Resource-based Verification for Robust Composition of Aspects

Ir. Pascal Eugène Alois Dürr

Ph.D. dissertation committee:

Chairman and secretary:

Prof. dr. J.J.W. van der Vegt, University of Twente, The Netherlands

Promoter:

Prof. dr. ir. M. Akşit, University of Twente, The Netherlands

Assistant promoter:

Dr. ir. L.M.J. Bergmans, University of Twente, The Netherlands

Members:

Prof. dr. E. Brinksma, University of Twente, The Netherlands
Dr. A. Garcia MSc, Lancaster University, United Kingdom
Prof. dr. W. Joosen, Katholieke Universiteit Leuven, Belgium
Dr. S. Katz, The Technion, Israel
Dr. L. Ferreira Pires MSc, University of Twente, The Netherlands
Prof. dr. C. Wohlin, Blekinge Institute of Technology, Sweden

CTIT Ph.D. thesis Series No. 08-119. Centre for Telematics and Information Technology (CTIT), P.O. Box 217 - 7500 AE Enschede - The Netherlands.

This work has been carried out as part of the *Ideals* project under the responsibility of the *Embedded Systems Institute*. This project is partially supported by the Netherlands Ministry of Economic Affairs under the SenterNovem TS program (grant TSIT3003). The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

ISBN 978-90-365-2685-2
ISSN 1381-3617 (CTIT Ph.D. thesis Series No. 08-119)
IPA Dissertation Series 2008-15.

Cover design: *Blue Lagoon*, Iceland Oct. 2006 by Pascal Dürr
Printed by PrintPartners Ipskamp, Enschede, The Netherlands

RESOURCE-BASED VERIFICATION FOR ROBUST COMPOSITION OF ASPECTS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit van Twente,
op het gezag van de rector magnificus,

Prof. dr. W.H.M. Zijm,

volgens het besluit van het College voor Promoties
in het openbaar te verdedigen

op donderdag 26 Juni 2008 om 16.45 uur

door

Ir. Pascal Eugène Alois Dürr

geboren op 12 December 1977
te 's-Gravenhage

This dissertation is approved by

Prof. dr. ir. M. Akşit (promoter)
Dr. ir. L.M.J. Bergmans (assistant promoter)

ISBN 978-90-365-2685-2

"I believe all suffering is caused by ignorance. People inflict pain on others in the selfish pursuit of their happiness or satisfaction. Yet true happiness comes from a sense of peace and contentment, which in turn must be achieved through the cultivation of altruism, of love and compassion, and elimination of ignorance, selfishness, and greed."

Tenzin Gyatso, Dalai Lama,
Nobel Peace Prize laureate.

Acknowledgments

The first person I want to thank, is my daily supervisor and assistant-promoter, Lodewijk Bergmans. A lot of the results achieved in this thesis can be partially contributed to his guidance and advice. He not only encouraged me to do a PhD in the first place but also ensured that I never regretted it. During our four years we never had a dull moment and I have yet to remember a meeting or trip were we did not have a lot of fun. Although our professional relationship ends on the first of July, I am sure that I will continue our excellent personal relationship with you and your wife Ingrid.

Secondly, my promotor Mehmet Akşit, his initial elegant brain-child "Composition Filters" has had a huge impact on the thesis and has motivated my quest to continue to improve and extent this approach to software composition. His feedback, advice and guidance can be found throughout this thesis.

Hereby I also thank the other members of my defense committee: prof. dr. ir. Ed Brinksma, dr. Alessandro Garcia, prof. dr. Wouter Joosen, dr. Shmuel Katz, dr. Luis Ferreira Pires, and prof. dr. Claes Wohlin.

I would like to thank all members of the Software Engineering group and of the Formal Methods group. In particular, Tom Staijen with whom I have enjoyed great hiking trips in Ireland and Vancouver. Gürcan Gülesir, with whom I worked closely in the Ideals project and shared the burden of traveling to Veldhoven

every week for three years, Wilke Havinga, Klaas van der Berg, Arend Rensink and my sushi-buddy Mariëlle Stoelinga.

The work described in this thesis has been carried out within the Ideals project. From this project, I would like to mention several people: Magiel Bruntink (CWI), Tom Tourwé (CWI), Joris van der Aker (ESI), Jan-Mathijs Wijnands (Sioux), Ad van Dongen (ESI) and Frans Beenker (ESI). Part of the work described in this thesis has been conducted at ASML. And a lot of the results of especially the first chapters can partially be contributed to the effort of Remco van Engelen. The WeaveC team at ASML really helped out with the experiment, in particular Steven de Brouwer and Istvan Nagy.

Conducting controlled experiments was a completely new endeavor for me. Fortunately, Vincent Buskens and Richard Zijdeman from the Department of Sociology at the University of Utrecht helped out with their expertise in the area of controlled experiments.

During my PhD, I continued to live in a dorm called *HabeDabeDebeDoe*. We had some great activities the past 6 years. Thanks for all the great times. I especially would like to mention: Daphne de Klerk, Niels Zijlstra, Steven Fokkenrood, Jeroen Jonker, Kirsten Rutgers and Ruben Wassink.

Ben Bruggeman also influenced this thesis, I will never forget our "Den-Helder" sessions. Piet van der Vlist, rose to the same challenge as me, finishing his PhD one year earlier under much tougher conditions.

Diana, although our timing could have been better, the prospect of being with you motivated me during the last months. I am thankful that I went to Nepal, where I met you and 21 other great friends.

Finally, I would to thank my family:

My oldest sister Natasha, her boyfriend Alex, and their two children Shanon and Meagan. Almost every summer during my PhD, I did some "real" engineering work with them, building a hangar and their future home.

My twin sister Nicôle and her husband Richard, my twin sister is my paranymph and my brother in law helped out with the statistics in this thesis, and will finish his PhD in September, good luck!

And last, but definitely not least, my parents Lia and Eugène. They have encouraged and supported me the entire four yours. Especially my father did not only impact the defense ceremony as a paranymph, but also a lot of his influence can be found throughout the thesis.

Abstract

Aspect Oriented Software Development has been proposed as a means to improve modularization of software in the presence of crosscutting concerns. Compared to object-oriented or procedural approaches, Aspect Oriented Programming (AOP) has not yet been applied in many industrial applications. In this thesis we investigate the application of AOP within an industrial context and propose a novel solution to the problem of behavioral conflicts between aspects.

We report on our experience transferring an aspect-oriented solution to a company called Advanced Semi-conductor Material Lithography (ASML). We investigate the acceptance criteria for AOP in industry, based on two industrial cases studies. We present a process that includes quantification of the benefits of AOP and elicitation of key worries expressed by stakeholders.

We conducted a controlled experiment to assess the advantages and disadvantages of an aspect-based approach using a tracing example. Twenty developers from ASML were requested to carry out five maintenance scenarios. This experiment has shown that, in case the tracing concern is implemented using an AOP implementation instead of a procedural language, the development effort is on average 6% reduced while the impact of errors is reduced by 77%, for maintaining code related to tracing. For a subset of the scenarios, the results were statistically significant on a confidence interval of 95%.

The so-called aspect interference problem is one of the major concerns in introducing AOP. Aspects can be developed independently and behave correct in isolation. However, due to intended or unintended composition of aspects, undesired behavior can emerge. In this thesis we focus on behavioral conflicts between aspects at shared join points. These are illustrated by a realistic example based on crosscutting concerns from ASML. We present an approach for the detection of behavioral interference that is based on a novel abstraction of the behavior of aspects, using resources and operations. This abstraction enables the expression of behavior in a simple manner that is suitable for automated detection of interference among aspects. The approach employs a set of conflict detection rules that can be used to detect both generic conflicts as well as domain specific conflicts.

Our approach is general for AOP languages, its application to one specific AOP language Composition Filters is also illustrated in this thesis. The application to Composition Filters demonstrates how the use of a declarative advice language can be exploited for automated conflict detection. We detail the analysis process and discuss what information is required from the aspect developer to be able perform the analysis. We also discuss when static analysis is insufficient for detecting behavioral conflicts. We present a run time extension aiming at detecting dynamic conflicts. We discuss optimizations for this run time approach, which exploits the static verification results.

Finally, we propose three improvements to the Composition Filters model to support automated and manual reasoning even further. The first improvement separates *what* behavior is executed from *when* this behavior is executed. Secondly, we introduce atomic filters that can be used to build more complex filters. The semantics of these filters are well defined. Although this approach has clear benefits from an automated reasoning perspective, the introduction of atomic filters results in the definition of numerous filters for specifying more complex behavior. Therefore, we introduce a filter composition language that enables the declarative composition of (atomic) filters, such that composed filter behavior can be reused elsewhere.

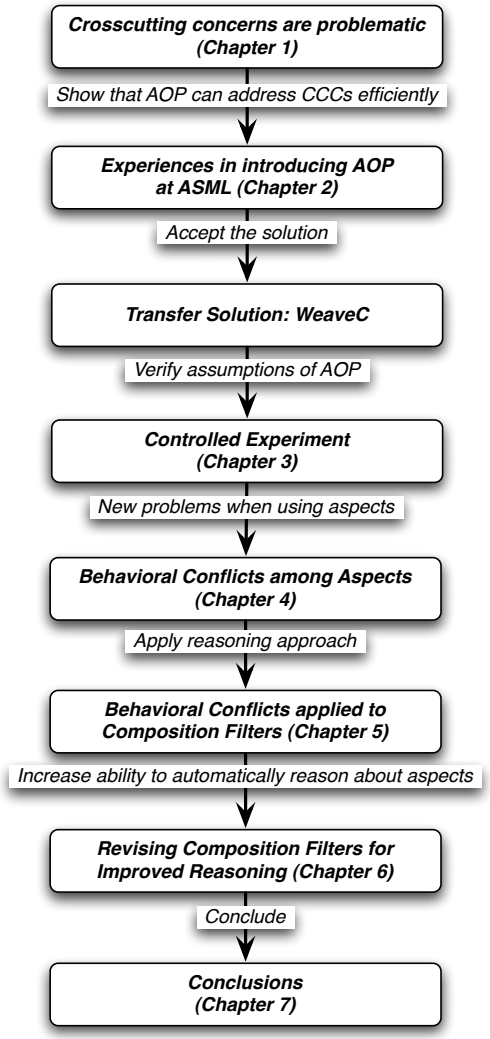
Contents

1 Introduction

The work in this thesis has been conducted as part of the Ideals [IDE] project. One of the assumptions of this project was that the software development effort increases in the presence of crosscutting concerns. ASML [ASM] was the industrial partner, providing case studies and motivating examples for the project. The line of reasoning and the structure of this thesis is presented in figure 1.1.

In this first chapter we discuss the symptoms of crosscutting concerns and their impact. The first step in the project was to show that aspect-oriented programming (AOP) could address these concerns efficiently: this is discussed in chapter 2. Once we demonstrated that aspect-oriented programming could be applied in the context of ASML, we transferred the solution: a tool called WeaveC. ASML matured the tooling and introduced it into the mainstream development process. Once the first users wrote program code without crosscutting concerns in the code, we tested the impact and benefits of AOP. We conducted a controlled experiment to measure the time it takes to execute a set of common change scenarios related to tracing. We also measured the severity of errors a developer makes while executing these scenarios (chapter 3).

If aspects are adopted, new problems can arise. In chapter 4, we discuss one such problem, namely behavioral conflicts among aspects: aspects may interfere with



Figuur 1.1: Overview of the thesis

the behavior of other aspects. In this chapter we present a concrete example of a behavioral conflict that is based on the actual crosscutting concerns used by ASML. We present a novel abstraction based on an (abstract) resource-operation model and discuss how conflicts are detected using a set of conflict detection rules. In chapter 5, we apply our conflict detection approach to Composition Filters and discuss how we utilize some unique properties of Composition Filters to implement (partially) automated reasoning. In the analysis of Composition Filters performed in chapter 5, we encountered some constructs in Composition Filters that hinder automated and manual reasoning. In chapter 6, we discuss these constructs and their limitations in detail and propose extensions to address these limitations. Finally, we conclude in chapter 7.

In the following sections, we discuss each item in figure 1.1 in detail.

1.1 Crosscutting Concerns

The Aspect-Oriented Software Development (AOSD) community has promoted aspects as a solution to complexities introduced by crosscutting concerns. Within the Ideals project, we focused on addressing crosscutting concerns in an industrial setting. In the project ASML waver scanners (machines that create computer chips) are used for providing case studies and motivating examples. These are complex embedded systems with over 16 million lines of (mostly) C code, structured into circa 200 components and 6 layers. The scanners use multiple (parallel) system boards and processors. Throughput and availability are key quality factors. To achieve these qualities, ASML has implemented several mechanisms for performance analysis, traceability and robustness. Many of these mechanisms have been implemented in software and exhibit the symptoms of crosscutting concerns: *Scattering*, *Code Replication* and *Tangling*.

Scattering occurs when one logical concern is distributed over multiple locations. Since, for example, measuring the performance of the machine has to be implemented at many different locations in the code-base, it is scattered over the base program. The resulting code is hard to maintain, i.e. “What if the program is extended, or crosscutting concerns are updated?”. Also, it is hard to keep an overview, i.e. “In which places is a concern implemented?”.

Code replication occurs due to the scattered implementation of a concern. The result is that the same or similar code is implemented in numerous locations in the program. This is often done via a “copy-paste-edit” process, where a developer copies a piece of code, pastes it in the new or modified function and adapts it to fit the context. This process still requires a lot of effort. Also, many

errors occur in such "boilerplate code". For example, in [BvDT06], Bruntink et. al. showed 154 faults in error handling code within 67 KLOC (i.e. 2 faults per 1000 lines of code). Changing a crosscutting concern may require many updates. In addition, often the same functionality is designed and implemented inconsistently in distinct components.

Tangling occurs when a program unit contains a mixture or interleaving of concerns. Most functions do not only implement the main functionality but also have to implement code for performance measurements, for example. This reduces comprehensibility, as one has to know which statements belong to which concern, and this may not always be obvious. Tangling reduces maintainability, because updating one concern may break surrounding code related to other concerns.

To illustrate the symptoms and the impact of crosscutting concerns, listing 1.1 presents a example function from ASML. The purpose of this function is to return a pointer, referring to mod_data.ROI, (line 16) to the caller, in essence this function is a *simple* getter function.

```
1  int get_roi(ROI_struct* ROI_ptr)
2  {
3    const char* func_name = "get_roi";
4    int result = OK;
5    timing_handle timing_hdl = NULL;
6    TIMING_IN;
7    trace_in(mod_data.tr_handle, func_name);
8    if((result == OK) && (ROI_ptr == null))
9    {
10     result = SYS_PARAMETER_ERROR;
11     SYS_Log(r, "f: Output parameter ROI_ptr is NULL.");
12   }
13   if (result == OK)
14   {
15     /* Retrieve current ROI */
16     *ROI_ptr = mod_data.ROI;
17   }
18   LC(result, GET_ROI_FAILED_obj);
19   trace_out(mod_data.tr_handle, func_name, result, ROI_ptr);
20   TIMING_OUT;
21   return result;
22 }
```

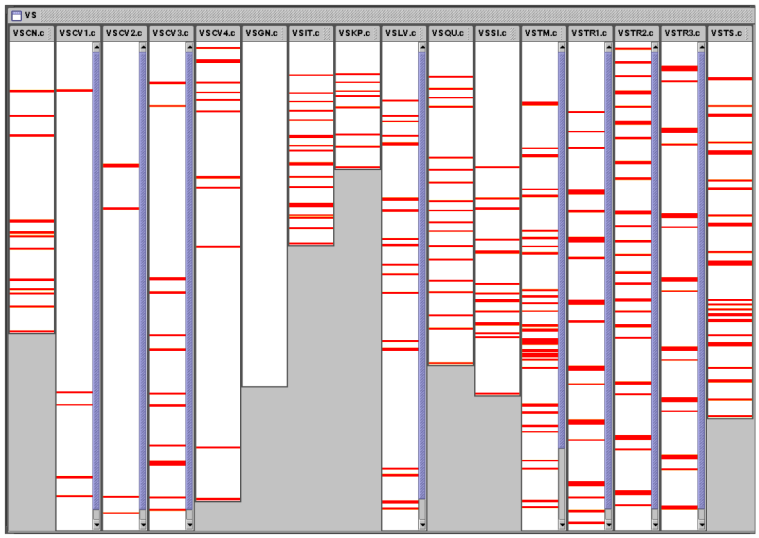
Listing 1.1: Example function in C

We can identify the following crosscutting concerns in this example function: Reflective Information (line 3), Exception Handling (lines 1, 4, 8, 10, 11, 13, 18 and 21), Profiling (lines 5, 6 and 20), Parameter Tracing (lines 7 and 19) and Parameter Checking (line 8). The overhead of crosscutting concerns is usually not as extreme as in this function. This function illustrates that even a simple "getter" function becomes complicated in the presence of multiple crosscutting

concerns.

To illustrate the impact of some of these crosscutting concerns on a large scale, we present several visualizations that represent the impact of crosscutting concerns on a component of ASML. We used AspectBrowser [oC] for creating these visualizations. All images are created from the same component CC from the codebase of ASML. Each vertical bar represents a file in this component and each gray line in such a vertical bar represents a line of code related to a particular concern. The height of each vertical bar relates to the size of the file, some large files have a scroll bar at the right side of the bar.

Figure 1.2 shows the impact of concern Parameter Tracing on component CC. Concern Parameter Tracing accounts for 8% of the statements in component CC.

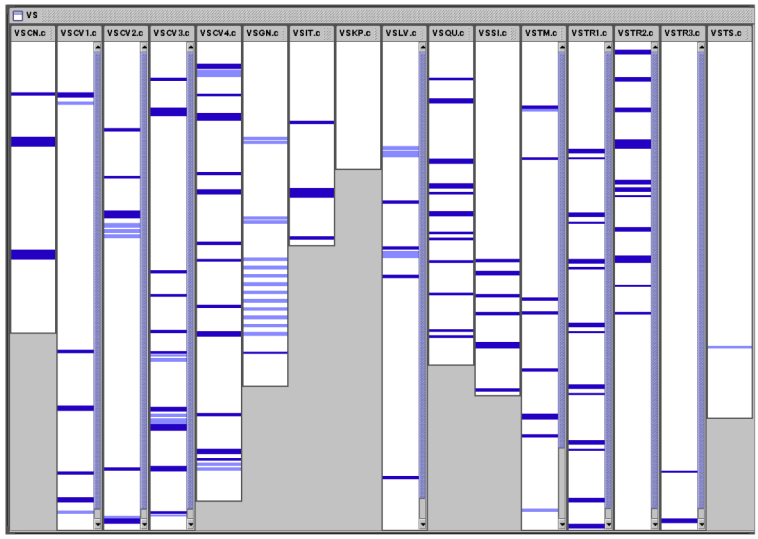


Figuur 1.2: Visualization of concern Parameter Tracing

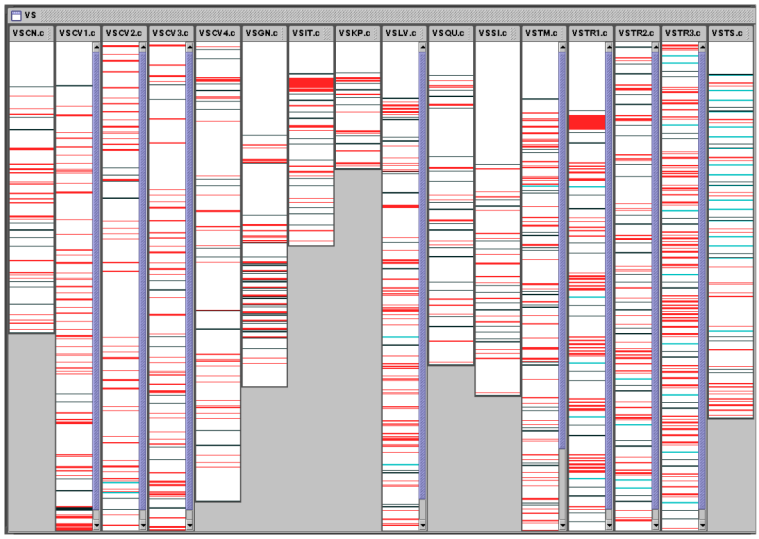
Figure 1.3 shows the impact of concern Parameter Checking on component CC. Concern Parameter Checking accounts for 7% of the statements in component CC.

Figure 1.4 shows the impact of concern Error Handling on component CC. Concern Error Handling accounts for 17% of the statements in component CC.

Figure 1.5 shows the impact of concern Memory Allocation Error Handling on component CC. Concern Memory Allocation Error Handling accounts for 5% of

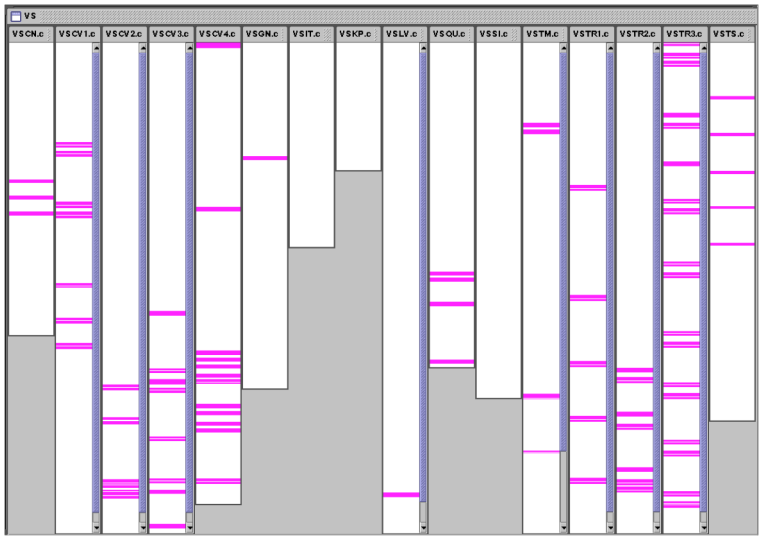


Figuur 1.3: Visualization of concern Parameter Checking



Figuur 1.4: Visualization of concern Error Handling

the statements in component CC.



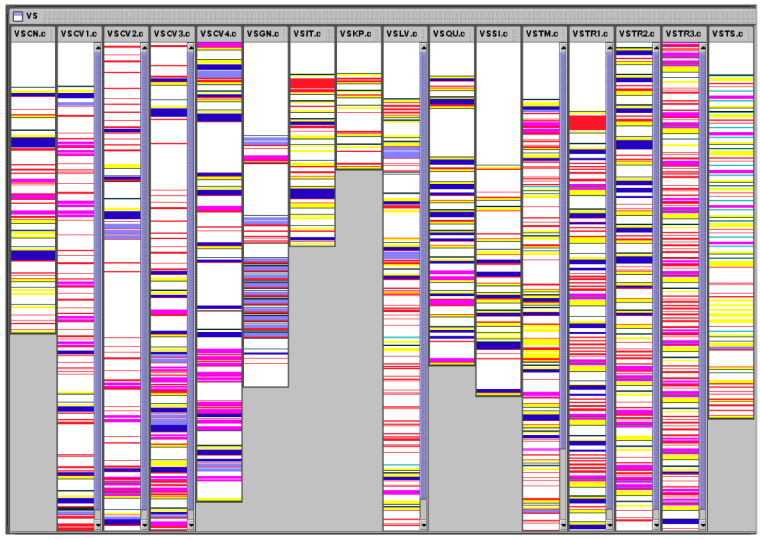
Figuur 1.5: Visualization of concern Memory Allocation Error Handling

We combined all four concerns in one figure, the result is shown in figure 1.6.

These four concerns together requires 29% of the statements in this component. This is less than the combined total of the individual concerns, which is 37%. This is caused by the overlap between concerns.

Separation of Concerns

Separation of concerns is one of the key goals of software and system engineering. The term was first coined by Dijkstra in 1982 [Dij82]. There have been many techniques in different fields which aim at separating concerns. A widely used example of separation of concerns today is the HyperText Markup Language (HTML) and Cascading Style Sheets (CSS), in web pages. Here the *content* that is written in HTML is separated from the *layout* specifications that are written in CSS. These specifications are combined by a web browser. Aspect-oriented software development (AOSD) aims at separating concerns at all levels in the software development life-cycle, from architecture to code implementations. In particular, aspect-oriented programming has been promoted as a means to address crosscutting concerns at the implementation level.



Figuur 1.6: Visualization of all four concerns

It is clear from figure 1.6 and listing 1.1, that the four crosscutting concerns Parameter Tracing, Parameter Checking, Error Handling and Memory Allocation Error Handling are not cleanly separated from the concern that implements the core functionality of this function, namely getting a value. If we would separate the four crosscutting concerns into separate modules, we could refactor the function in listing 1.1 to the function in listing 1.2.

```
1 ROI_struct get_roi()
2 {
3   return(mod_data.ROI);
4 }
```

Listing 1.2: Refactored function in C

In this particular example the savings are extreme: up to 26%, in general this is not so extreme.

Aspect-oriented programming separates *what* behavior should be executed from *when* and *where* this behavior should be executed. A crosscutting concern is implemented in a separate module as an *Aspect*. This aspect can contain two elements, an *advice* representing *what* behavior is to be inserted, and a distribution function or *pointcut* that specifies *where* and *when* to execute the behavior. A *pointcut* can select locations in the program, where each of these

locations is referred to as a *join point*. Aspects have to be combined with the program in some manner, to yield the complete program. Combining aspects with the base program is handled by a *weaver*. Such a *weaver* will resolve all *pointcuts* to a set of *join points* and insert the *advice* at those points. In this thesis we use the term *Concern* to indicate the behavior of a particular functionality and the term *Aspect* to refer to the implementation of such functionality in an aspect-oriented language. We use the term *Base System* and Base Program to indicate the system to which the join points are resolved, these can be aspects themselves.

In this thesis we show that aspect-orientation can indeed address these crosscutting concerns in the context of ASML. We also assess, using a controlled experiment, the merits of an aspect based approach to tracing.

Composition Filters

In this thesis we will make extensive use of the Composition Filters approach. The Composition Filters model has evolved from the first (published) version of the Sina language in the late 1980s [AT88, ABV92], to a version that supports language independent composition of crosscutting concerns [Unia, BA05]. In this chapter we use the language Compose* as an implementation of the Composition Filters model. A key design goal of the Composition Filters model has always been to contribute to object-oriented and aspect-oriented programming languages by improving their ability to compose.

The Composition Filters model can be applied to object-based systems. In such a system, objects can send messages to each other, e.g. in the form of method calls or events. In the Composition Filters model, these messages can be filtered using filters. Each filter has a filter type, which defines the behavior that should be executed if the filter accepts the message and the behavior that should be executed if the filter rejects the message. The matching behavior of a filter is specified by a sequence of filter expressions, which o?er a simple declarative language for state and message matching. Filters defining related functionality are grouped in so-called filter modules. Such filter modules can also encapsulate some internal state or share state with other objects. To indicate which filter modules should be applied (superimposed) to which objects, we use superimposition selectors. A superimposition selector selects a set of classes using a Prolog-based selector language. To the selected set of classes a certain filter module is applied. The result is that all messages sent to and received by all instances of those selected classes, are subjected to the filters within the filter module. In chapter 5 we provide a more detailed explanation and code

examples.

1.2 How to introduce AOP in Industry?

Aspect Oriented Software Development (AOSD) has been promoted as a solution to complexities introduced by crosscutting concerns. However, there is still no (reported) wide-spread industrial adoption of Aspect Oriented Programming (AOP). Several papers [CC04, WHM07, GSE+07, BCMS03] have stated that lack of industrial success stories and lack of experimental validation of the claims are two important barriers to wide-spread adoption of AOP by industry. For example, in the *(Informal Workshop Proceedings of the) first Workshop on Assessment of Aspect-Oriented Technologies* [GSE+07], the following comment was made. "Several workshop participants raised the issue that empirical evidence based on industrial-strength applications is crucial to convince the industry to adopt new AO techniques, and that there is a lack of such experiments in the AOSD field.". Similarly, in *How to Convince Industry of AOP*, by Wiese et. al. [WHM07], the following quote can be found: "We see a kind of a vicious circle here: Industry needs large scale success stories to be convinced. But, to produce such success stories you have to apply AO in industrial projects.". In chapter 2 we report on our experience in introducing AOP at ASML. The approach we used is a step-by-step process that not only elicits and addresses functional requirements, but also elicits quality requirements. In our experience, these qualities are important to address, before AOP can be adopted by a company. The process we used requires the following elements to be defined:

Context : The environment in which an AO solution should operate.
Aspects : The crosscutting concerns that are addressed by an AO solution.
Benefits : The expected gains when using an AO solution.
Worries : The possible problems, preventing the adoption of an AO solution.
Tooling : The previous four ingredients are requirements for the chosen tooling. In some case one can reuse existing tooling, while in others one has to develop new tooling or adjust existing ones.
Acceptance : Once an industrial prototype has been developed and all worries are addressed, the company has to decide whether to adopt an AO solution or not.

We have applied this process in two distinct projects, which are discussed in detail in sections 2.2 and 2.3.

1.3 Does AOP reduce the Software Development Effort?

As mentioned in the previous section, we do not only need success stories of industrial adoption of AOP, but also controlled experiments that assess the benefits of using AOP, for example in terms of development effort reduction and error reduction. These two variables are some of the key benefits of AOP. In chapter 3, we report on a controlled experiment to quantify an aspect-based approach to *Tracing*. The experiment was performed in an industrial setting at ASML. We believe that *Tracing* is a stereotypical aspect-oriented problem. Participants (20 ASML developers) of the experiment were requested to carry out five common and simple maintenance scenarios that affected *Tracing*. All participants were asked to execute these scenarios with and without aspects. We wanted to assess the benefits for developers while developing and maintaining base code, so we did not include any scenarios that required the development or maintenance of aspect code. In chapter 3, we explain the context, the design and the results of the experiment in detail.

1.4 Behavioral Conflicts among Aspects

In chapter 4, we discuss the problem of behavioral interference among aspects. Aspects can be developed independently and behave correct in isolation. However, due to intended or unintended composition of aspects, undesired behavior can emerge. We present one example conflict that we encountered at ASML. We also show examples of other behavioral conflicts. To illustrate the kinds of conflicts we focus on, we show an example here. Assume that there is a base system that uses a Protocol to interact with other systems. Class Protocol has two methods: one for transmitting data, sendData(String) and one for receiving data, receiveData(String). Now imagine, that we would like to secure this protocol. To achieve this, we *encrypt* all outgoing messages and *decrypt* all incoming messages. We implement this as an encryption advice on the execution of method sendData. Likewise, we superimpose a decryption advice on method receiveData. Next, imagine a second aspect that traces all the methods and possible arguments.

