `EXCEPTION_CODE()` macro from the exception library. This macro returns the current exception code, which is an integer value.

In some cases the exception variable is used outside of a handler, where it is not possible to replace the use by a call to `EXCEPTION_CODE()`. We replaced these uses by the exception constants (i.e, integer literals) that were implied by the contexts that the uses appeared in. In the CC10 component it was almost always possible to resolve the exception constant from the context.

A remaining issue consists of function calls implementing the tracing (crosscutting) concern that we described in detail in Chapter 6. These function calls appear at the start and end of each function definition to trace its input and output arguments, respectively (not shown in Figure 7.1. The tracing function call at the end of a function includes the value of the exception variable that will be returned by the function. In order to remove this use of the exception variable, we wrap the entire function body in a `TRY` block, and attach a `FINALLY` block containing the tracing call. Within the tracing call, the use of the exception variable is replaced by a call to `EXCEPTION_CODE()`. A `FINALLY` block here exactly mimicks the behavior of the RCI implementation: tracing code is executed whether or not an exception has occurred.

Remove exception variable declarations. All uses of the exception variables have now been eliminated, so finally the declarations of exception variables can be removed as well.

```
int ev = $Int;  ⟹
```

7.3.3 Tool Support

The reengineering process was supported by several tools. Prior to the reengineering process, any known faults in the application of the RCI were removed. These faults were first detected by our SMELL tool, and manually fixed. This step reduces possible confusion because of faults encountered during the actual reengineering. In terms of the renovation model described in Chapter 3 (Figure 3.1), SMELL is used here as an automatic concern (i.e., the exception handling concern) verification step.

We used the ASF+SDF Meta-Environment (van den Brand et al., 2001) to perform the actual code transformations. Several features of this environment turn out to be useful during a reengineering process. First, with an ANSI C grammar (extended with the XXL constructs), the Meta-Environment editor parses the code and provides precise syntax highlighting. After every change, the code is reparsed, making it possible to instantly recognize that a modification has not resulted in syntactically correct C code. Since the Meta-Environment uses a generalized LR parsing algorithm (implemented as SGLR (Visser, 1997)), the parsing result

SEH Implementation			
`TRY`	307	`CATCH`	0
`EXCEPT`	189	`FINALLY`	127
`THROW`	477		
`ev == OK`	4	`ev != OK`	5
`ev = ERR;`	4	`ev = func();`	0
`int ev = OK`	33		

Table 7.3: Number of constructs in the SEH implementations.

can be ambiguous. We deal with ambiguities by ignoring them in case they occur in parse trees that are not of interest (for instance, parameter lists). Otherwise, the code is changed in a reversible way such that the ambiguities do not occur during code transformation.

Second, within the ASF+SDF Meta-Environment, code transformations can be expressed as rewrite rules in the Algebraic Specification Formalism (ASF). Using such rewrite rules we were able to partly automate the transformation steps as outlined in Subsection 7.3.2. Transforming assignments, removing returns and removing declarations (respectively steps 1, 4, and 5) were automated fully, while transforming conditions and transforming other uses of exception variables (respectively steps 2 and 3) still required some manual work. Additionally, transforming assignments still required a manual step to verify using the SMELL tool that no functional code was executed on an exceptional path. Transforming conditions was fully automated for those conditions that just check whether no exception has occurred, i.e., `ev == OK`, and partly for conditions that check the opposite and start an exception handler, i.e., `ev != OK`. The `TRY` and `FINALLY` blocks that wrap each function that performs tracing (and reads the exception variable) were inserted fully automatically in step 3. Still other uses were transformed manually.

7.3.4 Validation

Actually testing whether functional equivalence holds has not been performed within this study: testing the CC10 component is an elaborate process that remains future work. Instead, functional equivalence was checked manually. The Meta-Environment was used to ease the manual work. Based on an ANSI C grammar, the structured editor of the Meta-Environment helps to ensure that the transformed code is still syntactically valid C. Alternatively, the level of validation could be increased by compiling the resulting C code (including the XXL library).

7.3.5 Results

We first observe that the NLOC for the SEH version is 11,200, which is a small increase compared to the original 11,134. Recall that `TRY` and `FINALLY` blocks were wrapped around each function definition that performs tracing, resulting in an increase of NLOC. A smaller

decrease was obtained by removing exception variable declarations, return statements, and guarding conditions.

Table 7.3 shows the number of times the main XXL constructs occur in the reengineered code. Note that `CATCH` statements were not used at all. This is due to the fact that in the CC10 component handlers are never attached to specific exceptions, i.e., `EXCEPT` can always be used instead of `CATCH`. Some `TRY` statements have no `EXCEPT` handler. Instead, those `TRY` statements only have a `FINALLY` handler.

The reengineered component no longer contains any RCI code, except in the case of 33 handlers that perform cleanup (among a total of 189 handlers). These cleanup handlers call functions that can throw exceptions as well. To deal with those exceptions, the cleanup function calls are wrapped in `TRY` and `EXCEPT` blocks of their own. From within those blocks, it is not possible to access the exception value of the (outer) cleanup handler block. The `EXCEPTION_CODE()` macro will instead return the exception that was raised during the cleanup function call. In these cases, an exception variable is still used to make the outer exception available within the inner handling blocks. In 9 cleanup handlers, these exception variables are also used within conditions that determine control-flow, resulting in 4 leftover occurrences of `ev == OK`, 4 of `ev = ERR` and 5 of `ev != OK`. Since the XXL library uses C integers to represent exceptions, the RCI and XXL constructs can be used together conveniently.

Figure 7.3 shows an example of a cleanup handler that still uses RCI code. At line 2 the exception active in the outer `FINALLY` block is assigned to the exception variable `ev`. If no exception is active in the `FINALLY` block (which is executed regardless of the occurrence of an exception), then `EXCEPTION_CODE()` returns OK (integer value 0). `ev` is used at line 8 to check whether an exception had occurred before the `mem_free` at line 5 threw an exception. If that is not the case, thus `ev == OK`, then just the exception that occurred during `mem_free` is handled, otherwise, specific handling is performed for handling two active exceptions.

7.3.6 Discussion

What are the advantages of using exception handling compared to using a return code idiom?

We argue that the main benefit provided by the reengineering from RCI to SEH is the elimination of almost all `ev == OK` like conditions. These have become unnecessary because the SEH constructs automatically provide the desired control flow in case of an exception. Likewise, exceptions being returned by a function no longer have to be saved explicitly in an exception variable. Hence the number of `ev = func();` statements is 0 in the SEH version. Programmers have less opportunity to make errors implementing the exceptional control flow since most of the task is now handled automatically by the SEH library. In Chapter 5 we showed that the majority of faults (out of a total of 2 faults per 1,000 LOC) occuring in the RCI implementation are unguarded assignments to the exception variable. In these cases, exception control flow was not programmed correctly. Such faults can be prevented by using SEH.

Instead of the manual programming of the exceptional control flow, a programmer using SEH must insert SEH constructs at the appropriate times. As we saw in our reengineering, this can actually result in a small increase of code size (11,200 NLOC versus 11,134). It is

```
1  FINALLY {
2    int ev = EXCEPTION_CODE();
3
4    TRY {
5      mem_free(&string);
6    }
7    EXCEPT {
8      if (ev == OK) {
9        LOG(MEM_ERR, EXCEPTION_CODE(), "free string failed");
10       THROW(MEM_ERR);
11     }
12     else {
13       /* handle multiple exceptions */
14     }
15   }
16 }
```

Figure 7.3: A cleanup handler with remaining RCI code.

also not possible to use SEH constructs exclusively in the case of some handlers that perform cleanup. A use of the return code idiom in such cases is probably acceptable as cleanup handlers do not constitute a majority (33 out of 189 handlers) and the effects remain localized within the handlers.

7.4 Reengineering to Aspect-Oriented Programming

What gains can be expected from reengineering SEH code to an aspect-oriented solution?

In general, this is an extremely hard question to answer. There are numerous variables to consider that either benefit or degrade an aspect-oriented solution of any problem. First, the *target aspect language* determines which features are available to factor the idiomatic code fragments into pointcuts and advice. Second, *aspect design* determines which idiomatic code fragments actually will be replaced by advice, and which remain as is. Third, *equivalence criteria* may be defined to limit the changes that the reengineering effort will be allowed to make. Typically, functional (or behavioral) equivalence of the code is required at some level in order to minimize the risk involved with changing an important system. In practice we have observed a far stricter equivalence criterion: syntactical equivalence between the old code and the aspectized code after the aspects have been woven in (in this practical case a source-to-source aspect weaver was used). Syntactical equivalence may be hard to achieve, but in some settings regression testing is considered extremely hard as well. ASML, our industrial partner, is prepared to spend significant effort on obtaining syntactical equivalence since otherwise they have to spend a great amount on regression testing.

We argue that the topics of aspect languages, aspect design, and equivalence criteria are not fully separable in the context of a renovation process. They all work together to determine

the quality of the end result. We will see that by focussing on the topic of equivalence criteria, we will also uncover some effects of aspect languages and aspect design.

7.4.1 The Tradeoff between Equivalence and Quality

If a strict equivalence criterion is being maintained, there may be consequences for the quality of the end result of the reengineering, due to the presence of variability (as also discussed in Chapter 6) in the legacy code. Variability consists in differences between idiomatic code fragments, and can be accidental (i.e., small differences, mistakes) or essential (i.e., larger differences, different idiom usages). For instance, three exception handling variations are shown in Figure 7.4. The differences between the variants are underlined. Variant F1 is an excerpt of exception handling code for function `f`. It calls a function `a(p)` with a pointer argument and if an exception is raised during that call, the log function is called linking an `F_ERROR` to the raised exception (obtained by `EXCEPTION_CODE()`), and finally `F_ERROR` is thrown as an exception. Variant F2 is an excerpt of a different piece of exception handling code from function `f`. The only difference between variants F1 and F2 is the log message provided at line 5.

Variant G is an excerpt of a function `g` from the same component as `f`. If performs the same call to `a(p)`, but the handling code for exceptions coming out of the call to `a(p)` is different: `f` and `g` throw different exceptions after failure of function `a`, `F_ERROR` and `G_ERROR`, respectively.

Suppose variants F1, F2 and G would be reengineered to use an aspect the exception handling for functions `f` and `g`. Ideally, this would result in an aspect consisting of a single pointcut and attached advice containing the exception handling code. However, the differences that exist between the variants may be such that it is not be possible to arrive at this ideal result. Which log message must be provided or which exception must be thrown at the end both depend on the surrounding function. Unless the aspect language makes the name of the surrounding function available to an advice, it may not be possible to factor out these differences such that a single pointcut and advice re-generates the original exception handling behavior.[3] Instead, the aspect would use three pointcuts with matching advice for each variant. Such a solution would re-generate the original behavior, but be less desirable in terms of quality (a bigger aspect).

A syntactical equivalence criterion demands that the reengineered code re-generates all the variations that are present in the legacy code. As we demonstrated in the example above, this could reduce the quality of an aspect-oriented solution. What would happen if we stepped down from the strict syntactical equivalence criterion just a little bit? On the one hand, the difference between the two log messages may really be accidental, i.e., it is caused by a programmer mistake, and therefore it could be ignored by the equivalence criterion. Variants F1 and F2 would then be unified in the sense that a single pointcut and advice would be able to re-generate F1 and F2 according to the relaxed equivalence criterion. On the other hand, the difference between variants F1 and F2, and variant G, i.e., the exception to be thrown after handling the exception coming out of `a`, is of a more essential kind. Unless the exception

[3] An advice could also use (run-time) reflection in, for example, Java. Arguably, this would be an undesirable solution.

handling aspect is assumed to be able to re-generate the correct exceptions, variants F1 and F2 should not be unified with variant G. We are then left with an aspect consisting of two pointcuts and advice pairs, one for variants F1 and F2, and one for variant G.

Variant F1:

```
1   TRY {
2     a(p);
3   }
4   EXCEPT {
5     LOG(F_ERROR, EXCEPTION_CODE(), "exception from a");
6     THROW(F_ERROR);
7   }
8 }
```

Variant F2:

```
1   TRY {
2     a(p);
3   }
4   EXCEPT {
5     LOG(F_ERROR, EXCEPTION_CODE(), "received exception from a");
6     THROW(F_ERROR);
7   }
8 }
```

Variant G:

```
1   TRY {
2     a(p);
3   }
4   EXCEPT {
5     LOG(G_ERROR, EXCEPTION_CODE(), "received exception from a");
6     THROW(G_ERROR);
7   }
8 }
```

Figure 7.4: Three exception handling variations.