These two advices are superimposed on the same join point, in this case Protocol.sendData. We only focus on join point Protocol.sendData, but a similar situation presents itself for join point Protocol.receiveData. As the advices have to be sequentially executed, there are two possible execution orders here. Now assume that we want to ensure that no one accesses the data before it is encryp-

ted. This constraint is violated, if the two advices are ordered in such a way that advice tracing is executed before advice encryption. We may end up with a log file that contains "sensitive" information. The resulting situation is what we call a behavioral conflict. Both orderings are correct and can be executed without problems, but they yield different behavior, where one order is characterized as a conflict.

In chapter 4, we present our approach for detecting behavioral conflicts that is based on an abstraction of the behavior of advices in terms of operations on resources. We verify that conflict rules, which reason about the possible patterns of operations on resources, are not violated. We apply the resource-based approach to Composition Filters in chapter 5. We show that the Composition Filters model has several unique characteristics that enable (partially) automated reasoning about filters.

1.5 Limitations of Automated Reasoning

Chapter 6 presents some constructs of Composition Filters that hinder manual and automated reasoning. We explain why these constructs reduce the ability to reason about filters, and we present extensions to Composition Filters to overcome these restrictions. The chapter addresses three problems:

- In most AOP languages, one can apply advice either before, after or around the execution of a join point. In Composition Filters this execution moment is encapsulated inside the filter definitions. This hinders reasoning and reuse, since the filters do not only encapsulate *what* behavior should be executed, but also *when* this behavior is executed.
- The current set of predefined filter types are not atomic with respect to the actions carried out. Most filters manipulate certain properties of the message and affect the control flow.
- Composition Filters offers a declarative way for composing filters within a filter set and filter modules. However, there is no way to compose filters from either filter actions or from a sequence of other (atomic) filters.

Table 1.1 shows the problems, solutions and in which chapters these are discussed, using a language, problem and solution pattern.

Tabel 1.1: Thesis contents overview

Language	**Problem**	**Solution**	**Chap.**
Object-Oriented and Imperative	Crosscutting Concerns	Aspect-Oriented Programming	1
Aspect-Oriented	How to introduce AOP in Industry?	Experiences in introducing AOP at ASML	2
Aspect-Oriented	Does AOP reduce the Software Development Effort?	An Assessment of an Aspect-based Approach to Tracing	3
Aspect-Oriented	Behavioral Conflicts among Aspects	Behavioral Conflict Detection Tools	4 & 5
Composition Filters	Limitations of Automated Reasoning	Improved Composition Filters Design	6

Experiences in introducing Aspect-Oriented Programming at ASML

2

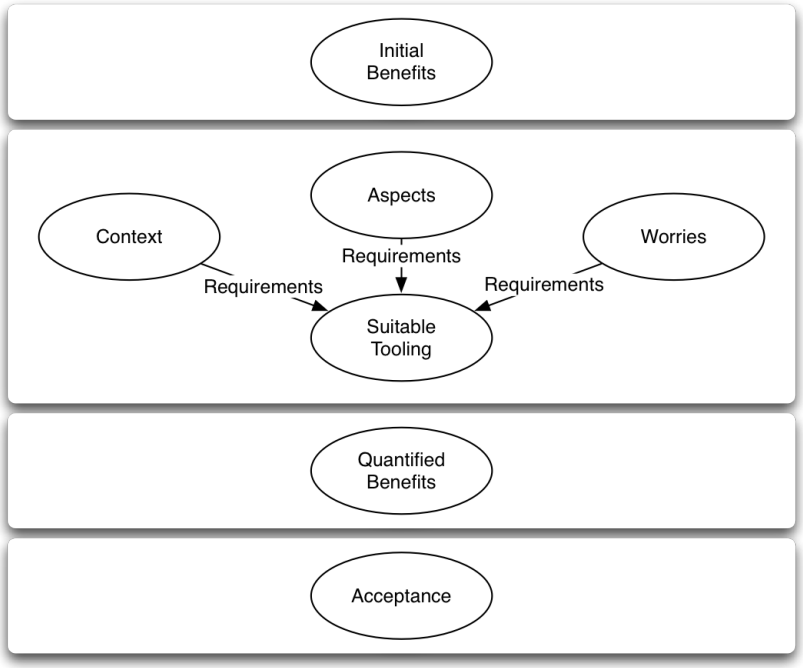
In this chapter we report on our experience on the introduction of aspect-oriented programming (AOP) at ASML. This chapter should be considered as a necessary step for adopting AOP at ASML, and creates the context for the next chapters.

We believe the presented experience can be useful for transferring AOP solutions to practice. The approach we used is a step-by-step process which not only elicits and addresses functional requirements, but also elicits quality requirements. In our experience, these qualities are important to address, before AOP can be adopted in industry.

This chapter is structured as follows. First we outline our approach, which consists of several steps. Next we show our experience with the approach in two projects. We conducted both projects at ASML. The first project addresses three crosscutting concerns within the C code base. The second project addresses one crosscutting concern within the .NET code base. Finally, we relate our work to others and conclude.

2.1 Approach

First we have to know exactly what the problem is, we want to address. This is imperative as all steps outlined are dependent on finding, quantifying and addressing the problem. Also, convincing a company of a certain technology requires showing them that they indeed have the problem you are trying to address. Once the problem definition is clear we can commence our process.



Figuur 2.1: Overview of technology acceptance process

We can distinguish four phases in our process (see figure 2.1). Initially, we have some expected benefits. In most cases these benefits are expressed in papers that cover the topic. Next, there are three main drivers for determining a suitable tool or process. These three drivers are Context, Aspects and Worries. Once a suitable tool or process is selected, we can quantify the benefits in the real case. These might not be the same as the initial benefits, as the tool itself can introduce some overhead. Finally, the tool or process has to be accepted. The

next sections discuss the benefits, drivers, tooling and acceptance in more detail.

In this chapter, we focus on the problem of crosscutting concerns. Chapter 1 detailed the symptoms and problems of crosscutting concerns. It is important to realize that most companies already experience the symptoms and worries related to crosscutting concerns. They may not label them as such, but use terms such as *idiom*, *coding conventions* or *coding guidelines*.

2.1.1 Initial Benefits

The benefits are usually the key drivers to introduce a new technology. Sometimes the benefits may not be immediately visible. The benefits of both case studies included a better software structure and less code (2.2.6 and 2.3.6). Especially, the latter benefit translates into error, effort and lead-time reduction. Code size reduction can serve as a primary driver to adopt AOP. There are studies, e.g. [S. 04], that show that even in rigorously tested and verified software, there are still 3 bugs in every 1000 lines of code. Thus code size reduction increases the quality of the code and decreases the number of bugs in the code. At this point in our approach we can not yet verify these benefits since we have yet to implement the solution.

2.1.2 Context

The first step in our process is to determine in which context the solution should operate. This is important, as some companies use slightly different or modified programming languages and libraries. The context is also the environment in which the solution has to operate. The context helps to determine whether certain concerns are inherent to the usage of a certain programming language, domain, application or inherent to a previous design choice.

Some crosscutting concerns may be inherent to a specific programming language or paradigm, an example is error propagation in C code. Other languages like Java and .NET provide a native exception handling mechanism, with automated exception propagation usually along the call stack.

2.1.3 Aspects

Once we know the details of the context, we can start to search for crosscutting concerns. As mentioned previously, the chosen programming language or libraries

may already provide some indicators to crosscutting concerns. Also, investigating software and architecture documentation and manuals can provide information about possible crosscutting concerns.

In parallel one can use tooling that detects recurring patterns in the code. In[BvDvET05], Bruntink et. al. propose a technique based on clone detection. But even simpler tooling like grep and AspectBrowser [oC] can provide hints as to which crosscutting concerns may be present.

Once a set of crosscutting concerns has been identified, we have to describe these concerns clearly. All elements, dependencies and interactions must be documented. These elements form part of the requirements for the language and tooling.

Examples of these elements are: if the concern uses a library, there could be dependencies to import, using or include statements. To be able to insert these kind of statements, one must have a language that supports some sort of introductions. Another example is the execution time of the behavior, e.g. *before* or *after* the execution of a function.

All details should be documented carefully, as this directly drives the requirements of the tooling and expressiveness of the aspect-oriented language. Also, from a migration point of view, it is imperative to know exactly which elements are required to replace the original functionality with a similar behavior. Exact behavior migration may not always be feasible or desirable, for example there could be many deviations from the idiom.

2.1.4 Worries

In this phase we elicit the worries that stakeholders express about the introduction of the tooling or process. There are usually many roles and tasks in an organization, all with different focus. Examples of these roles and tasks are: test, integration, core and aspect developers, customer support, management, hardware design, etc. The stakeholders responsible for these tasks and roles can be impacted by the proposed tooling or process. Before these stakeholders accept the solution, we first need to elicit their worries and address them accordingly.

Once all worries are collected, we need to prioritize them. Usually, there is limited time and we have to trade-off certain quality requirements against each other. Some of these quality requirements might even contradict each other. In the case studies, we selected around seven worries. After discussions with the stakeholders, we selected a set of worries that were deemed the most important.

After this selection and prioritization, we must address these worries. For some worries it could be a matter of documenting or providing training. In other projects, a change in the design of the tooling or process might be required. Also, the costs of addressing the worries should be in balance with the expected benefits. For example, in the first case AOP was incrementally introduced within the company. The costs of the initial implementation was quite high compared to the benefits of the approximately first ten users. However, since there was clear potential to scale up to 600 developers, as such the initial costs were accepted.

2.1.5 Tooling

Once the precise requirements for the crosscutting concerns, context and worries are documented, we can start to build new or use and possibly adapt existing tooling which addresses these concerns. There are off-the-shelf aspect-oriented solutions for most popular programming languages. These differ mostly in granularity of join points, pointcut language and advice language. However, there are cases for which no appropriate tooling exists and needs to be developed. Other cases might simply require adaptation of the current tooling.

The tooling should not only be driven by functional requirements but also by quality requirements, discussed in the previous step. It is advisable to execute this phase in parallel with the previous phase, since some of the quality factors may limit or alter your choice of tooling, or the design of new tooling.

The result of this phase should be a tool chain or prototype that can address the selected crosscutting concerns on a representative component or set of source files. This prototype should also address the key worries. The prototype is an academic prototype and as such does not have to be industry strength yet.

2.1.6 Quantified Benefits

Once we have implemented the solution to problem we want to address, we can quantify the benefits. The benefits will vary from project to project. Once the context and impact of the selected crosscutting concerns are clear, we can start to determine the benefits of modularizing these concerns. Lines of code is usually a key indicator of the expected effort reduction. Also, talking to developers about the problems they encounter can be insightful. In [BvDT06], Bruntink et. al. present the results of a study that tries to detect faults in manual exception handling. They showed that, for one component, there were 2 errors per 1000 lines of code, w.r.t. error handling. This illustrates that even in *boiler plate*

code, people make mistakes. Code size reduction thus reduces the number of errors and can serve as a clear motivation to adopt AOP.

2.1.7 Acceptance

After all steps have been executed and there is a working prototype, we can convince a company to use such a tool. If all worries are addressed and the benefits are accepted by all stakeholders, there is no technical reason not to accept the solution. The process presented in this thesis is no guarantee for acceptance. However, it does try to cover the best conditions for acceptance.

A prototype implementation can serve as a guideline for a more robust version, once the project is accepted. In some projects the developed prototype will have a limited scope and possibly a limited feature set. For example, in the first project we were unable to address all worries within the limited time frame, but we did show that there is no fundamental problem. This was sufficient to convince the company.

In the next two sections we present two examples of projects that we were involved in at ASML. In both projects we introduced AOP to address crosscutting concerns.

2.2 Aspects in C

The first project started in September of 2005, and the deadline for the project was end of November of the same year. This short time frame forced us to limit the scope of the tooling and the number of crosscutting concerns we could address.

2.2.1 Initial Benefits

The expected benefits of an aspect-oriented approach are: statement reduction, effort and lead-time reduction, a general solution to address crosscutting concerns and a in general a better software structure. We cannot quantify these claims at this point. Once we have an implementation we can determine the exact statement reduction, for example.

2.2.2 Context

For this project, we used a component called CC1[1], composed of 30458 lines of source code, containing 6389 executable statements. The former definition of lines is code is defined as *Physical Lines of Code* (PLOC). Whereas the latter definition is defined as *Logical Lines of Code* (LLOC) This component was selected by ASML, because it was considered representative in terms of crosscutting concerns. In addition, component CC1 was restructured for better modularity, which would increase our chances to identify the *real* crosscutting concerns, i.e. not the ones due to the deterioration of the design of the component.

Listing 2.1 shows function get_roi from component CC1, this is the same function as shown in listing 1.1. Most of the functions contain more base functionality than this one. As a result, the crosscutting concern overhead is not always as extreme as in this example.

```
1  int get_roi(ROI_struct* ROI_ptr)
2  {
3    const char* func_name = "get_roi";
4    int result = OK;
5    timing_handle timing_hdl = NULL;
6    TIMING_IN;
7    trace_in(mod_data.tr_handle, func_name);
8    if((result == OK) && (ROI_ptr == null))
9    {
10     result = SYS_PARAMETER_ERROR;
11     SYS_Log(r, "f: Output parameter ROI_ptr is NULL.");
12   }
13   if (result == OK)
14   {
15     /* Retrieve current ROI */
16     *ROI_ptr = mod_data.ROI;
17   }
18   LC(result, GET_ROI_FAILED_obj);
19   trace_out(mod_data.tr_handle, func_name, result, ROI_ptr);
20   TIMING_OUT;
21   return result;
22 }
```

Listing 2.1: Example function in C

The details of listing 2.1 are explained in the next section. For now it suffices to state that line 16 implements the base functionality of this function, the other lines address different crosscutting concerns.

ASMLs C code base uses the *Gnu Compiler Collection* (GCC) as the main programming environment. The C version of GCC is a modified version of the ANSI-C programming language. The build environment uses ClearCase as a

[1]For confidentiality reasons, all identifiers in the text and in the code listings have been altered. This does not affect the work presented here.

source versioning system and a collection of make scripts. The proposed solution had to integrate with these version control and build systems.

2.2.3 Aspects

To identify crosscutting concerns, we manually investigated source code and used automatic tooling like grep and AspectBrowser [oC]. Some of these concerns were already mentioned in the *Software Architecture Manual* of ASML. We identified six crosscutting concerns in component CC1. Almost all of these crosscutting concerns could be identified in listing 2.1. These crosscutting concerns were:

Reflective Information (line 3): This concern provides meta information about the function, file or component, in this case the name of the function. There are other kinds of information, like build target and component name.

Error Handling (lines 1, 4, 8, 10, 11, 13, 18 and 21): In the C programming language, one has to manually implement exception handling, since C does not feature a native exception handling mechanism. In most cases this is done through the use of an integer return value. A value not equal to zero indicates a failure, otherwise success.

Profiling (lines 5, 6 and 20): Profiling is used to measure the execution time of functions. This information is used to determine performance bottlenecks.

Parameter Tracing (lines 7 and 19): Tracing ensures that the value of all parameters is written to a log file, this includes function arguments and the return value.

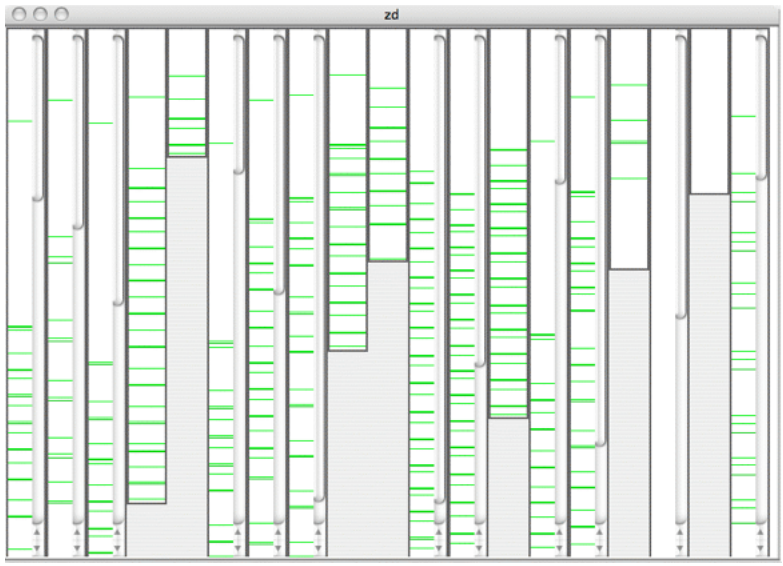
Parameter Checking (lines 8 to 12): This is an implementation of the Design by Contract methodology [JM97]. This concerns checks for null pointers and prevents potential memory leaks.

Memory Handling (not shown in the listing): C and its derivatives do not feature automatic memory management. As such , one has to explicitly allocate and free memory at the appropriate locations in the code.

Out of the identified crosscutting concerns, we only addressed concerns Parameter Tracing, Profiling, and Reflective Information. These three crosscutting concerns already made up 28% LLOC, and realizing them using aspects was feasible within our limited time frame. We will now detail each of the three selected concerns. These concerns are not component-specific, but occur in all components.

2.2.3.1 Concern "Profiling"

Figure 2.2 illustrates the impact of concern Profiling on component CC1. The figure has been created using AspectBrowser[oC].



Figuur 2.2: Visualization of Profiling code in component CC1

Figure 2.2 visualizes component CC1 and the impact of concern Profiling. Each vertical bar in the figure represents a file in component CC1. The height of each vertical bar relates to the size of the file, some large files have a scroll bar at the right side of the bar. Each shaded horizontal gray line indicates a line in a source file that deals with, in this case, concern Profiling.

Purpose: The goal of the concern is to measure the execution time of each function in the component. This information is used to determine potential performance bottlenecks and to provide indications which functions to consider for optimization.

Design: There is a library that implements functionality to start and stop profiling. This library also maintains a database that stores the execution time of functions. Each function in the component must call the start (line 6) and stop (line 20) functions at the appropriate locations, i.e. after declaration of the local variables and just before returning to the caller. The start and stop functions

are defined in two macros called TIMING_IN (line 6) and TIMING_OUT (line 20). These start and stop functions must be called with a variable that identifies the function, this is a so-called timing handler (line 5).

Example: Lines 5, 6 and 20 of listing 2.1 show an example snippet of concern Profiling.

Implications: The calls to the profiling library and the corresponding initialization and registration of the handler are crosscutting the component.

Impact: This concern accounts for 3% of PLOC and 14% of LLOC of that component.

Implementation details: The following 5 actions are required to successfully implement concern Profiling. These are thus functional requirements.

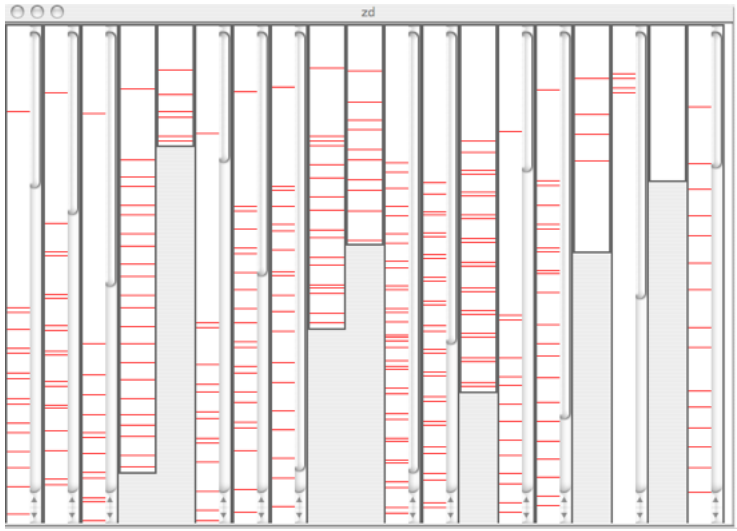
- Introduce a global profiling variable or handler in the structure which contains all the general information about this file or module. A module is considered the same as a file.
- In the start-up function of the module, register this module to the profiling library, using the global profiling handler.
- Introduce a local profiling handler in each function of the module.
- Call a function to start the profiler (TIMING_IN) at the start of the function with the local profiling handler.
- Call a function to stop the profiler (TIMING_OUT) at the end of the function with the local profiling handler.

2.2.3.2 Concern "Parameter Tracing"

Figure 2.3 visualizes component CC1 and the impact of concern Parameter Tracing.

Purpose: The goal of the concern is to trace the values of the parameters of all functions in component CC1. The parameters can either be input, output or both. Input parameters are only read in a function, whereas output parameters are only written in a function. Input and output parameters are both read and written in a function. At the end of the function the return value must be traced. The return value is a special case of output parameter.

Design: Input parameters must be traced at the start of a function, using function trace_in (line 7). Output parameters must be traced at the end of a function, using function trace_out (line 19). Input and output parameters should be traced at both locations. The return value must also be traced at the end of a function. There is a library that provides functions with variable



Figuur 2.3: Visualization of Tracing code in component CC1

arguments to implement trace_in and trace_out.

Example: Lines 7 and 19 of listing 2.1 show an example of concern Parameter Tracing.

Implications: The calls to the tracing library and the corresponding initialization and registration of the handler are crosscutting the component.

Impact: This concern accounts for 2% of PLOC and 8% of LLOC. The actual impact can be larger, since this component does not trace all functions.

Implementation details: The following 3 actions are needed to successfully implement concern Tracing. These are thus requirements for our solution.

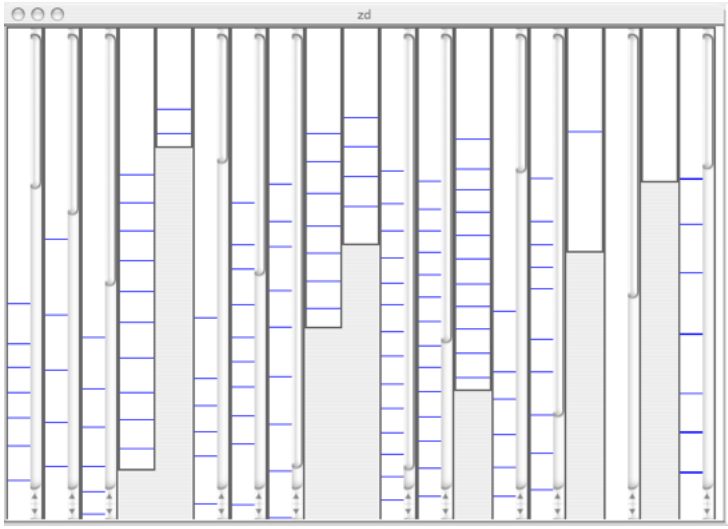
- Introduce a global tracing handler in the main module struct. This is similar to concern Profiling.
- For functions *without* arguments;
  - At the start of a function, call function trace_in of the tracing library using the global tracing handler and the name of the function.
  - At the end of a function, call function trace_out of the tracing library using the global tracing handler, the name of the function and the return value of the function.
- For functions *with* arguments;

– At the start of a function, call function trace_in of the tracing library using the global tracing handler, the name of the function and any arguments that are read.
– At the end of a function, call function trace_out of the tracing library using the global tracing handler, the name of the function, any arguments that have been written and the return value of the function.

To automatically determine which parameters are input, output or both, we have to do detailed control and data flow analysis of each function, since we have to determine which parameters are read and which are written. We did not do this in this project, due to limited time. However, the theory behind this is well known, and there are tools which provide this information, such as CodeSurfer[Gra]. We have chosen to use annotations attached to functions and arguments to provide the necessary information.

2.2.3.3 Concern "Reflective Information"

Figure 2.4 visualizes component CC1 and the impact of concern Reflective Information.



Figuur 2.4: Visualization of Reflective Information code in component CC1

Purpose: The goal of the concern is to provide meta information to all functions in component CC1. For this project we were only concerned with the name of the function.

Design: A local constant contains the name of the function. This could also be generated at compile time, using, for example, the construct __FUNCTION__ of GCC.

Example: Line 3 of listing 2.1 shows an example of concern Reflective Information.

Implications: This concern is relatively trivial. However, there are several reasons why we also addressed this concern. Firstly, keeping this kind of meta information consistent with the actual implementation is hard. For example if the file is relocated to a new component, we have to update this information. Secondly, if we want to introduce additional meta information, we can do this in one single location.

Impact: This concern accounts for 1% of PLOC and 5% of LLOC.

Implementation details: The following is required to successfully implement concern Reflective Information:

- Introduce a local constant that holds the name of the function.

2.2.3.4 Composition of Concerns

The three concerns are all applied to all functions within the same component. As such we need to determine if the order of application matters, and if so in which order should these concerns be applied. Concern Reflective Information is order independent of the other two concerns, and as such can be woven in before or after the other two concerns. The execution order of concerns Profiling and Tracing does affect the behavior. If concern Profiling is executed before concern Tracing we will also measure the time it takes to trace this function. If we apply the two concerns the other way around, we will measure the execution time of a function without tracing. ASML expressed the desire to include tracing in the execution time of a function. This resulted in the requirement to be able to specify the order of concerns.

2.2.4 Worries

This section describes some of the worries that we elicited from several stakeholders. Although we elicited these in an ASML context, we believe that they are applicable for many organizations.

2.2.4.1 Migration

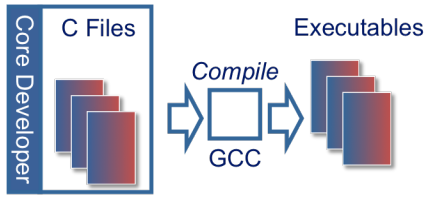
ASML has a lot of legacy code, actually up to 15 million lines of code. Obviously, it would be nice to experience the benefits of AOP not only for new code, but also for legacy code. This requires a transformation from the old manual implementation to the new one. Concerns Profiling and Reflective Information were implemented on separate lines in the code and were easy to identify with "simple" tooling. Migrating these concerns involves removing the lines. However, migrating concern Tracing was more difficult, because determining whether to trace a variable at the start or end of a function relies on the parameters input and output specification. We implemented this by annotating which parameters are input, output or both. Automatically generating these annotations requires control flow analysis and read-write analysis of the parameters. Also, in [BvDDT07], Bruntink et. al. showed that there is a lot of variation in the implementation of tracing, e.g. some components have 10 different ways of tracing. This variation hinders automatic migration. Manually migrating the concerns is possible, albeit laborious. We investigated automatic migration, but it turned out to be impractical. This worry was addressed by only applying our solution to newly developed code. This was not considered a major issue as for this project a lot of components were rewritten.

2.2.4.2 Availability of the tooling and maturity of the process

Our solution was implemented in a tool called WeaveC, see section 2.2.5. WeaveC has been developed as a prototype. Clearly, WeaveC had to mature to reach industry strength. This required (substantial) investments of money and people. The use of AOP requires a seamless integration into the development and build process. In an industrial development process there are usually many specific developer roles, with different responsibilities. These roles should remain clearly separated even in the presence of aspects. If one uses AOP, there will be additional roles in the development life-cycle. Most notably, an aspect developer, a weave tool maintainer, and an integrator. The latter is responsible for the correct behavior of the entire system. If problems arise due to the composition

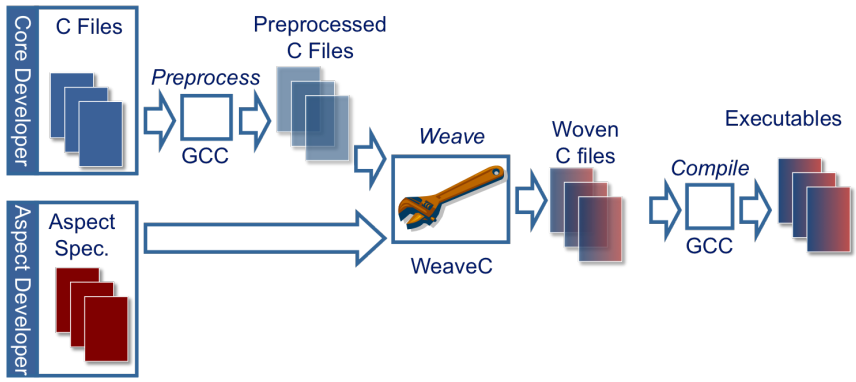
of aspects, we have to have tooling that verifies this composition. This problem is addressed in chapter 4.

Figure 2.5 depicts the current build process.



Figuur 2.5: Build process without AOP

Figure 2.5 shows a set of C files developed and maintained by developers. These are compiled using GCC and result in a set of executables. Figure 2.6 depicts the build process with AOP.



Figuur 2.6: Build process with AOP

Figure 2.6 shows that we now have two sets of source files: a set of C files that are developed and maintained by core developers, and a set of aspect files that are maintained by aspect developers. The first step is to pre-process the C files, this is a requirement of WeaveC. The preprocessed C files and the aspect specifications are inputs for WeaveC. The result is a set of woven C files that have the combined functionality of both the core and aspect code. These are compiled using GCC, and the result is a set of executables. The semi-transparent shaded files in figure 2.6 are intermediate results.

2.2.4.3 Ability to control the application of aspects

It should be possible to control the application of aspects for two reasons. Firstly, for error purposes; if the aspect code contains a bug, the system should still be able to compile without a specific aspect. This is only possible if there are no base-aspect or aspect-aspect dependencies. For example, profiling information is typically not used during the execution of the program, but error handling can impact the execution of the program.

Secondly, to control aspect application. For example, some functions can not be traced by design, since these functions are active before the tracing framework is active. Other methods should not be traced due to performance issues. Some parameter types need to be traced differently. In order to facilitate this can we use annotations to control the weaver. In the prototype we only implemented an annotation that turned tracing off for a specific function.

2.2.4.4 Understandability

ASML uses the C programming language, hence adopting AOP is a change of paradigm. This was considered a possible worry. This could be addressed by a clear separation between aspect developers and core developers. Aspect developers will need be trained in writing and maintaining aspects. Core developers will only have to be trained how to control the weaving process using annotations. Core developers can still see the result of weaving, since there is an intermediate source file.

2.2.4.5 Compile-time Performance

In large systems with many developers and many dependencies between components, the build time can be quite substantial. One of the key problems with aspects is that the impact of the aspect can be quite high in case of a large pointcut.

The prototype we developed introduced a 100% performance penalty on the build time. This did not reach the targeted 30% overhead. However, as there were no optimization efforts made on the prototype, we expect that the build overhead would have roughly matched the requirement.

2.2.4.6 Run-time performance

As the throughput at run-time must be guaranteed, a run-time performance penalty was not acceptable. A static weaving approach was implemented instead of more dynamic types of weaving. WeaveC resolves all pointcuts and advices statically. As such, no run-time penalty is introduced by WeaveC itself. Run-time penalties are purely caused by the aspects themselves.

2.2.4.7 Ability to Debug

Debugging a program with aspects is considered to be an issue. Especially, if an error occurs, who is responsible for fixing this error? First of all, this requires determining whether the error is in the aspect code, in the core code or in the composition of aspect with core code. Core developers indicated that they did not want to see the woven code. As such, they may not be aware of which aspects are applied to their code.

We explored the following three options for debugging:

1. Debug with the inserted code visible. This may *surprise* core developers, since these may not be aware of aspects and obfuscates the base code.
2. Debug with the inserted code hidden, similar to the way macros are handled.
3. Debug normally and jump to the aspect definition while executing the inserted code.

A preference for the second option was expressed by the stakeholders, such as architects.

One implication of using static weaving was that we broke the standard debugging facility of GCC, called *GNU Debugger* (GDB). Since, our solution introduced aspect code, the line directives of the original program are no longer valid. We did not address this issue in our prototype, but we did show that it was feasible to implement each of the three options, using adjusted line directives.

2.2.5 Tooling

In this project we developed an improved version of WeaveC[Unib]. This is a source-to-source weaver for the C programming language, and it was developed at an earlier time in the Ideals project. It uses an XML input format and supports the following join points and introductions:

- Function call
- Function execution
- Local variable introduction in a function
- Global variable introduction in a file
- Field introduction in a global structure

The first two elements are join points, where one can apply before and after advice. These join points and introductions were sufficient to address Tracing, Profiling and Reflective Information. The prototype implementation uses a grammar which can parse only preprocessed C code. There is also limited support for annotations and plug-ins to add more complex advice. Plug-ins are written as Java classes and have access to the full program model and weaving library.

To illustrate how one can write aspects using WeaveC, we show the implementation of concern Profiling in listing 2.2.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <aspect id="ProfilingConcern">
3    <pointcut id="generic profiling">
4      <elements files="*.c" data=".*" identifier="function"/>
5      <advices>
6        <adviceapplication id="profiling_handler_introduction" type="before"/>
7        <adviceapplication id="start_profiler" type="before"/>
8        <adviceapplication id="end_profiler" type="after"/>
9      </advices>
10   </pointcut>
11   <pointcut id="register profiler module">
12     <elements files="*.c" data="%MODULE_NAME_UPPER%_startup" identifier="function"/>
13     <advices>
14       <adviceapplication id="profiler_module_registration" type="before"/>
15     </advices>
16   </pointcut>
17   <pointcut id="add profiler handle to module struct">
18     <elements files="*.c" data="%MODULE_NAME_UPPER%_module_data_struct" identifier="struct"/
         >
19     <advices>
20       <adviceapplication id="profiler_handler_addition" type="before"/>
21     </advices>
22   </pointcut>
23   <advice id="profiling_handler_introduction" type="function_variable_introduction">
24     <code>
25       <![CDATA[ Profiler_handle timing_hdl = NULL; ]]>
26     </code>
27   </advice>
28   <advice id="start_profiler" type="execution" priority="10">
29     <code>
30       <![CDATA[ CC_start_profiler(func_timing_hdl, %MODULE_NAME%_mod_data.ti_handle,
             func_name, &timing_hdl); ]]>
31     </code>
32   </advice>
33   <advice id="end_profiler" type="execution" priority="10">
34     <code>
35       <![CDATA[ CC_stop_profiler(func_timing_hdl, %MODULE_NAME%_mod_data.ti_handle,
             func_name, &timing_hdl); ]]>
36     </code>
37   </advice>
```

```
38   <advice id="profiler_module_registration" type="execution" priority="10">
39     <code>
40       <![CDATA[
41         if (result == OK)
42           {
43             result = CC_register_module("%MODULE_NAME_UPPER%", & %MODULE_NAME%_mod_data.
                   ti_handle);
44         }
45       ]]>
46     </code>
47   </advice>
48   <advice id="profiler_handler_addition" type="structure_introduction">
49     <code>
50       <![CDATA[ CC_module_handle ti_handle; ]]>
51     </code>
52   </advice>
53 </aspect>
```

Listing 2.2: Concern Profiling in WeaveC

Listing 2.2 defines aspect Profiling. It defines three pointcuts:

generic profiling (lines 3 to 10): this selects (line 4) all functions in all C files. Selection is based on regular expressions. To the join points selected by this pointcut, we apply three advices:

- **profiling_handler_introduction:** this advice is specified at lines 23 to 27, and inserts a local profiling handler in the functions.
- **start_profiler** : this advice is specified at lines 28 to 32, and is applied before the execution of the functions. The result is that a call to function CC_start_profiler (line 30) is inserted just after the variable declarations of each function.
- **end_profiler** : this advice is specified at lines 33 to 37, and is applied before the execution of the functions. The result is that a call to function CC_stop_profiler (line 35) is inserted just before returning to the caller in the functions.

register profiler module (lines 11 to 16): this selects (line 12) all functions with the name %MODULE_NAME_UPPER%_startup. %MODULE_NAME_UPPER% is a internal variable that is substituted for the file name in uppercase characters. To the resulting join points, we apply advice profiler_module_registration, defined at lines 38 to 47. The result is that before the execution of each startup function we call function CC_register_module (line 43).

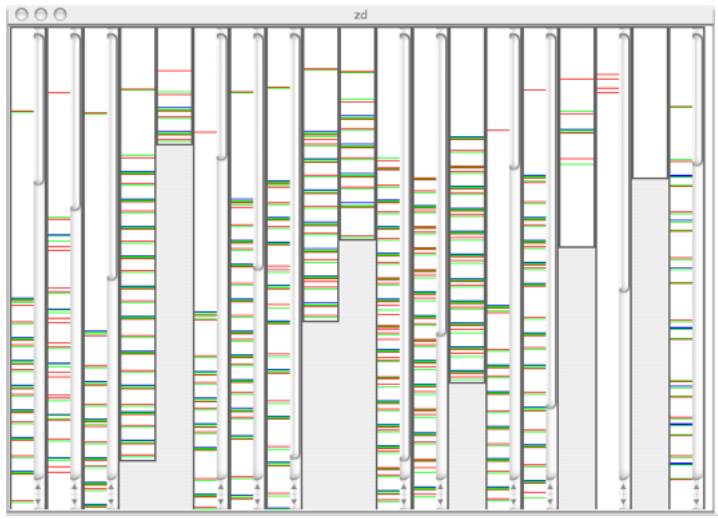
add profiler handle to module struct (lines 17 to 22): this selects (line 18) all structures named %MODULE_NAME_UPPER%_module_data_struct. To this structure we add a field ti_handle of type CC_module_handle (line 50).

We have implemented rudimentary support of advice ordering at shared join points. Each advice has an optional priority (e.g. line 28), advices with higher priorities are woven before lower priority advices. This mechanism was sufficient for the case study. In general a more detailed and fine-grained ordering mechanism might be necessary.

2.2.6 Quantified Benefits

This section explains the results we achieved with our project. Most of these are considered to be the key motivations for using AOP. We discuss these in the context of the project.

First, we show the combined impact of the three concerns on the component (figure 2.7).



Figuur 2.7: Visualization of three aspects in component CC1

Even though these three concerns are simple, they do have a large impact on the base code. In this case 6% PLOC and 27% LLOC. One quarter of the statements that implement this component corresponds to one of these three concerns.

2.2.6.1 Statement reduction

As stated previously, our project was carried out on a component of about 6K LLOC. In our project, we achieved a reduction of 26% of LLOC, compared to the original code. The tracing annotations and the aspect definition added 1% LLOC to the code. We could achieve such a dramatic reduction (96%) of the code related to aspects, since we addressed three highly homogenous aspects. In our project, we only implemented three aspects. If we would also address parameter checking and error handling, the reduction would be up to 50%. However, the migration path, especially for the error handling concern, is more difficult as the error handling code is highly tangled with the base code.

2.2.6.2 Effort and Lead time reduction

We used an intermediate *COnstructive COst MOdel* (CoCoMo) II model, to estimate the reduction of effort and lead time. This model was created and instantiated with the details of ASML by the Ideals liaison officer. The results show about 7% effort reduction and about 3% lead time reduction. This may not seem that much, but our solution can be scaled up to around 600 developers at ASML. Hence a substantial amount of person years reduction can be achieved. Chapter 3 quantifies the benefits in terms of effort and error reduction, using a controlled experiment. There is a discrepancy between the effort reduction (7%) and the statement reduction (26%). We cannot quantify this discrepancy, but this may be caused by developers that may spend more time developing the core functionality, writing other crosscutting concern code or other tasks.

2.2.6.3 Generality of solution

The solution we proposed was not specific to the current component. We also applied concern Profiling to a component without profiling, within 10 minutes. We also demonstrated that one could easily change an aspect definition to adapt or implement new functionality. This clearly showed that the tooling was indeed generic and not specific to either a component or set of aspects.

2.2.6.4 Local deviations

The solution we proposed still allows for local deviations. For example, there are functions which are called often and for these functions tracing is removed. There are also low level components which do not perform tracing, as these are

active before the logging and tracing component is initiated. Although we do not encourage these deviations, they can be implemented with annotations and an appropriate pointcut description language.

2.2.6.5 Better software structure

A better software structure has been stated as one of the primary benefits of AOP, and separation of concerns in general ([LB05, GSF+05, GBF+07]). The pointcut and advice mechanisms provide an easy way to deal with changing coding guidelines. This flexibility, in advice, could partly be achieved in the original implementation through the use of macros in C. However, macros do not offer a mechanism comparable to *superimposition*, so that one always has to explicitly import macro libraries in each file and place macro calls in the appropriate locations.

2.2.7 Acceptance

At the end of this project, we had a meeting with a group leader and several developers. We demonstrated the prototype and, using a presentation, discussed all details, benefits and worries. These were the worries that they themselves had expressed, they saw the benefits and were confident that all worries were addressed accordingly.

The prototype was accepted, initially only for concern Tracing and a transfer project was started. This transfer project developed the tooling from scratch. This was needed as the prototype lacked the ability to perform control and data flow analysis. Also, there were optimizations possible that did not fit into the design of the prototype. The development of the tooling required almost a year and required around 4 persons. In January 2007, the first source files were automatically woven. In April 2007, the first components could use the tooling. In November 2007, 42 components used the weaver, to weave 1007 source files and aspect Tracing was used.

2.3 Aspects in .NET

The next project we conducted, was on a different system at ASML. This was a new system that was developed using the Microsoft .NET framework [Micb] and mostly used the C# programming language [Mica]. This was a new approach for

the company, as all previous systems were built using C. We were approached by the architect of the project, whether we could help them in preventing the problems with crosscutting concerns that had plagued the C based system. They realized that they now had the opportunity to develop differently from the start, thus avoiding migration problems later. However, they had no idea about what kind of tooling was available.

2.3.1 Initial Benefits

Similar to section 2.2.1, the expected benefits of an aspect-oriented approach are: statement reduction, effort and lead-time reduction, a general solution to crosscutting concerns and a in general a better software structure. We cannot quantify these claims at this point. Once we have an implementation we can determine the exact statement reduction, for example.

2.3.2 Context

The new system was developed using the .NET framework, version 2. *Visual Studio 2005* [Micc] and *MSBuild* [Micd] were used as development environment. In listing 2.3, we provide here an example method found in this new system.

```
1  public void PickFromFoup(Port fromPortId, int fromSlotNumber)
2  {
3      Logging.TraceMethodEntry(Constants.ComponentName, fromPortId, fromSlotNumber);
4
5      checkSlotNumber(fromSlotNumber);
6      // ...
7      m_efem.FaceStation(m_WaferStageStation, m_WaferStageSlot, m_UnloadGripper);
8
9      Logging.TraceMethodExit(Constants.ComponentName, null);
10 }
```

Listing 2.3: Example method in C#

The details of the method will be elaborated upon in the next section. One can see that the method has two trace statements (lines 3 and 9). Also notice, that the error handling code is no longer scattered and tangled in this method, since C# offers a native exception handling mechanism.

2.3.3 Aspects

The .NET framework offers a native exception handling mechanism. Therefore, we were not expecting error handling to have a major impact on the code base.

We received a copy of the source code and we investigated this. The only crosscutting concern we found was:

Parameter Tracing (lines 3 and 9 in listing 2.3): Tracing ensures that the value of all parameters is written to a log file, which includes method arguments and the return value.

This concern was also present in the previous project. We do expect more crosscutting concerns once more code is developed, but at the time of the project, concern Tracing had the highest priority. Concern Profiling was mentioned as a likely candidate. The architect of the system stated that they wanted general purpose tooling, and not tracing-specific tooling, since he was aware that in the future new crosscutting concerns could be addressed using the same tooling.

2.3.3.1 Concern "Parameter Tracing"

The definition of concern Parameter Tracing, or Tracing for short, is almost exactly the same as in the previous project (see section 2.2.3.2).

Purpose: The goal of the concern is to trace the values of the formal parameters of all methods within a certain namespace or package. Tracing should be turned on and off for specific components at run-time.

Design: Input arguments should be traced at the start of a method, using method Logging.TraceMethodEntry (line 3). Output arguments and the return value should be traced at the end of a method, using method Logging.TraceMethodExit (line 9). Input and output arguments should be traced at both location. There was a library that provided methods with variable arguments to implement TraceMethodEntry and TraceMethodExit. This design of tracing can not be satisfactory implemented using inheritance, since it would still require inserting code at the start and end of each method as well as modifying one or more common super classes.

Example: Lines 3 and 9 of listing 2.3 show an example snippet of concern Tracing.

Implications: The calls to the tracing library are crosscutting the component.

Impact: The impact is unknown, since this is newly developed system. However, each class has to import the tracing library and call the tracing library twice in each method.

Implementation details: The following 2 elements are required to successfully implement concern Tracing. These are thus requirements for our solution.

- Trace the name of the component and any arguments that have the implicit in specifier at the start of a method.
- Trace the name of the component, any arguments that are have the explicit out specifier and the return value at the end of a method.

C# has explicit support for input and output parameters, using keywords in and out respectively, and to our knowledge these were used consistently and correctly. This property can be queried using reflection, as such there was no need for annotations, as used in the WeaveC project.

2.3.4 Worries

We now discuss the worries that were expressed by the stakeholders of the project. Some of these worries are shared with the previous case. We matured the tooling and addressed the worries detailed in the rest of this section, with funding from ASML. This project was called *StarLight*. StarLight has been released on the Compose* SourceForge website: `http://composestar.sourceforge.net`.

2.3.4.1 Migration

Migration was not considered an issue, as the system was newly developed. There were already some crosscutting tracing calls in the source of the application. However, these could be identified and removed easily using simple tooling.

2.3.4.2 Availability and maturity of tooling and process

As mentioned previously, Compose*.NET was developed as an academic prototype for research purposes. The focus therefore had not been on robustness and maturity. With funding from ASML we were able to create robust and mature tooling. We verified the robustness by weaving on a test version of the new system. All tasks and tests succeeded, and a huge trace file was generated.

2.3.4.3 Ability to control the application of aspects

Currently only aspect Tracing is implemented and this aspect can be removed in case of problems. However, this does affect diagnostics capabilities in the field. As Compose* already supported annotations we could easily enable developers to add annotations that prevented weaving of tracing code or other aspects.

For example, when we tested the system in simulation mode, there was a huge performance hit of around 600%. We investigated this further and found out that it was solely caused by one library. This was an image processing library that performed a large number of complex mathematical operations on a large 20 Mega Pixel image. However, the implementation of the image processing library was such that all get and set access of each individual pixel was traced. This produced a staggering amount of mostly useless tracing information. We therefore introduced an annotation that could be attached to a .NET assembly (library) and that prevented weaving on that assembly.

2.3.4.4 .NET Framework Version 2 Compatibility

The original implementation of Compose* only supported .NET Framework version 1. Version 2 introduced numerous enhancements, such as generics, to the intermediate language. As a result, our original version did not work with code from version 2. This thus required redesigning and re-implementing parts of the Compose* tool set. However, this was required any way, since we switched from an interpreter based implementation to an in-lining implementation (see section 2.3.4.6).

2.3.4.5 Compile-time Performance

The requirement for the compile-time performance was at most 100% extra build time overhead. At the end of the project we had achieved an overhead of around 65%, with clear potential to reduce this even further. We have chosen to increase the compile-time overhead slightly to gain more performance at runtime. We evaluated all superimpositions and filters as much as possible statically. We only inject code relating to filter actions which can be executed in a specific method. This requires more analysis and thus more compile time. However, the runtime overhead is decreased.

2.3.4.6 Run-time Performance

The performance of the machine at run time was only allowed to be decreased by at most 10%. We reduced the performance of the machine by 0.9%, when not tracing and were faster than the manually inserted tracing statements, when tracing was enabled for a specific component. We analyzed the base code in detail and stored this analysis. This original implementation used reflection

to gather input and output parameter specifications. The result was that our tracing implementation was 65% faster than the original tracing implementation.

Conditional Superimposition: Although we had adequate performance in our testing environment, when we tested the system on the actual machine, we found that implementation was 60% slower when tracing was disabled. This was caused by the repeated construction of context information that was required to determine whether to trace or not. Building up this context takes a substantial amount of time. However, only after creating the context, the condition that checked whether to trace or not was evaluated. We solved this problem by implementing a new feature in Compose*, called conditional superimposition. Conditions that are specified in such a construct are evaluated before the context object is created. As such they only have access to a limited set of context information. With this extra feature we reached the 0.9% performance reduction.

2.3.4.7 Ability to Debug

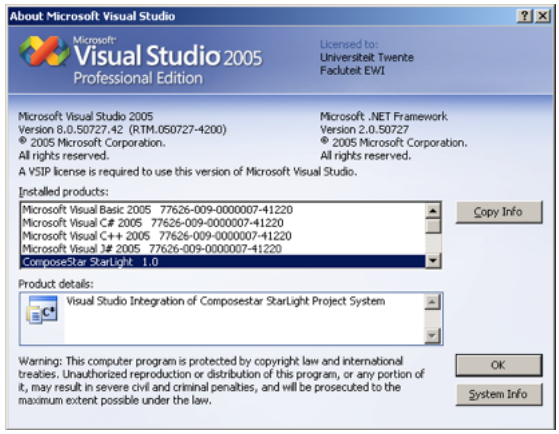
As with the C project, we limited the ability to debug, since we changed the intermediate language. We had to update the corresponding *Program Debug DataBase(PDB)* files to reflect this change. We used the CECIL [Eva] framework for manipulating the IL code. The latest version of the framework also updates the debug database to reflect the changes in the intermediate language. This restored debugging of the original code and allowed debugging of the filter actions. Debugging of the concern code itself was not supported, but not required either.

2.3.4.8 Integration within the Development Environment

ASML uses *Visual Studio* and *MSBuild* for developing the system in this project. To ensure a minimal learning curve, a tight integration with this environment was required. Figure 2.8 shows the about screen of *Visual Studio 2005*. The last item in the installed features is StarLight.

Our Visual Studio integration provides several features:

- Compose* language services, supporting keywords, limited command completion and early syntax error detection.
- A Compose* project type that can be used to define a project with filters. All referenced libraries of this project will be subjected to weaving.
- MSBuild integration. Compose* uses MSBuild for incremental compilation and for the compilation process itself.



Figuur 2.8: About screen of *Visual Studio 2005*

Especially the first item was important, since none of the developers was familiar with the Compose* language or AOP.

2.3.5 Tooling

We used Compose*.NET [Unia] as a starting point for the project. Compose* has been developed at the University of Twente. It is an implementation of the Composition Filters model. At the time the project started, we had an academic prototype implementation based on a run time interpreter suitable for .NET framework version 1. Compose* features a lot of advanced features and analysis tools. We used Compose* to demonstrate our solution on their code. The developers were immediately convinced of its functionality and potential.

Listing 2.4 presents the implementation of concern Tracing.

```
1  concern TracingConcern
2  {
3    filtermodule TracingModule
4    {
5      inputfilters
6        trace : Tracing = { [*.*] }
7    }
8
9    superimposition
10   {
11     conditions
12       shouldTrace : TracingLib.Tracer.ShouldTraceComponent;
13     selectors
```

```
14      traceClasses = { Class | isClass(Class), not(classHasAnnotation(Class, Annotation)),
                isAnnotationWithName(Annotation,'NoTrace') };
15    filtermodules
16      shouldTrace => traceClasses <- TracingModule;
17  }
18 }
```

Listing 2.4: Implementation of concern Tracing in StarLight

Listing 2.4 defines concern TracingConcern. This has a single filter module (lines 3 to 7) and a superimposition specification (lines 9 to 17). Filter module TracingModule declares only one input filter, called trace of type Tracing (lines 5 and 6). The exact implementation of filter type Tracing is not shown here. Filter trace accepts all messages, and will trace these accordingly. We superimpose filter module TracingModule on all classes that are not annotated with a NoTrace annotation (lines 14 and 16). We only superimpose the filter module (line 16), if condition shouldTrace is true, this condition is a call to method TracingLib.Tracer.ShouldTraceComponent (line 12).

2.3.6 Quantified Benefits

We now discuss the benefits of our AOP solution. These are similar to the previous project. We do not have as detailed information about the system as in the other project though, since most of the code was still being developed.

2.3.6.1 Potential Statement reduction

This system was newly developed, and as such we had no idea about the possible amount of tracing code that would have been manually inserted. It is hard to make an estimation, but we estimate around 4% LLOC. As this 4% does not have to be written, the developers can spend their time on other parts of the code.

2.3.6.2 Effort and Lead time reduction

We did not calculate the effort and lead time reduction for this specific system. We expect that the results are similar to those results gained in the previous project. The effort reduction and lead time reduction will be smaller, since we only address one concern instead of three. Therefore, we expect about 2-3% effort reduction and about 1-2% lead time reduction. Again, this reduction scales up to all software developers of this system, currently around 60.

2.3.6.3 Generality of solution

We used an already existing tool, Compose* to implement the solution. This is a general purpose and extensible software composition environment that also supports AOP. We showed that we could also profile methods using Compose*.

2.3.6.4 Local deviations

Compose* supports the usage of annotations. This enabled the developers to turn off tracing or even weaving for certain components or external libraries. This was primarily required to achieve better performance.

2.3.6.5 Better software structure

This benefit has been stated as one of the primary benefits of AOP and separation of concerns in general. The pointcut and advice mechanisms provide an easy way to deal with changing coding guidelines or changes in the tracing idiom. This project also showed that even with a modern language like C#, there are still crosscutting concerns that are not adequately addressed.

2.3.7 Acceptance

At the beginning of 2007, we delivered a version of StarLight that addressed all worries. At that time though, there were problems with the core functionality of the machine that required more immediate attention. As such it is still unclear whether ASML will adopt StarLight. ASML has continued to express interest in the tooling. We are hopeful that we will be able to transfer the tooling to ASML in the future. We expect the need for this tooling to only increase.

2.4 Related Work

There are numerous papers about the importance of industrial success stories, such as [WHM07], [GSE+07] and [BCMS03]. However, papers actually reporting on success stories are rare.

There are some success stories about the industrial adoption of AOP. In [BF06], Bodkin et. al. report an experience where the authors applied aspects to provide

feedback on user behavior, system errors, and to provide a robust solution for a widely deployed diagnostic technology for DaimlerChrysler. Aspects are used as a reflective means to gather information about the system.

In [CC04], Colyer and Clement discuss large scale AOSD for middleware. The authors report on a project they conducted at IBM. They present a set of challenges they faced while executing the project. One of the aspects tackled in the project is also *Tracing*. They also showed that even a "simple" concern like *Tracing* can be an excellent driver to adopt AOP. The conclusion of the authors is that AOP can be used successfully on a large scale.

In [MKL97], Mendhekar, Kiczales and Lamping presented the results of a project that compared the implementations of an image processing system. The authors compared the performance of a naïve OO implementation, an optimized OO implementation and an AOP implementation. The authors showed that the performance of the AOP solution was comparable to the performance of the optimized OO version. The performance of the naïve OO implementation was much slower. The AOP solution required 88% fewer lines of code (including the weaver).

2.5 Conclusions

In this chapter we motivated that there is currently no wide-spread reported adoption of AOP in industry. In the Ideals project, we conducted two case-studies for transferring an aspect-oriented solution to ASML. These case studies were executed on two distinct systems at ASML. These systems had widely different development methods. In this chapter, we reported on our experience introducing AOP at ASML. We proposed a process that consists of several steps that aim at providing a solution which fits into the context of the company. This context consists of the programming language, design methods, software development process, etc. Understanding this context is in our view imperative for a successful adoption of AOP. Knowing the context also helps in better expressing the benefits and drawbacks of the solution. Finally, we have to elicit and address key quality requirements, before an aspect-oriented solution can be transferred.

The proposed process does not guarantee acceptance of an aspect-oriented solution, as one project demonstrated. However, it provides guidelines to create the optimal conditions for technology transfer. In one of the described projects the proposed technology and prototype have been transferred to the company. Parts of the work described in this chapter have been published in [DGB+06].

An Assessment of an Aspect-based Approach to Tracing

3

In this chapter, we aim at providing reliable evidence of the advantages and possible limitations of AOP languages so that adoption of this technology can be based on rational arguments. We first explain concern tracing, which we used in our experiment. Next we explain the design of the experiment, here we discuss the subjects, environment, treatments, objects, variables and hypotheses. Subsequently, we discuss results of the experiment and verify our hypotheses. Finally, we provide a detailed discussion about possible validity treats, the generalizability of the experiment and conclude.

3.1 Tracing

AOP is suitable to address many (complex) crosscutting concerns besides *Tracing* and *Logging*, as motivated in [Lad06]. *Tracing* was chosen as the main driver for the adoption of AOP by ASML. The concern accounted for around 7% [DGB+06] of the executable statements, for one specific component. Although this varies per component, the amount of scattered and replicated code of concern *Tracing* is large. Concern *Tracing*, at first glance, seems to be a "simple" and "trivial" aspect. However, in practice this is not the case. In [BvDDT07], Bruntink et. al.

discuss the variation of the implementation of concern *Tracing*. This variation does not impact the representativeness of the tracing aspect, since this variation was mostly syntactic and did not impact the essence of the concern. The paper argues that for automated migration even this mostly syntactic variation is a problem. The functionality implemented by concern *Tracing* is required for effective diagnostics of the machines. As such, moving to an AOP solution should exceed, or at least match, the requirements stated for concern *Tracing*. We only elaborate on those details of concern *Tracing* and its aspect implementation, which are relevant to the developers of the base code. First, we show the manual solution of tracing. Second, we present aspect Tracing, which is now in use at ASML and which was used for the experiment.

3.1.1 Concern Tracing

We show an example of concern *Tracing* as it is currently implemented in C. The current tracing implementation uses the so-called THXA framework, which is a tracing framework developed internally at ASML.