We are particularly interested in this tradeoff between the strictness of equivalence criteria on the one hand, and the resulting quality of the aspect-oriented system, on the other. The variability that exists in the idiomatic exception handling fragments gives rise to this tradeoff, as we will see. Of course the target aspect language is an important factor, but we consider language choice out of scope for this chapter. Instead, we consider the (abstract) aspect language that we describe in Subsection 7.4.2. We will evaluate a number of equivalence criteria for the reengineering of the SEH version of the CC10 component to an AOP solution.

Note that we did not consider the reengineering of the RCI code to SEH constructs (see Section 7.3) in terms of equivalence criteria. In that case we were not interested in unifying

exception handling variations to obtain a solution consisting of fewer fragments. We only expressed the existing exception handling code in another idiom while maintaining all existing variations. Therefore, the tradeoff between the strictness of equivalence criteria and the resulting quality is different from the AOP case that we consider here.

7.4.2 Approach

The two main components of most aspect languages are pointcuts and advice, i.e., respectively *where* (or when) something should happen, and *what* should happen at that location (or time). A simple quality criterion can be based upon these two basic notions. Given a concern, such as exception handling, the fewer pointcuts and advices required to describe that concern, the higher is the quality of the aspect-oriented implementation. Here we will never consider pointcuts and advice separately, and a pointcut will always be associated with exactly one piece of advice. It thus suffices to consider just the number of pointcuts. We will use this simple quality criterion to judge the quality of the aspect-oriented solutions that we obtain.

To facilitate the analysis of the reengineering of the SEH solution to an AOP solution, we will also make the following assumptions: The AOP solution replicates the behavior of the SEH solution according to a well-defined equivalence criterion. Most importantly, the set of locations at which exception handling occurs in the SEH solution remains unchanged in the AOP solution. The equivalence criterion may only be relaxed to allow changes in the handler code that occurs at these locations. In our evaluation the equivalence criterion will be relaxed in a step-wise fashion to investigate the effect it has on the expected quality of the end result.

We also implicitely assume that it is always possible to actually move a piece of exception handling code to an aspect. Any accesses by advice to local variables or other contextual information are assumed to be resolved by the aspect weaver. Alternatively, we can assume that *enabling transformations* are performed to expose the necessary context (Binkley et al., 2006).

The AOP implementation will consist of pointcuts and advice. A pointcut captures a number of joinpoints that each identify a location in the source code at which exception handling occurs. A pointcut is then associated with an advice that contains the actual handling code. In this chapter joinpoints are considered to be of the *static* kind. That is, the joinpoints we consider here can be identified in the source text. In contrast, *dynamic* joinpoints, which represent events in the system's execution, are not considered.

Concretely, the following mapping is made from SEH constructs to pointcuts and advice:

- A `TRY` block maps to a joinpoint. The code within the block of the `TRY` identifies the joinpoint.

- All the handlers, i.e., `EXCEPT`, `CATCH`, and `FINALLY` blocks that occur together with a `TRY` form the advice that is associated with a pointcut that contains the `TRY` joinpoint.

For now, we have allowed joinpoints to be coincide with blocks of arbitrary code, such that all `TRY` blocks can be considered as joinpoints. Current aspect languages are far more restricted. We will later limit our set of joinpoints to a more realistic set. However, *enabling transformations* can be performed to expose joinpoints for arbitrary blocks of code: the code

occurring in a TRY block can be refactored into a function and replaced by a call to the new function, which is typically available as a joinpoint in most aspect languages.

A trivial AOP solution can now be defined. It consists of a pointcut for each single joinpoint, i.e. TRY block, and an associated advice that consists of the handling code belonging to that TRY block. This implementation clearly satisfies the requirement that its behavior equals the SEH implementation. However, no real improvement is offered by this implementation, since all handling code is merely moved into an aspect as separate advices. We will use this implementation as a base line upon which to improve.

The following defines an AOP implementation more formally:

$$Aspect = \{\langle Pointcut_1, Advice_1\rangle, \langle Pointcut_2, Advice_2\rangle, \dots\}$$

$$Pointcut = \{Joinpoint_1, Joinpoint_2, \dots\}$$

The trivial AOP implementation described above consists of 307 $\langle Pointcut, Advice\rangle$ pairs for the 307 TRY blocks that occur in the SEH implementation (see Table 7.3).

We will now merge $\langle Pointcut, Advice\rangle$ pairs to try and arrive at more beneficial AOP implementations merging as the equivalence criterion is relaxed. Binkley et al. (2006) propose a similar process called pointcut abstraction. As mentioned before, the total set of joinpoints will remain constant during this process, i.e., the set of locations where handling code is woven in will not change. Pointcuts and advices can be merged if both the pointcuts and the advice are considered equivalent according to the criterion.

We define the merge operation for two $\langle Pointcut, Advice\rangle$ pairs as follows:

$$merge(\langle Pointcut_1, Advice_1\rangle, \langle Pointcut_2, Advice_2\rangle) = \langle Pointcut_1 \cup Pointcut_2, Advice_1\rangle$$

In words, the pair obtained by merging two pairs consists of the union of both pointcuts, and one of the advices (which are considered equivalent by the criterion).

7.4.3 Equivalence Criteria

Intuitively, two pointcut–advice pairs can be merged if those pairs perform equivalent handling at an equivalent location in the source code. We identify a number of interesting equivalence criteria that capture this intuition for both joinpoints and advices. First, many joinpoints in the CC10 component turn out to be either function calls (a TRY wrapping just a function call) or complete function bodies (a TRY wrapping a complete function body). The latter are referred to as function *execution* joinpoints in AOP terminology. We base two equivalence criteria on function call and execution joinpoints.

- **Call same + exec**. Two joinpoints are equivalent if they both call the same function, or they both are the body of the same function (execution).
- **Call any + exec**. Two joinpoints are equivalent if they both call any function, or they both are the body of the same function (execution).

Second, for advices we identify the following equivalence criteria:

- **Syntax**. Two advices are equivalent if the Abstract Syntax Trees (ASTs) of the two advices are identical. Effectively, syntactical equivalence is the same as textual equivalence disregarding white space and comments.
- **Syntax + EH**. Two advices are equivalent if they are syntactically equivalent while also disregarding the arguments of calls to the LOG and THROW macros (see Figure 7.2). It is assumed that these arguments can be generated correctly by the exception handling aspect.
- **Syntax + EH + Trace**. Two advices are equivalent if they are equivalent according to Syntax + EH, while also ignoring the arguments of any tracing calls. This criterion assumes that a separate tracing aspects correctly generates the tracing calls at the beginning and end of each function. These tracing calls were discussed before in Subsection 7.3.2 and Chapter 6. Currently, such a tracing aspect is being implemented at ASML.

7.4.4 Results

The CC10 component that was reengineered to SEH in Section 7.3 is considered again here, in particular, we consider its SEH version. The `TRY` statements, and the associated handlers were extracted automatically (using the ASF+SDF Meta-Environment) to form the trivial aspect solution that was described above. In this section we will obtain several results by (automatically) merging pointcuts under a number of different equivalence criteria. We will consider two sets of joinpoints. First, the set of joinpoints we consider in the CC10 component, i.e., all `TRY` blocks. Second, the set of those joinpoints that are function calls or executions, i.e., all `TRY` blocks that wrap just a function call or an entire function body. Function call joinpoints are of the form:

```
TRY {
  $FunctionCall;
}
$Handlers
```

while function execution joinpoints occur in the following context:

```
$FunctionSignature {
  TRY {
    $Body
  }
  $Handlers
}
```

where `$Body` represents the actual joinpoint. As we will see, most (262 out of 307) joinpoints are either function call or execution joinpoints. This set represent a realistic set of joinpoints in the sense that function call and execution joinpoints are typically available in real aspect languages.

For both sets, we start with the trivial aspect solution that consists of a pointcut for each single joinpoint (and its advice). The merging process is applied until all possible $\langle Pointcut, Advice \rangle$ pairs are merged.

Arbitrary joinpoints. Table 7.4 shows the effect of merging the pointcuts under the various equivalence criteria. In the tables, the criteria for advices (horizontal) are represented in a cumulative shorthand, i.e, '+EH' refers to the equivalence criterion 'syntax+EH', and '+trace' refers to 'syntax+EH+trace.' The top part of Table 7.4 shows how many pointcuts remain of the initial 307 pointcuts. Thus, 130 pointcuts remain after merging under the 'call same+exec' and 'syntax+EH' equivalence criteria.

Recall that a joinpoint represents a location in the source code where exception handling code occurs, i.e., a `TRY` statement. We can thus conclude that for 130 locations where exception handling occurs, that joinpoint is unique, or if there exists an equivalent joinpoint, the handling at the other joinpoint is different according to 'syntax+EH.' As the equivalence criteria are relaxed, the number of pointcuts remaining decreases. 62 pointcuts remain in case the most relaxed equivalence criteria are used, i.e., 'call any+exec' and 'syntax+EH+trace.'

The bottom part of Table 7.4 shows the minimum, median, and maximum number of joinpoints that occur in the merged pointcuts. As the median value is 1 in all cases, pointcuts tend to consists of only 1 joinpoint.

Number of pointcuts for 307 joinpoints

advices → / joinpoints ↓	syntax	+EH	+trace
call same + exec	206	130	88
call any + exec	206	104	62

Minimum, median, and maximum number of joinpoints in pointcuts

advices → / joinpoints ↓	syntax	+EH	+trace
call same + exec	1, 1, 60	1, 1, 60	1, 1, 113
call any + exec	1, 1, 60	1, 1, 68	1, 1, 113

Table 7.4: Results for arbitrary joinpoints.

Function call and execution joinpoints. The same merging process as before is applied to these joinpoints. Table 7.5 shows the results. 17 pointcuts are required to capture all exception handling occurrences in case of the most relaxed equivalence criteria 'call any+exec' and 'syntax+EH+trace.' The median number of joinpoints in pointcuts is still 1 except when advices are considered equivalent under 'syntax+EH+trace,' where the median value is 2.

Note that the reduction in number of pointcuts does not change when considering only function call and execution joinpoints. The top parts of Tables 7.4 and 7.5 are equivalent if we add the difference between the number of joinpoints considered ($307 - 262 = 45$) to the numbers in the top part of Table 7.4. We can therefore conclude that only joinpoints of type function call or execution could actually be merged under the equivalence criteria used here.

Number of pointcuts for 262 joinpoints

advices → / joinpoints ↓	syntax	+EH	+trace
call same + exec	161	85	43
call any + exec	161	59	17

Minimum, median, and maximum number of joinpoints in pointcuts

advices → / joinpoints ↓	syntax	+EH	+trace
call same + exec	1, 1, 60	1, 1, 60	1, 2, 113
call any + exec	1, 1, 60	1, 1, 68	1, 2, 113

Table 7.5: Results for function call and execution joinpoints.

7.5 Discussion

It is clear that relaxing the equivalence criteria has a significant effect on the number of pointcuts needed to include all exception handling joinpoints. The best results are obtained when merging under the most relaxed equivalence criteria, i.e., 'call any+exec' and 'syntax+EH+trace.' We will discuss four pointcuts of particular interest, and some general results. First, the following interesting pointcuts are found under the 'call any+exec' and 'syntax+EH+trace' criteria.

1. The top pointcut consists of 113 joinpoints. These are the function execution joinpoints representing the `TRY` and `FINALLY` blocks that wrap an entire function body. Their purpose is only to execute tracing code whether an exception has occurred or not. In other words, those `FINALLY` blocks consist of just a tracing call. Note that this pointcut only appeared on top by considering the 'syntax+EH+trace' equivalence criterion, which assumes that a suitable tracing aspect generates the arguments of the tracing calls.

2. Pointcut number 2 consists of 68 joinpoints that are calls to memory allocation, and de-allocation functions that have syntactically equivalent handling. In fact, the same pointcut appears under all stricter equivalence criteria with 60 joinpoints. These are all the memory allocation function calls. The deallocation function calls with similar handling are added to the pointcut when relaxing to 'call any+exec.'

3. The third pointcut has 39 joinpoints that represent function calls to various functions. These calls have similar handling, though not syntactically equivalent. Only when the exception handling aspect is presumed to be able to fill in the `LOG` and `THROW` macros can these handlers be considered equivalent (i.e., under '+EH').

4. The fourth pointcut again consists of memory de-allocation function calls. In these 13 cases, more elaborate handling is performed that includes cleanup and possible handling of exceptions that arise during cleanup.

5. until 11. These pointcuts contain only a small number of joinpoints and are not discussed separately. Pointcuts of rank lower than 11 contain only 1 joinpoint.

The first pointcut raises an important issue: (crosscutting) concerns may be interdependent, and thus one must be careful when reengineering either of them into aspects. In fact, the interdependence between the exception handling and tracing aspects was already exposed during the reengineering of the RCI to SEH. The occurrence of exceptions causes control to skip the remainder of a function unless the exception is caught and handled. It turned out that tracing functionality at the end of functions had to be moved into `FINALLY` handlers in order to mimic the RCI implementation.

This move may have eased our analysis here. Inspection of the merging results earlier on showed many separate joinpoints that had only a `FINALLY` advice. These turned out to consist of only tracing code. Since ASML is actually implementing a tracing aspect that can regenerate the original (idiomatic) tracing situation to a large degree, the '+trace' equivalence criteria seemed to be an obvious addition. We thus conclude that the benefits expected from the reengineering of one concern may be dependent on the proper aspectization of other concerns. The interdependence of exception handling and tracing would have caused worse results for the aspectization of exception handling would a tracing aspect not have been present.

Table 7.4 shows the merging results for all joinpoints, including joinpoints that represent arbitrary pieces of code. Aspect weavers are typically not expected to weave at such joinpoints. The results in Table 7.5 are limited to function call and execution joinpoints and are hence considered more applicable to real aspect weaving. The difference between these sets of joinpoints is a set of 45 joinpoints that are never merged into other pointcuts. These joinpoints, and their associated advices, represent unique exception handling that should not be implemented as an aspect.

Figure 7.5 gives an idea of the relative effects of the equivalence criteria. It shows the cumulative number of joinpoints included by the biggest n pointcuts. Pointcuts of size 1 are not shown, but ultimately all pointcuts considered together include 100% of the joinpoints. The straight ticked line at 85% represents the maximum percentage that can realistically be expected of an aspect-based solution. The remaining 15% are neither function call nor execution joinpoints.

Under the most strict equivalence criteria that make no assumptions about the tracing or exception handling aspects, many (54) pointcuts are required to include a reasonable percentage (say 50%) of joinpoints (line with boxes). If we relax the equivalence criteria to '+EH', only 3 pointcuts are required to reach 50% (line with crosses). Recall that '+EH' means that it is assumed that the exception handling aspect is capable of generating the arguments of `LOG`

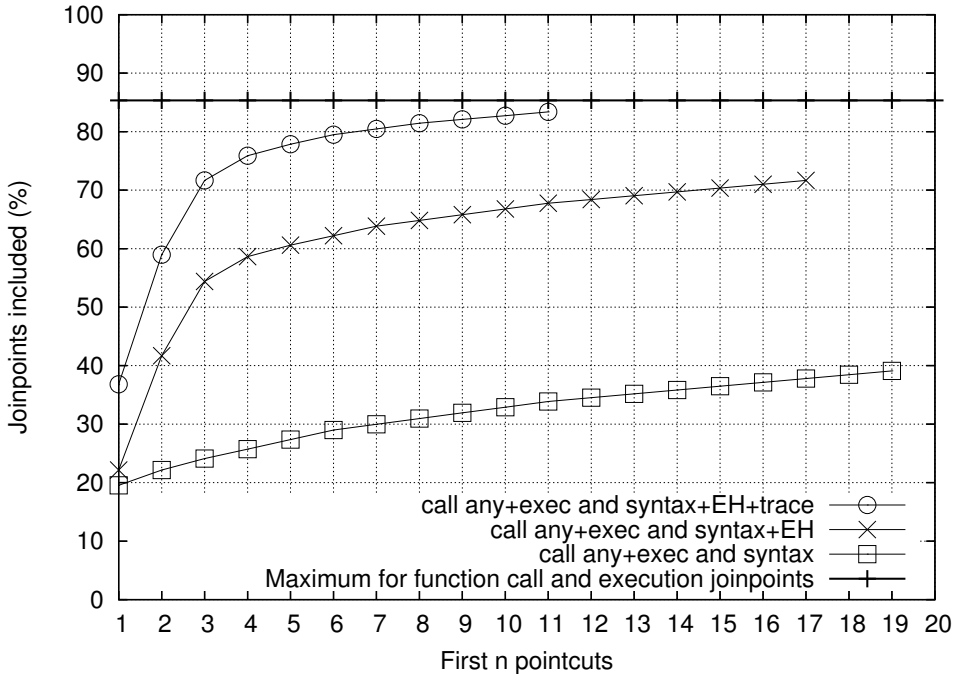

Figure 7.5: Percentage of joinpoints included in the first 19 pointcuts, under different equivalence criteria.

and `THROW` macros based on context, and thus those arguments can be ignored, which can result in more pointcuts being merged. Relaxing further to '+trace' improves the situation even more: 3 pointcuts cover more than 70% of joinpoints (line with circles).

Considering the analysis of the results, the question now arises whether exception handling would benefit from an AOP implementation. Clearly, not if one would naively move exception handlers into an aspect while merging the syntactically equivalent handlers. One would be left with an aspect consisting of 206 pointcuts and pieces of advice (see Table 7.4). However, the expected benefits can be improved by taking into account specifics of the current exception handling implementation. The `LOG` and `THROW` macros can probably be provided with their arguments by the aspect (assuming the aspect language exposes the necessary context). As a consequence more handlers can be merged, resulting in fewer pointcuts. Similarly, assuming the presence of a tracing aspect further had the effect of further reducing the number of pointcuts.

However, there appear to be parts of the exception handling code that will not benefit from aspect-orientation according to our analysis. 15% of the joinpoints, i.e., `TRY` blocks, are not of the common function call or execution type, and hence cannot be moved easily into an aspect. Also, since these joinpoints were never merged into other pointcuts, they represent unique handling code or handling code that occurs within a unique context.

7.5.1 Limitations

We assumed in our analysis that handling code may not be changed beyond what the (strict) equivalence criteria allow. One could argue that a realistic reengineering project could probably unify large fragments of code in an ad-hoc fashion. However, our experiences with ASML developers planning such a reengineering project, are that changes performed at a large scale (100 to 1,000 KLOC) are considered very risky and must be controlled as much as possible. In that setting, strict equivalence criteria seem a suitable means of analyzing expected benefits.

The analysis performed here should be interpreted as an upper bound, in the sense that actual reengineering efforts may be complicated by weaver limitations. We assumed that all fragments of exception handling code could be moved into an aspect, and that references to context are automatically resolved by the weaver. References to local variables, however, can not be resolved by current weavers. We also saw before that some fragments of exception handling code do not map onto function call or execution joinpoints. Therefore, some fragments of exception handling code may not be easily moveable to an aspect in reality. Again, enabling transformations could alleviate this problem.

7.5.2 Future Work

Chapter 6 proposes a method, based on formal concept analysis (FCA), to explore the variations present in a (crosscutting) concern. The analysis presented here, based on the equivalence criteria, is closely related. The FCA approach provides an overview (a lattice) of the differences and commonalities of a concern. In that approach, equivalence criteria remain fixed, and hence their effects cannot be judged. It would therefore be interesting to combine both approaches, obtaining different lattices for different equivalence criteria.

7.6 Related Work

Reengineering exception handling. Adams and de Schutter (2007) propose an AOP solution for the exception handling idiom of the same component that was studied in this chapter. They demonstrate how a new kind of joinpoint that is based on continuations can be used to express the exception handling idiom using aspects. Furthermore, their proposal includes annotations that allow developers to specify alternate behavior.

Mortensen and Ghosh (2007) demonstrate how to refactor a return code idiom in C++. They identify several exception handling strategies, which are essentially variations upon the ‘standard’ return code idiom, and define separate aspects for each of them.

Several studies by Filho et al. focus on the refactoring of exception handling code in Java programs to aspects, which are provided by AspectJ (Kiczales et al., 1997). In Filho et al. (2007) the authors propose a categorization of exception handling code that helps judging whether a refactoring to aspects is worthwhile. They demonstrate that a refactoring that takes the recommendations made by their categorization into account results in systems of a higher quality compared to bluntly refactoring all exception handling code to aspects. A categorization approach like theirs could probably guide C reengineering projects as well.

Earlier work discusses their lessons learned during the aspectization of the exception handling code of four Java applications (Filho et al., 2006). They conclude that the aspectization of exception handling code in applications may not always results in solutions of higher quality.

Lippert and Videira Lopes (2000) studied the reengineering of exception handling code in a Java application framework (JWAM) to AspectJ. They conclude that the aspect-oriented solution they obtained offers benefits in terms of reuse, evolvability and readability.

Robillard and Murphy (2003) study the flow of exceptions through Java programs using a static analysis tool. Such a tool would be extremely valuable during the reengineering projects described in this chapter. In particular, any changes in the exception flows would need to be brought to the attention of the reengineer.

Reverse engineering. Several specialized reverse engineering approaches are related to our work. Aspect mining attempts to discover opportunities for the use of aspects in existing (legacy) software. These techniques are typically more generic in nature than our work, i.e., they do not focus specifically on one concern (like exception handling), but instead they try to find aspects for all kinds of functionality. Three such techniques are discussed by Ceccato et al. (2006).

Clone detection techniques search source code for the occurrence of code clones, i.e., fragments of code that are very similar or even identical. Code clones can indicate opportunities for reengineering, as is shown by Baxter et al. (1998). Their approach finds clones that can be factored into a function, and subsequently replaces those clones by a call to the newly defined function. Our merging process for pointcuts (see Section 7.4) can be seen as a form of clone detection in the sense that we also search for equivalent code fragments among the advices in the exception handling aspect. However, we are also interested in deriving new equivalence criteria, and hence require insight into the differences between code fragments, which is not typically covered by clone detection techniques.

7.7 Conclusion

This chapter analyzed the step-wise reengineering of a small C component (11,134 NLOC) of a large-scale embedded system using SEH and AOP.

Replacing the RCI by SEH. The idiomatic implementation of the exception handling in this component was reengineered to an implementation that uses SEH constructs. This chapter showed that it is possible to almost completely eliminate the legacy idiom usage using SEH constructs, at the expense of a slight increase (0.6%) in code size. The main benefit of the SEH implementation is argued to be that the programmer's task of maintaining exceptional control flow has become simpler. Using the RCI exception control flow is programmed manually by inserting many explicit checks. The reengineered component does no longer contain such explicit checks (with a few exceptions).

Aspects for Exception Handling. This chapter presented an analysis of the expected benefits of reengineering SEH code to AOP. We found that reengineering exception handling

idioms using aspects may not result in great benefits. The benefits were shown to depend on the desired level of equivalence between the SEH and AOP implementations. We observed that the (industrial) context in which reengineering processes are performed can fix the desired level of equivalence using strict equivalence criteria.

We found that strict equivalence criteria, e.g. those that demand syntactical equivalence, will result in unfavorable aspect-oriented implementations of exception handling. However, we also found that relaxing the equivalence criteria can increase the benefits offered by aspect-oriented implementations. The analysis showed that the benefits also depend on the availability of aspects for other concerns such as tracing, and on features of the exception handling aspect itself.

This chapter provided quantitative data on the effect of various equivalence criteria on the expected benefits of the use of aspects. In particular, we found that 15% of the SEH code in the studied component is probably not suitable for implementation as an aspect.

Chapter 8

Conclusion

This thesis is concluded by a summary and evaluation of the research contributions made while trying to answer the research questions. We also discuss the use of the industry-as-laboratory research method (Potts, 1993) by reviewing challenges we encountered during the research project, recommendations for future projects, and expected industrial results.

8.1 Contributions and Evaluation

Research Question 1

> *Can idiomatic crosscutting concerns be identified automatically? In particular, are clone detection tools suitable for this purpose?*

Chapter 2 compared the results of three clone detection tools (Bauhaus' ccdiml, CCFinder, and Komondoor's PDG-DUP) to the annotated source code of five crosscutting concerns. It turned out that all three clone detectors can identify good aspect candidates. However, the recall and precision measures varied strongly for the different crosscutting concerns. Partly, this is due to the clone detection tools not being designed with this particular purpose in mind, and the fact that some crosscutting concern code simply is not cloned. Also, the clone detection technique (e.g., similar subtrees in ASTs) that a clone detector implements turned out to be a significant factor. The three clone detectors obtained different levels of recall and precision for most crosscutting concerns (similar recall and precision levels were obtained for exception handling and tracing).

A problem not solved here is that other code (i.e., not crosscutting code) can also be cloned. In Chapter 2 the reference body of crosscutting concern code is used to filter irrelevant clones. In a real aspect mining scenario such a reference body could be missing, since aspect mining is tasked with actually identifying crosscutting concerns. Therefore, the real aspect mining performance of clone detection tools for aspect mining cannot be judged fully based on the results of Chapter 2.