```
1  int CCXA_change_item_ids(ITEM_DEF *item_def_1, ITEM_DEF *item_def_2)
2  {
3    int result = OK;
4    int item_id = LO_UNDEFINED_ITEM_ID;
5
6    THXAtrace("CC", TRACE_INT, __FUNCTION__, "> (item_id_1 = %d, item_id_2 = %d, active = %b)",
7             item_def_1 != NULL ? item_def_1 -> item_id : 0,
8             item_def_2 != NULL ? item_def_2 -> item_id : 0, MACHINE_IS_ACTIVE);
9
10   if(result == OK && ( item_def_1 == NULL || item_def_2 == NULL )
11   {
12     result = CC_PARAMETER_ERROR;
13   }
14
15   if(result == OK && MACHINE_IS_ACTIVE )
16   {
17     item_id = item_def_1 -> item_id;
18     item_def_1 -> item_id = item_def_2 -> item_id;
19     item_def_2 -> item_id = item_id;
20   }
21
22   THXAtrace("CC", TRACE_INT, __FUNCTION__,"< (item_id_1 = %d, item_id_2 = %d) = %R",
23            item_def_1 != NULL ? item_def_1 -> item_id : 0,
24            item_def_2! = NULL ? item_def_2 -> item_id : 0, result);
25
26   return result;
27 }
```

Listing 3.1: Tracing example in C

Listing 3.1 declares one function called CCXA_change_item_ids. This function exchanges the identifiers of two items. Lines 17 to 19 show the implementation of these two item identifiers. This function also implements error handling (see lines 3, 10, 12, 15 and 26). Concern *Parameter Checking* is implemented at lines 10-13. More interestingly are lines 6-8 and 21-24. At these lines two trace calls are stated using function THXAtrace.

Trace call THXAtrace at lines 6-8, has several arguments. The first is a textual representation of the component in which this file is located, in this case CC. The second is a constant passed to indicate whether this is an internal or external tracing call. For this chapter, this distinction is not relevant. The third argument is a GCC meta variable(__FUNCTION__) is used, which is replaced by the name of the function at compile-time. The fourth argument is a format string which controls how this trace entry should look like. In this case we are tracing the item identifiers of both arguments and a global variable called MACHINE_IS_ACTIVE. The next two arguments are the actual argument values which should be inserted in the appropriate places in the format string. We also verify that the arguments are not NULL. If they are NULL and we do refer to the field item_id, since this could cause a segmentation fault in the system, effectively shutting down the machine. Implementing a guard ensures that if the arguments are null, a zero is printed in the trace file. Since global variable MACHINE_IS_ACTIVE is being read inside this function, we pass this variable as the last argument of the THXAtrace call.

The trace call at lines 21-24 is similar to the former trace call. One difference is that since global variable MACHINE_IS_ACTIVE has not been changed in this function, we no longer have to trace this variable. Another difference is that we trace the return value of this function, since the return value indicates whether the function executed successfully. Similar to the previous trace call, the two item identifiers are traced, as these may be modified in this function. Again, we check whether any of the arguments are not NULL.

In general all functions within the codebase are to be traced in the manner illustrated above. There are some exceptions, e.g. there is a subset of functions which cannot be traced, as the tracing framework has not been initialized yet when these functions executed. All parameters of all traceable functions should be traced. The parameters of a function include the following:

- Function arguments that are either queried, manipulated, or both in this function.
- Global variables that are either queried, manipulated, or both in this function.
- The return value of a function.

Furthermore, we distinguish between parameters that are input, output or both:

- *Input* parameters are those parameters which are only read.
- *Output* parameters are those parameters which are only manipulated, this includes those parameters passed to other functions.
- *InOutput* parameters are those parameters which are read and manipulated.

Concern Tracing should behave as follows:

- At the start of a function, trace the following items:
  - the name of the component,
  - an internal or external trace specifier,
  - the name of the function,
  - *Input* and *InOutput* parameters.
- At the end of a function, trace the following items:
  - the name of the component,
  - an internal or external trace specifier,
  - the name of the function,
  - *Output* and *InOutput* parameters.
  - the return value of the function.

The just described behavior shows that concern *Tracing* is far from trivial. Even modern AOP languages have problems addressing this effectively. Implementing this concern requires detailed control and data flow analysis to determine the *Input*, *Output* and *InOutput* classification for all arguments and variables that are used inside a specific function. Most AOP approaches do not support this detailed analysis.

3.1.2 Aspect Tracing in WeaveC

Now that the requirements of concern *Tracing* have been elaborated, we present the details of the aspect specification that are required to understand the experiment. We used an industrial-strength C weaver developed by ASML, called *WeaveC*, and an AOP language, called *Mirjam*, which are detailed in [NvEvdP07]. Aspect *Tracing* uses the THXA framework for tracing the values.

The requirements of concern *Tracing*, can be implemented solely in a single aspect specification. However, there are cases where developers need to deviate from the concern, mostly for performance reasons. To allow such deviations, *WeaveC* supports annotations. The following annotations have been defined for aspect Tracing:

- $trace(TRUE \mid FALSE)$: controls if a module(file), function, parameter, variable or type should be traced.
- $trace\_as(fmt = \text{“}\ldots\text{”}, expr = \text{“}\ldots\text{”})$: traces a parameter, variable or type in a different manner.

In short, aspect *Tracing* can be defined as:

- All functions should be traced
  - Except for functions that are annotated with $trace(FALSE)$ or whose module is annotated with $trace(FALSE)$.
- For each traceable function, trace the input parameters at the start and the output parameters at the end of the function.
  - Except for parameters annotated with $trace(FALSE)$ or whose type is annotated with $trace(FALSE)$.

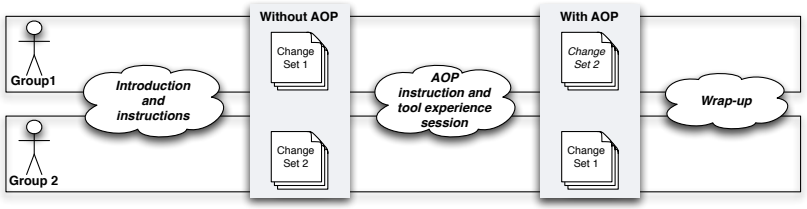
Concern *Tracing* has been implemented by ASML. Within ASML there are only a few developers that write aspects. Most developers write base code that is subjected to these aspects. Therefore, the impact of aspects on base-code development is much more significant. In this experiment we purely focused on the development and maintenance of base code. Base code developers will not see the aspect specification, since it is assumed that all functions the base developers write are traced. The base developers only need to write annotations to influence the aspects, if required.

The development of the aspect and weaver are out of the scope of this experiment. The developers of the weaver also created the aspect specification of tracing, and used it as a test case for the tooling. Also, since the tracing aspect still uses the THXA framework, the extra overhead of writing the aspect is minimal. For the experiment we assume that the weaver and an implementation of aspect tracing are present.

3.2 Experiment Setup

We first informally present the setup of the experiment. For the statistical results we used the theory and guidelines as presented in [WRH+00] and [KPP+02]. The experiment was conducted in combination with a training on *WeaveC*. Figure 3.1 presents an overview of the training and the experiment.

The goal of the experiment was to determine whether using an aspect-oriented approach to *Tracing* helps to reduce the development and maintenance effort. We determined this by measuring both the time it takes to implement tracing-related



Figuur 3.1: Overview of the experiment and training. Change Set 1 and Change Set 2 represent the same type of change requests on two different source codes.

change scenarios, and the errors introduced while implementing these scenarios.

The subjects were split into two groups while executing the change scenarios. To prevent possible differences between the subjects in the groups, all subjects were present at the instructions. The introduction included not only the overview of the training but also a brief explanation of concern *Tracing*. This ensured that all subjects had the same notion of tracing. During the introduction a balanced selection of groups was made, based on the years of software engineering experience and the years of working at ASML. Section 3.4 provides a discussion about the differences between the groups.

After the introduction, the groups were split. Group 1 executed change scenario set 1, while group 2 executed change scenario set 2. Both groups had to implement 5 change scenarios using the current, manual, way of tracing. Change sets 1 and 2 contained similar scenarios. However, the code base on which the scenarios were executed was distinct. The subjects were allowed to spend at most 45 minutes on this part of the experiment.

After the first session, a general introduction into AOP and a specific explanation on the way of working with *WeaveC* was presented, again all subjects were present. Next, the subjects had the opportunity to familiarize themselves with the new process and tooling, using a set of simple exercises.

Subsequently, group 1 had to implement change scenario set 2 using *WeaveC*, while group 2 implemented change scenario set 1 using *WeaveC*. Again the subjects executed up to 5 scenarios within 45 minutes. We used this so-called *cross-over design* to prevent or minimize learning effects between sessions without *WeaveC* and with *WeaveC*. Section 3.4 discusses the implications of this choice on the validity of the experiment.

For each completed scenario, we determined the time and the error classification.

We used the command logs and version control information for this.

3.2.1 Subjects

The experiment was conducted as part of a training of *WeaveC*. As such we had no control over the selection of the subjects. The possibility of a training was published within the company to software group leaders. Subsequently, subjects could register themselves for the training.

We have gathered some characteristics of the subjects. We asked the subjects to fill in a questionnaire before the training started. The characteristics in the questionnaire were:

- Years of experience in software development
- Years working at ASML
- Gender
- Age
- Education Level
- C Level
- Regular THXA user

We used these characteristics for two reasons. Firstly, it allowed us to find possible interesting correlations between the performance of the subjects and these characteristics. Secondly, we distributed the subjects into two groups. We used the sum of the years of software development and years working at ASML to distribute the subjects into two groups. We made an ordered list with these sums. Next we assigned subjects to groups in an alternating fashion. This process provided a somewhat balanced selection of groups.

3.2.2 Environment and Tooling

The experiment and training were conducted at an external location, i.e. not ASML. The two groups were both located in different rooms. During the presentations and instructions all persons were in the same room.

We used an external location to prevent the impact of the working environment on the experiment. The subjects could use their own build environment and tools, using remote log-in. This ensured that the build performance would not influence the experiment as the subjects used the build farm at the company. We restricted the usage of shell aliases, to prevent overriding our command logger and timing facilities.

We gathered data from two data sources: command log data, and source versioning data. The first was a list of commands which were entered in the terminal, with time stamps. In addition the exit codes of the build commands were logged. An example is shown here:

```
1 Tue Jul 3 09:34:37 MEST 2007, cleartool co -nc CCEWdata.c
2 Tue Jul 3 09:44:40 MEST 2007, > ccmake CCEWdata.osparc
3 Tue Jul 3 09:45:13 MEST 2007, < ccmake 1
4 Tue Jul 3 09:45:39 MEST 2007, > ccmake CCEWdata.osparc
5 Tue Jul 3 09:46:04 MEST 2007, < ccmake 0
6 Tue Jul 3 09:46:18 MEST 2007, cleartool ci -nc CCEWdata.c
```

Line 1 states a checkout of file CCEWdata.c. After ten minutes the subject proceeded to build the file (line 2). This build fails after 30 seconds (exit code 1). After 20 seconds the subject builds the file again, see line 4. This build executed successfully (exit code 0), see line 5. Finally, at line 6 the user checks in the file.

The second data source was the check-in and checkout information from a version control system. We used this as a backup, if for some reason the key logger did not work. Here is a line from this data source, corresponding to the above mentioned data source of file CCEWdata.c:

```
1 CCEWdata.c;20070703.115123;20070703.094618;20070703.093438;20070703.093438
```

This line has to be read from right to left. One can see the same time stamps as in the previous example. To determine the error classification, we examined the committed version of the edited source files to classify the changes the subjects had made to the functions in the files.

3.2.3 Treatments

We used change requests as treatments to determine the possible gain of using *WeaveC*. We only selected change requests which affect *Tracing*. Since there was only limited time to execute the scenarios, we selected the top four requests. This selection was made with input from ASML to decide which requests were the most frequently occurring and most valuable, according to ASML. These were the selected change requests:

1. Add tracing to a (traceless) function.
2. Change the parameters of a function.
3. Remove tracing from a function.
4. Selective tracing - only trace a specific parameter.
5. Remove a function.

We added a fifth scenario, that was executed first, to account for an initial overhead (see section 3.4). We did not include a scenario related to changing concern *Tracing*. We only focus on the scenarios which impact the developers of the base code. At ASML only a small set of developers will write or change aspects. The vast majority will write base code that is subjected to aspects. Also, changing concern *Tracing* is not often done. With AOP this becomes a lot easier but this is out of the scope of this experiment. For each scenario we will now show what is required to fulfill these requests without and with AOP.

1. **Add tracing to a traceless function**:
   Without WeaveC: Add the appropriate tracing code to the function, accounting for parameter usage and null pointer checks.
   With WeaveC: Nothing
2. **Change the parameters of function**: (from Input to InOutput)
   Without WeaveC: This requires adding null pointer checks and adding or altering tracing at the end of a function.
   With WeaveC: Nothing
3. **Remove tracing from a function**:
   Without WeaveC: Remove trace statements from the function.
   With WeaveC: Attach a $trace(FALSE) annotation to the function.
4. **Selectively tracing - Only trace a specific parameter**:
   Without WeaveC: Alter the trace statements to reflect this situation, removing all the other parameters from these statements.
   With WeaveC: Attach a $trace(FALSE) annotation to all parameters except the one which should be traced.
5. **Remove a function**: This is used to reduce the initial overhead and to measure possible learning effects between scenarios.
   Without WeaveC: Remove function body.
   With WeaveC: Remove function body.

In the last scenario, we cannot remove the entire function, since the call sites would have to be adjusted then as well, to enable a successful build. The subjects did not have to reflect changes to the core functionality only changes to the code related to tracing.

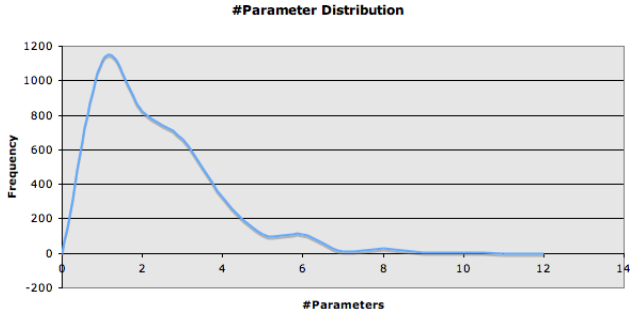
3.2.4 Objects

The code we used in our treatments has been taken from the ASML codebase. We used the largest component(CC) as a source for objects. We measured several metrics of the code to ensure that we selected representative objects. We used three metrics to select representative functions: *Number of parameters*, *Lines*

of Code and *McCabe cyclomatic complexity*. We chose the first metric because the complexity of tracing directly relates to the number of parameters. The latter two are well established metrics for the perceived complexity of source code [McC76].

3.2.4.1 Number of Parameters

Figure 3.2 shows the distribution of the number of parameters within component CC. There are many functions which have one parameter. This is mostly caused because the return value of the functions is used for error handling and even simple "getter" functions use a parameter. Based on the graph and the average (2.71), we chose to select only functions with between 2 and 4 parameters.



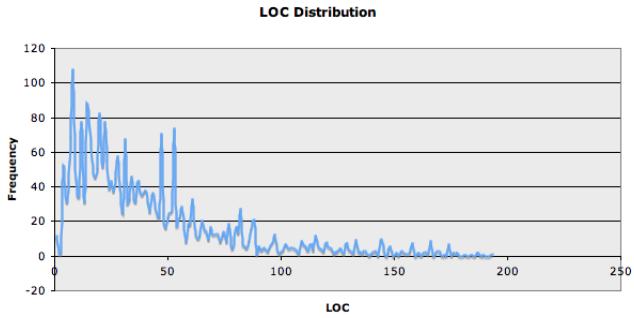
Figuur 3.2: Parameter Distribution

3.2.4.2 Lines of code(LOC)

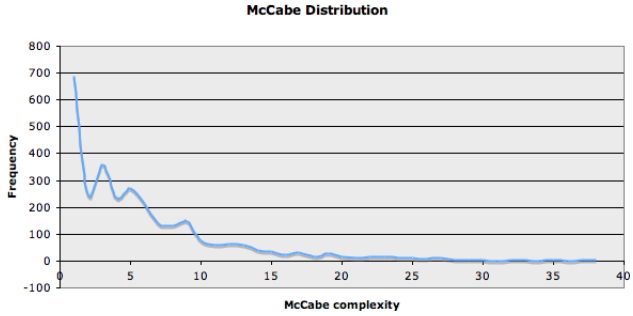
Figure 3.3 shows the distribution of LOC within component CC. We chose to select only those functions that have between 40 and 55 lines of code. This was based on the graph and the average (47.08),

3.2.4.3 McCabe cyclomatic complexity

Figure 3.4 shows the distribution of the McCabe complexity of functions within component CC. Based on the graph and the average (6.16), we chose to select only functions with McCabe complexity between 5 and 10.



Figuur 3.3: LOC Distribution



Figuur 3.4: McCabe Distribution

3.2.4.4 Values of the metrics

Table 3.1 provides the values for these metrics for each object and treatment combination. This combination is called a *Change Scenario*.

During the design of the experiment we had a rough idea of the average time of the execution of all five scenarios. This was 36 minutes. There was a chance the subjects would take a lot more time than anticipated. Since we used a paired sample test for analyzing the experiment, we had to ensure that we had the most paired scenarios from the subjects. We also wanted to ensure that we had results for the most important set of scenarios. We ordered the scenarios with input from ASML, to get the most valuable information out of the experiment.

We used a process called clustered randomization to randomize the scenarios.

Change Set	Change Scenario	Description	#Params	#LOC	McCabe
CS1	CS1_1	Add tracing	3	44	9
CS1	CS1_2	Change the parameters	2	49	6
CS1	CS1_3	Remove tracing	3	44	5
CS1	CS1_4	Selectively trace a parameter	3	53	8
CS1	CS1_5	Remove a function	3	53	5
CS2	CS2_1	Add tracing	2	50	6
CS2	CS2_2	Change the parameters	4	48	8
CS2	CS2_3	Remove tracing	2	48	5
CS2	CS2_4	Selectively trace a parameter	2	51	8
CS2	CS2_5	Remove a function	3	52	7

Tabel 3.1: Metric values for each change scenario.

$CS5$ was always put first, since this captures the initial overhead. Next $CS1$ and $CS2$ were randomly selected. Finally, $CS3$ and $CS4$ were randomly selected. The following permutations were possible:

- CS5 CS1 CS2 CS3 CS4
- CS5 CS1 CS2 CS4 CS3
- CS5 CS2 CS1 CS3 CS4
- CS5 CS2 CS1 CS4 CS3

We assigned the order within a change scenario set to the subjects at random.

3.2.4.5 An example scenario

We show part of an instruction sheet for one scenario, in this case CS1_1: Change Set 1 - Adding tracing to a function.

1. Navigate to directory: CC/CCQM/int/bin:

```
cd CC/CCQM/int/bin
```

2. Check out file: CCEWdata.c:

```
cleartool co -nc CCEWdata.c
```

3. Locate the following function: CCEWDA_set_wafer_state,
4. Generate tracing code for this function.
5. Build the file:

```
ccmake CCEWdata.osparc
```

6. In case of build errors, please go to step four and fix the build errors and rebuild the system, until there are no more build errors.

7. Check in file: CCEWdata.c

```
1 cleartool ci -nc CCEWdata.c
```

3.2.5 Variables

3.2.5.1 Factors

WeaveC is the only factor of this experiment. This factor is measured in the nominal scale, at two levels: without WeaveC and with WeaveC.

3.2.5.2 Independent Variables

The independent variables of our experiment are:

- **Years of experience in software development**: Numeric
- **Years working at ASML**: Numeric
- **Gender (0=Male, 1=Female)**: Numeric
- **Age**: Numeric
- **Education Level (4=PhD, 3= MSc, 2=BSc, 1=Practical Education)**: Numeric
- **C Level (5=Expert, 1=None)**: Numeric
- **Regular THXA user (1=Yes, 0=No)**: Numeric

3.2.5.3 Dependent Variables

The dependent variables of our experiment are:

- **Time** to execute a change scenario, in secs: Numeric
- **Error** classification of a change scenario: Numeric

3.2.5.4 Error classification

We chose to use an error classification instead of simply counting the number of errors. Most of the time only one error is introduced into the concern code. Also, the impact of the errors differ. Therefore, we used the following error classification:

0 : No errors,

1 : Typo in tracing text string,
2 : Tracing less than is required,
3 : Wrong tracing,
4 : No parameter checking code for possible invalid or null pointers.

An error of class 4 can result in a segmentation fault, and is as such considered the worst case. A higher number indicates a more severe error. In case of two errors we used the worst case, this only occurred once in the data set. We manually inspected the committed version to determine which class of errors were introduced, if any. Since a scenario was only completed if there were no compilation errors, we purely focus on those errors that are detected at testing or execution time.

3.2.6 Hypotheses

We defined the following null hypotheses:

- *WeaveC* does not reduce the development time, while developing and maintaining code related to *Tracing*:
  H_0^{time}: $time_{without\_WeaveC} \leq time_{with\_WeaveC}$
- *WeaveC* does not reduce the severity of errors, while developing and maintaining code related to *Tracing*:
  H_0^{error}: $error_{without\_WeaveC} \leq error_{with\_WeaveC}$

For this experiment we defined the following alternative hypotheses:

- *WeaveC* does reduce the development time, while developing and maintaining code related to *Tracing*:
  H_a^{time}: $time_{without\_WeaveC} > time_{with\_WeaveC}$
- *WeaveC* does reduce the severity of errors, while developing and maintaining code related to *Tracing*:
  H_a^{error}: $error_{without\_WeaveC} > error_{with\_WeaveC}$

These hypotheses will be tested in the next section for each of the five scenarios.

3.3 Experiment Results

We split the total group of subjects into two groups. The training was scheduled for one day with a morning and an afternoon session.

3.3.1 Subjects

Table 3.2 presents the so-called descriptives of the subjects. These contain; the number of scenarios, the minimum values, the maximum value, the mean and the standard deviation.

Characteristic	N	Min	Max	Mean	Std. Dev
Age	17	26	45	34.88	5.73
Education	17	2	3	2.41	0.51
Years at ASML	17	0	7	3.40	2.78
Years in SW development	16	1	20	9.12	5.62
Clevel	17	1	5	3.71	0.99
THXA	17	0	1	0.59	0.51

Tabel 3.2: Descriptives of the Subjects

In the morning session, two subjects were accidentally included in group one. This is why, in the morning session, group one had more subjects than group two. Section 3.4 discusses this discrepancy more thoroughly. Furthermore, we excluded three subjects and a series of data points (see section 3.4 for more details).

3.3.2 Initial processing

Since the number of subjects is low, it is hard to get statistically significant results. Before we present these statistical results, we will first present our observations based on these results.

Table 3.3 presents the number of data points and the average development effort of each scenario respectively without and with *WeaveC*, as well as the delta for these scenarios. A negative delta indicates that the subjects executed the scenario faster with *WeaveC*.

Scenario	without WeaveC		with WeaveC		Effort
	#	Average	#	Average	Δ
Adding Tracing	15	824	17	328	-60.2%
Changing Parameters	14	523	15	422	-19.4%
Removing Tracing	9	143	11	230	60.6%
Selectively Tracing	8	292	13	227	-22.2%
Remove Function	17	250	17	185	-26.2%

Tabel 3.3: Development Effort

Table 3.4 presents the number of data points and the average severity of errors of each scenario without and with *WeaveC*, as well as the delta for these scenarios. A negative delta indicates that the subjects introduced less errors with *WeaveC*.

Scenario	without WeaveC		with WeaveC		Error
	#	Average	#	Average	Δ
Adding Tracing	15	0.67	17	0.00	-100.0%
Changing Parameters	14	2.57	15	0.20	-92.2%
Removing Tracing	9	0.00	11	0.00	0%
Selectively Tracing	8	1.38	13	0.92	-32.9%
Remove Function	17	0.00	17	0.00	0%

Tabel 3.4: Errors

Observations
Based on the results in tables 3.3 and 3.4, we observe the following:

- Overall, users were able to implement 10 more change scenarios with the use of *WeaveC*.
- Overall, users were 6% faster with the use of *WeaveC*.
- Overall, users made 77% less severe errors with the use of *WeaveC*.
- Most scenarios require less effort with *WeaveC* than without *WeaveC*.
- Removing tracing from a function requires more effort with *WeaveC* than without. This is probably caused by the lack of experience in using the newly introduced annotations.
- Adding tracing and changing parameters introduces almost no errors with *WeaveC*, whereas manually implementing these changes result in more severe errors.
- Selectively tracing a function with *WeaveC* introduces (on average) less severe errors than without *WeaveC*.
- *WeaveC* introduces no new errors while removing tracing and a function.
- The manual implementation contained 8 critical errors, which could have resulted in a segmentation fault during run-time.

Table 3.5 presents the distribution of the errors for the five scenarios, without and with *WeaveC*. For each error class the number of errors for this class is presented in the corresponding cells.

3.3.3 Development Effort

We now present the statistical results from the experiment. We used SPSS Version 13.0 for Windows[SPS] to process the raw data and to execute the tests.

Scenario	without WeaveC					with WeaveC				
	0	1	2	3	4	0	1	2	3	4
Adding Tracing	7	1	1	2	0	15	0	0	0	0
Changing Parameters	3	3	0	0	7	15	0	0	0	0
Removing Tracing	9	0	0	0	0	11	0	0	0	0
Selectively Tracing	4	0	2	1	1	9	0	0	4	0
Remove Function	17	0	0	0	0	17	0	0	0	0

Tabel 3.5: Error Distribution over the scenarios

We have removed all statistical outliers from the data-set (see section 3.4).

3.3.3.1 Descriptives

Table 3.6 presents several details for all scenarios and development effort (measured in seconds), namely; the number of scenarios, the minimum values, the maximum values, the means and the standard deviations.

Scenario	N	Min	Max	Mean	Std. Dev.
Adding Tracing w/o *WeaveC*	11	254	902	583	227
Changing Parameters w/o *WeaveC*	12	76	648	412	177
Removing Tracing w/o *WeaveC*	9	91	210	143	45
Selectively Tracing w/o *WeaveC*	7	142	339	249	72
Remove Function w/o *WeaveC*	15	84	347	189	80
Adding Tracing with *WeaveC*	13	106	379	218	79
Changing Parameters with *WeaveC*	15	153	910	422	243
Removing Tracing with *WeaveC*	10	59	333	217	86
Selectively Tracing with *WeaveC*	13	89	421	227	120
Remove Function with *WeaveC*	15	106	253	179	46

Tabel 3.6: Descriptives of Effort

3.3.3.2 Significance

We calculated the significance through the use of a standard paired sample test with a confidence interval of 95%. Table 3.7 shows the results for each scenario.

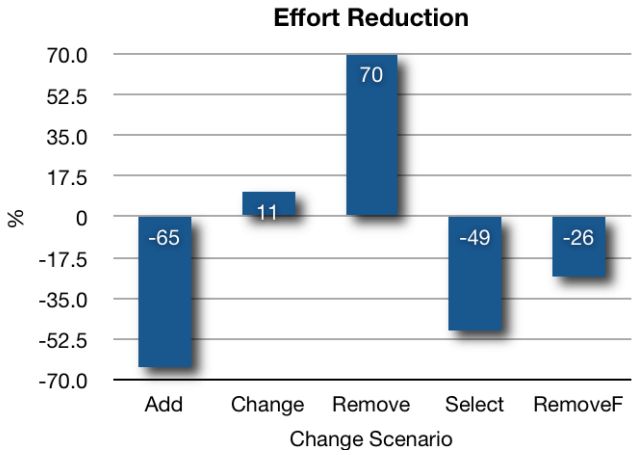
Figure 3.5 presents the differences of each change scenario.

3.3.3.3 Observations

The following observations can be made with a statistical significance of 95%:

Scenario	N	Mean	Std. Dev.	Sig.(2-tailed)
Adding Tracing	8	500	179	0.00
Changing Parameters	11	-47	370	0.68
Removing Tracing	6	-91	75	0.03
Selectively Tracing	6	95	78	0.03
Remove Function	13	16	69	0.41

Tabel 3.7: Significance of Effort



Figuur 3.5: Effort Reduction

- Adding tracing to a function takes less time with *WeaveC* than without.
- Removing tracing from a function takes more time with *WeaveC* than without.
- Selective tracing the parameters of a function, takes less time with *WeaveC* than without.

3.3.3.4 Correlations

We found the following correlations between the subjects and their performance without and with *WeaveC*:

- There was a strong, positive correlation between the difference in effort while changing a function and the number of years in software development [correlation factor=.624, number of measurements=11, significance≤.05]. In other words: subjects with more years of experience in software devel-

opment, worked faster when using *WeaveC*.

3.3.4 Errors

We now present the statistical results from the experiment with respect to the errors.

3.3.4.1 Descriptives

Table 3.8 presents several details for all scenarios related to the errors, namely the number of scenarios, the minimum values, the maximum values, the means and the standard deviations.

Scenario	N	Min	Max	Mean	Std. Dev.
Adding Tracing w/o *WeaveC*	11	0	3	0.82	1.25
Changing Parameters w/o *WeaveC*	12	0	4	2.58	1.78
Removing Tracing w/o *WeaveC*	9	0	0	.00	.00
Selectively Tracing w/o *WeaveC*	7	0	4	1.57	1.62
Removing Function w/o *WeaveC*	15	0	0	.00	.000
Adding Tracing with *WeaveC*	13	0	0	.00	.000
Changing Parameters with *WeaveC*	15	0	0	.00	.000
Removing Tracing with *WeaveC*	10	0	0	.00	.000
Selectively Tracing with *WeaveC*	13	0	3	0.92	1.44
Removing Function with *WeaveC*	15	0	0	.00	.000

Tabel 3.8: Descriptives of Errors

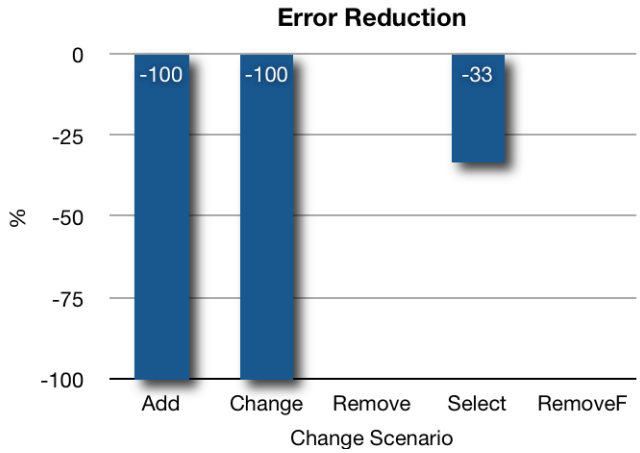
3.3.4.2 Significance

We calculated the significance through the use of a standard paired sample test with a confidence interval of 95% (see table 3.9).

Scenario	N	Mean	Std. Dev.	Sig.(2-tailed)
Adding Tracing	8	.0.38	0.74	0.20
Changing Parameters	11	2.50	1.81	0.00
Removing Tracing	6	0	0	-
Selectively Tracing	6	0.50	2.60	0.66
Removing Function	13	0	0	-

Tabel 3.9: Significance of Errors

Figure 3.6 presents the error reduction of each change scenario.



Figuur 3.6: Error Reduction

3.3.4.3 Observations

The following observations can be made with a statistical significance of 95%:

- Changing the parameters of a function manually introduces more errors than with *WeaveC*.

3.3.4.4 Correlations

We found the following correlations between the subjects and the difference between without and with *WeaveC*:

- There was a strong, negative correlation between the difference in errors while changing the parameters of a function and the number of years in software development [correlation factor=-.723, number of measurements=11, significance≤.05]. In other words the subjects introduced less severe errors, with *WeaveC*, if they were more experienced in software development.
- There was a strong, negative correlation between the difference in errors while changing the parameters of a function and the age of subjects [correlation factor=-.624, number of measurements=11, significance≤.05].

In other words the subjects introduced less severe errors, with *WeaveC*, if they were older.

- There was a strong, negative correlation between the difference in errors while selectively tracing a parameter of a function and the number of years in software development [correlation factor=-.879, number of measurements=7, significance≤.01]. In other words the subjects introduced less severe errors, with *WeaveC*, if they were more experienced in software development.
- There was a strong, negative correlation between the difference in errors while selectively tracing a parameter of a function and the age of the subjects [correlation factor=-.879, number of measurements=7, significance≤.05]. In other words the subjects introduced less severe errors, with *WeaveC*, if they were older.
- There was a strong, negative correlation between the difference in errors while selectively tracing a parameter of a function and the experience with C [correlation factor=-.789, number of measurements=7, significance≤.05]. In other words the subjects introduced less severe errors, with *WeaveC*, if they had more experience in C.

3.3.5 Verification of the Hypotheses

We now verify the hypotheses that were stated in section 3.2.6.

We can reject the H_0^{time} for scenarios Adding Tracing and Selectively Tracing. We can thus accept alternative hypothesis H_a^{time} for these scenarios.

We cannot reject H_0^{time} for scenarios Changing Parameters and Remove Function. Also, we have to accept H_0^{time} for scenario Remove Tracing. Therefore, we cannot accept alternative hypothesis H_a^{time} for these scenarios.

We can reject H_0^{error} for scenario Changing Parameters. We can thus accept alternative hypothesis H_a^{error} for this scenario.

We cannot reject H_0^{error} for scenarios Adding Tracing, Remove Tracing, Selectively Tracing and Remove Function. Therefore, we cannot accept the alternative hypothesis H_a^{error} for these scenarios.

Concluding, the results from the experiment showed with statistical significance of 95% that adding tracing to a function and selectively tracing (only one parameter) takes less time to implement with the use of *WeaveC*. The results also showed that changing the parameters of a function reduces the severity of errors substantially with the use of *WeaveC*.

3.4 Validation

There are many possible threats to the execution and validity of the results of the experiment. In this section we present some of the threats which could have impacted the experiment, discuss our counter measures and the possible impact of these threats. Although, we use an informal question answer style, all questions can be mapped to the categorization proposed in [CC79]. These categories are: construct, internal and external validity and reliability. We used the guidelines proposed in [KPP+02] to cover all validity threats.

You combined the results from both groups, is this valid? Table 3.10 shows the descriptives of the subjects in both groups.

Group	Characteristic	N	Min	Max	Mean	Std. Dev
1	Age	12	26	45	35.42	5.66
	Education	12	2	3	2.50	0.52
	Years SW	11	1.0	20.0	9.00	5.80
	Years ASML	12	0	7.0	4.21	2.71
	Clevel	12	1	5	3.50	1.09
	THXA	12	0	1	0.58	0.52
2	Age	5	26	43	33.60	6.35
	Education	5	2	3	2.20	0 .45
	Years SW	5	1.0	17.0	9.60	5.86
	Years ASML	5	0	5.0	1.46	2.03
	Clevel	5	4	5	4.20	0.45
	THXA	5	0	1	0.60	0.55

Tabel 3.10: Groups

From table 3.10 we can observe the following:

- Subjects in group one had (on average) worked longer at ASML.
- Subjects in group two had (on average) more experience with C.

We do not feel that this impacts the results of the experiment very much, especially since we did not find any significant correlation between these observations and the performance of the subjects. Also, intuitively these two differences may compensate each other. As one can see there were more subjects in group 1 than in group 2, there are two reasons for this. Firstly, three subjects we excluded were in group 2. Secondly, two subjects accidentally received the instructions of group 1 in the without *WeaveC* session. We chose to move them to group 1 once we discovered this.

Could the separation over two sessions have influenced your results? Similar to the previous question, we also verified whether there we no major

differences between the subjects participating in the morning session and in the afternoon session. We observed the following:

- Subjects in the morning session had (on average) worked longer at ASML.
- Subjects in the morning session had (on average) worked longer in Software Engineering.

We believe that this does not impact the results of the experiment a lot, as we did not find any significant correlation between this time of day variable and the performance of the subjects. More importantly, we take the composed results of the both sessions which eliminates this threat. The subject executed the same treatments and we did not measure the difference between the sessions.

Were the subjects representative? We had no control over the selection of the subjects. Therefore, we could not make a representative selection from the population. Also, we do not have any statistics about the characteristics of the population, therefore we cannot extrapolate our results to a specific population. However, we believe that the characteristics of the subjects are not ASML or industry-specific and, as such, can serve as an indicator for the whole software industry. However, we cannot support this claim with statistical evidence.

Could the subject's knowledge about the source code have any impact? We used the largest component(CC) as a source for objects. We cannot ensure the subjects were not familiar with the selected functions. We tried to minize the this threat by selecting functions from different subcomponents. Also, there are 600 developers developing and maintaining 15 MLOC, the change that all subjects were familiar with this component was slim, since there were also new developers. We do not think that the impact of the threat would be that severe, since tracing is a general concern and not component specific. Using artificial examples would have introduced more validity threats in our opinion.

Why are not all scenarios significant? There may be several reasons why we did not get significance for some scenarios. One of these reasons can be that the sample size is too small. Another can be that the expected benefits may not exist in those scenarios, although most scenarios indicate that there are benefits.

Were the without and with WeaveC treatments equivalent?

1. **Adding Tracing:** In one scenario the subjects had to trace four input parameters and one output parameter. In the other scenario the subjects had to trace two input parameters and two output parameters. There is a difference in complexity, and we also saw indications in the results that this might have impacted the results. Group 1, which first executed the hard scenario without *WeaveC* and then the simple scenario with *WeaveC*,

benefited more from *WeaveC* than group 2. Group 2 executed the simple scenario first and then the hard scenario, and thus benefited less from *WeaveC*. However, we are unable to establish the precise impact of this difference.

2. **Changing parameters of a function:** In one scenario, the subjects had to change one parameter from an input `int` to an input and output `int`. The other scenario had a similar change but this required changing two parameters from input to input and output. There is a small complexity difference here that may have impacted the experiment. However, we saw no evidence of this in the results.
3. **Removing Tracing:** Removing trace statements from the functions in the scenarios has the same complexity in both scenarios. Therefore, we do not expect an impact on the results.
4. **Selectively Tracing:** Only tracing one specific parameter is equally complex in both scenarios. Therefore, we do not expect an impact on the results.
5. **Removing Function:** Removing a function is equally complex in both scenarios.

Did you exclude any subjects? We excluded three subjects from the results. The check-in and check-out time stamps for these subjects were not valid. This was either caused by a bug in the monitor tooling, or because the subjects did not check-in the files.

Did you exclude any data points? We removed 8 individual scenarios from the session without *WeaveC*, and 6 individual scenarios from the session with *WeaveC*. These were all statistical outliers related to the development effort. These statistical outliers were determined using SPSS [SPS]. We removed the individual scenarios, and after wards we verified that there were no more statistical outliers related to the errors. This was indeed the case.

Was there an impact of the tooling on the results? We had to ensure that the build performance did not influence the experiment results too much, i.e. approaching the scenario times or an order of magnitude slower than a regular build. We used the command logger to determine the build times. The results are presented in table 3.11.

#C builds	Average C build time	#AOP builds	Average AOP build time	Δ
65	24.71	88	51.82	210%

Tabel 3.11: Build times

From the table we can observe that the average build time is roughly doubled. However, as this is a realistic delay which we expect will only decrease as the tooling is improved, we did include the build times in our experiment results.

Why didn't you inquire about prior knowledge about AOP? We did not introduce a variable representing the knowledge on AOP, since the subjects were base code developers and had no or minimal prior knowledge on *WeaveC* and aspect tracing. More importantly, prior knowledge about AOP does not affect the results of the experiment, since we only execute scenarios related to the base code, and not scenarios related to the aspect code.

Did you address the initial start up effects of the subjects? To prevent an initial start up time from influencing our results, we added a scenario (Remove Function) as the first scenario. Any initial start up effects should be captured by this first scenario.

Was there a learning effect between scenarios? We used clustered randomness to prevent learning effects between scenarios. This ensured that even if there was insufficient time for the participants to finish all scenarios, we would have results from the two most important scenarios. The order in which the scenarios were executed in each session was randomly assigned.

Was there a learning effect between sessions? We introduced scenario Remove Function not only to determine the initial start up effects, but also to determine whether there was a learning effect between the without *WeaveC* session and with *WeaveC* session. The actions required for executing this scenario are the same in both sessions. However, we are unable to state whether there was a learning curve as there are no significant results for this scenario. Also there is no clear separation between the times of the initial start up effects and the learning curve.

How did you ensure that the users were motivated? As the experiment was part of a voluntary training, we can assume that most subjects were motivated to learn *WeaveC*. To further ensure the motivation, we promised the best 10 subjects a small gadget (around 10 euro). We explained to the subjects that the definition of "the best" was a function of both time and errors. This should have prevented the users from rushing the scenarios.

How did you ensure that all subjects had a similar notion of tracing? At the start of the course we presented the current tracing concern to all users. The subjects were all present during the same presentation, to remove the possible influence of different instructors. Similarly, we presented to the subjects the concepts of AOP and the new way of working, before the second session. Again the subjects attended the same presentation, presented by the same instructor.

How did you ensure that the users were familiar with the tooling and process? As part of the training, we included a tool experience session before the session with *WeaveC*. In this session, the subjects had to go through all the steps of the new way of working and use all features of the weaver. Therefore, all subjects were familiar with the tooling and process before they started with the *WeaveC* session.

Did you satisfy the requirements for the T paired sample test? The T paired sample test requires normally distributed results. We verified the normality of effort and severity of errors using the Kolmogorov-Smirnov statistics and histograms produced by SPSS [SPS]. Almost all scenarios were normally distributed. Five scenarios were reasonably normally distributed, probably as a result of the low number of measurements and the variation between these measurements. These scenarios related to the effort were Remove Tracing w/o WeaveC and Selectively Tracing with WeaveC. Regarding the severity of errors these scenarios were Add Tracing w/o WeaveC, Change Function Parameter w/o WeaveC and Selectively Tracing with WeaveC.

Is the raw data available? Due to confidentiality reasons we cannot publish the raw results of the experiment. We can be contacted to provide these results based on a nondisclosure agreement or a similar construction.

3.5 Survey

A survey was conducted three months after the training and experiment. The purpose of this survey was to determine the current usage and acceptance of *WeaveC*, as well as to determine any issues users faced.

Out of 50-60 current users of *WeaveC*, 26 users responded. In this group were 8 persons who also attended the training and experiment. The survey consisted of several questions about the current acceptance of the tooling and usage of *WeaveC* within ASML. The users were also questioned about the usage, any issues and benefits of *WeaveC*.

The respondents stated that they would like to have more training or documentation about the usage of annotations. They also proposed several new annotations. This strengthens the assumption that the negative development effort while removing tracing, is caused by the newly introduced annotations. Similarly, for changing parameters where the subjects also still introduced errors. Both scenarios use annotations.

The users also stated that they would like to see more (complex) aspects like

Profiling and especially *Error Propagation*. This implies that the users have confidence in *WeaveC* and want to use it for more aspects than just *Tracing*. The following benefits of *WeaveC* were expressed by the 8 respondents that were also subjects in the experiment:

- Better handling of crosscutting concerns.
- Lead time reduction.
- Effort reduction.
- Less boring work.
- Better quality: less errors and cleaner code.

The results from the survey confirm our results from the experiment. Overall the users experience the benefits of *WeaveC*. They also mention the need for appropriate and more elaborate documentation and training for those elements of *WeaveC* which are new to them, especially annotations.

3.6 Related Work

As mentioned in the introduction there are numerous papers about the importance of industrial controlled experiments and success stories, e.g. [WHM07], [GSE+07] and [BCMS03]. However, papers about controlled experiments with AOP are scarce.

In [SHH+05], Sjoberg et. al. present a survey of controlled experiments in software engineering. The authors calculated that only 1.9% of journal and conference articles in a representative set of journals and conferences reported on controlled experiments in software engineering. Only 0.2% of the papers reported to use professionals for the experiments. Similarly, 0.3% reported on experiments which measured the effects of changing the code. This indicates that empirical validation of software engineering methodologies which improve development effort is rare. In the area of AOP this is even more rare.

There are some success stories about the industrial adoption of AOP. In [BF06], Bodkin et. al. report an experience where the authors applied aspects to provide feedback on user behavior, system errors, and to provide a robust solution for a widely deployed diagnostic technology for DaimlerChrysler. Aspects are used as a reflective means to gather information about the system. Although the authors present a discussion about benefits and challenges, they do not provide a quantitative assessment of the benefits of AOP.

In [CC04], Colyer and Clement discuss large scale AOSD for middleware. The authors report on a case-study they conducted at IBM. They present a set of

challenges they faced while executing the case-study. One of the aspects tackled in the case study is also *Tracing*. This supports our statement that even a "simple" concern like *Tracing* can be an excellent driver to adopt AOP. The conclusion of the authors is that AOP can be used successfully on a large scale. They do not provide a quantitative assessment of the benefits of AOP.

In [MKL97], Mendhekar, Kizales and Lamping presented the results of a case-study that compared the implementations of an image processing system. The authors compared the runtime performance of a naïve OO implementation, an optimized OO implementation and an AOP implementation. The authors showed that the runtime performance of the AOP solution was comparable to the performance of the optimized OO version. The performance of the naïve OO implementation was much slower. The AOP solution required 88% less lines of code (including the weaver). They only considered the runtime performance and not the development time and effort in a controlled manner.

In [LB05], Lopes et. al. study the evolution of the design of a web services application. The authors observe the effects of applying aspect-oriented modularization. In this study the authors compare the Net Options Value (NOV) of a design without AOP with the NOV of an aspect-oriented design. NOV has been developed by Baldwin and Clark [BC00], and can be used to evaluate modular design structures. NOV states that the outcome of a design is unknown and models this uncertainty. NOV assumes the expected value of a module to be a normally distributed random variable. It includes the costs of making a design based on the complexity of individual modules and its dependencies. The study by Lopes et. al. shows that after applying aspect-oriented modularization the NOV increased significantly, thus indicating a better design. The work supports our conclusions. However, it uses different dependant variables from our work and it is not a controlled experiment, it does support our conclusions.

In several papers (e.g. [CSF+06, GSF+05]) Garcia et. al. present studies that assess the benefits of AOP for design patterns. In [GSF+05], the authors present a quantitive study for verifying whether AOP improves the modularization of crosscutting concerns with respect to design patterns. The paper compares object-oriented implementations of the 23 Gang-Of-Four patterns [GHJV95], to the same patterns implemented using aspect-oriented approaches. The authors measured several attributes of both implementations, namely coupling, cohesion, size and separation of concerns. These four attributes were derived from a set of metrics, like Depth Inheritance Tree, Lines of Code, Number of Attributes, Concern diffusion over Components, etc. Based on these metrics, for each design pattern in an object-oriented and an aspect-oriented implementation, the authors concluded that for most of the design patterns the aspect-oriented

implementations improve the separation of concerns. Some patterns result in higher coupling, more complex operations and more lines of code.

In [CSF+06], the authors verify whether AOP improves the ability to compose design patterns. The paper presents a study on how aspect-oriented programming handles the modularization of pattern-specific concerns in the presence of pattern interactions. The authors use four software attributes to assess object-oriented and aspect-oriented implementations: separation of concerns, coupling, cohesion and conciseness. The results from the study show that the benefits of aspectizing patterns depend on the patterns in question, the interaction between these patterns and application requirements. As such no firm conclusion can be drawn about applying aspects to patterns in general for all compositions of patterns.

All of the above related work is based on collecting code metrics rather than measuring the performance of software developers. The next two paragraphs discuss related work based on the performance of software developers.

In [GBF+07], Greenwood et. al. present the setup and results of an empirical study to assess the impact of aspectual decompositions on design stability. The study compares the design stability between object-oriented and aspect-oriented implementations, while executing some software maintenance tasks. The authors analyze the execution of these tasks in terms of modularity, change propagation, concern interaction, identification of ripple-effects and adherence to well-known design principles. The authors present two categories of results. The first is where aspectual decomposition did improve design stability and a second where aspectual decomposition did not improve design stability. In the first category, the authors state that aspectized concerns tend to show superior modularity design, require less intrusive modification. Also, both object-oriented and aspect-oriented implementations exhibited stability of high-level design structures. In the second category the authors observe significant violation of pivotal design principles, such as narrow interfaces and low coupling, and aspectizing exception handling showed no improvements.

Walker, Murphy and Baniassad [WBM99, MWB99, WBM05] present an initial assessment of AOP. Here the authors present several experiments. One experiment to determine whether users of AOP were able to more quickly and easily find and correct faults in a multi-threaded program. The results of this experiment showed that AspectJ users were indeed faster for one scenario, while for two other scenarios the difference was smaller compared to the regular Java developers. The intention of the second experiment was to investigate whether the separation of concerns provided in AOP enhances the ability to change the functionality of a multi-threaded, distributed program. The results of this experiment showed that the users of AOP(Cool, Ridl and JCore) were slower than

the users who manually changed the program using Emerald. There are several differences between our experiment and the one described by the authors. The main difference is that the authors conducted semi-controlled empirical studies rather than "statistically valid experiments" [MWB99]. Other differences are the smaller set of subjects in the studies and that the subjects are not professionals, but students and professors. However the results from these experiments confirm our findings.

3.7 Generalizability of the experiment

In this chapter we have presented and discussed the design and results of our experiment. In this section we provide a discussion about the generalizability of the design of experiment and the experiment results. We can generalize the design and results of the experiment in several different ways: for other concerns, aspect and base languages and organizations.

3.7.1 Other concerns

Although tracing is a (in)famous example of a crosscutting concern, there are many other concerns. Examples have been mentioned in the first chapter, like error propagation and parameter checking.

Design: The design can be reused to a large extent for other concerns, only the change scenarios would require adoption.

Results: The results of the experiment are hard to generalize, since the complexity of aspects can differ significantly. However, concerns with a similar dependency on the signature of a function, like for example parameter checking, may show similar results.

3.7.2 Other aspect languages

In our experiment we used *WeaveC* and its language called *Mirjam*. How would our experiment design and results be effected, if there were other language and tools capable of expressing tracing, as defined in this chapter?

Design: The design can remain the same, only the scenarios need to be adjusted, since the current set of change scenarios uses *WeaveC*-specific annotations.

Results: Since we only assess the benefits of aspect-orientation for base code developers, we would expect similar results. Comparing different AOP implementations is out of the scope of this work.

3.7.3 Other base languages

The tracing concern is not specific for C, as illustrated by section 2.3.3, where concern tracing was implemented in C#. Other base languages like Java or PHP could also be used.

Design: Again, we can reuse the design of the experiment to a large extent. However, the change scenarios will differ in their exact implementation.

Results: We would expect similar results, since crosscutting concerns are not addressed differently in most languages.

3.7.4 Other organizations

The experiment was conducted with 20 ASML developers. Since crosscutting concerns are a widespread problem, we discuss the impact of generalizing to other companies.

Design: Again the design would remain the same, but only the scenarios may differ.

Results: If we assume that the same base language, weaver and aspect are used, then the reusability of the results would depend on the independent variables of the specific company. Most companies in large scale industries will use bachelor and master student. As such the results should be in line with our findings.

To summarize, the design of the experiment can be reused for other experiments. The results are not necessary generalizable to all aspects for all companies.

3.8 Conclusions

The experiment we report on in this chapter has adopted the *Tracing* aspect, explained in section 3.1. Tracing is sometimes dismissed as a trivial aspect, however, we showed that this is not the case in the actual realization of Tracing at ASML. Additionally, this "simple" aspect can serve as an excellent driver to adopt AOP, as also mentioned by Colyer et. al.. [CC04]. Aspects can be used

to address a wide range of crosscutting concerns, but the benefits of adopting AOP are not as clear for heterogeneous or more complex concerns, as with homogeneous concerns like *Tracing* and *Profiling*. As such these more "simple" aspects should not be overlooked, but rather be embraced as effective examples to introduce AOP in industry.

One of the contributions of this chapter is the design of a controlled experiment that can be used to quantify the benefits of AOP in other organizations. The design can easily be adopted for other aspects, other base code, and other scenarios. We believe that the proposed integration with an introductory course is an elegant solution to reduce the reluctance of a development organization for spending time on an experiment.

We conducted the experiment with 20 ASML developers. The developers had to execute five simple tracing-related change scenarios twice. First, the developers had to implement the scenarios manually. After this the subjects had to implement the scenarios using *WeaveC*. We offer a detailed discussion of validity threats in section 3.4 and we have tried to reduce validity threats through careful design of the experiment (see section 3.2).

The results from this experiment, presented in section 3.3, showed that, overall, the subjects were able to execute the scenarios 6% faster using the AOP solution. There were scenarios where the subjects were slower with AOP. Since the actual amount of work was much less, we *believe* that this is caused by the fact that the subjects were new to the tooling, especially the required annotations. More prominent however, was the reduction in errors when using the AOP solution by 77%. We can safely conclude that in the experiment case, a substantial reduction of errors was achieved with even slightly less effort.

The statistical results of the experiment are limited, as —probably due to the limited number of participants— not all executed treatments give results that are statistically significant (at a 95% confidence interval). For two common change scenarios there is a statistically significant benefit in terms of development effort w.r.t. *Tracing*. There is a statistically significant severity error reduction for one scenario.

The profile of the subjects in terms of education and software development experience (see section 3.3.1) seems representative for ASML and (medium- to large-sized) software development organizations, but we had no comparable profile information about these populations to substantiate such a conclusion.

A survey was conducted amongst 26 users of *WeaveC*. Firstly, the respondents expressed the need for more complex aspects, like errors handling. This indicates that the respondents were confident in *WeaveC*. Secondly, the respondents

expressed the need for more explanation about annotations, as this was a new concept. This strengthens our belief that there is still some extra gain in development effort, which we were unable to measure.

Concluding, the survey confirms that the users do "experience" the benefits of AOP.

The current set of subjects is limited. We hope to conduct more experiments in the future to validate our results and achieve —presumably— statistically more significant results.

The contributions of this chapter are:

- A detailed description of a real-world aspect: *Tracing*. Although this is usually considered simple, we show that this aspect is complex (Section 3.1).
- A design of a controlled experiment that can be used to quantify the benefits of using an aspect-based approach to *Tracing* in an industrial setting (Section 3.2).
- The results of a controlled experiment with 20 professional software developers, using an aspect-based approach to *Tracing* (Section 3.3).

Behavioral Conflicts among Aspects

4

In this chapter we discuss the problem of behavioral conflicts among aspects. We present an example conflict based on an industrial case study. We also show examples of other behavioral conflicts. We use a problem space of composition conflicts to discuss the related work and to show which kind of composition conflicts we address. Next, we present our approach, which is based on creating abstractions of the behavior of aspects as resources and operations. Next, we apply our approach to the above mentioned example conflicts. Finally, we discuss the generality of the approach and conclude. In this chapter we only discuss the problem of behavioral conflicts and present an approach for detecting these behavioral conflicts. In chapter 5, we show in detail how this can be used in a concrete AOP language: Composition Filters.

4.1 Motivation

In this section we present an example application, which serves two purposes: first, it should define of conflicts we address in our approach. Second, since the example has been identified in the context of a large industrial application, it is intended to motivate the relevance of the problem. The example has been

identified within the Ideals project [IDE]. The presented example is taken from this project. We present two aspects[1] that we have identified, namely *Parameter Checking* and *Error Propagation* (section 1.1).

4.1.1 Parameter Checking

The Design by Contract [JM97] approach to software development is based on the principle that the interface between modules of a software system should be bound by precise specifications. These specifications can be pre-conditions, post-conditions and invariants. One application of design by contract is to check whether the parameters of a method or function are valid. ASML adopts this in its wafer scanner software to ensure the validity of the parameters. We call this concern *Parameter Checking*. Parameters can be one of three types: input, output and in- and output. This distinction depends on whether a parameter is read, written or both. ASML employs two checks to verify the validity of the parameters, and thus the contract. First, at the start of a function, function input and in- and output pointer parameters should not be empty (i.e. not null) . If the input parameter pointer is null, it could yield a fatal error whenever this parameter is accessed. Second, every output pointer parameter must be null at the start of a function. An output parameter is a parameter that is written in the function body. If such a parameter points to a memory location that is already in use, data in this location can be accidentally overridden, which is undesirable. An example of this concern, as currently implemented by ASML, applied to the function compare_data() is shown in listing 4.1.

```
1  static int compare_data(
2      const DATA_struct*                  p1, /* input */
3      const DATA_struct*                  p2, /* input */
4      bool*                               changed_ptr) /* output */
5  {
6    int result = OK;
7
8    /* Check preconditions */
9    if (p1 == NULL)
10   {
11     result = INVALID_INPUT_PARAMETER_ERROR;
12   }
13   if (p2 == NULL)
14   {
15     result = INVALID_INPUT_PARAMETER_ERROR;
16   }
17   if (changed_ptr != NULL)
18   {
19     result = INVALID_OUTPUT_PARAMETER_ERROR;
20   }
```

[1] The example aspects presented here are slightly altered for reasons of confidentiality. However, this does not affect the essence of the examples.

```
21
22    // code that compares the structures and sets the changed_ptr boolean accordingly
23
24    return result;
25  }
```

Listing 4.1: Example of the Parameter Checking code

The function compares the two input parameters p1 and p2 (declared in lines 2 and 3), and sets the changed_ptr boolean output parameter accordingly (line 22, not shown in detail here). At lines 9 to 20, the checks for the input and output parameters are shown. Typically, the parameter checking concern accounts for around 7% of the number of statements in the code, although the exact percentage varies among components.

4.1.2 Error Propagation

The C programming language does not offer a native exception handling mechanism. The typical way to implement exception handling in C, is to use the return value of a function. The function returns 'OK' in case of success and an error number in case of failure. The identifier 'OK' has been defined in a macro and resolves to integer value 0. This means that the caller of the function should always get the return value and verify that the value is still OK, otherwise it should either handle the error or return the error to its caller.

Exception propagation at ASML consists of (a) passing the error state through a so-called *errorvariable* and as the return value of the function, (b) ensuring that no other actions are performed in an error state, and (c) if an error state is detected, it is logged. Listing 4.2 shows an example of the exception handling scheme employed at ASML.