However, since the results do show the level of cloning within known crosscutting concerns, conclusions can be drawn about the best possible aspect mining performance that can

be expected of the studied clone detection tools. On the one hand, Chapter 2 showed that none of the three clone detection results is able to identify exception handling code well. This result carries over to aspect mining (using one of the three clone detectors), which can therefore not be expected to find good aspects for exception handling. On the other hand, the parameter checking concern (called NULL-value checking in Chapter 2), and to a lesser extent the tracing concern, are identified well enough by all three clone detection results.

Research Question 2

> *Is it possible to renovate idiomatic crosscutting concerns? What are the challenges for an automatic approach?*

Renovation Approach. In Chapter 3 we proposed a systematic approach to renovation of idiomatic crosscutting concerns using aspects. This approach is based on a concern (or idiom) verification tool that processes legacy source code and reports on adherence to the idiom. Applications and violations of the idiom are reported by the tool such that subsequent phases can use those data. Idiom violations that are clearly faults may need to be fixed, while minor violations may need to be retained. Figure 8.1 (repeated from Chapter 3) shows an overview of the approach.

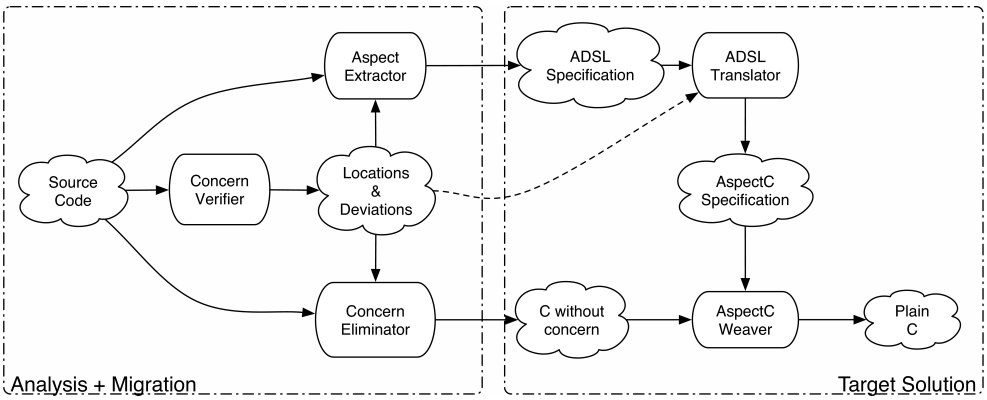

Figure 8.1: An overview of Chapter 3's renovation approach.

Idiom applications ultimately need to be eliminated (i.e., removed) from the legacy source code and replaced by an aspect-oriented implementation. Aspects are extracted (by reverse engineering) from the list of idiom applications (and minor violations), and ultimately woven back into the source code by an aspect weaver. In addition to the use of aspects, Chapter 3 proposed the use of domain-specific aspect languages that are particularly suited for the expression of a specific crosscutting concern. The domain-specific aspects are then translated into a general purpose aspect language and woven by a general purpose weaver. In this approach the weaving technology of the general purpose aspect language is reused. Chapter 3

demonstrated this approach by means of PCSL, a domain specific aspect language for the parameter checking concern.

It turned out in Chapters 5 and 6 that violations to the idiom (e.g., faults, variations, or inconsistencies) can be expected to occur frequently (see also the discussion below for Reseach Question 3). This situation had not been appreciated fully at the beginning of the research project when the approach described in Chapter 3 was defined. In hindsight, a variability exploration tool such as the one presented in Chapter 6 would need to be included in our renovation approach. Such a tool is needed to obtain a clear picture of the number of idiom violations that are present in the legacy system. In Section 8.2 we further discuss the consequences of idiom violations on the renovation process.

Tool interoperability. Chapter 4 proposed a solution, source-based mappings (SBMs), to linking existing analysis and transformation tools such that parts of the renovation process can be automated. In particular, Chapter 4 focused on the situation in which tools have different schema's of the source code that they process.[1] For the implementation of the various steps described in Figure 8.1 we were interested in reusing as many existing tools as possible. For instance, we implemented many analysis tools (including the idiom verification tool SMELL described in Chapter 5) as plugins for Grammatech's CodeSurfer, a commercial program analysis toolkit (Anderson et al., 2003; CodeSurfer, 2007).

CodeSurfer is an industrial strength tool, that is, it can process source code at the scale required for real renovation projects and has adequate documentation and support. Like many tools in its class, it uses fixed and sometimes proprietary source code schema's that cannot be changed by a user. Proprietary schema's can furthermore deny a user an accurate correspondence between schema elements and source code. Nonetheless, tools such as CodeSurfer represent many years of research and development, and thus we felt justified to expend effort to make such tools useable within our research project.

Tools using proprietary schema's are not the only contributing factor to this problem. Source code schema's (public or proprietary) can represent very delicate specifications of source code structure (or semantics), and can be the result of years of work (e.g., context-free grammars for programming languages like C that can be used in a renovation setting). They can also be geared toward a particular purpose, such as either analysis or transformation (Cordy and Vinju, 2006). In those cases it is undesirable to change the schema just to accommodate tool interoperability. We conclude that a means to cope with different, immutable, source code schema's is a requirement for renovation projects that seek to use existing, industrial strength, tools for their purpose.

Unfortunately it is not always possible to accurately transfer results between tools that use different schema's. However, SBMs can be used under certain conditions pertaining to the accuracy of the correspondence between source code and schema elements of the tool. Chapter 4 described two applications of SBMs that are linked to work described in other chapters. First, the elimination of idiom applications for parameter checking (see Chapter 3) was facilitated by an SBM. In that setting, an idiom verification tool (a CodeSurfer plugin) first processed the source code and identified idiom applications (and violations). The idiom applications were annotated in the source code itself, such that ASF+SDF (van den Brand

[1] Cordy and Vinju (2006) also discuss this situation.

Crosscutting Concern	Chapter(s)
Parameter checking	2, 3
Tracing	2, 6
Exception handling	2, 5, 7

Table 8.1: Crosscutting concerns studied in this thesis.

et al., 2001) could perform the actual elimination later on. Second, insertion of annotations to facilitate aspect weaving was done similarly. Code analysis results (in this case, parameter usage information) were stored inside annotations that were attached to function signatures in the source code. Finally, the extraction of the interesting parts of the tracing idiom in Chapter 6 was facilitated using SBMs.

Research Question 3

Are idiomatic crosscutting concerns sources of implementation faults or inconsistencies?

Both faults and inconsistencies appear to be associated with idiomatic crosscutting concerns. First, Chapter 5 showed a fault rate of 2.1 faults per KLOC[2] in the ASML exception handling code. This fault rate was discovered by means of an automatic bug finding tool (SMELL) that is based on a specific fault model for the ASML exception handling idiom. Many of these faults could be prevented by adopting a better idiom or programming language. Second, Chapter 6 exposed an unexpected number of variations of inconsistencies in the ASML tracing crosscutting concern. The idiomatic fragments of code that implement tracing appear cloned, but actually constitute non-trivial variations. These variations present a challenge for renovation efforts, since the question arises whether they should not retained, or whether they should replaced by a generic implementation. This challenge is discussed further in Section 8.2.

Research Question 4

What are the benefits offered by renovating idiomatic crosscutting concerns using aspects?

Table 8.1 shows a list of the three crosscutting concerns that were studied in detail in this thesis. We will now discuss the benefits and limitations we observed while considering aspect-oriented implementations of those crosscutting concerns.

[2] 1,000 lines of code.

Parameter checking. Chapter 2 showed that parameter checking code is often duplicated, and can be captured in a relatively small number of clone classes. An aspect-oriented solution will potentially eliminate much of the duplication parameter checking code, as demonstrated by the results for obtained using the PCSL domain-specific language (see Chapter 3). Another potential benefit is obtained as a result of localizing parameter checking code in a (domain-specific, e.g., PCSL) aspect. The aspect allows for increased uniformity of the parameter checks since a smaller number of fragments of parameter checking code will need to be maintained. However, this turns out to be a mixed blessing. Making the parameter checking code more uniform could also remove variations that turn out to be relevant, and is therefore a risky process. The aspect language will need to accommodate variations such that the introduction of aspects will not necessarily eliminate variations, but allows them to be unified later on. The PCSL language has limited support for variations by allowing annotations to switch off checking for specific parameters.

Unexpectedly, the aspect-oriented solutions obtained using PCSL and AspectC do not reduce code size for all the components that were considered in Chapter 3. Code size changes instead ranged from -86% to +173%. These results are explained by the different levels of variation that occur within each component. The PCSL language was not capable of dealing effectively with all variations, or else the code size reductions would have been more uniform.

Tracing. Similar as for the parameter checking concern, we observed significant duplication in tracing code. However, the level of duplication was not as high as for parameter checking. Again, variations and inconsistencies seem to be present. Chapter 6 studied the variations and inconsistencies of the tracing concern in detail, trying to answer the question whether tracing could still be usefully expressed as an aspect. In a case studied (performed at ASML) related to Chapter 6, we encountered a very strict requirement of the renovation process: the renovated tracing code had to be textually identical to the legacy code at compile time after all code transformations had been applied (Chapter 7 provided further discussion of this phenomenon). The FCA method presented in Chapter 6 fulfills a pressing need in the context of such a strict requirement. The question is: Would a renovation using aspects still offer an improvement, given that all variations must be (textually) retained? Using the FCA method, one can reach an answer by inspecting the resulting concept lattices. Chapter 6 furthermore concluded that obtaining a beneficial aspect-oriented solution for tracing would be difficult due to lacking language support. Much effort would need to be spend on developing a sufficiently generic aspect language that can cope with the variations and inconsistencies of the legacy tracing implementation.

Exception Handling. Of the crosscutting concerns studied, exception handling contains the least duplication (as observed in Chapter 2). Chapter 5 also showed that the idiom used for exception handling at ASML is also fault prone. The question raised by the presence of these faults is whether a renovation process should fix them prior to introducing an aspect-oriented solution. On the one hand, additional risk will emerge by fixing these faults since legacy systems are in operation, and other processes may depend on even the faults of the legacy system's implementation. On the other hand, faulty exception handling code may be hard to be reengineer since it may be unclear what the code was supposed to do in the first

place. Chapter 5 presented a tool (SMELL) that is capable of finding faults in the ASML exception handling code, providing renovators and developers with awareness of potential problems in the code. They will then need to decide upon a strategy of coping with the actual faults during their activities.

Chapter 7 demonstrated a renovation effort for the exception handling concern in an ASML component. Faults were first discovered by the SMELL tool and eliminated manually. Then, the code was transformed from the ASML return-code idiom (described in detail in Chapters 5 and 7) to the more modern try/catch idiom. A very slight increase in code size (0.5%) was observed as a result of this renovation, but almost all statements that idiomatically implemented exceptional control flow could be eliminated. In Chapter 5 we showed that the majority of faults in the exception handling implementation regard the exceptional control flow, and therefore a reduction of fault proneness can be expected of the renovated component using the try/catch idiom. The next step in the renovation process consisted of analyzing the benefits aspects would offer for the exception handling implementation. It turned out that 15% of the exception handling code could be considered unique in the sense that either the handling code, or the context in which it appears, is not repeated within the legacy component. An aspect-oriented solution for this fraction of exception handling code is not beneficial. The remaining 85% exception handling code could increasingly benefit from the use of aspects given that the legacy and aspect-oriented implementations are allowed to diverge. Chapter 7 showed how much the expected benefits could increase by allowing for more divergence.

8.2 Synthesis

Idiomatic implementation of crosscutting concerns is associated with idiom violations, as we saw in Chapters 5 and 6. A renovation process will benefit from knowing the number and nature of idiom violations that occur in a legacy system because of three important related factors that could be considered external (i.e., given) to the renovation process: *equivalence criteria*, *system quality* and *language technology*. We will now discuss the factors in turn:

Equivalence criteria. Equivalence criteria capture the level of equivalence that a renovation process is expected to maintain between the legacy system and the renovated system. An example equivalence criterion is the successful execution of a regression test suite. As we discussed above, in the context of ASML we encountered a very strict equivalence criterion: the source code has to be textually identical at compile time after all code transformations have been applied. This strict equivalence criterion is a realistic example that occurred during our research at ASML.

A strict equivalence criterion could require that idiom violations are retained by a renovation process. Conversely, idiom violations can becomes less troublesome if a more relaxed equivalence criterion is adopted. Chapter 7 showed the influence different equivalence criteria have on the renovated system.

System quality. The end result of a renovation process should be a superior system. As we saw in Chapters 3, 6, and 7, idiom violations can limit the improvements that are offered

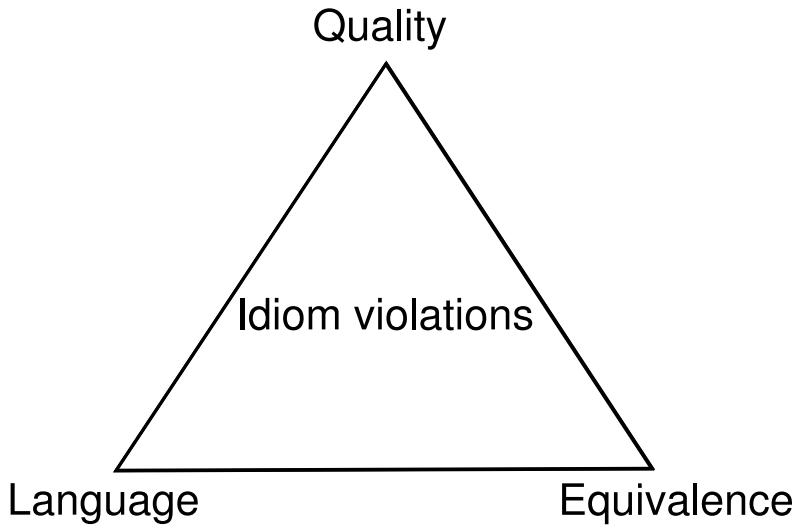

Figure 8.2: Tradeoff for dealing with idiom violations during renovation.

by the renovated system. Expecting smaller quality improvements could be a means to cope with many idiom violations.

Language technology. The language used to implement the renovated system may limit its capability to deal with the idiom violations that must be retained from the legacy system. For instance, in aspect-oriented programming, a pattern language is typically used to specify pointcuts. The expressiveness of this pattern language determines how well exceptions to the aspect can be expressed. Alternatively, annotations of the base code are a means of specifying exceptions to an aspect's pointcuts. In Chapter 3, the PCSL language demonstrated how a domain-specific language could concisely express a crosscutting concern while still allowing for exceptions by allowing them to be specified explicitly. The introduction of new language technology by a renovation process is not always desirable, however, since the costs of personnel training, and embedding in existing processes can be high.

The way a renovation process chooses to deal with idiom violations is subject to a tradeoff between these three factors as depicted in Figure 8.2. The tradeoff consists in choosing which of three factors are 'sacrificed' in order to successfully complete a renovation process in the presence of idiom violations. Sacrificing equivalence implies adopting more relaxed equivalence criteria, i.e., allowing for more differences between the legacy and renovated system. Idiom violation may then become less of a problem since they no longer need to be retained. Sacrificing quality implies lowering expectations of the quality improvement the renovation process is going to offer. For instance, as we saw in Chapter 7 a larger number of aspects could be used to capture all idiom violations. Finally, sacrificing in the area of language technology entails spending effort to extend the language used, and coping with

consequences like language adoption and programmer training.

In order to make an informed choice with respect to this tradeoff, a renovation process needs to be aware of the idiom violations that occur in a legacy system. A toolkit that provides insight into the violations made to an idiom is therefore called for. In Chapter 6 we already presented a tool that gives an overview of idiom violations. This tool could be combined with the analysis presented in Chapter 7 to gain insight into the effects on system quality of equivalence criteria that a renovation process may consider using. Furthermore, the analysis should be extended to allow for the evaluation of the benefits of adding new language features.

8.3 Extrapolations

The research presented in this thesis was performed in the specific industrial context provided by ASML. We discuss here how the results of thesis can be extrapolated to different contexts.

- The use of idioms, i.e., idiomatic implementation, is a common practice in software engineering. Idioms appear in the form of architecture patterns (Buschmann et al., 1996) or design patterns (Gamma et al., 1995), coding conventions and templates. We have shown that applying such idiomatic practices in an industrial setting can result in implementations that frequently violate the idioms used. This observation can probably be repeated for numerous other software systems that employ idioms.

- The SMELL tool presented in Chapter 5 is a means of improving upon the practice of idiomatic implementation. The tool is capable of finding implementation faults and can thus be integrated with an idiomatic practice to reduce the fault proneness of the resulting implementation. SMELL is geared specifically for the ASML exception handling idiom, but the general technique used to implement SMELL is not. The Metal language developed by Engler et al. (2000) could be used to specify similar tools for other idioms. Furthermore, CodeSurfer (2007) and its sibling application CodeSonar are both extensible program analysis toolkit that allow for the rapid development of idiom checker tools like SMELL.

- Chapter 6 demonstrated a generic method to explore idiom violations in legacy source code. This method is based on formal concept analysis, which can be applied in many contexts. However, it is required that relevant objects and attributes are identified within the context in order to obtain meaningful results. Significant knowledge of the idiom that is being studied is required to initialize this method, but interpretation of the results is based on generic properties of formal concept analysis that are not limited to the ASML context. This method is currently being applied to explore the variability in another crosscutting concern: contract enforcement in the Pidgin[3] instant messenger client.

- Renovation processes typically consist of separate analysis and transformation phases that must use the results of one another. The source-based mapping approach that was developed in Chapter 4 provides a semi-generic solution to the problem of linking

[3]Formally known as GAIM.

separate analysis and transformation tools together. The current implementation of this approach, called SCATR, is limited to a specific transformation tool, i.e., the ASF+SDF Meta Environment (van den Brand et al., 2001), but generically supports any analysis tool that is capable of generating SCATR's format.

- Exception handling is a concern in any industrial software system. Chapter 7 provides evidence that using aspects for exception handling in legacy systems cannot be expected to result in great benefits. These results tie in with other evidence presented within the aspect-oriented programming community (Filho et al., 2007; Lippert and Videira Lopes, 2000).

8.4 Industry as Laboratory

8.4.1 Research Approach

The work in this thesis has been carried out as part of *Ideals*: a research project initiated by the Embedded System Institute in Eindhoven, that closely collaborates with ASML as the industrial partner. Idiomatic crosscutting concerns were studied at different levels within the project, ranging from modeling to source code levels. All research within the project was performed on a case study basis, in the industry-as-laboratory style that was proposed by Potts (1993). This style of research entailed close integration of research and industry goals and ways-of-working.

Generation of hypotheses occurred either within the company or the research team. Starting from a hypothesis an initial feasibility study (or proof of concept) was defined to test the basic soundness of the idea. This initial study was performed using relevant artifacts, e.g., source code, models, documentation, from within the company. If successful, a follow-up case study (or transfer project) was started. The objective of such a case study was to apply the research idea in practice. During such a study collaboration with the company was extensive. The steps followed are roughly: advocacy of the idea, resource planning, design and implementation, and embedding in the company way-of-working. All steps required input from both the company and the research team.

Both feasibility studies and case studies were shared with the research community as soon as possible. The planning of either study type contained a target forum (e.g., conference or journal), which typically occurred either during or near the end of the actual study. Especially case studies could run for longer periods of time, which allowed for multiple interactions with the research community during the study.

The work presented here consisted of both kinds of studies. For instance, the study into automatic bug finding in error handling code (Chapter 5) has passed the feasibility study phase and is currently being defined as a technology transfer project within the company. The goal of this project is to implement the functionality offered by the SMELL prototype in the ASML development process. Chapters 3 and 4 studied the feasibility of renovation of idiomatic crosscutting concerns, and formed the basis of the case study reported on in Chapter 6. In the near future we expect such renovations to be carried out for the ASML tracing concern.

8.4.2 Challenges and Recommendations

Some of the challenges described by Potts (1993) have surfaced during our project. The main challenge Potts mentions, an overemphasis on the short term, has been found to be only a minor problem. Our experience was that the most serious challenge occurs at the level of prioritization of company resources. Companies are bound by market deadlines, and can change priorities even during running projects. However, the research project had been defined up front such that state-of-the-art research topics could be investigated. Nonetheless, case studies still remained challenging activities. The company and the research teams were closely integrated during case studies, and care had to be taken to keep the focus on research goals.

Second, technology transfer turned out to be an activity that demanded (at least) as much effort as scientific study itself. Both the research team and the company had to invest in order to successfully transfer results. Furthermore, significant effort had to be expended in order to prepare the road ahead for the research team. An extremely important task consisted of exploring the company in order to find a problem owner, i.e., a person responsible for handling a certain problem, within the company that would actually be interested in adopting a solution devised in research. Problem owners do not tend to line up at the desks of the researchers by themselves. They typically have a working solution to their problem, which may be suboptimal in the eyes of the research team but which the company considers good-enough for the time being. Significant effort is required to convince problem owners otherwise. Furthermore, advocacy of the researched solution is a prerequisite for further adoption of the solution within a company. Both exploration and advocacy are tasks that require specialists that are capable of talking with both the research team and the company.

Third, company concerns with respect to privacy limited the degree of disclosure that the research team could assume. Each publication was processed in order to check for unwanted disclosure. Despite these concerns, aggregate results, or sanitized examples of artifacts such as code examples, have been shared with the research community. These data still represented artifacts and measurements of real software engineering activities. In the context of the software engineering research discipline, it is imperative that as much real data is accumulated in the public domain.

Finally, the lack of scientific control mentioned by Potts is a reality. A company like ASML is a sprawling hub of activity, which continuously changes project goals, staffing, and priorities. Performing controlled research within the context of such a company is a daunting task. However, companies like ASML constitute reality for the software engineering research discipline. It is in this industrial context that one can find the processes that software engineering research aims at studying and improving. The software engineering research discipline should ultimately develop a method that provides researchers with solid ground within the marshlands of industry.

Bibliography

B. Adams and K. de Schutter. An aspect for idiom-based exception handling: (using local continuation join points, join point properties, annotations and type parameters). In *Proceedings of the 5th Software-Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT'07)*, New York, NY, USA, 2007. ACM Press. URL `http://doi.acm.org/10.1145/1233843.1233844`.

B. Adams and T. Tourwé. Aspect-Orientation in C: Express Yourself. Appeared at the Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT'05), co-located with the 4th International Conference on Aspect-Oriented Software Development (AOSD'05), March 2005.

E. N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

P. Anderson, T. W. Reps, T. Teitelbaum, and M. Zarins. Tool support for fine-grained software inspection. *IEEE Software*, 20(4):42–50, 2003.

R. S. Arnold. *Software Reengineering*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1993. ISBN 0818632712.

AspectC++. Website, September 2007. URL `http://www.aspectc.org`.

Aspect-oriented C. Website, September 2007. URL `http://www.aspectc.net`.

L. Aversano, L. Cerulo, and M. D. Penta. How clones are maintained: An empirical study. In *Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 81–90, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2802-3.

B. S. Baker. On finding duplication and near-duplication in large software systems. In *Second Working Conference on Reverse Engineering (WCRE'95)*, pages 86–95. IEEE Computer Society Press, July 1995.

M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 98–107. IEEE Computer Society Press, November 2000.

T. Ball and S. K. Rajamani. The slam project: debugging system software via static analysis. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–3. ACM, January 2002.

E. L. A. Baniassad, G. C. Murphy, C. Schwanninger, and M. Kircher. Managing crosscutting concerns during software evolution tasks: an inquisitive study. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development (AOSD'02)*, pages 120–126, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-469-X.

Project Bauhaus. Website, September 2007. URL `http://www.bauhaus-stuttgart.de`.

I. D. Baxter. Design maintenance systems. *Communications of the ACM*, 35(4):73–89, 1992. ISSN 0001-0782.

I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 368–377. IEEE Computer Society Press, November 1998.

J. A. Bergstra and P. Klint. The discrete time TOOLBUS — a software coordination architecture. *Science of Computer Programming*, 31(2-3):205–229, 1998.

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.

D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2006.

B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

M. van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Software: Practice and Experience*, 30(3):259–291, 2000a.

M. van den Brand, M. P. A. Sellink, and C. Verhoef. Generation of components for software renovation factories from context-free grammars. *Science of Computer Programming*, 36 (2-3):209–266, 2000b.

M. 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, *Proceedings of Compiler Construction (CC '01)*, volume 2027 of *Lecture Notes in Computer Science*, pages 365–370. Springer-Verlag, 2001.

M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser. Stratego/XT 0.16. a language and toolset for program transformation. *Science of Computer Programming*, 2007. To appear.

S. Breu and J. Krinke. Aspect mining using event traces. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 310–315. IEEE Computer Society, September 2004.

M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces, and the Incremental Approach*. Morgan Kaufmann, 1995. ISBN 1-55860-330-1.

M. Bruntink. Aspect mining using clone class metrics. In *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE'04)*, number SEN-E0502 in CWI technical reports, pages 23–27, November 2005.