```
1  static int compare_data(
2      const DATA_struct*                  p1,
3      const DATA_struct*                  p2,
4      bool*                               changed_ptr)
5  {
6    int result = OK;
7
8    if (result == OK)
9    {
10     result = example_action1(...);
11     if(result != OK)
12     {
13       LogError(result);
14     }
15   }
16
17   return result;
18 }
```

Listing 4.2: Example of the Error Propagation code

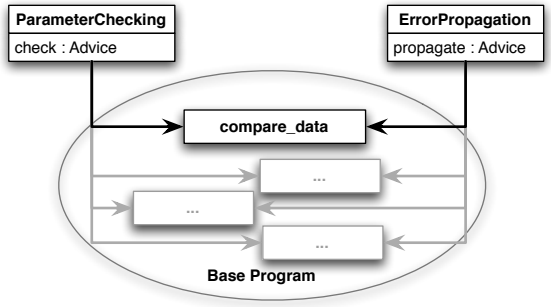
The code in listing 4.2, first (line 6) initializes a variable, result, to hold the current error state, which is referred to as the *errorvariable*. To determine whether to continue with normal execution, a check is placed which guards the execution (line 8). In this case this might seem useless as the *errorvariable* already contains OK. The reason it is inserted is that as the code evolves, such a check might be required, if another statement is inserted before this. Next, a call to a regular function (*example_action*1(...)) is done (line 10). If an error is detected, this error is logged (lines 11 to 14). Finally, the *errorvariable*, here named result, is returned at line 17.

It is out of the scope of this thesis to elaborate on the alternatives for exception handling, especially since this was a given situation at ASML, and changing the standard completely at once is not feasible. However, the integer return value exception handling contributes substantially to the lines of code, in some extreme cases even up to 25% of the code, depending on the component. Error handling can be divided into three main elements: detection, propagation and handling, but we only focus here on the propagation part. Detection and handling of errors is highly context dependent at ASML, thus refactoring this into an aspect is hard, and perhaps not even desirable. In [FGR07], Filho et. al. discuss provide situations where modularizing exception detection and handling into aspects seems beneficial or harmful. In this section we purely focus on error propagation, since it follows a more common pattern which can be refactored into an aspect more easily.

4.1.3 An aspect-based design

We will now discuss how to refactor the crosscutting concerns above into an AOP solution. Concern *ParameterChecking* should check the input and output pointer parameters of each function to ensure the contract of the function is not violated. We implement this functionality as an advice, named *check*. Concern *ErrorPropagation* should check whether the system is not in an erroneous state (i.e. the *errorvariable* is equal to zero), if the *errorvariable* is zero it should execute the original call. If this call yields an error, we should log this. Similar to the *ParameterChecking* concern, we also implement the functionality of error propagation as an advice, named *propagate*. If we apply both concerns to a base system, the resulting system is shown in figure 4.1.

At the top of the picture, the two concerns ParameterChecking and ErrorPro-



Figuur 4.1: Parameter Checking and Error Propagation example

pagation and their advices are presented, namely check and propagate. The figure also shows our example C function, *compare_ data(. . .)*. This is one of the functions that form the base system. The arrows show where each advice is applied. The advices are superimposed on the same join point, in this case *compare_ data(. . .)*. However, concerns ParameterChecking and ErrorPropagation implement coding conventions and are applied to all functions in the system, as such there are many such shared join points. These are indicated by the gray arrows and rectangles. Advices at shared join points have to be executed in some order. We assume that advice *propagate* is applied before *check*; in this case, the errors detected by *check* are never propagated to the caller. The alternative ordering is not problematic.

If we examine the conflict more carefully we see that the conflict is caused by a dependency between the two advices. The *propagate* advice reads the errorvariable to determine the current error state and can subsequently write the errorvariable and log the error if an error is detected. Advice *check* verifies that the arguments are valid, and possibly writes the errorvariable. In this case the presence of the conflict depends on a specific ordering of advice. Later in this chapter, we present some examples where the ordering does not matter.

4.2 Problem Statement

We will now elaborate more on concerns ParameterChecking and ErrorPropagation and the conflict between them. Individually, both aspects are consistent with their requirements and therefore they can be considered sound. From the

language compiler point of view, the program with either orderings of advices can be considered as a valid program with no errors, there are no syntactical or structural problems, and both orderings can be executed. However, once these aspects are applied at the same join point, new behavior emerges. Such new behavior may be undesirable, in which case we call it a behavioral conflict.

We use the terms *aspect* and *advice* inter-changeability in the context of behavioral conflicts. Although a conflict is caused by interference between the behavior of advices, since we consider only conflicts at shared join point, we have to include the pointcut designator as well, and as such the complete aspect.

In the ASML example, if a developer is aware of such (potentially) conflicting case and the order in which aspects are applied matters, an ordering can be enforced. For example, it is possible to enforce an ordering in AspectJ [Asp], using the declare precedence construct. However, the ordering constraint models in some AOP languages may not be sufficiently expressive to specify the intended ordering. For example, in the case of AspectJ one can only indicate a precedence between aspects not between advices in different aspects. AspectJ does offer precedence between advices within the same aspect, using the declaration order. Composition Filters offers ordering language at the granularity of filter modules.

In practice, detecting emerging conflicts is hard, especially if the conflicting aspects crosscut the entire base application and share many join points. The conflicts are especially hard to detect when the conflicting situation emerges only at specific shared join points. This implies that the conflict presents itself given a specific context, the dependency between the aspects is not direct but rather indirect, through some properties of the join point. These base system-dependent conflicts are not addressed in this thesis, but we focus on conflicts caused solely by the aspects themselves.

As aspects are becoming more and more adopted and widespread, these kinds of emerging behavioral conflicts will become more prominent. Behavioral conflicts are not new to programming languages. The same kinds of problems can occur when using object-oriented or imperative programming languages. However, since aspects are usually independently specified and can impact a large section of the application, the problem is more prominent and harder to detect. It is therefore necessary to develop techniques and tools that reason about (potential) behavioral conflicts between aspects.

4.3 Other examples of Behavioral Conflicts

To further exemplify behavioral conflicts, and demonstrate that these may occur in many different domains and applications, we introduce several other examples. We assume that all these aspects apply to same join point.

Authorization and Persistence : An authorization aspect checks whether an "action" is allowed to be executed for the current user. A persistence aspect writes data to some persistent structure. A conflicting situation arises if we first write the required data to the persistent structure, before checking whether the user was allowed to perform this action.

Authorization and Authentication : An authentication aspect verifies the identify of the user. Here a conflicting situation arises, if the authorization aspect is executed before the authentication aspect, as the identify of the user has not yet been verified.

Compression and Logging : A data compression aspect can decrease the amount of traffic on a communication protocol. If this aspect is composed with an aspect which logs all traffic, we have to ensure that the logging advice is executed before compressing the data, else the log file will be unreadable.

Encryption and Tracing : A comparable situation to the previous one presents itself for encryption and tracing aspects. However, in this case the correct order of the advices depends on application specific requirements. If the application resides in a hostile environment, we would like to ensure that all data is encrypted before the tracing aspect reads this data, to ensure the safety of the data in the log file. However, if the application operates in a safe environment and we would like to verify the data, we would like to see the plain text data and not the encrypted data. This conflict is an example of an application-specific, or even deployment-specific conflict. In these cases, the correct ordering of advice cannot be determined automatically.

Data modification : Imagine two aspects at a shared join point which both write the same field or table of a datastore. We assume that this is a conflict as the value written by one aspect is overwritten by another aspect. In this case there is no (in)valid ordering, since in any ordering a value is overwritten. If one aspect would do a non-destructive update, which first reads the value and then writes an updated value[2], a valid ordering is possible (i.e. *update* after *write*).

Real-time constraints and Concurrency : Assume that a real time aspect

[2]In general, an update can be considered nondestructive if the update is reversible because no information is lost.

enforces some timing constraints on a certain action. If we now also apply an aspect which enforces a synchronization constraint on that same action, we can never ensure that the deadline is met, because of potential blocking. This problem is again a conflict that cannot be solved by reordering the aspects, unlike the first four example conflicts above.

4.4 Background and Related Work

In this section we discuss a part of the problem space of conflicts caused by the composition of aspects, and position our approach into this space. We discuss the problem space along the following dimensions:

1. Composition type (advice-base or advice-advice)
2. Type of superimposition (structural or behavioral)
3. Type of interaction (control-flow based or state-based)
4. Type of join point (shared or distinct)
5. Ordering (order-dependent or order-independent conflicts)
6. Generality (generic, domain-specific or application-specific conflicts)
7. Advice specification form (imperative vs. declarative, Turing-completeness)

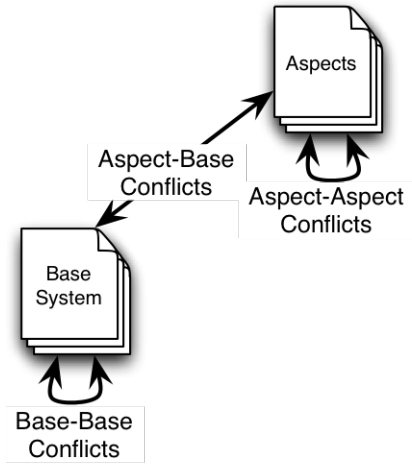
These dimensions are not fully orthogonal, but in the presented order they allow us to scope the context and to indicate the category of problems our approach focuses on.

4.4.1 Composition type

One can identify three types of composition between aspects and base code, as illustrated by figure 4.2.

We can distinguish three types of composition:

- *Base-base composition*: the composition of the behavior of two elements in the base system.
- *Advice-base composition*: the composition of the advice behavior with the behavior of the base system.
- *Advice-advice composition*: the composition of the advice behavior with one or more other advice behaviors; this happens when several advices are superimposed at locations where their behavior 'joins'. As mentioned in section 4.2, we divide this type of composition into two categories:
  - *Advice-advice*: In this case a conflict occurs due to the composition of



Figuur 4.2: Aspect composition conflicts

the aspects themselves. As such this conflict will occur at any shared join point.

– *Advice-advice-base*: In this case the conflicts occurs due to the composition of the aspects, but only for specific join points.

Base-base composition, e.g. using inheritance or method call, is usually already verified, by for example the type checker of the compiler. This only guarantees that the types are compatible or that a certain method call is valid. To verify the correctness of the behavior after composition, we still need techniques like model checking. Although advice-base composition is a very relevant category, and in principle our approach could be applied here as well, we only focus on interference among aspects: the main motivation for this is that automated reasoning about the behavior of (Turing-complete) base code is notoriously difficult. We consider the annotation of a complete base program with clues about its behavior to be impractical in all but very special situations. As we explain in the next chapter, we believe it is much more feasible to (automatically) derive a sufficiently informative specification of its behavior from a declarative aspect language.

4.4.2 Type of Superimposition

In general, a distinction can be made between *behavioral* and *structural superimposition*: behavioral superimposition refers to the adornment of a program (at join points) with behavior (expressed as advice). Structural superimposition refers to various disciplined forms of transformation of the program, typically through the addition of program elements such as methods and fields. This is also called *introductions* or *inter-type declarations*. One aspect module may consist of a combination of behavioral and structural superimposition.

In our approach, we focus on the possible behavioral conflicts between composed pieces of behavior. Although this applies in principle to any form of behavior composition, to restrict the scope of the discussion we focus on behavioral superimposition, which is also the most common of the two types of superimposition.

4.4.3 Type of interaction

Conflicts between aspects are caused by some form of interaction. In general, we can distinguish between *control-flow* based interaction and *state* based interaction. Control-flow based interaction means that the composition affects the flow of execution in the program. State-based interaction means that the composition brings the program into a state that would not occur (at that location) in the program without that particular composition. It should be stressed that an interaction is not necessarily bad, or should be considered a conflict; many interactions are in fact desirable, and compositions may be specified exactly to achieve such an effect on the control flow or state. One of the goals of our approach is to present a means to distinguish among desired interactions and undesired interactions (i.e. conflicts).

In [CL02], Leavens and Clifton propose a classification of advice based on whether or not the advice changes the specification of the base behavior where the advice is superimposed. So called *observers* do not alter this effective specification, whereas *Assistants* do.

In [Kat93, KG99], Katz et.al. propose three categories of aspects: *spectative*, *regulative* and *invasive*. Spectative aspects do not influence the underlying system, but only query the state. Regulative aspects can alter the control flow of the underlying system. Finally, invasive aspects change the state of the underlying system and may alter the control flow. In [GK06], a further distinction is made between *weakly invasive* and *strongly invasive*. A weakly

invasive advice always returns to a valid state of the base system, whereas this is not the case for a strongly invasive advice.

Rinard et.al. [RSB04] have proposed a classification system for the possible effects of advice on base code. The classification system distinguishes the dimension of control flow and the dimension of state. For the dimension of control flow, the following categories are distinguished:

Augmentation : the join point is always executed.
Narrowing : the join point is conditionally executed.
Replacement : the join point is not executed.
Combination : a combination of the base system and the aspect is executed.

For the dimension of state the following types of interaction are distinguished:

Orthogonal : the aspect and base system access disjoint fields.
Independent : neither the aspect nor the base system writes a field that the other may read or write.
Observation : the aspect may read one or more fields that the base system may write, but they are otherwise independent.
Actuation : the aspect may write one or more fields that the base system may read, but they are otherwise independent.
Interference : the aspect and base system may write the same field.

In our approach we focus on both control flow and data flow related conflicts.

4.4.4 Type of Join Point

We mentioned that advice-advice composition may yield conflicts when the advices 'join'. This only occurs when there is some direct influence or interaction between advices. This depends on the relative locations where the respective advices are superimposed; these join points can be either:

- *shared join points*:Multiple advices are applied to the same join point, so they may interact.
- *distinct join points*: Advices are applied at distinct join points, they may still influence each other, albeit less likely. For example, assume that a first advice affects a state that another advice depends on (or changes as well). If the second advice is within the control flow or data flow of the first advice, the changed state may very well affect the second advice.

In our approach we focus on advice-advice composition at shared join points. The primary reason for this restriction is that detecting the mutual influence

of advices at distinct join points requires extensive analysis of the base code of programs, something that is not the primary aim of this work. However, our approach could as far as we can envision be applied as well to aspect interaction at distinct join points when such an analysis would be available.

Douence, Fradet and Sudholt [DFS02] state that two aspects do not interact if they are independent of each other. Here, (in)dependence is expressed in terms of having shared join points. They distinguish two forms of independence. The first, *Strong Independence*, occurs when two crosscutting specifications never overlap for any base system, whereas in the second form, *Independence w.r.t. a program*, the independence of the aspects is relative to a given base system.

4.4.5 Ordering

Conflicts caused by advice-advice composition can be divided into two categories:

- *order-dependent*: when the conflict only occurs in a specific ordering of advices.
- *order-independent*: when the conflict is independent of the ordering of the advices.

For advices applied at distinct join points, at least for imperative languages, the ordering cannot be changed without changing the pointcuts or changing the execution flow. In case of conflicts as a result of side-effects, this distinction has only practical relevance for advices applied at shared join points. Many AOP languages have support for ordering advices at shared join points, such as aspect precedence declaration in AspectJ [Xer] and declarative ordering constraints in Compose* [NBA05].

4.4.6 Generality

Most behavioral conflicts are not cases where the execution of composed advice leads directly to execution problems, or is otherwise fundamentally wrong or impossible, i.e. would result in an not compilable or executable program. Typically, behavioral conflicts are detected through the use of *domain knowledge*. The advice composition is problematic because in a specific domain or application context, such a composition does not make sense for that context, since the resulting behavior is semantically incorrect or undesired. For example, the composition may lead to deadlock, incorrect scheduling, invalid application data, advice that is *accidentally* not executed at all, and so forth. We can distinguish the following categories conflicts:

- *Generic*: conflicts that are general to computing and program execution, and hence may occur in any type of program. For example, when two advices write the same variable consecutively, the value that was written by the first advice may be lost, which means the intended behavior of the first advice is not represented in the composed application.
- *Domain-specific*: conflicts that are specific to a particular domain, such as concurrency, persistence, exception handling or security. Each of these domains imposes specific rules about correct compositions. For example, the combination of two advices may cause a wrong synchronization, which can lead to e.g. deadlock or (indirectly) corrupt data.
- *Application-specific*: conflicts that only arise due to specific constraints that can be traced back to the requirements of the application. For example, when an encryption advice is composed with a logging advice, it depends on the application requirements whether the encrypted, or the unencrypted, information should be logged.

The approach presented in this chapter is applicable to all three categories. However, for the domain-specific and especially the application-specific conflicts, the programmer may need to provide specific information, e.g. in the form of annotations or rules, so that the these more specific behavioral conflicts can be identified.

4.4.7 Advice specification form

Finally, we briefly discuss the various forms of advice specification, and their relationships with advice composition conflicts and the detection of such conflicts. First, we can distinguish between advices that are specified in an *imperative* manner and those that are specified in a *declarative* manner. The first group is the common approach in most AOP languages, such as AspectJ, where advice is expressed just like a method body in the base language. The latter group is less common, and most examples appear in domain-specific aspect languages, such as the COOL and RIDLE [Lop97] aspect languages for distributed programming. There are definite advantages for programmers to be able to express advices in the base language they are familiar with, and interfacing between the base language and the advice languages is also straightforward in such cases, for example such as sharing data structures. However, a declarative advice specification may avoid certain categories of conflicts, and in general it is easier to analyze the specification in such a language for the purpose of detecting conflicts.

A second categorization relates to the expressiveness of the advice language. Most advice languages are expressed in an (imperative) Turing-complete langu-

age. Some usually domain-specific advice languages, however, have restricted expressiveness and are not Turning-complete. This is a trade-off for the language designer between expressiveness, and ease of understanding for both programmers and machines (such as conflict detection tools).

Our approach is not restricted to any advice specification form. However, in the next chapter we demonstrate that our approach can leverage the characteristics of a declarative non Turing-complete advice language, to allow for better automated conflict detection.

4.5 Approach

To reason about the behavior of advices and detect behavioral conflicts between them, we introduce a formalization that enables us to express behavior, and conflict detection rules over that behavior. A complete formalization of the behavior of advice in general would be too complicated to reason about. In most case the behavior is expressed using Turing complete languages, which are notoriously hard to reason about, e.g. termination problems and undecidability issues. Techniques like model checking have already encountered these issues and use a variety of techniques to prevent state-space explosion and reduce computational complexity. One of these techniques is the use of abstractions as proposed in our approach.

Imagine an encryption advice which encrypts the arguments of a message. To do this, this advice first reads the arguments, calls an encrypt method to encrypt them, sets the encrypted arguments and calls the original method with the encrypted arguments . Without extra information it is impossible to derive that a read, followed by a call to some method (which does the actual encryption), followed by a write on the arguments, is considered an encrypt operation. At the implementation level this important information is lost. There are many different ways to encrypt data, abstracting from these different ways is important for detecting behavioral conflicts.

Therefore, we propose an abstraction that can represent the essential behavior of advice, such that it can be used to detect behavioral conflicts between advices. At the same time, the abstraction will reduce or simplify details such that undecidability issues are avoided and computational complexity of the reasoning process is reduced.

Our abstraction consists of a *resource-operation* model to abstract the relevant behavior of advice. We have chosen to adopt a resource-operation model, since

this is a simple abstraction model that can represent both concrete, low-level, behavior as well as abstract high-level behavior. The resource-based abstraction is not unlike the idea of Abstract Data Types [KM80]: representing an abstraction through its operations. Our approach to conflict detection can also be viewed as an generalization of the Bernstein conditions [Ber66] for stating concurrency requirements. A similar approach is also used for detecting and resolving (concurrency) conflicts in transaction systems, such as databases [LMWF93]. However, our approach generalizes from these domain-specific approaches.

The most primitive actions on shared data resources are read and write operations. However, if desired by the programmer, we allow such actions to be modeled at a higher level of abstraction, and thus introduce more specific information into the model. These more specific operations can be derived from a specific domain, e.g. the pass and free operations on a semaphore, or can even be application specific. Like operations, resources can be refer to concrete elements in the application, e.g. the target of a message or more abstract elements like a semaphore or thread. Operations in our model are abstractions of a (complex) action executed by an advice. We do not describe the implementation of an operation, just its notion. Since, there may be many implementations that can be mapped to the same operation, e.g. there are many ways to encrypt data.

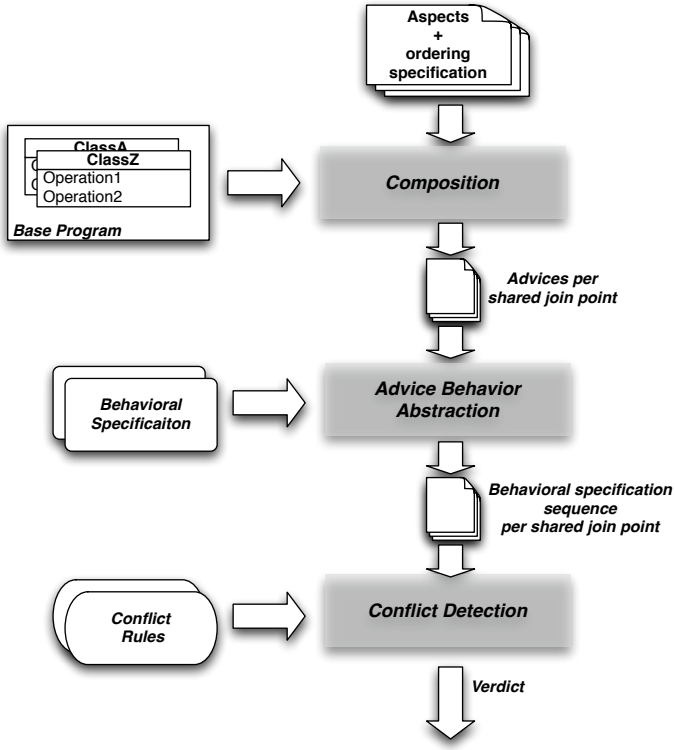
The key idea of our approach is that for a conflict to occur, there must be an interaction. A conflict is an interaction with undesirable consequences. This interaction can be modeled by operations on one or more shared (abstract) resources. A conflict is then modeled as the occurrence of a certain pattern of operations on a shared resource.

We first explain the general outline of our approach. In the next section we exemplify this approach using the ASML example. Figure 4.3 presents the behavior analysis process and its relationships to the program. We will use this image as a guideline throughout sections 4.5.1 to 4.5.3.

Each shaded block in figure 4.3 presents a step in the behavioral conflict analysis process, with its inputs and outputs. The conflict analysis process contains three steps: *Composition*, *Advice Behavior Abstraction* and *Conflict Detection*. Sections 4.5.1 to 4.5.3 each explain one such step, including the specific inputs, how these are transformed, and the resulting outputs.

4.5.1 Composition Phase

Inputs This phase in the process has two kinds of inputs:



Figuur 4.3: An overview of the approach

- **Aspects**: These are shown at the top of figure 4.3. These aspects typically contain:
  - **Advice**: This is the behavior that should be inserted.
  - **Pointcut Designators**: These specify where the behavior should be inserted.
  - **Ordering Specifications**: These allow the developer to influence the ordering of advice at shared join points. These can be complete or partial ordering specifications.
- **Base Program**: These are the classes or files that are subjected to the aspects. The pointcuts are resolved on this base program, yielding a set of join points.

Transformation During this phase all pointcut designators are evaluated with respect to the base program. If a pointcut matches, a weaver typically inserts the behavior that is specified in the advice into the join point. If there are multiple advices, an execution order must be determined, based on the (partial) ordering specifications. Most AOP approaches only support serial execution of advice, hence we only consider this situation. For example, AOP approaches that execute advices in parallel are out of the scope of our approach. Resolving the pointcuts and ordering advice is usually carried out by an aspect oriented weaver.

Outputs The result of this phase is a set of join points with a sequence of advices attached to them. For conflict analysis, we only need to consider join points with more than one superimposed advice. We could also carry out conflict analysis for join points with one advice attached, however we assume that the implementation of a single advice is correct. Our approach can equally be applied to detect conflicts within a single advice.

4.5.2 Advice Behavior Abstraction Phase

Inputs This phase in the process has two kinds of inputs:

- **Advice Execution Sequences**: This is the result of phase *Composition*. This is a set of join points with a sequence of advices attached to them.
- **Behavioral Specifications**: This is the behavioral specification for each advice or filter type. This specification consists of a set of resources with a sequence of operations attached to them. We assume that all operations are executed sequentially, i.e. we do not take conditional or repetitive

execution into account, and assume the worst case.

We now discuss the way resources and operations are specified and the relationship between these two concepts.

Resource : A resource represents a possible interacting area. A resource has two elements:
- A string that represents the name of the resource.
- A set of operations that is the alphabet of operations for this resource. Only operations that are in this alphabet are allowed to be carried out on this resource.

Operation : An operation represents the effect an advice has on a certain resource. An operation can by identified by a name. A possible extension would be to also include arguments, thus providing parametrized operations. For now we ignore this extension. An operation has only one element:
- A string that represents the name of the operation.

Transformation During the abstraction phase, the sequence of advices is transformed into a sequence of operations per resource per shared join point.

Outputs The result of this phase is a sequence of operations per resource per shared join point.

4.5.3 Conflict Detection Phase

Inputs

- **Conflict detection rules** (or conflict rules for short): A pattern that describes for a set of resources what combination of operations are allowed to occur on this resources. There are two types of rules : *conflict* rules and *assertion* rules. A rule consists of three elements:
  - A set of resources for which this rule applies. This may be all resources.
  - An expression describing the conflict pattern. In this thesis, we work mostly with extended regular expressions (IEEE standard 1003.1 [GI04]), but other options like linear temporal logic (LTL) [Pnu77] would be possible as well.
  - A message that is provided to a developer if a conflict rule matches, or if an assertion rules fails to match.
- **Behavioral sequence of operations per resource per shared join point**: The output of the previous phase.

Transformation Each conflict rule is transformed to an automaton. We can do this since we use regular expressions or LTL (see [Sud97]). Also, for each shared join point and for each resource, there is a resulting sequence of operations. For each such sequence we determine whether the automata, that represent the conflict detection rules, accept this sequence. If a conflict rule automaton accepts the sequence, this indicates a conflict. If an assertion rule automaton does not accept the sequence, this also indicates a conflict.

Outputs For each resource and each join point we determine whether the conflict rules match or the assertion rules do not match. If so, we report an error or warning to the user.

4.6 Application to the ASML example

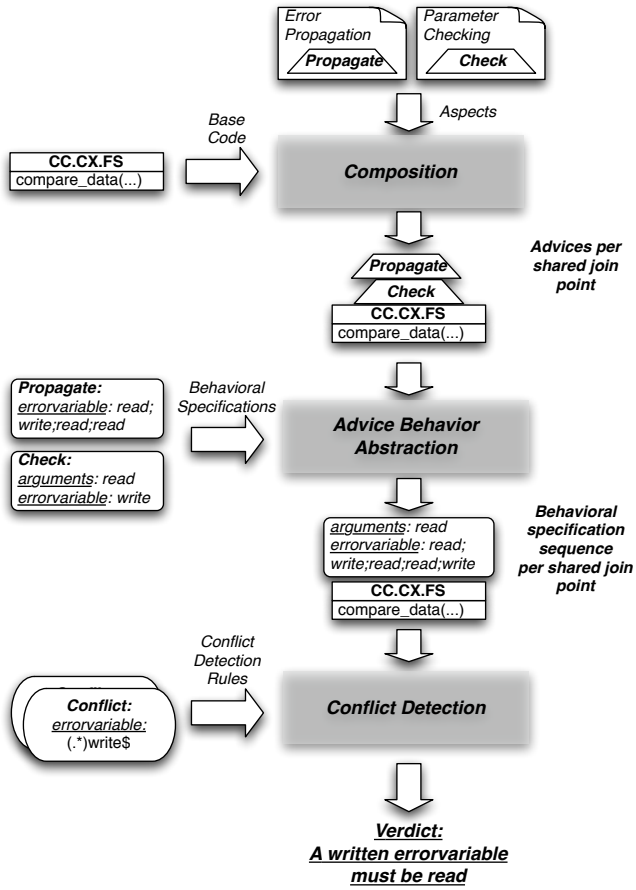
In this section, we illustrate our approach by applying it to the ASML example. We do this by enumerating the three phases of our approach and we discuss the artifacts involved. We have instantiated the approach illustrated in figure 4.3 for the ASML example, as shown by figure 4.4.

Figure 4.4 has the same structure and layout as the overview of the approach in figure 4.3. However, now all artifacts are instantiated for the ASML example. Similar to section 4.5 we discuss each phase of the approach.

4.6.1 Composition Phase

Inputs This phase in the process has two kinds of inputs:

- **Aspects**: In this case we have two aspects: Error Propagation and Parameter Checking. These aspects contain the following elements:
  - **Advice**: Aspect Error Propagation contains advice Propagate. Aspect Parameter Checking contains advice Check.
  - **Pointcut Designators**: Since both aspects implement coding conventions, we assume that both advices are applied to numerous places in the base program. However, aspect Error Propagation is only applied to functions returning an integer, which should be the case for the vast majority of functions.
  - **Ordering Specifications**: In this case we do not provide an ordering specification.



Figuur 4.4: Approach instantiated for the ASML example

- **Base Program**: There are numerous files and functions in the base program that should be considered for aspects Error Propagation and Parameter Checking. Here, we only focus on one function in particular, namely function compare_data(...). The discussion and our approach applies equally well to other functions.

Transformation Function compare_data(...) is a shared join point, as both advice Propagate and Check are applied to this point. Since we have to sequentially compose both advices, an ordering has to be chosen. We assume that advice Propagate is executed before advice Check; this is the order that results in a conflict.

Outputs The result of this phase is one shared join point: function compare_data(...). This join point has a sequence of advices attached to it, this sequence is: Propagate followed by Check.

4.6.2 Advice Behavior Abstraction Phase

Inputs This phase in the process has two kinds of inputs:

- **Advice Execution Sequence**: This is the result of phase *Composition*. In this case this is one shared join point, function compare_data(...). This join point has a sequence of advices attached to it, this sequence is: Propagate followed by Check.
- **Behavioral Specifications**: We have two behavioral specifications, in this example. One for advice Propagate and one for advice Check. These behavioral specifications are shown here:
  - **Propagate**:

    ```
    errorvariable: read;write;read;read
    ```

    This specification is only concerned with resource errorvariable. This resource directly corresponds to local variable *result*, as shown in listing 4.1 (line 6). It states that on this resource we perform four operations, namely: a read, followed by a write, followed by a read and finally a read is performed. The first read corresponds to line 8 in listing 4.2. The write operation corresponds to line 10 in listing 4.2. The next two reads are caused by lines 11 and 13 in listing 4.2, respectively.
  - **Check**:

```
arguments: read
errorvariable: write
```

Advice Check reads the arguments and can possibly write the errorvariable.

Transformation For each shared join point we have to compose the behavioral specifications of the individual advices. Since these advices are sequentially composed, we can also compose the behavioral specifications sequentially. We have to do this for each resource, in this case for resources arguments and errorvariable.

Outputs The result is a sequence of operations per resource for each shared join point. In this case we only have one shared join point and two resources. The results is as follows:

```
arguments: read
errorvariable: read;write;read;read;write
```

4.6.3 Conflict Detection Phase

Inputs

- **Conflict Detection Rules**: In the ASML example we stated that if an error is detected, it should be logged. A conflicting situation occurs if the operation sequence for resource errorvariable ends with a write, or if we have two consecutive writes in the operation sequence for resource errorvariable. These two cases can be translated into the following rules:

  ```
  Conflict(errorvariable): (write)$
  Conflict(errorvariable): (write)(write)
  ```

  We have expressed the rules using extended regular expressions.
- **Behavioral sequence of operations per resource per shared join point**: The sequences of operations from the previous phase:

  ```
  arguments: read
  errorvariable: read;write;read;read;write
  ```

Transformation In this example, we see that the conflict rule: `Conflict(errorvariable): write$`, accepts the sequence `read;write;read;read;write` of resource errorvariable. As such we have encountered a conflict.

Outputs The verdict for the ASML example is: advices Propagate and Check are conflicting on resource errorvariable for rule: `Conflict(errorvariable): write$`. If the advices were ordered differently, this conflict would not occur.

4.7 Application to other examples

As an additional illustration, we show how several of the behavioral conflict examples implemented in section 4.3 can be described using the resource-operation model. We discuss the following potential behavioral conflicts:

Authorization and Persistence : If data is made persistent before data access is authorized, this is considered a conflict.

Authorization and Authentication : A conflict is present if the authorization aspect is executed before the authentication aspect, as the identify of the user has not yet been verified.

Data modification : If two aspects both (destructively) write the same field or entry of a datastore, a conflict occurs, as the value written by one aspect is overwritten by another aspect. In this case there is no order that does not yield a conflict; either way a value is overwritten.

Again, we would like to stress that there are many possible ways to model the above behaviors (and conflicts); in this section we aim to show that this *can* be done using resources and operations, and we do so in the simplest possible way. All these behaviors can be expressed upon the same abstract resource, data, which represents some storage container. The data resource can model either a single (pseudo-)variable, a field or record in a database, a complex data structure or a complete database.

We now discuss the operations that can express each of the above individual behaviors:

Authorization : This can be expressed by a single operation *autr*, which indicates that an authorization check is executed. This check verifies that the current user is allowed to access the *data* resource. To keep things simple, we assume that there is a clear notion of 'current user' (e.g. associated with the current execution thread), and we do not distinguish among the various forms of access, such as read, write or execute. The latter could be modeled easily by introducing multiple different authorization operations (and would then also require more refined conflict detection rules that distinguish among these forms of access to the data resources).

Persistence : We only need to model operations that make the data resource

persistent. This could be a dedicated operation, such as persist, but we generalize this to the generic access operation write. The assumption is that operation write commits the data to some persistence structure.

Authentication : We propose a single operation autn on the data resource to represent authentication. We assume that this operation corresponds to some actions (e.g. password requests, biometrics) that verify whether the notion of 'current user' can be considered to be correct.

Data(base) modification : We use the general read and write operations to model data access.

The following conflict detection rules can express potential behavioral conflicts among the above behaviors:

Data access without authorization : The following assertion rule guarantees that no data access takes place without preceding authorization:
$Assert(data)$: ^ (!((*read*) | (*write*))* | !((*read*) | (*write*))* *autr* .* $
This rule demands that either there are no data access operations at all, or all operations before the authorization operation are not data access operations, after operation *autr*, any operation (.*) is allowed. This has to be the case in the entire sequence, from start ('^') to the end ('$').

Authorization without authentication :
$Conflict(data)$: ^ [!(*autn*)]* (*autr*) $; this detects a conflict if there is no authentication preceding an authorization operation.

All other conflicts between the data access operations are addressed by the following conflict rules (caused by distinct advices):

- $Conflict(data)$: *read write*: In this case the read operations obtains a value that is no longer valid, since the resource is changed afterwards.
- $Conflict(data)$: *write write*: In this case the effect of the first write operation is lost due to a subsequent write operation.

As mentioned previously, these rules do not distinguish between conflict caused within a single advice or between multiple pieces of advice. The tooling, implementing our approach should take care of this distinction.

4.8 Discussion

In this section we discuss to what extent our approach is general applicable to a wide range of situations. The key questions to consider are: can we describe conflicts among aspects at all (i.e. how expressive is our model)? Secondly, can we describe these in a generic way, independent of a specific application, or can

we describe application-specific conflict models, or both? We will argue that the method we propose is indeed general enough to represent both generic conflict models and domain- or application-specific conflict models.

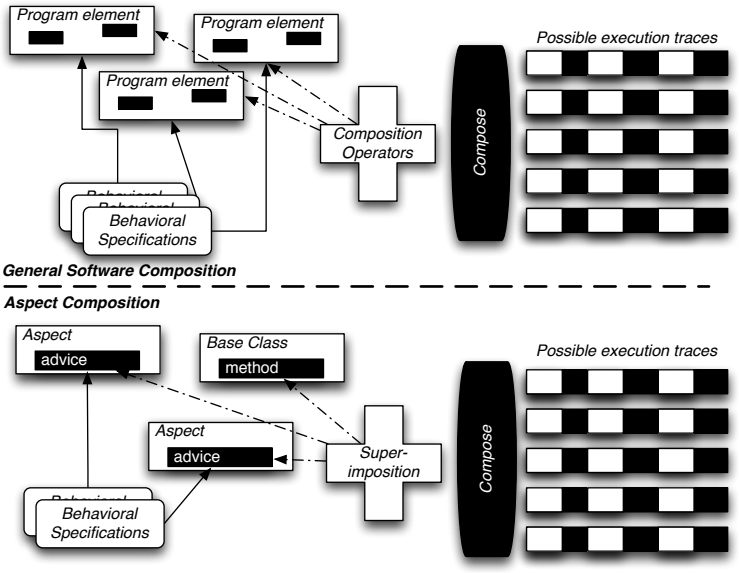
4.8.1 Can all behavior be modeled as a sequence of operations?

We currently only investigated a fully serialized list of operations per resource: although it is always possible to create such a serialization (in a deterministic way), and it appears to suffice in many cases, this does abstract from the control flow logic among the operations. For example, if a certain behavior is implemented within a control structure, and hence does not always occur, there are two options how to model this in a serialized form: first, one can assume that it is safer to ignore this conditional operations. In this way, assertions that require such an operation to occur would not match and yield a conflict. Second, one can assume it is safer to include it, which ensures that a conflict rule that matches upon such an operation is always triggered (but possibly including cases where this would not apply). Clearly, modeling the alternatives is a better way, where the matching algorithm can encode the most safe (or optimistic) assumption for both the assertion and the conflict rules. We envision also using an automaton to describe the behavior, since this can accommodate control flow.

4.8.2 Is it applicable to any paradigm or approach?

Our behavioral conflict detection approach is in fact independent of Aspect-Oriented Programming, and can be used to describe any behavioral composition and related behavioral conflicts. Figure 4.5 serves to illustrate that our approach can also be applied to general software composition verification.

Figure 4.5 is divided into two sections, the top section shows our approach for general software composition verification. We assume that there are some program elements, and a set of composition operators that compose these program elements into a system. Each program element also has a behavioral specification, in terms of resources and operations. All these ingredients are composed to create the complete application. The application can be viewed as a set of possible execution traces. These can be statically determined or monitored at run time. We try to match a set of conflict rules on the possible execution traces. True concurrency cannot be modeled using this approach. However, concurrent executions that are serialized can be represented. In this case we can



Figuur 4.5: The instantiation of the approach for respectively general and aspect-oriented software composition.

still apply our approach.

The lower part of figure 4.5 presents the approach for aspect composition. The program elements are aspects and base classes. Aspects have advices with a behavioral specification. Classes have methods which are subjected to advices. The composition operator in this case is superimposition of advice on methods. For each join point, we can create a set of possible execution traces, as explained in the previous sections. This is also detailed in chapter 5. Similar to generic software composition, we try to match a set of conflict rules on the possible execution traces.

There are two reasons why we believe AOP, and especially composition filters, are a more suitable domain for applying our method. The first reason is that, due to obliviousness, the risk of having unplanned interactions among aspects is much more serious in AOP. The second reason is that a suitable aspect language can help or ease the (automated) derivation of behavior from code. In general, we believe that, from a practical perspective, automatic derivation of the resources and operations from regular Turing complete code is computationally complex and suffers from termination and undecidability problems. We think that with the addition of domain-specific and application-specific information and our abstract resource-operations, (partial) automatic derivation of behavioral specifications from (declarative) aspect languages is feasible, as illustrated by chapter 5.

4.8.3 Can all behavior be specified?

This depends on the expressiveness of our resource-operation model. The key contribution of this model is that it allows for the encapsulation of possibly complex implementation details within abstract operations. This means that it is not necessary to be able to accurately represent those implementation details.

4.8.4 Can all conflicting patterns be detected?

This depends on the chosen language for expressing the conflict rules. In this thesis, and in our current implementation, we use extended regular expressions to specify patterns over a sequence of operations on a single resource. However, other languages could be chosen instead. For example, we plan to investigate the applicability of linear temporal logic as an alternative. In addition, one may consider patterns that express (un)desired interaction among operations between resources.

4.8.5 Which types of conflicts can be modeled?

A key underlying assumption of our method is that conflicts occur because of shared state or shared behavior among aspects. This would be reflected in conflict rules that look for multiple operations on shared resources. In fact, such rules can also identify conflicting patterns that do not come from different advices, but are written in a single advice. This category of detected conflicts can be turned off in tools. The proposed method is suitable for cases where (potentially complex) behavior can be abstracted into one or more operations, and a conflict can be described as the occurrence of a certain pattern of those operations. Some other categories of conflicts would be detectable in different ways, e.g. by a complex data flow (dependency) analysis. It is hard to determine for which types of conflicts our approach is or is not applicable.

4.8.6 What is required for and what is the effect of detecting different categories of conflicts?

In this thesis we distinguish between three categories: *Generic*, *Domain-specifc* and *Application-specific*. These are not on a discrete scale, since a mixture of categories is possible. We compare the three categories using three properties:

Reuse : To what extent are the resources, operations and rules reusable in other applications?

Specification : To what extent can the resources, operations be automatically derived?

Number of false positives and negatives : Does the choice between general or more specific resources, operations and rules influence the errors in conflict detection?

We now discuss these three properties for each category.

4.8.6.1 Generic conflicts

An example of a generic conflict would be: a data resource that can only be read or written, is not allowed to be written twice.

Reuse : The resources, operations and rules are reusable for all applications. All imperative programming languages assume that there is some data and that one performs transformations, e.g. read and write, on this data. As such generic conflicts can be detected in all applications.

Specification : In imperative languages, resources can represent local or global variables or parameters. Read and write operations on these resources can be automatically extracted using appropriate tooling, e.g. CodeSurfer [Gra] for C code.

Number of false positives and negatives : General resources, operations and rules are prone to identify too many conflicts since there is no specific behavioral information. The rule in the ASML example detects all duplicate writes, but if a write is actually an update; in that case we might detect a problem where there is none.

4.8.6.2 Domain-specific conflicts

An example of a domain-specific conflict would be the ASML example, as described in section 4.1.3.

Reuse : The resources, operations and rules are reusable for a specific domain, e,g. synchronization, security or exception handling. A common set of resources, operations and rules should be defined for a specific domain. With this set one should be able to express all domain-specific behavior, and detect conflicts.

Specification : Usually, domain specific resources, operations and rules cannot be automatically derived from a program. However, as there are probably general patterns for a given domain, we can use this information to partially automate behavior specification extraction. Some manual specification may still be required for a given domain.

Number of false positives and negatives : The number of false positives and negatives, depends on the accurate specification of the domain. Assuming that this specification is accurate, a more detailed analysis can be performed. Also, domain-specific conflict rules only identify those patterns that are conflicting within the domain. As such the number of false positives and negatives should be reduced.

4.8.6.3 Application-specific conflicts

Reuse : Resources, operations and rules that are specific for one application or even for a part of application are only reusable to variations of that application.

Specification : In practice, it is be very hard to extract application specific behavioral specification automatically from the code. This category of conflicts requires detailed information from the developers. This can be

achieved by manually annotating the source code with detailed behavioral specifications.

Number of false positives and negatives : An application specific behavioral model will capture numerous, if not all, details for that application. As such the number of false positives and negatives should be low.

Which category of behavioral conflicts one wants to detect is a trade-off between the three described properties. For example, in a critical application, we might want to gain better conflict detection by specifying many details. The approach we propose in this thesis does not make any assumptions about a specific instantiation. However, we favor domain-specific conflicts, since this seems to us the optimal trade-off between the three properties.

4.9 Conclusions

In this chapter we have discussed the problem of behavioral conflicts among aspects. We explained the problem of behavioral conflicts using an example which we encountered at ASML. This illustrates the relevance of the problem. The main contribution of this chapter is to present an approach for detecting behavioral conflicts, based on a novel abstraction of advice behavior in terms of resources and operations. Such an abstraction has several advantages:

- It allows for expressing behavior (and conflicts) without involving (needless) implementation-level details.
- It allows to express not only generic, or universal, conflicts, but also domain- and application-specific conflicts.
- It strongly reduces the computational complexity of the conflict detection analysis.

In section 4.4 the relation with other work in the area of aspect composition problems is described. The next chapter will provide a detailed instantiation of the approach for Composition Filters. In the next chapter, we also discuss the related work in more detail.

The current and earlier versions of the abstract conflict detection approach have been published in [DSBA05] and [DBA06].

Behavioral Conflict Reasoning applied to Composition Filters

5

In this chapter we show how the approach that was introduced in the previous chapter is instantiated for one specific aspect-oriented language, namely Composition Filters. We first explain the rationale for using Composition Filters. Next we will go through the three main phases of our approach for behavioral conflict detection, and describe in detail how to apply each phase to the Composition Filters approach. We also explore how suitable language constructs can aid in (partially) automating the reasoning process. Finally, we discuss the related work in this area and conclude.

5.1 Motivation

Chapter 4 explained the motivation for behavioral conflict detection among aspects. That chapter also presented an approach for the detection of behavioral conflicts using a resource-operation based abstraction of advice behavior. This approach can be used to model both generic behavioral conflicts, as well as more specific behavioral conflicts.

Chapter 4 also showed an example of a behavioral conflict among aspect that we

encountered at ASML. This example illustrated the need for behavioral conflict detection in industry. Hence, we would like to demonstrate an implementation of our behavioral conflict detection approach.

We demonstrate behavioral conflict detection on the Composition Filters approach and its implementation in Compose* [Unia]. The Composition Filters model has some characteristics that support (partial) automatic derivation of behavioral specifications:

- Filters encapsulate domain or application knowledge for which it is usually possible to provide a specification of the behavior, in terms of resource and operations.
- Given a message and system state, a filter can either accept or reject the message. In both cases a filter action is executed.
- The filter pattern language is a declarative language for message matching and substitution, which can be analyzed statically.
- Superimpositions are expressed in Prolog, which resolve to a set of join points.
- The ordering of filters within a filter module is defined by the declaration order.
- Optional (partial) ordering specifications can be provided by the programmer, to determine the ordering constraints among filter modules, at the same join point. If there are multiple orderings, the compiler selects an arbitrarily one.

5.2 Composition Filters

Section 1.1 discussed the basic concepts behind Composition Filters. In this section we discuss all elements in detail, including a precise specification of the elements in the language. We use the Vienna Development Method (VDM) [Jon90, Jon92, Daw91] notation. We first define some primitive types:

$String = char^*$

$Name = String$

$Undefined = String$

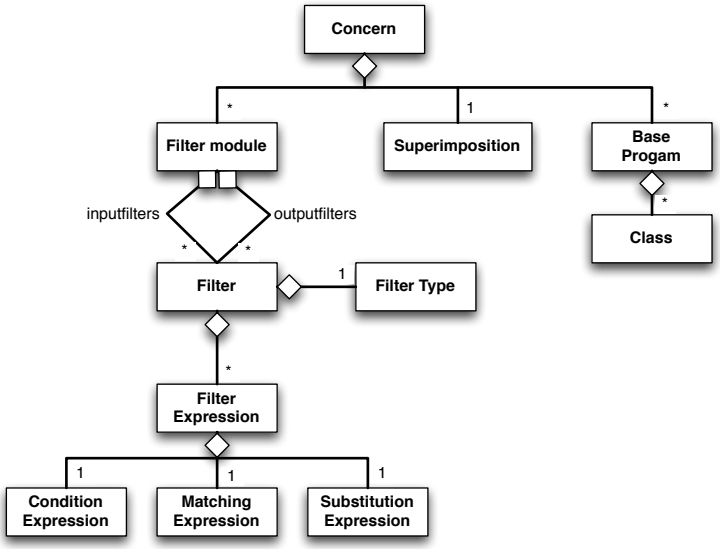
$Object = Name$

$Class = Name$

$Method = Name$

$MessageObject = Name$

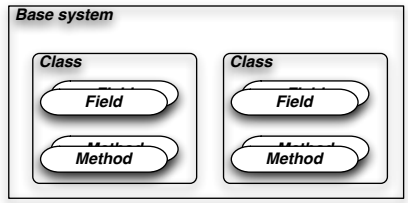
In VDM, a '*' indicates a sequence and a **compose** statement a record. Figure 5.1 presents an overview of the elements in the Composition Filter model and their relations.



Figuur 5.1: Overview of Composition Filters

We have two types of elements in figure 5.1. The first is the *Base program* and the other elements are part of the Composition Filters model.

The base system, depicted in figure 5.2 represents a program. In this program we can have classes, containing methods and fields. We use here an object-oriented program, but Composition Filters is not limited to object-oriented languages, procedural languages can also be used.

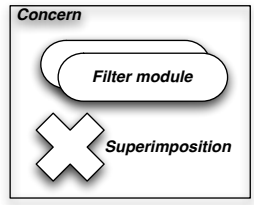


Figuur 5.2: Base system

Base System

A Composition Filters Concern (figure 5.3) is a module that addresses (crosscutting) concerns in the base system.

Concern



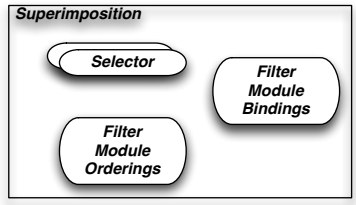
Figuur 5.3: Concern in Composition Filters

A Concern in Composition Filters consists of a set of Filter modules and a Superimposition specification. A Filter Module specifies *what* behavior should be executed. Superimposition specifies *where* the Filter Modules should be applied. To formalize:

compose *Concern* **of**
filermodules : *FilterModule*-**set**,
superimposition : *SuperImposition*-**set**
end

Superimposition

Superimposition provides a mechanism to address crosscutting concerns. A superimposition specification selects a set of relevant classes in the base program and superimposes filter modules on each class in this set.



Figuur 5.4: Superimposition in Composition Filters

A superimposition specification (figure 5.4) is composed of:

- A set of Selectors, which refer to elements in the base program. These selectors are Prolog queries over a representation of the base program. The selectors yield a set of classes, i.e. the join points.
- Filter Module Bindings link selectors to filter modules.
- Filter Module Orderings allow the developer to indicate an ordering between filter modules at join points where multiple filter modules are superimposed.

Superimposition can be formalized as follows:

$$Selector = Class\text{-}\mathbf{set}$$

$$OrdeningsExecutionConstraint = String$$

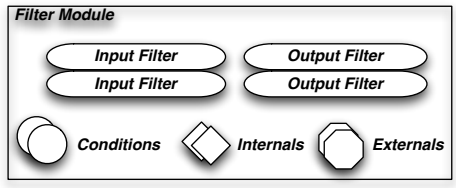
compose *SuperImposition* **of**

$$\begin{aligned} selectors &: String \xrightarrow{m} Selector, \\ filterModuleBindings &: Selector \xrightarrow{m} FilterModule\text{-}\mathbf{set}, \\ filterModuleOrderings &: OrdeningsExecutionConstraint\text{-}\mathbf{set} \end{aligned}$$

end

Filter Module

A Filter Module (figure 5.5) is the primary unit of superimposition and reuse, it can be compared to an advice. A filter module is composed of:



Figuur 5.5: Filter Module in Composition Filters

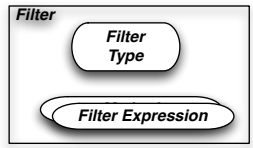
- A sequence of input filters, which are filters that match incoming messages. Messages pass through filters in the sequence.
- A sequence of output filters, which are filters that match outgoing messages.
- A set of conditions, which are references to methods in the base program that return a Boolean value. Conditions are used within filter expressions, these expressions explained shortly.
- A set of internals, which are classes that are instantiated each time a filter module is superimposed. They represent the internal state of a filter module.
- A set of externals, which are instances of classes that are shared between all superimpositions of this filter module.

Formally, a filter module can be defined as:

compose *FilterModule* **of**
$$\begin{aligned} \mathit{inputfilters} &: \mathit{Filter}^*, \\ \mathit{outputfilters} &: \mathit{Filter}^*, \\ \mathit{internals} &: \mathit{Name} \xrightarrow{m} \mathit{Object}, \\ \mathit{externals} &: \mathit{Name} \xrightarrow{m} \mathit{Object}, \\ \mathit{conditions} &: \mathit{Name} \xrightarrow{m} \mathit{Method} \end{aligned}$$
end

Filter

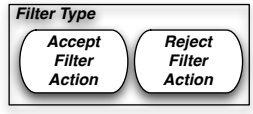
Filters (figure 5.6) in Composition Filters execute behavior, depending on a set of filter expressions. A filter has a Filter Type and a sequence of Filter Expressions. We define a filter as:



Figuur 5.6: Filter in Composition Filters

compose *Filter* **of**
type : *Name*,
filterexprs : *FilterExpression**,
acceptaction : *FilterAction*,
rejectaction : *FilterAction*
end

Filter Type



Figuur 5.7: Filter Type in Composition Filters

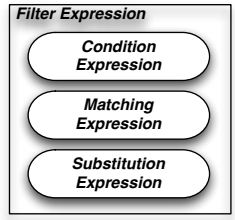
A Filter Type (figure 5.7) encapsulates reusable (domain-specific) behavior with powerful parametrization. It is composed of two filter actions:

- An Accept Filter Action is executed if the filter accepts a message. A Filter Action is implemented as a special method in the base program. This method has a single context object as a parameter. This context object allows the filter action to query and manipulate properties of a message and affect the control flow.
- A Reject Filter Action is executed if the filter does not accept a message.

A FilterAction is defined as:

compose *FilterAction* **of**
name : *String*,
effect : *Undefined*
end

Filter Expression



Figuur 5.8: Filter Expression in Composition Filters

Whether a filter accepts or rejects a message depends on the filter expressions (figure 5.8). A Filter Expression can be use to query the state, to select a specific message or substitute certain properties of the message. A Filter Expressions is composed of three elements:

- A Conditional Expression is a Boolean expression composed of Conditions and Boolean operators: And (&&), Or (||) and Not (!).
- A Matching Expression can match on two properties of the message: *target* and *selector*: the object, respectively the method for which this message is intended. One can match on specific *targets* and *selectors* or use wildcards (*). Composition Filters support two kinds of matching: *name* matching and *signature* matching. *Name* matching is purely syntactical, whereas signature matching checks whether the message is in the signature of the target object. A Matching Expression is only evaluated if the condition expression resolves to true.
- A Substitution Expression can substitute the *target* and *selector* of a message. This enables declarative rewriting of the message. A Substitution Expression is only executed if the matching expression accepts a message.

Formally, a filter expression is defined as:

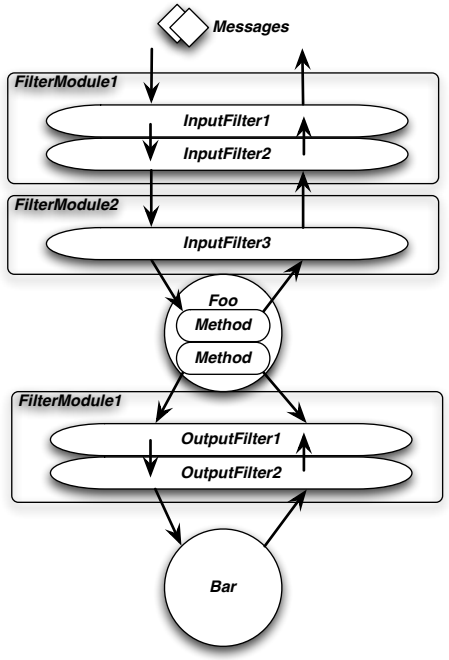
$$ConditionalExpression = \mathbb{B}$$

compose *FilterExpression* **of**
conditionExpr : *ConditionalExpression*,
matchingExpr : *TargetSelectorTuple*,
subsitutionExpr : *TargetSelectorTuple*
end

compose *TargetSelectorTuple* **of**
Target : *Name*,
Selector : *Name*
end

Dynamic Behavior of Filters

We now elaborate on the dynamic behavior of filters and filter modules. Consider the situation in figure 5.9.



Figuur 5.9: Message Filtering in Composition Filters

Filter modules FilterModule1 and FilterModule2 are superimposed on class Foo.

All messages sent from, or to, instances of class Foo will be subjected to the filters inside these two filter modules. A Message represents an interaction or event in the system, and it has several properties:

Sender : The object that initiated the message.
Server : The object that initially received the message.
Target : The object for which the message is intended.
Selector : The name of the method to which the message is directed.
Arguments : The arguments of the message.
Return value : The return value of the message.

Assume that a message is sent to an instance of class Foo. It is first subjected to the input filters in filter module FilterModule1, in this case first to filter InputFilter1. Filter InputFilter1 can either accept or reject this message, depending on condition and matching expression. The exact behavior that is executed in case of acceptation or rejection is filter type-specific. A filter can execute some behavior on an incoming message and on the return of a message. This behavior is linked, meaning that if the message accepts on an incoming message, it will also execute the related accept return behavior. This does not have to be the same behavior, obviously. Also, behavior that is executed on the return of messages, is executed in the inverse order of the filter order, i.e. bottom-up rather than top-down.

A filter can choose to continue with filter evaluation or redirect the message. Assuming no filter affects the control flow, the message will pass through filters InputFilter2 and InputFilter3. At this point it reaches the end of the input filters of filter module FilterModule2. The message is sent to the current target and selector. We assume that a method of class Foo is executed. Once this method returns, the return behavior of the filters is executed in the reverse order.

If a method in class Foo calls a method of another class, in this case class Bar, the message first has to pass through the output filters superimposed on class Foo. In this example there are two output filters: OutputFilter1 and OutputFilter2. These filters are processed in the same way as input filters.

5.3 Application of Behavioral Conflict Detection to Composition Filters

We now show a detailed instantiation of the approach presented in the previous chapter. To structure the discussion in this chapter, we use a similar three phase process as presented in the previous chapter. Figure 5.10 shows the instantiation

of this process for the Composition Filters model.

Figure 5.10 shows the three phases of our approach. The first phase (*Composition*) deals with *superimposition* and *ordering*. The result of this phase is a sequence of filter modules per shared join point, i.e. per class. These sequences are subsequently transformed (phase *Advice Behavior Abstraction*) into a structure that enables automated reasoning about the control flow within the filter modules. This structure is annotated with the resource-operation tuples from the behavioral specifications. Finally in phase *Conflict Detection*, all conflict rules are translated into a suitable structure and compared with the annotated representation of the filters. From this comparison a verdict is expressed. These three phases are explained in more detail in the next three sections.

5.4 Composition Phase

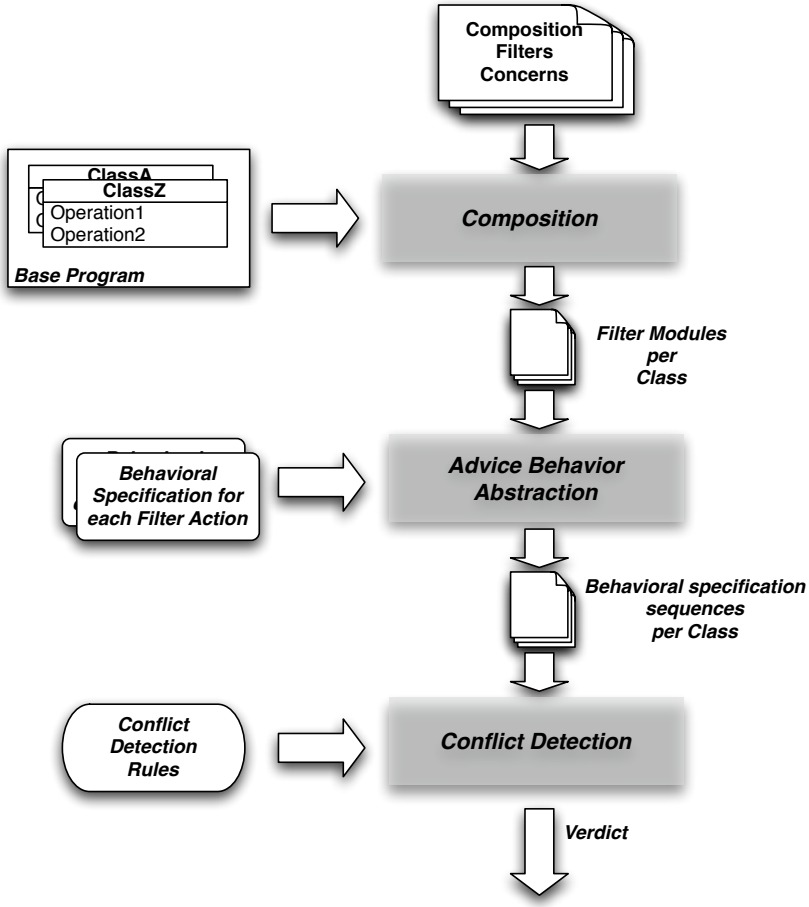
5.4.1 Inputs

Phase *Composition* expects a set of concerns and a base program as its inputs. We first explain concerns *Parameter Checking* and *ErrorPropagation* from the ASML example in Compose*. Next, we briefly discuss an example base program.

5.4.1.1 The Parameter Checking Concern in Compose*

We explain the composition filters model with the use of the example of the parameter checking concern presented in the previous chapter. Listing 5.1 shows the implementation of the parameter checking concern in Compose*. The behavior implementing the checking of parameters (as shown in listing 4.1) is implemented in a filter type called ParameterChecking (not shown here).

```
1  concern ParameterChecking
2  {
3    filtermodule check
4    {
5      internals
6        checker : ParameterChecker;
7      conditions
8        inputwrong : checker.inputParametersAreInvalid();
9        outputwrong : checker.outputParametersAreInvalid();
10     inputfilters
11       paramcheckfilter : ParameterChecking = {
12                                   inputwrong || outputwrong => [*.compare_data] *.* }
13   }
14
15   superimposition
```



Figuur 5.10: Approach instantiated for Composition Filters

```
16  {
17    selectors
18      pcsel = {Class | isClassWithName(Class, 'CC.CX.FS')};
19    filtermodules
20      pcsel <- check;
21  }
22 }
```

Listing 5.1: Source of the ParameterChecking concern

Before we discuss the code in listing 5.1, we first explain some generic concepts of Composition Filters. In Composition Filters, the basic abstraction is a *concern*; this is a generalization of both (object-oriented) classes and aspects. In this example, the concern ParameterChecking corresponds to an aspect that implements a crosscutting concern, i.e. the contract enforcement for the parameters of a function.

In general, composition filters concerns consist of three main parts, all of them optional:

filter modules : are the unit of superimposition corresponding to the general notion of advice. A filter module defines a certain behavior that is to be superimposed at specific locations in the program. In the *ParameterChecking* example, there is one filter module, called check (lines 3 to 13). We shortly discuss the filter module specification in more detail.

superimposition : is the specification of the actual crosscutting locations. These specifications define the pointcut designators (called selectors), and the binding of the behavior (expressed by filter modules) at the selected join points. In the example, the filter module check is superimposed (line 20) at the locations indicated by the selector pcsel that designates all relevant classes, in this case the class with the name CC.CX.FS (line 18).

implementation : the implementation part contains the definition of the object behavior of a concern: this can be expressed in an arbitrary object-based language (assuming it is supported by the implementation). In the example no implementation part is necessary.

We now explain concern ParameterChecking in listing 5.1 in more detail. The behavior of the filter module check (lines 3 to 13) is as follows: for each object where this filter module is superimposed, internal object *checker* is instantiated. For every incoming message, it first verifies whether both conditions inputwrong and outputwrong are true. In case one of the two condition is true, the filter tries to match the selector of that message to *compare_data*. In the example, the target of the message is ignored. If the selector matches, no substitution has to take place in this case, as indicated by the wildcards, and the filter *accepts* the message. If the selector does not match *compare_data* the filter rejects

and the message is passed to the next filter. This triggers the *accept* action of the ParameterChecking filter, the implementation details of this filter and filter action are not shown here. This accept action sets the error variable. In a programming language that supports exceptions, we could have used the general *Error* filter, however since C does not support exceptions, and we have to set the error variable to a specific value, and we used a dedicated filter for this. If both conditions inputwrong and outputwrong are false, the filter will *reject* and the message continues to the subsequent filter, if any.

Conditions inputwrong and outputwrong are implemented in the base programming language and have to return a Boolean value. These conditions receive a context object as a parameter. This context object provides reflection on the current join point.

The superimposition part of the concern definition in line 15-21 of listing 5.1, starts with a section labeled selectors. A selector corresponds to a pointcut designator; it selects a number of join points within the program. This is expressed using Prolog predicates, making use of a number of primitive predicates that express properties of the program. For example, line 18 defines a selector with identifier pcsel; its intention is to select all relevant classes. The parameter checking concern implements a coding guideline, as such we may select all classes in the system. Selector pcsel selects the specific class "CC.CX.FS", to demonstrate the selection language and to ensure a single concrete shared join point, for the discussion in this chapter. The first part of the selector defines the unbound variable that refers to all the join points identified by this selector, in this case Class. The selector consists of a predicate, isClassWithName, which narrows down the possible values for Class to those classes with the name "CC.CX.FS". A class is equal to a file and a method to a function, in the Compose* implementation that targets C. This enables us to use the same matching language in the three ports of Compose*: .NET, Java and C. After the declaration of the selectors, the superimposition of filter modules is defined by binding selectors and filter modules. A filter module will be superimposed on each join point identified by the selector.

5.4.1.2 The Error Propagation Concern in Compose*

As a second aspect, we show how the error propagation concern can be described, this is shown in listing 5.2. The behavior implementing the propagation of errors (as shown in listing 4.2) is implemented in a filter type called ErrorPropagation (not shown here).