M. Bruntink. Linking analysis and transformations tools with source-based mappings. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 107–116. IEEE Computer Society Press, September 2006.

M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. An evaluation of clone detection techniques for identifying crosscutting concerns. In *Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 200–209. IEEE Computer Society Press, September 2004a.

M. Bruntink, A. van Deursen, and T. Tourwé. An initial experiment in reverse engineering aspects. In *Proceedings of the 11th Working Conference on Reverse Engineering (WCRE'04)*, pages 306–307. IEEE Computer Society, 2004b.

M. Bruntink, A. van Deursen, M. D'Hondt, and T. Tourwé. Simple crosscutting concerns are not so simple – analysing variability in large-scale idioms-based implementations. In *Proceedings of the Sixth International Conference on Aspect-Oriented Software Development (AOSD'07)*, pages 199–211. ACM Press, March 2007.

M. Bruntink, A. van Deursen, and T. Tourwé. Isolating idiomatic crosscutting concerns. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 37–46. IEEE Computer Society Press, September 2005a.

M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 242–251. ACM Press, May 2006.

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, October 2005b.

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.

M. Bush. Improving software quality: the use of formal inspections at the jpl. In *Proceedings of the International Conference on Software Engineering*, pages 196–199. IEEE Computer Society, 1990.

W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. A qualitative analysis of three aspect mining techniques. In *Proceedings of the International Workshop on Program Comprehension (IWPC'05)*, pages 13–22. IEEE Computer Society Press, May 2005.

M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwe. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 3(14):209–231, September 2006.

H. Chen and D. Wagner. Mops: an infrastructure for examining security properties of software. In *ACM Conference on Computer and Communications Security*, pages 235–244. ACM, November 2002.

E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990. ISSN 0740-7459.

F. Christian. *Exception handling and tolerance of software faults*, chapter 4, pages 81–107. John Wiley & Sons, 1995.

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.

CodeSurfer. Website, September 2007. URL `http://www.grammatech.com`.

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. ISBN 1-58113-842-3.

A. Colyer, A. Rashid, and G. Blair. On the separation of concerns in program families. Technical report, Computing Department, Lancaster University, 2004.

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

J. R. Cordy. Generalized selective XML markup of source code using agile parsing. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC)*, pages 144–153. IEEE Computer Society, May 2003.

J. R. Cordy. The TXL source transformation language. *Science of Computer Programming*, 61(3):190–210, 2006. ISSN 0167-6423.

J. R. Cordy, K. A. Schneider, T. R. Dean, and A. J. Malton. HSML: Design directed source code hot spots. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC)*, pages 145–156. IEEE Computer Society, May 2001.

J. R. Cordy and J. J. Vinju. How to make a bridge between transformation and analysis technologies? In J. R. Cordy, R. Lämmel, and A. Winter, editors, *Transformation Techniques in Software Engineering*, number 5161 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum (IBFI). URL `http://drops.dagstuhl.de/opus/volltexte/2006/426`.

M. Das, S. Lerner, and M. Seigle. Esp: Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 57–68. ACM, May 2002.

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

A. van Deursen. *De software-evolutieparadox*. Technische Universiteit Delft, 23 februari 2005. Inaugural Lecture.

A. van Deursen, P. Klint, and C. Verhoef. Research issues in the renovation of legacy systems. In *Proceedings of the Second Internationsl Conference on Fundamental Approaches to Software Engineering (FASE '99)*, volume 1577 of *Lecture Notes In Computer Science*, pages 1–21, London, UK, 1999. Springer-Verlag. ISBN 3-540-65718-5.

A. van Deursen and T. Kuipers. Building documentation generators. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 40–49. IEEE Computer Society, 1999a.

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, 1999b.

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.

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

S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 109–118. IEEE Computer Society Press, 1999.

P. Durr, G. Gulesir, L. Bergmans, M. Aksit, and R. van Engelen. Applying AOP in an industrial context. URL `http://trese.cs.utwente.nl/publications/files/0407BPAOSD2006.pdf`. Appeared at the Workshop on Best Practices in Applying Aspect-Oriented Software Development (BPAOSD'06), co-located with the 5th International Conference on Aspect-Oriented Software Development (AOSD'06), 2006.

M. Dyer. The cleanroom approach to quality software development. In *Proceedings of the 18th International Computer Measurement Group Conference*, pages 1201–1212. Computer Measurement Group, 1992.

J. Ebert, K. Kontogiannis, and J. Mylopoulos, editors. *Dagstuhl Seminar Interoperability of Reengineering Tools*, Schloss Dagstuhl, Germany, January 2001. Internationales Begegnungs- und Forschungszentrum (IBFI).

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

E. van Emden and L. Moonen. Java quality assurance by detecting code smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE '02)*, pages 97–. IEEE Computer Society, November 2002.

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

R. Ettinger and M. Verbaere. Untangling: A slice extraction refactoring. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 93–101. ACM Press, March 2004.

R. Fanta and R. Václav. Removing clones from the code. *Journal of Software Maintenance: Research and Practice*, 11(4):223–243, July/August 1999.

N. E. Fenton and S. L. Pfleeger. *Software Metrics: A rigorous and Practical Approach*. PWS Publishing Company, second edition, 1997.

R. Ferenc and Á. Beszédes. Data exchange with the columbus schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 59–66. IEEE Computer Society, March 2002.

F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: the devil is in the details. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 152–162. ACM Press, 2006.

F. C. Filho, A. Garcia, and C. M. F. Rubira. Extracting error handling to aspects: A cookbook. In *Proceedings of the International Conference on Software Maintenance (ICSM'07)*, pages 134–143. IEEE Computer Society, October 2007.

R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical Report 01.12, Research Institute for Advanced Computer Science, May 2001. Workshop on Advanced Separation of Concerns, OOPSLA 2000.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245. ACM, 2002.

C. Fu, A. Milanova, B. G. Ryder, and D. G. Wonnacott. Robustness testing of java server applications. *IEEE Transactions on Software Engineering*, 31(4):292 – 311, 2005.

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

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

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

N. H. Gehani. Exceptional C or C with exceptions. *Software Practice and Experience*, 22 (10):827–848, 1992. ISSN 0038-0644.

J. B. Goodenough. Exception handling: issues and a proposed notation. *Commununications of the ACM*, 18(12):683–696, 1975. ISSN 0001-0782.

W. G. Griswold, J. J. Yuan, and Y. Kato. Exploiting the map metaphor in a tool for software evolution. In *Proceedings of the International Conference on Software Engineering (ICSE'01)*, pages 265–274. IEEE Computer Society Press, March 2001.

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

J. Hannemann and G. Kiczales. Overcoming the prevalent decomposition in legacy code. URL `http://www.cs.ubc.ca/~jan/amt`. Appeared at the Workshop on Advanced Separation of Concerns, co-located with the 23rd International Conference on Software Engineering (ICSE'01), May 2001.

R. Holt. Structural manipulations of software architecture using tarski relational algebra. In *Proceedings of the 5th Working Conference on Reverse Engineering (WCRE'98)*, pages 210–219. IEEE Computer Society, October 1998.

R. C. Holt, A. E. Hassan, B. Laguë, S. Lapierre, and C. Leduc. E/R schema for the datrix C/C++/Java exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 284–286. IEEE Computer Society, November 2000a.

R. C. Holt, A. Winter, and A. Schürr. GXL: Toward a standard exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*, pages 162–171. IEEE Computer Society, November 2000b.

D. Jin and J. R. Cordy. Ontology-based software analysis and reengineering tool integration: The OASIS service-sharing methodology. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 613–616. IEEE Computer Society, September 2005.

J. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the IBM Centre for Advanced Studies Conference (CASCON'93)*, pages 171–183. IBM Press, October 1993.

S. Johnson. Lint, a C program checker. Technical Report 65, Bell Laboratories, Dec. 1977.

T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):645–670, July 2002.

C. Kapser and M. W. Godfrey. "Cloning considered harmful" considered harmful. In *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006)*, pages 19–28, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2719-1.

Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proceedings of the International Conference on Software Maintenance (ICSM'01)*, pages 736–743. IEEE Computer Society, 2001.

A. Kellens, K. Mens, and P. Tonella. A survey of automated code-level aspect mining techniques. *Transactions on Aspect-Oriented Software Development*, 4:145–164, 2007.

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

M. Kim, L. Bergman, T. A. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'04)*, pages 83–92. IEEE Computer Society Press, August 2004.

M. Kim, V. Sazawal, D. Notkin, and G. C. Murphy. An empirical study of code clone genealogies. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/SIGSOFT FSE)*, pages 187–196. ACM Press, September 2005.

B. A. Kitchenham, G. H. Travassos, A. von Mayrhauser, F. Niessink, N. F. Schneidewind, J. Singer, S. Takada, R. Vehvilainen, and H. Yang. Towards an ontology of software maintenance. *Journal of Software Maintenance*, 11(6):365–389, 1999. ISSN 1040-550X.

R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *Proceedings of the 8th International Symposium on Static Analysis (SAS'01)*, volume 2126 of *Lecture Notes In Computer Science*, pages 40–56. Springer-Verlag, July 2001.

J. Kort and R. Lämmel. Parse-tree annotations meet re-engineering concerns. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 161–170. IEEE Computer Society, 2003.

R. Koschke, E. Merlo, and A. Walenstein, editors. *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Schloss Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI). URL `http://drops.dagstuhl.de/opus/volltexte/2007/972`.

J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the Eight Working Conference On Reverse Engineering (WCRE'01)*, pages 301–109. IEEE Computer Society Press, 2001.

J. Lang and D. B. Stewart. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. *ACM Transactions on Programming Languages and Systems*, 20(2):274 – 301, 1998.

P. A. Lee. Exception handling in C programs. *Software: Practice and Experience*, 13(5): 389–405, 1983.

M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. Academic Press, London, 1985. ISBN 0-12-442440-6.

T. Lethbridge, S. Tichelaar, and E. Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. *Electronic Notes in Theoretical Computer Science*, 94:7–18, 2004.

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

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.

J.-L. Lions. Ariane 5 flight 501 failure. Technical report, ESA/CNES, 1996.

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

B. Littlewood. Dependability assessment of software-based systems: state of the art. In *Proceedings of the International Conference on Software Engineering*, pages 6–7. ACM Press, 2005. ISBN 1-59593-963-2.

A. J. Malton, K. A. Schneider, J. R. Cordy, T. R. Dean, D. Cousineau, and J. Reynolds. Processing software source text in automated design recovery and transformation. In *Proceedings of the 9th International Workshop on Program Comprehension (IWPC)*, pages 127–134. IEEE Computer Society, May 2001.

A. Marcus and J. I. Maletic. Identification of high-level concept clones in source code. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 107–114. IEEE Computer Society, November 2001.

M. Marin, A. van Deursen, and L. Moonen. Identifying crosscutting concerns using fan-in analysis. *ACM Transactions on Softare Engineering and Methodology*, 2006. To appear.

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

J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM'96)*, pages 244–254. IEEE Computer Society Press, November 1996.

M. Messier and J. Viega. XXL. Website, October 2007. URL `http://www.zork.org/xxl`.

G. Mishne and M. de Rijke. Source code retrieval using conceptual similarity. In *Proceedings of the 2004 Conference on Computer Assisted Information Retrieval (RIAO'04)*, pages 539–554, Paris, April 2004. C.I.D.

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.

L. Moonen. Generating robust parsers using island grammars. In *Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE)*, pages 13–22. IEEE Computer Society, October 2001.

M. Mortensen and S. Ghosh. Refactoring idiomatic exception handling in C++: Throwing and catching exceptions with aspects. URL `http://aosd.net/2007/program/industry/I2-RefactoringExceptionsC++.pdf`. Appeared at the industry track of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07), March 2007.

H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

G. C. Murphy, W. G. Griswold, M. P. Robillard, J. Hannemann, and W. Leong. Design recommendations for concern elaboration tools. In R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, pages 507–530. Addison-Wesley, Boston, 2005. ISBN 0-321-21976-7.

G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating features in source code: An exploratory study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 275–284. IEEE Computer Society, 2001.

M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the 5th symposium on Operating systems design and implementation (OSDI'02)*, pages 75 – 88. USENIX Association, 2002.

S. Paul and A. Prakash. A framework for source code search using program patterns. *IEEE Transactions on Software Engineering*, 20(6):463–475, 1994.

C. Potts. Software-engineering research revisited. *IEEE Software*, 10(5):19–28, 1993. ISSN 0740-7459.

J. F. Power and B. A. Malloy. Program annotation in XML: A parse-tree based approach. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pages 190–198. IEEE Computer Society, October 2002.

J. M. Purtilo and J. R. Callahan. Parse tree annotations. *Communications of the ACM*, 32 (12):1467–1477, 1989.

M. Rieger, S. Ducasse, and G. Golomingi. Tool support for refactoring duplicated OO code. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'99)*, pages 177–178, Germany, June 1999. Springer.

E. S. Roberts. Implementing exceptions in C. Technical Report 40, Digital Systems Research Center, 1989.

M. Robillard and G. C. Murphy. Regaining control of exception handling. Technical Report TR-99-14, Department of Computer Science, University of British Columbia, 1999.

M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Transactions on Software Engineering and Methodology*, 12(2):191–221, 2003. ISSN 1049-331X.

F. van Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)*, pages 126–130. IEEE Computer Society Press, September 2003.

F. van Rysselberghe and S. Demeyer. Evaluating clone detection techniques from a refactoring perspective. In *Proceedings of the 9th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 336–339. IEEE Computer Society Press, September 2004.

M. P. A. Sellink and C. Verhoef. Scaffolding for software renovation. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 161–172. IEEE Computer Society, February 2000.

D. Shepherd, E. Gibson, and L. L. Pollock. Design and evaluation of an automated aspect mining tool. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP'04)*, pages 601–607. CSREA Press, June 2004.

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.

S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in java programs. In *Proceedings of the International Conference on Software Maintenance*, pages 265–. IEEE Computer Society, 1999.

D. B. Smith, H. A. Mller, and S. R. Tilley. The year 2000 problem: Issues and implications. Technical Report CMU/SEI-97-TR-002, Software Engineering Institute, 1997.

R. T. Snodgrass and K. Shannon. Supporting flexible and efficient tool integration. In *Advanced Programming Environments, Proceedings of an International Workshop*, Lecture Notes in Computer Science, pages 290–313. Springer, June 1986.

E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

P. Tarr, H. Ossher, W. Harrison, and S. M. J. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software engineering (ICSE'99)*, pages 107–119. IEEE Computer Society Press, May 1999.

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.

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. ISBN 0-7695-2144-4.

T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 91 – 100. IEEE Computer Society, 2003.

W. N. Toy. Fault-tolerant design of local ess processors. In *Proceedings of IEEE*, pages 1126–1145. IEEE Computer Society, 1982.

C. van Rijsbergen. *Information Retrieval*. Butterworths, London, 2nd edition edition, 1979.

N. P. Veerman. Revitalizing modifiability of legacy assets. *Journal of Software Maintenance*, 16(4-5):219–254, 2004.

J. J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. PhD thesis, University of Amsterdam, 2005.

E. Visser. Scannerless generalized-LR parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997. URL `http://ftp.science.uva.nl/pub/programming-research/reports/1997/P9707.p%s.Z`.

A. Walenstein. Problems creating task-relevant clone detection reference data. In *Proceedings of the Tenth Working Conference on Reverse Engineering (WCRE'03)*, pages 285–294. IEEE Computer Society Press, November 2003.

A. Walenstein and A. Lakhotia. Clone detector evaluation can be improved: Ideas from information retrieval. URL `http://citeseer.ist.psu.edu/761253.html`. Appeared at the 2nd International Workshop on the Detection of Software Clones (IWDSC'03), held in conjunction with the 10th Working Conference on Reverse Engineering (WCRE'03), November 2003.

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

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

H. Winroth. Exception handling in ANSI C. Technical Report ISRN KTH NA/P–93/15–SE, Department of Numerical Analysis and Computing Science, Royal Institute of Technology, Stockholm, Sweden, 1993.

J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *6th Symposium on Operating System Design and Implementation*, pages 273–288. USENIX Association, 2004.

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.

C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 130–139. ACM, March 2003.

C. Zhang and H.-A. Jacobsen. PRISM is research in aSpect mining. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*, pages 20–21, Boston, MA, 2004. ACM.

Summary

This thesis investigates the phenomenon of idiomatic crosscutting concerns in embedded software systems. In particular, we consider the renovation of idiomatic crosscutting concerns using Aspect-Oriented Programming (AOP).

Crosscutting concerns are phenomena that are present in almost any (embedded) software system. They arise if the implementation of a concern –a requirement or design decision– does not fit neatly into the modular decomposition of a software system. A crosscutting concern cannot be confined to a single modular unit and therefore becomes scattered across the system and tangled with other concerns. This thesis focuses on the specific class of *idiomatic crosscutting concerns*, which are crosscutting concerns that are *idiomatic* in the sense that they are implemented manually by applying an idiom, resulting in many similar pieces of source code.

The approach taken is that of *renovation*, i.e., a step-wise improvement process aimed at easing the evolution of *legacy software systems*. The legacy software system that is studied in this thesis is the (embedded) control software of an ASML wafer scanner, a device used in the manufacturing process of integrated circuits. This software system consists of 15 million lines of C code. We study whether the use of AOP is beneficial compared to the idiomatic style of implementation used in the ASML software system.

In systems developed without AOP, code implementing a crosscutting concern may be spread over many different parts of a system. Identifying such code automatically (aspect mining) could be of great help during maintenance of the system. In Chapter 2, we evaluate the suitability of clone detection as a technique for the identification of crosscutting concerns. To that end, we manually identify five crosscutting concerns in the ASML C system, and analyze to what extent clone detection is capable of finding them.

Chapter 3 reports on our experience in automatically renovating the crosscutting concerns of the ASML C system using AOP. We present a systematic approach for isolating crosscutting concerns, and illustrate this approach by zooming in on one particular crosscutting concern: parameter checking. Additionally, we compare the legacy solution to the AOP solution, and discuss advantages as well as disadvantages of both in terms of selected quality attributes. Our results show that automated renovation is technically feasible, and that adopting an AOP approach for parameter checking can lead to significant improvements in source code quality, if carefully designed and managed.

Automatic renovation requires a source code analysis and transformation infrastructure to be carried out. In the case studies presented in this thesis different technologies are used to support various renovation tasks. Chapter 4 therefore discusses an approach to linking

separate analysis and transformation tools, such that analysis results can be used to guide transformations. The approach consists of two phases. First, the analysis tool maps its results to relevant locations in the source code. Second, a mapping in the reverse direction is performed: the analysis results expressed as source positions and data are mapped to the abstractions used in the transformation tool. We discuss a prototype implementation of this approach in detail, and present the results of two applications within the context of the ASML C system.

Idiomatic implementation is a manual and repetitive task. Crosscutting concerns, if implemented idiomatically, may therefore be particularly fault prone. In Chapter 5 we analyze the exception handling idiom of the ASML C system. Like many systems implemented in classic programming languages (e.g, C), the ASML system uses the popular return-code idiom for dealing with exceptions. Our goal is to evaluate the fault-proneness of this idiom, and we therefore present a characterization of the idiom, a fault model accompanied by an analysis tool, and empirical data. Our findings show that the idiom is indeed fault prone, but that a simple solution can lead to significant improvements.

Chapter 6 describes a method for studying idioms-based implementations of crosscutting concerns, and our experiences with it in the context of the ASML C system. In particular, we analyze 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 AOP could help prevent it, how it can paralyze (automated) renovation efforts, and which AOP 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.

Some legacy programming languages, e.g., C, do not provide adequate support for exception handling. As a result, users of these legacy programming languages often implement exception handling by applying an idiom. An idiomatic style of implementation has a number of drawbacks: applying idioms can be fault prone and requires significant effort. Modern programming languages provide support for Structured Exception Handling (SEH) that makes idioms largely obsolete. Additionally, AOP is believed to further reduce the effort of implementing exception handling. Chapter 7 investigates the gains that can be achieved by re-engineering the idiomatic exception handling of an ASML C component to these modern techniques. First, we re-engineer the legacy component such that its exception handling idioms are almost completely replaced by SEH constructs. Second, we show that the use of AOP for exception handling can be beneficial, even though the benefits are limited by inconsistencies in the legacy implementation.

The research presented in this thesis was performed in the context of ASML. The results can be generalized as follows:

- We have shown that applying idioms in an industrial setting can result in implementations that frequently violate the idioms used. This observation can probably be repeated for numerous other software systems that employ idioms. (Chapters 5, 6, and 7)

- The SMELL tool presented in Chapter 5 is a means of improving upon the practice of idiomatic implementation. SMELL is geared specifically for the ASML exception handling idiom, but the general technique used to implement SMELL is not and can be applied in different contexts. The Metal language could be used to specify similar tools

for other idioms. Furthermore, CodeSurfer and its sibling application CodeSonar are both extensible program analysis toolkit that allow for the rapid development of idiom checker tools like SMELL.

- Chapter 6 demonstrated a generic method to explore idiom violations in legacy source code. This method is based on formal concept analysis, which can be applied in many contexts. Significant knowledge of the idiom that is being studied is required to initialize this method, but interpretation of the results is based on generic properties of formal concept analysis that are not limited to the ASML context.

- Renovation processes typically consist of separate analysis and transformation phases that must use the results of one another. The source-based mapping approach that was developed in Chapter 4 provides a semi-generic solution to the problem of linking separate analysis and transformation tools together. The current implementation of this approach, called SCATR, is limited to a specific transformation tool, i.e., the ASF+SDF Meta Environment, but generically supports any analysis tool that is capable of generating SCATR's format.

- Exception handling is a concern in any industrial software system. Chapter 7 shows the limits of using aspects for exception handling in a case study at ASML. This case study provides evidence that using aspects for exception handling cannot be expected to result in great benefits. These results match other evidence presented within the aspect-oriented programming community.

Samenvatting

Dit proefschrift onderzoekt idiomatische crosscutting concerns die voorkomen in embedded softwaresystemen. In het bijzonder beschouwt het proefschrift de renovatie van idiomatische crosscutting concerns met behulp van aspect-georiënteerd programmeren (AOP).

Crosscutting concerns komen voor in bijna elk (embedded) softwaresysteem. Ze ontstaan als de implementatie van een bepaalde functie van een softwaresysteem niet netjes past binnen de modulaire structuur van het systeem. Het gevolg is dat de functie op verschillende plaatsen in het systeem moet worden geïmplementeerd. Dit leidt tot een versnipperde implementatie die bovendien vervlochten kan raken met de implementatie van andere functies. Als bij de implementatie bovendien gebruik is gemaakt van een idioom spreek ik van een idiomatisch crosscutting concern. Het gebruik van een idioom heeft tot gevolg dat de implementatie bestaat uit meerdere, sterk gelijkende, stukjes programmacode. Deze stukjes programmacode zijn versnipperd over de modulestructuur van het softwaresysteem, en kunnen vervlochten zijn met andere stukjes programmacode.

De aanpak die in het proefschrift wordt gebruikt gaat uit van een renovatieproces dat als doel heeft om de evolutie van bestaande softwaresystemen makkelijker te maken. In het bijzonder beschouw ik het controlesysteem van een ASML wafer scanner, een apparaat dat ingezet wordt bij de productie van chips. Een wafer scanner bevat een immens softwaresysteem dat bestaat uit ruwweg 15 miljoen regels programmacode. Mijn onderzoek richt zich op de vraag of het softwaresysteem van een wafer scanner verbeterd kan worden door het gebruik van idiomen te vervangen door het gebruik van AOP. Het gebruik van idiomen kenmerkt zich door het relatief veel voorkomen van gelijkvormige stukjes programmacode, die zijn geproduceerd aan de hand van een voorschrift (het idioom).

In softwaresystemen die geen gebruik maken van AOP kan het voorkomen dat de implementatie van crosscutting concerns versnipperd is geraakt. Het onderhoud van crosscutting concerns kan hierdoor worden bemoeilijkt. Als de versnipperde programmacode van een crosscutting concern automatisch zou kunnen worden aangewezen, dan zou daarmee de taak van onderhoud verlicht kunnen worden. In hoofdstuk 2 evalueer ik in welke mate dat doel zou kunnen worden bereikt door het gebruik van drie automatische duplicatie-detectietechnieken. Met de hand worden vijf crosscutting concerns aangeduid in de programmacode, die vervolgens vergeleken worden met de resultaten van de automatische duplicatie-detectietechnieken. Uit deze vergelijking blijkt dan in welke mate de automatische duplicatie-detectietechnieken in staat zijn om crosscutting concerns zelfstandig aan te wijzen.

Hoofdstuk 3 beschrijft de ervaringen die zijn opgedaan bij het automatisch renoveren van crosscutting concerns in het ASML softwaresysteem. Hierbij is steeds gekeken naar een

alternatief voor de huidige implementatie, waarbij het alternatief gebruik maakt van AOP. Er wordt eerst een systematische aanpak beschreven, die daarna wordt geïllustreerd aan de hand van een concreet crosscutting concern: de controle van parameters. De alternatieve implementatie die gebruikt maakt van AOP wordt vervolgens vergeleken met de oude implementatie. Hiervoor worden een aantal bekende kwaliteitsmaten ingezet. De resultaten laten zien dat het technisch haalbaar is om automatische renovatie in te zetten, en dat met behulp van AOP significante verbeteringen kunnen worden behaald. Er blijkt echter ook dat het nodig is om eerst een geschikt ontwerp te maken, en zorgvuldig met AOP om te gaan.

Het automatisch renoveren van programmacode veronderstelt een infrastructuur die in staat is om programmacode te analyseren en te transformeren. In de case studies die in dit proefschrift gepresenteerd worden zijn daarvoor verscheidene analyse- en transformatietools gebruikt. De ervaring leert dat het noodzakelijk is om analyseresultaten te kunnen overdragen aan de transformatietools, zodat die vervolgens de analyseresultaten kunnen gebruiken. In hoofdstuk 4 bespreek ik daarom hoe ik de gebruikte analyse- en transformatietools koppel, zodanig dat uitwisseling van resultaten mogelijk is. Hiervoor wordt de volgende aanpak gebruikt: Eerst wijst het analysetool die plekken in de programmacode aan die relevant zijn voor de analyseresultaten. De analyseresultaten worden vervolgens op die plekken ingevoegd in de programmacode. Tenslotte leest het transformatietool de programmacode weer in, en koppelt de analyseresultaten aan de abstracties die relevant zijn voor de uit te voeren transformaties. In hoofdstuk 4 wordt een prototype besproken dat deze aanpak implementeert, gevolgd door een tweetal demonstraties van de toepassing van dit prototype op het ASML softwaresysteem.

De implementatie van crosscutting concerns met behulp van idiomen is repetitief, handmatig, werk. Het zou dus kunnen voorkomen dat implementaties van idiomatische crosscutting concerns bijzonder veel defecten bevatten. In hoofdstuk 5 wordt het foutafhandelingsmechanisme van het ASML softwaresysteem beschouwd. Dit mechanisme is geïmplementeerd met behulp van het zogenaamde return-code idioom, dat veel voorkomt in softwaresystemen die in oudere programmeertalen, zoals C, beschreven zijn. Het doel in dit hoofdstuk is om de de dichtheid van defecten te bepalen in het foutafhandelingsmechanisme van het ASML softwaresysteem. Daartoe wordt het return-code idioom eerst beschreven, en worden defecten die mogelijk kunnen optreden bij het toepassen van het idioom vastgelegd in een defectmodel. Het defectmodel vormt de basis voor een tool dat in staat is de defecten te vinden in het foutafhandelingsmechanisme. Dit tool wordt toegepast op de programmacode ten einde de dichtheid van defecten te bepalen. De verkregen meetresultaten laten zien dat de dichtheid van defecten, zoals verwacht, hoog is. Tenslotte beschouwt hoofdstuk 5 een simpele oplossing die significante verbetering kan bieden.

In hoofdstuk 6 beschrijf ik een methode waarmee idiomatische crosscutting concerns kunnen worden bestudeerd, en de ervaringen die zijn opgedaan met deze methode tijdens de uitgevoerde cases voor het ASML softwaresysteem. In het bijzonder wordt er ingegaan op het crosscutting concern dat verantwoordelijk is voor het traceren van de executie van de software (tracing). Dit schijnbaar eenvoudige crosscutting concern blijkt significant veel variatie te bevatten, ondanks dat er een idioom is voorgeschreven aan de programmeurs. Ik bespreek hoe de waargenomen variatie een probleem vormt voor automatische renovatie, hoe AOP mogelijk dit soort variatie kan voorkomen, en welke eigenschappen een AOP-taal zou moeten bezitten ten einde een wenselijk alternatief mogelijk te maken. Tenslotte wordt

verder ingegaan op de bruikbaarheid van de gekozen methode, en de representativiteit van de behaalde resultaten.

Sommige oudere programmeertalen, zoals C, leveren onvoldoende ondersteuning voor de implementatie van foutafhandelingsmechanismes. Het gevolg daarvan is dat gebruikers van zulke oude programmeertalen terugvallen op het gebruik van een idioom, zoals bijvoorbeeld het return-code idioom. Het gebruik van idiomen heeft als nadeel dat er gemakkelijk fouten mee gemaakt kunnen worden, en dat het een arbeidsintensieve praktijk is. Moderne programmeertalen bevatten daarom vaak een vorm van structured exception handling (SEH), dat het gebruik van idiomen grotendeels onnodig maakt. Daarnaast zou het gebruik van AOP voordelen bieden voor de implementatie van foutafhandelingsmechanismes. In hoofdstuk 7 onderzoek ik de voordelen van deze beide technieken voor het foutafhandelingsmechanisme van het ASML softwaresysteem. Er wordt hierbij uitgegaan van de renovatie van een bestaande softwarecomponent, die stapsgewijs herschreven wordt. Eerst laat ik zien dat het mogelijk is om het oude idioom bijna geheel te vervangen door SEH. Daarna toon ik aan dat het gebruik van AOP ook voordelen kan bieden voor de implementatie van foutafhandelingsmechanismes. Er blijkt echter ook dat inconsistenties die voorkomen in het gebruik van het oude idioom de voordelen van AOP kunnen begrenzen.

Het onderzoek dat in dit proefschrift beschreven wordt heeft plaatsgevonden in de context van ASML. Desalniettemin stel ik de volgende generalisaties van de resultaten voor:

- Er is meerdere malen gebleken dat het toepassen van idiomen op industriële schaal kan leiden tot frequente schendingen van de voorgeschreven idiomen. Het is zeer waarschijnlijk dat dit ook voorkomt bij andere (industriële) softwaresystemen. (Hoofdstukken 5, 6, en 7)

- Het tool SMELL dat wordt beschreven in hoofdstuk 5 zou kunnen worden ingezet om het gebruik van idiomen tot minder defecten te laten leiden. Hoewel SMELL is toegespitst op het foutafhandelingsmechanisme van ASML zou de onderliggende techniek ook in andere contexten kunnen worden toegepast. De taal Metal zou bijvoorbeeld geschikt zijn om dergelijke tools te maken. Daarnaast bieden de uitbreidbare tools CodeSurfer en CodeSonar de mogelijkheid om snel tools als SMELL te ontwikkelen.

- Hoofdstuk 6 beschouwt een algemene methode om idioomschendig-en in bestaande programmacode in kaart te brengen. Deze methode is gebaseerde op (formele) conceptanalyse, en is voor de interpretatie van de resultaten niet strikt afhankelijk van specifieke kenmerken van ASML. De toepassing van deze methode op een ander idioom (binnen een andere context) vereist wel dat er veel kennis over het idioom wordt gebruikt om voldoende gegevens te verzamelen.

- Automatische renovatieprocessen bestaan typisch uit losstaande analyse- en transformatietools die moeten samenwerken. In hoofdstuk 4 wordt een mogelijke overdracht van gegevens tussen zulke tools beschreven. Het huidige prototype dat deze koppeling implementeert, genaamd SCATR, kan worden ingezet voor elk analysetool dat in staat is gegevens in SCATR's formaat te genereren. Op dit moment kan SCATR echter alleen omgaan met één enkel transformatietool, namelijk de ASF+SDF Meta-omgeving.

- Foutafhandelingsmechanismes komen voor in elk (industriëel) softwaresysteem. De resultaten in hoofdstuk 7 wijzen erop dat het gebruik van AOP voor de implementatie van foutafhandelingsmechanismes kan leiden tot verbeteringen, maar ook dat deze verbeteringen sterk beperkt zijn. Deze waarnemingen zijn in overeenstemming met het werk van andere onderzoekers naar het gebruik van AOP.

Curriculum Vitae

Full name
Magiel Bruntink

Date of birth
February 6, 1980

Place of birth
Delfzijl, the Netherlands

Nationality
Dutch

Education

October 2003 — March 2008
PhD student (AiO) at the Centrum Wiskunde & Informatica (CWI), Amsterdam, under the supervision of prof. dr. Arie van Deursen and prof. dr. Paul Klint. The research conducted during this period was done in the context of the Ideals research project, in cooperation with ASML and the Embedded Systems Institute in Eindhoven.

September 1998 — Augustus 2003
MSc and BSc degrees (both cum laude) in Computer Science, specialized in Software Engineering, at the University of Amsterdam. The MSc phase was completed by an internship at the Software Improvement Group, Amsterdam. The results of this internship were published in an international journal.

The first year of the BSc consisted of the Bèta-Gamma Propaedeusis programme at the University of Amsterdam. The Bèta-Gamma Propaedeusis is an interdisciplinary programme which offers the student a combination of natural- and social science courses. The major field of study during this first year was computer science, while the minor field of study consisted of economics. The first year examination (propedeuse) was passed cum laude in August 1999. For obtaining outstanding results during the first year of study, the “Civi Propedeuseprijs” was awarded.

Titles in the IPA Dissertation Series since 2002

M.C. van Wezel. *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects*. Faculty of Mathematics and Natural Sciences, UL. 2002-01

V. Bos and J.J.T. Kleijn. *Formal Specification and Analysis of Industrial Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02

T. Kuipers. *Techniques for Understanding Legacy Software Systems*. Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03

S.P. Luttik. *Choice Quantification in Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04

R.J. Willemen. *School Timetable Construction: Algorithms and Complexity*. Faculty of Mathematics and Computer Science, TU/e. 2002-05

M.I.A. Stoelinga. *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-06

N. van Vugt. *Models of Molecular Computing*. Faculty of Mathematics and Natural Sciences, UL. 2002-07

A. Fehnker. *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems*. Faculty of Science, Mathematics and Computer Science, KUN. 2002-08

R. van Stee. *On-line Scheduling and Bin Packing*. Faculty of Mathematics and Natural Sciences, UL. 2002-09

D. Tauritz. *Adaptive Information Filtering: Concepts and Algorithms*. Faculty of Mathematics and Natural Sciences, UL. 2002-10

M.B. van der Zwaag. *Models and Logics for Process Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11

J.I. den Hartog. *Probabilistic Extensions of Semantical Models*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12

L. Moonen. *Exploring Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13

J.I. van Hemert. *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining*. Faculty of Mathematics and Natural Sciences, UL. 2002-14

S. Andova. *Probabilistic Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2002-15

Y.S. Usenko. *Linearization in μCRL*. Faculty of Mathematics and Computer Science, TU/e. 2002-16

J.J.D. Aerts. *Random Redundant Storage for Video on Demand*. Faculty of Mathematics and Computer Science, TU/e. 2003-01

M. de Jonge. *To Reuse or To Be Reused: Techniques for component composition and construction*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02

J.M.W. Visser. *Generic Traversal over Typed Source Code Representations*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03

S.M. Bohte. *Spiking Neural Networks*. Faculty of Mathematics and Natural Sciences, UL. 2003-04

T.A.C. Willemse. *Semantics and Verification in Process Algebras with Data and Timing*. Faculty of Mathematics and Computer Science, TU/e. 2003-05

S.V. Nedea. *Analysis and Simulations of Catalytic Reactions*. Faculty of Mathematics and Computer Science, TU/e. 2003-06

M.E.M. Lijding. *Real-time Scheduling of Tertiary Storage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07

H.P. Benz. *Casual Multimedia Process Annotation – CoMPAs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08

D. Distefano. *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09

M.H. ter Beek. *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components*. Faculty of Mathematics and Natural Sciences, UL. 2003-10

D.J.P. Leijen. *The λ Abroad – A Functional Approach to Software Components*. Faculty of Mathematics and Computer Science, UU. 2003-11

W.P.A.J. Michiels. *Performance Ratios for the Differencing Method*. Faculty of Mathematics and Computer Science, TU/e. 2004-01

G.I. Jojgov. *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving*. Faculty of Mathematics and Computer Science, TU/e. 2004-02

P. Frisco. *Theory of Molecular Computing – Splicing and Membrane systems*. Faculty of Mathematics and Natural Sciences, UL. 2004-03

S. Maneth. *Models of Tree Translation*. Faculty of Mathematics and Natural Sciences, UL. 2004-04

Y. Qian. *Data Synchronization and Browsing for Home Environments*. Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications*. Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications*. Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems*. Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs*. Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell*. Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation*. Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets*. Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics*. Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station*. Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents*. Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures*. Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra*. Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling*. Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-*. Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of

Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architecturest*. Faculty of Electrical Engineering, Mathematics, and Computer Science Delft University of Technology. 2007-16

A.H.J.Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W.Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology. 2008-03