```
1 concern ErrorPropagation
```

```
2  {
3    filtermodule propagate
4    {
5      inputfilters
6        errorpropagationfilter : ErrorPropagation = { [*] }
7    }
8
9    superimposition
10   {
11     selectors
12       epsel = {Class | isClass(Class) };
13     filtermodules
14       epsel <- propagate;
15   }
16 }
```

Listing 5.2: Source of the ErrorPropagation concern

The concern ErrorPropagation defines one filter module named propagate, which consists of an input filter named errorpropagationfilter of type ErrorPropagation. The filter, defined on line 6, matches all messages, since the filter expression is `[*]`, and thus will always execute the accept action of the filter. The accept action of the error propagation filter ensures that all calls are only executed in a non erroneous state and that, if an error is detected, it will be logged and properly propagated to the caller. The filter module propagate is to be superimposed on all classes in the system, because of the selector `epsel = {Class | isClass(Class)}`, see lines 12 and 14.

5.4.1.3 Base Program

The base program on which both aspects are superimposed is also input for the *Composition* phase. We only show a sample from the base program. The implementation details are not important for this discussion. Listing 5.3 shows function compare_data, and we assume that this function resides in file FS in directories CC and CX, and is thus selected by selectors pcsel and epsel in listings 5.1 and 5.2.

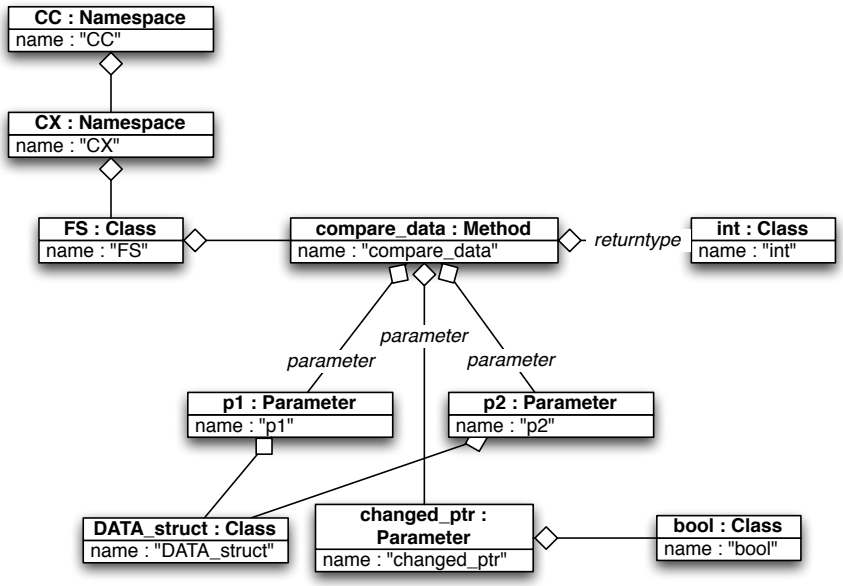
```
1  static int compare_data(
2      const DATA_struct*                p1,
3      const DATA_struct*                p2,
4      bool*                             changed_ptr)
5  {
6    int result = OK;
7
8    // code that compares the structures and sets the changed_ptr boolean accordingly
9
10   return result;
11 }
```

Listing 5.3: Example base program

5.4.2 Transformation

The first step is to construct a model of the base program. This must include all entities that can be referred to in the superimposition specifications. Examples of elements in this model are: Namespaces, Packages, Classes, Methods, Parameters, Interfaces, etc. This model also contains all relationships between the elements, for example: which methods does a class have, which interfaces are implemented by a class, what is the superclass of a particular class, etc. Explaining all the details about this program model is beyond the scope of this thesis, see [Hav05] for more information.

In the ASML example, the program model is relatively simple: there is only one class FS that belongs to namespace CC.CX and this class has a single method, called compare_data. This is depicted by the object diagram in figure 5.11.



Figuur 5.11: Object model of the example base program

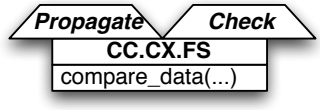
Figure 5.11 shows the program model for our base program. In the top left corner we see the two namespaces (CC and CX) and class FS. Class FS has one method called compare_data. This method has two parameter p1 and p2 of type DATA_struct and one parameter changed_ptr of type bool. Method

compare_data has class int as its declared return type. For larger base programs a program model gets larger as well.

Once the program model is constructed, we iterate over all superimposition selectors and resolve these w.r.t. the base program model. In this case there are two superimposition selectors. The first at line 18 in listing 5.1, and this is: `pcsel = {Class | isClassWithName(Class, 'CC.CX.FS')};`. The second selector is defined at line 12 in listing 5.2, and this is: `epsel = {Class | isClass(Class)};`.

For the given program model, these two selectors resolve to the same set of classes, namely class CC.CX.FS. In larger applications this set will contain many more classes.

Next, we iterate over all filter module bindings and attach the filter modules to the join points, i.e. the classes resulting from evaluating the selectors. The result is a set of filter modules per class. In our example there two bindings. The first one is defined at line 20 in listing 5.1, and this is: `pcsel <- check;`. The second binding is defined at line 14 in listing 5.2, and this is: `epsel <- propagate;`. The result of superimposition is depicted in figure 5.12.



Figuur 5.12: Resolved superimposition

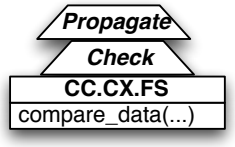
Figure 5.12 shows a single join point, in this case class CC.CX.FS, and two filter modules: propagate and check.

We have not yet determined an ordering between filter modules at shared join points. As explained before, the order of filter modules can be affected using either a full or partial ordering specification. In this example we have not included this specification. For examples and more details about the ordering specification, we refer to [Nag05] and [NBA05].

In this example case, we will assume the following order, as this order exposes the conflict we are interested in: first propagate, then check. The result of ordering is thus a sequence of filter modules per class.

5.4.3 Output

The result of this first phase is a sequence of filter modules per class. In the example case the result is illustrated by figure 5.13. Figure 5.13 shows class CC.CX.FS with the two filter modules superimposed on it. In this case these are propagate followed by check.



Figuur 5.13: Sequence of filter modules for class CC.CX.FS

We can define function Composition as follows:

$$SharedJoinPointMap = Class \xrightarrow{m} FilterModule^*$$

$$\begin{aligned} &Composition : SuperImposition\text{-}\mathbf{set} \times FilterModule\text{-}\mathbf{set} \\ &\qquad \rightarrow Class \xrightarrow{m} FilterModule^* \end{aligned}$$

$$\begin{aligned} &Composition(superimps, fms) \quad \triangleq \\ &\quad SharedJoinPointMap = \ldots \end{aligned}$$

$$\begin{aligned} \mathbf{post}\ &\forall class \in \mathbf{dom}\ SharedJoinPointMap \cdot class \in \mathbf{rng}\ superimps.selectors \\ &\wedge \mathbf{rng}\ SharedJoinPointMap \subseteq fms \\ &\wedge \forall fm\text{-}seq \in \mathbf{rng}\ SharedJoinPointMap \cdot \mathbf{card}\ fm\text{-}seq >= 2 \end{aligned}$$

Composition can be presented as a function that takes a set of join points, i.e. a set of classes, and set of filter modules. Function Composition returns a map named SharedJoinPointMap, which maps a class to a sequence of filter modules.

We constrain this map in three ways. First, we state that each class in the domain of this map, should be an element of the set of join points. Second, we state that the range of this map should only contain filter modules present in the program. Finally, we state that each sequence of filter modules in the range of map SharedJoinPointMap, should contain at least two filter modules. This

last constraint ensures that map SharedJoinPointMap contains only shared join points.

5.5 Advice Behavior Abstraction Phase

The next phase in our approach is to transform the filter module sequences to a structure that enables automated reasoning. To do this we also need to have behavioral specifications from the filters that are used. Again, we start the discussion of this phase by stating the inputs.

5.5.1 Inputs

This phase has two inputs:

- The first is the sequence of filter modules per class, which was the output of the previous phase.
- The second input is the behavioral specification of the filters and filter actions involved.

The behavior of a filter is consists of filter instance-specific behavior and filter type behavior. The first behavior represents the condition, matching and substitution behavior of a filter. The second behavior is defined per filter type and represents the intention of the filter. The filter type-specific behavior is composed from the following three categories.

- A filter can access the properties of a message.
- A filter can alter the control flow of filter evaluation and join point,
- A filter can execute some specific actions, e.g. encryption or compression.

We first explain the filter instance-specific behavior, followed by a discussion on filter type-specific behavior and then discuss the behavior in terms of resources and operations for each category.

5.5.1.1 Filter instance-specific behavior

Filter instance-specific condition, matching and substitution expressions also operate on resources. In table 5.1 we show examples of filter expressions and their corresponding resource models. We also include the ASML example filter expression from filter parameterchecking. Resources between angled brackets are conditions. In the table we have assumed the worst case scenarios, since in

Composition Filters if a target match fails, we do not match the selector. This conditional evaluation is addressed in section 5.5.2.2

Tabel 5.1: Resource model for example filter expressions

Filter Expression	**Target**	**Selector**	**<cond>**	**<input-wrong>**	**<output-wrong>**
`[t.s]*.*`	read	read			
`[t.*]*.*`	read				
`[*.s]*.*`		read			
`[*.*]*.*`					
`[*.*]t.s`	write	write			
`[*.*]t.*`	write				
`[*.*]*.s`		write			
`[*.*]*.*`					
`cond => [t.s]t.s`	read write	read write	read		
`inputwrong || outputwrong => [*.compare_data]`		read		read	read

Table 5.1 illustrates that wildcards in matching and substitution expressions are ignored. In the last row, the matching expression from the *Parameter Checking* filter (line 11 in listing 5.1) is shown. The two conditions *inputwrong* and *outputwrong* are (possibly) read as is the selector.

5.5.1.2 Message properties related behavior in filters

Message property related interference can be caused by one advice altering a message property on which a second advice depends.

Resources: Message properties can be inspected or manipulated by advice. These properties are usually bound via explicit context bindings or via pseudo variables, like *thisJoinPoint* in AspectJ or *JoinPointContext* in Compose*. We use the terminology of message based interception to describe the meaning of these common resources .

Sender : The object that sent the message.
Server : The object that initially received the message, this remains the same even if an if the message is dispatched to another object.
Target : The object for which the message is directed.
Selector : The name of the method to which the message is directed.
Arguments : The arguments of the message.
Returnvalue : The return value of the message.

We have chosen here to model the set of arguments as one simple resource. This abstraction might lead to false positives, e.g. two advices can operate on distinct arguments. However, we have chosen for this abstraction as writing a behavioral specification which depends on the ordering of arguments is very fragile.

Operations: For each of the above resources, we assume the following operations:

read : This operation queries the state of the resource on which it operates,
write : This operation overrides the state of the resource on which it operates.

5.5.1.3 Control-related behavior in filters

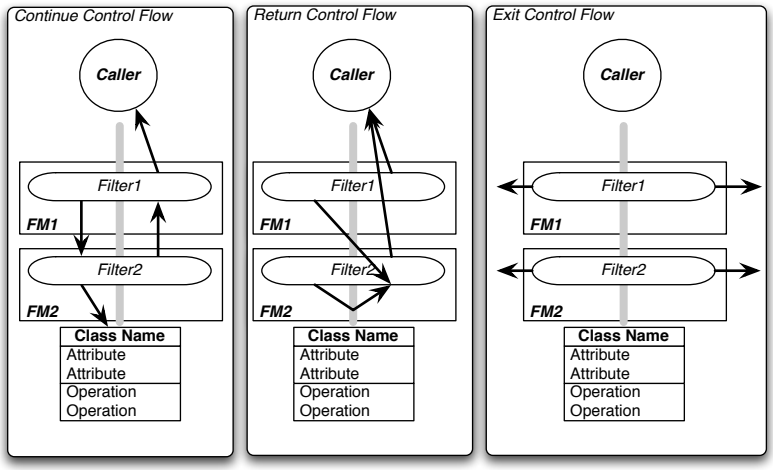
Control-related interference can be caused by one advice affecting the control flow of another advice. In this section we define the resources and operations that capture the possible behavioral conflicts related to control flow.

Resources: We model control flow behavior as operations on the abstract controlflow resource. All advices on the same join point operate on this single resource.

Before we show the operations that can be carried out on resource controlflow, we first have to elaborate on the different ways one can alter the control flow within Composition Filters. A filter in Composition Filters can affect the control flow in three ways: a filter can continue the control flow, return the control flow, or exit the control flow. These three ways are from the perspective of a filter in a filter module. Figure 5.14 shows the three ways the control flow can be affected by the filters.

Figure 5.14 shows three similar situations. Two filter modules (FM1 and FM2) are superimposed on a class. Filter module FM1 has one filter (filter1), filter module FM2 has also one filter (filter2). There is a solid vertical gray line in the middle of the figure. The parts of the filters that are to the left of this line are executed before the execution of the join point, the parts to the right are executed after the execution of the join point. Also, the filters on the returning side are evaluated bottom up, while the filter on the incoming side are evaluated top down.

In the left situation in figure 5.14, a black line indicates a continue control flow action. This flow action is implicit in the composition operator between filters, and corresponds to the semicolon between filters. This is thus the default, unless other flow actions are specified. The semantics of the continue control flow action depends on the location within the entire chain of filters. In most cases a



Figuur 5.14: Affecting the control flow in Composition Filters

continue control flow action results in passing the current message to the next filter. The compiler inserts a default dispatch filter (not shown here) at the end of the filter set, which ensures that messages reaching the end of the filter set will be dispatched to the current target and selector. If we reach the last filter, in this case filter1, on the return of the message, a continue control flow action results in retuning the current return value to the caller.

The second control flow action one can do is called return. Return actions can be identified in the middle situation in figure 5.14. Again the semantics of this action depend on the location within the filter chain. Before the execution of the join point, issuing a return control flow action results in the control flow being transferred to the first of the filters that execute on the return of the message, in this case filter filter2. If a return flow action is issued on the returning side, we immediately return to the caller. If a filter executes an accept filter action for an incoming message, the accepting return action also executes, when the message returns. The same situation holds for reject actions.

Finally, one can issue an exit control flow action. These can be identified in the right situation in figure 5.14. In all cases, this results in completely stopping the evaluation of the remaining filters. An example of such a case would be the Error filter, which throws an exception.

Operations: The above mentioned control flow actions are translated to the

following operations:

continue : The advice does not change the control flow.
return : The advice returns immediately, and as such the original join point is no longer executed.
exit : The advice terminates the entire control flow, e.g. a exception is thrown or an exit call is made,

5.5.1.4 Filter types

In this section we can translate the current set of filters to the above mentioned resources and operations. Compose* has several predefined filter types and supports user-defined filter types. We first discuss these predefined filter types. Next, we discuss how these filter types are mapped to accept and reject actions. Finally, we map these actions to specific operations on resources. For more detailed information about these and other filter types we refer to [BA05] and [DBA05]. We have also included the two filters from the ASML example in the list below.

Dispatch : The dispatch filter can be used for delegation or simulating (multiple) inheritance. If the filter accepts, it performs a DispatchAction action, else the next filter is evaluated, this is considered a NopAction action. Dispatch filters can only be used in an input filter set.
Send : The send filter is similar to a dispatch filter, however it can only be used in a output filter set. Also, it sets the sender of a message to the server of that message and sets the server of a message to the current target of that message. If the filter accepts, it performs a SendAction action, else the next filter is evaluated, a NopAction action.
Error : The error filter can be used for assertion specifications, e.g. pre and post conditions. If the error filter accepts, it performs a NopAction action, if it rejects it executes an ErrorAction (i.e. throws an exception).
Meta : The meta filter enables the developer to create user-defined advice. If the filter accepts, the message is reified and passed to a method as an argument. Acceptance of a meta filter is referred to as the execution of a MetaAction. In this advice, the user can introspect or manipulate the properties of the message and the message execution. This method is called an AdviCe Type (ACT) method. Rejection of this filter results in the execution of a NopAction action.
Before : A before filter executes a method in the base program,. This is similar to the meta filter. The only difference is that this filter only matches and executes accepting action BeforeAction on incoming messages. If a before

filter fails to match a NopAction filter action is executed.

After : An after filter is similar to a before filter, only this filter matches and executes accepting action AfterAction on the return of messages. If an after filter fails to match a NopAction filter action is executed.

Substitute : The substitution filter allows the user to explicitly change the target and selector of a message, without using a meta filter. If the filter accepts, the specified substitutions are carried out, a SubstituteAction action. If the filter rejects, the message continues to the next filter, which corresponds to a NopAction action.

ErrorPropagation : If the error propagation filter accepts, it executes action PropagateAction, meaning that any encountered error is logged. In case the filter rejects, the next filter is evaluated, which corresponds to a NopAction action.

ParameterChecking : In case of acceptance, action CheckAction is executed. This action set the error variable. The message will be passed to the subsequent filter, if the filter rejects, a NopAction action is executed.

Table 5.2 summarizes the filters and filter actions described above.

Tabel 5.2: Filters mapped to filter actions

Filter	Accept Filter Action	Reject Filter Action
Dispatch	DispatchAction	NopAction
Send	SendAction	NopAction
Error	NopAction	ErrorAction
Meta	MetaAction	NopAction
Before	BeforeAction	NopAction
After	AfterAction	NopAction
Substitute	SubstituteAction	NopAction
ErrorPropagation	PropagateAction	NopAction
ParameterChecking	CheckAction	NopAction

We now translate these filter actions to our resource model. Table 5.3 shows this translation.

Table 5.3 presents the translation from filter actions of the predefined filter to the resource and operation model. Three filters of them execute some functionality that is implemented in the language of the base program. These three filters are Meta, Before and After, which are respectively mapped to filter actions MetaAction, BeforeAction and AfterAction. We are unable to extract the precise behavior of these three filter actions, since we do not reason about code in the language of the base program. For these filter actions we assume a worst case situation, i.e. all message properties in the scope of an advice are read and

Tabel 5.3: Effect of the current filter actions

Filter Action	Sender	Server	Target	Selector	Args	Return Value	Control Flow
DispatchAct.			write	write			return
SendAction	write	read write	read write	write			return
ErrorAction							exit
NopAction							continue
SubstitutionAct.							continue
MetaAction	read write	read write	read write	read write	read write	read write	continue return exit
BeforeAction	read	read	read write	read write	read write		continue
AfterAction	read	read	read write	read write	read write	read write	continue

written. Since a MetaAction can also affect the control flow, we assume that a MetaAction can continue, return and exit the control flow. It is however possible to override this by manually annotating the base program methods that are called by these filters with a behavioral specification.

Table 5.4 shows the two filters that are used in the ASML example. In section 4.6.2 this abstraction was explained in detail.

Tabel 5.4: Effect of the two example filter actions

Filter Action	arguments	errorvariable
PropagateAction		read write read read
CheckAction	read	write

The behavioral specification for filter actions are defined in an XML file. This file is input for the Compose* toolset to automatically detect behavioral conflicts among aspects. The module that reasons about behavioral conflicts is called the Semantic Reasoning Tool (SECRET). Listing 5.4 shows the behavioral specification in XML for the two filters from the ASML example; the other filters are also included.

```
1 <secret>
2   <resources>
3     <resource name="sender" alphabet="read,write"/>
```

```
4     <resource name="target" alphabet="read,write"/>
5     <resource name="selector" alphabet="read,write"/>
6     <resource name="args" alphabet="read,write"/>
7     <resource name="returnvalue" alphabet="read,write"/>
8     <resource name="errorvariable" alphabet="read,write"/>
9     <resource name="controlflow" alphabet="continue,return,exit"/>
10  </resources>
11  <filters>
12    ...
13    <filter type="ErrorPropagation">
14      <accept action="PropagateAction"/>
15      <reject action="NopAction"/>
16    </filter>
17    <filter type="ParameterChecking">
18      <accept action="CheckAction"/>
19      <reject action="NopAction"/>
20    </filter>
21  </filters>
22  <actions>
23    ...
24    <action name="CheckAction">
25      <operation name="read" resource="arguments"/>
26      <operation name="write" resource="errorvariable"/>
27    </action>
28    <action name="PropagateAction">
29      <operation name="read" resource="errorvariable"/>
30      <operation name="write" resource="errorvariable"/>
31      <operation name="read" resource="errorvariable"/>
32      <operation name="read" resource="errorvariable"/>
33    </action>
34  </actions>
35  <constraints>
36    ...
37    <conflict pattern="(write)(write)" resource="errorvariable" message="The previous
          value of the error variable is overwritten!" />
38    <conflict pattern="write$" resource="errorvariable" message="A written error variable
          must be read!" />
39  </constraints>
40 <secret>
```

Listing 5.4: Behavioral specification XML file

The XML file has a main tag called secret. This tag contains four sub tags:

resources : These describe which resources there are and what operations are in the alphabet of these resources (see section 4.5.2). Since their details have been discussed before, we will not explain these resources here.

filters : This states which filters are present in the system and what the accept and reject actions for these filters are. We have omitted the filter specifications for the common set of filters, since these can be derived easily from table 5.2.

actions : This maps filter actions to resource-operation tuples. These operations are only allowed to occur, if they are defined in the alphabet of the resource they operate on. We have omitted the mapping of the common filters,

these can be derived from table 5.3.

constraints : This defines all the conflict rules. Here we only show the two rules from our example. These rules are discussed in more detail in the next section.

5.5.2 Transformation

In the transformation phase we transform the sequence of filter modules to a representation that we can reason about and that incorporates the behavioral specification of the individual filters. To accomplish this, three steps have to be executed:

1. Transform a sequence of filter modules to a sequence of filters.
2. Transform a sequence of filters to a message flow graph.
3. Annotate the message flow graph with behavioral specifications.

5.5.2.1 Step 1 - Transform a sequence of filter modules to a sequence of filters

The first step is to transform the sequence of filter modules to a sequence of filters. The compositional operator between filters modules in Composition Filters is a sequential composition operator. As such we can assume the following:

$$\begin{aligned} &\mathit{FilterModulesToInputFilters} : \mathit{Class} \xrightarrow{m} \mathit{FilterModule}^* \\ &\qquad\qquad\qquad\qquad \rightarrow \mathit{Class} \xrightarrow{m} \mathit{Filter}^* \end{aligned}$$

$$\begin{aligned} &\mathit{FilterModulesToInputFilters}(\mathit{fms}) \triangleq \\ &\quad \textbf{for } \mathit{class} : \mathit{Class} \textbf{ in elems dom } \mathit{fms} \\ &\quad \textbf{do } \mathit{FilterInputSequenceMap}(\\ &\quad \mathit{class} \mapsto \textbf{for } \mathit{fm\text{-}seq} : \mathit{FilterModule}^* \textbf{ in elems } \mathit{fms}(\mathit{class}) \\ &\quad \textbf{do } \overleftarrow{\mathit{FilterInputSequenceMap}(\mathit{class})} \curvearrowright \mathit{fm\text{-}seq.inputfilters} \\ &\quad) \end{aligned}$$

$$FilterModulesToOutputFilters : Class \xrightarrow{m} FilterModule^* \rightarrow Class \xrightarrow{m} Filter^*$$

$$FilterModulesToOutputFilters(fms) \triangleq$$
for $class :\ Class$ **in elems dom** fms
do $FilterOutputSequenceMap($
$class \mapsto$ **for** $fm\text{-}seq :\ FilterModule^*$ **in elems** $fms(class)$
do $\overleftarrow{FilterOutputSequenceMap(class)} \frown fm\text{-}seq.inputfilters)$

We define two functions FilterModulesToInputFilters and FilterModulesToOutputFilters which take the map from the previous phase (see section 5.4.3), i.e. that maps classes to a sequence of filter modules. Functions FilterModulesToInputFilters and FilterModulesToOutputFilters both produce a map that maps classes to a sequence of input and output filters. This is accomplished by iterating over every shared join point, i.e. class, and concatenating ($\frown$) all enclosed filters in the sequence of filter modules. An overbar over a variable indicates that we use the previous value of that variable, a $\mapsto$ indicates a mapping relationship, and the notation $FilterInputSequenceMap(class)$ means that we want to get the element (in this case a sequence of filter modules) associated with this class.

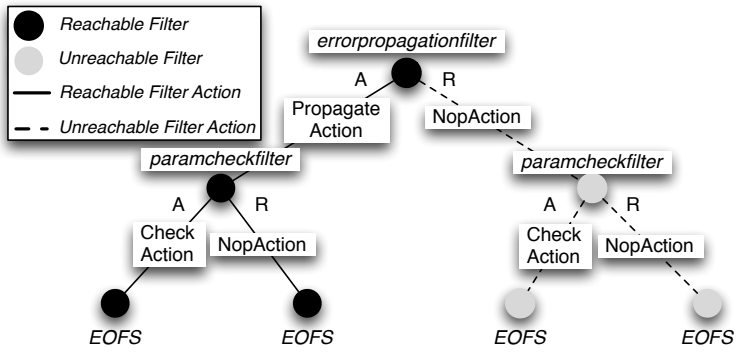
In the case of the ASML example map FilterInputSequenceMap looks like:

$CC.CX.FS \mapsto [errorpropagationfilter: ErrorPropagation;$
$paramcheckfilter: ParameterChecking]$

5.5.2.2 Step 2 - Transform a sequence of filters to a message flow graph

A message flow graph models the control flow information of the sequence of filters for a shared join point. This graph is not only used for behavioral conflict detection, but is also required for consistency checking, signature calculation and inlining of the filter code into the base program. This graph is thus not specific for behavioral reasoning. As mentioned in the previous section we distinguish between behavior that is filter type-specific and behavior that is filter instance-specific. We construct a filter action graph, named G_{action} and a filter evaluation graph G_{eval}. A message flow graph is the combination of both graphs: $G_{msgflow} = G_{action} \times \forall filters \cdot G_{eval}$. We first explain the filter action graph, next we explain the filter evaluation graph and show the complete message flow graph. We do not explain all implementation details of how such a graph is constructed, but only explain the message flow graphs themselves. For more detailed information see [dR07].

The first step is to translate the filters into a filter action graph. This graph is a binary tree, where the nodes are filters and the edges indicate either the execution of an accept or reject filter action. Figure 5.15 shows this graph for the ASML example.



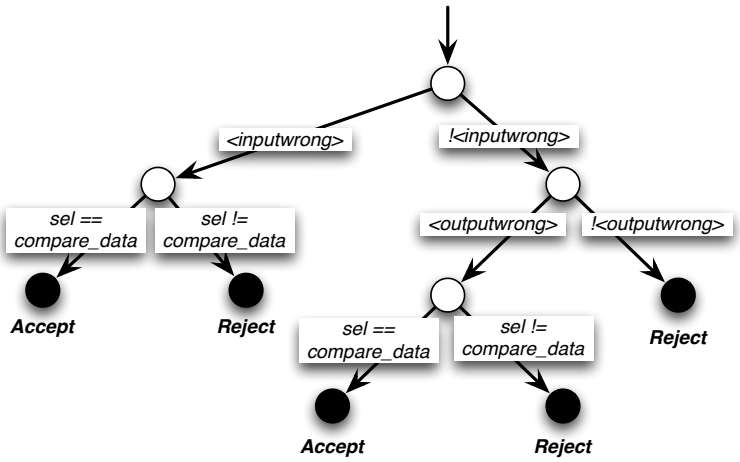
Figuur 5.15: Filter action graph (G_{action}) from the ASML example

A couple of observations can be made from figure 5.15 . The first is that the number of execution paths grows exponentially with the number of filters. Since filters can either accept or reject, we get a number of paths of at most 2^n, where n is the number of filters in the filter sequence. The second observation we can make is that not all paths are possible. For example. we know that filter errorpropagationfilter always accepts, due to its matching expression (`{ [*] }`), which is discussed in more detail shortly. In this case we see that the number of paths we have to analyze is only two as the other two paths are not reachable. In practice, there will probably always be a reduction in the number of paths. The number of filters will be also limited, since we know from experience that most filter modules contain on average three to four filters, and the number of filter modules per shared join point will also be limited, since the number of concerns will probably be limited.

An optimization from which our approach could benefit is to do local reasoning. All join points with the same combination of filter modules yield the same result for an analysis by SECRET. Therefore, we only have to analyze distinct shared join points. This optimization has not been implemented and is not discussed in our approach.

We now explain how the condition, matching and substitution parts in the

filter expressions affect the message evaluation. If we would not consider filter expressions and always assume the worst case, there would be many false positives. We represent the evaluation of the matching expression, `inputwrong || outputwrong => [*.compare_data]`, using a graph (G_{eval}). Figure 5.16 presents the filter evaluation graph for filter paramcheckfilter of concern ParameterChecking. We do not show the graph for filter errorpropagationfilter since it always matches.



Figuur 5.16: Filter evaluation graph (G_{eval}) for filter paramcheckfilter in the ASML example.

Initially, the start node in the graph has two edges. In case the condition inputwrong (we use the notation '<' and '>' to indicate a condition in the graph) is *true*, we try to match the selector of the current message to: *compare_ data*. If the match is successful, the filter will *accept* the message, and *reject* it otherwise. Another path can be taken when inputwrong yields false, and outputwrong yields true. As figure 5.16 shows, there are five ways for the filter to either accept or reject. Condition inputwrong is always read, whereas condition outputwrong is only sometimes read. Furthermore, we can assume that if the filter accepts, it will have read the selector of the message, and written resource errorvariable.

Once all behavioral and control flow information is gathered, we can create a so-called message flow graph. A message flow graph is a combination of G_{action} and G_{eval}. A message flow graph G_{mflow} is a directed acyclic graph and is defined as $< V, E >$, where:

V is a set of vertices representing the evaluable composition filters elements.

These can be filter actions, condition expressions, matching expressions and substitution expressions.

E : is the set of edges connecting the vertices, such that $E = \{(u, v) \bullet u, v \in V \wedge u \neq v\}$.

For each shared join point a message flow graph G_{mflow} is created. This graph is subsequently simulated to detect impossible or dependent paths through the filter set. It is out of the scope of this thesis to discuss the full implementation of this simulation (see [dR07]).

The general idea is that for each set of messages that have the same flow through the sequence of filters, we create a so-called message class. In the worst case the number of classes is equal to the number of individual messages that can be accepted by the filters. However, in general this number is smaller, since most filters will match the same set of messages. This transformation has been implemented in Compose*, and uses graph transformation rules to build up the message flow graph and to simulate the effect of messages on this graph.

We now show the message flow graph for the ASML example. These were the two filters in the ASML example:

```
1 errorpropagationfilter : ErrorPropagation = { [*] };
2 paramcheckfilter : ParameterChecking = {
3                         inputwrong || outputwrong => [*.compare_data] *.* }
```

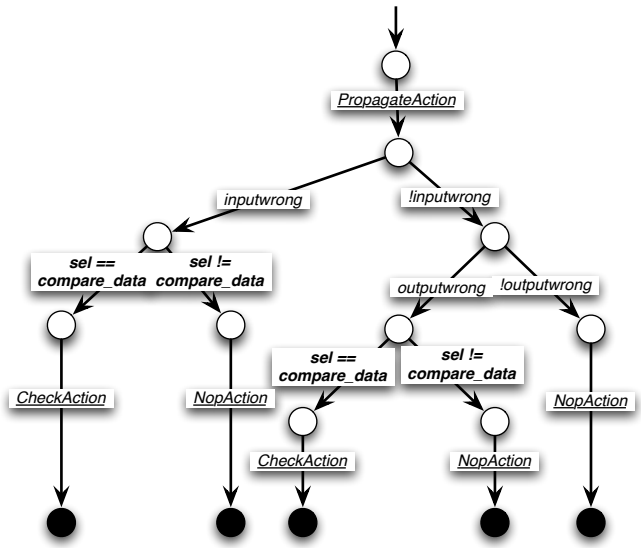
Figure 5.17 shows the message flow graph of the ASML example.

Message flow graph G_{mflow_asml} presents the control flow for the ASML example. The **bold** labels on edges refer to matching expressions. In this case these edges correspond to checking whether the selector is equal to compare_data. The underlined labels on edges refer to the execution of a particular filter action. In this case there are three filter actions, namely PropagateAction, CheckAction and NopAction. The label on edges that are not underlined or bold refer to the evaluation of a particular condition, in this case inputwrong and outputwrong. For more detailed information about this transformation we refer to [dR07].

5.5.2.3 Step 3 - Annotate the message flow graph with behavioral specifications

Now that we have a representation of the message flow through a filter set we can annotate the message flow graph with the behavioral specifications. This requires four transformations:

Condition expressions Each time a condition is evaluated, we translate this



Figuur 5.17: Message flow graph of the ASML example(G_{mflow_asml})

as a read on the resource representing this condition. In our example case we have two conditions inputwrong and outputwrong, which are both read.

Matching expressions Each time a message is matched, using either name or signature matching (the distinction is not relevant here), we interpret this as a read on either target, selector or both, depending on the exact matching expression. In the ASML example we only verify that the selector is equal to compare_data, we thus only read the selector. A matching expression is only evaluated if the related condition expression yields true.

Substitution expressions Similar to the matching expressions, we transform each substitution expression to a write on either target, selector or both, again depending on the exact substitution expression. In the ASML example, we do not use a substitution expression. A substitution expression is only executed if the related condition and matching expression match.

Filter actions This is the last transformation step. For each filter action in the message flow graph, we retrieve the behavioral specification, i.e. the sequence of resource-operation tuples for this action, and attach it to the appropriate edge. In the ASML example there are three filter actions, namely PropagateAction,

CheckAction and NopAction. The behavioral specifications for these three actions are repeated here:

```
1 PropagateAction ==> errorvariable: read;write;read;read
2 CheckAction ==> arguments: read, errorvariable ==> write
3 NopAction ==>
```

If we apply these four transformation rules on the message flow graph (G_{mflow}) we get the annotated message flow graph (G_{amflow}). G_{amflow} is a directed acyclic graph and is defined as: $< V, E, L >$, where:

V is a set of vertices representing the evaluable composition filters elements. These can be filter actions, condition expressions, matching expressions and substitution expressions.

E : is the set of edges connecting the vertices, such that $E = \{(u, v) \bullet u, v \in V \land u \neq v\}$,

L : is the set of resource-operation labels attached to the edges, such that
$L = \{(e, rsrcop) \bullet e \in E \land rsrcop \in ResourceOperations\}$
$ResourceOperations = Resource \mapsto Operation\text{-}\mathbf{set} \bullet$
$\forall rsrc \in \mathbf{dom}\ ResourceOperations \cdot$
$ResourceOperations(rsrc) \subseteq rsrc.alphabet$

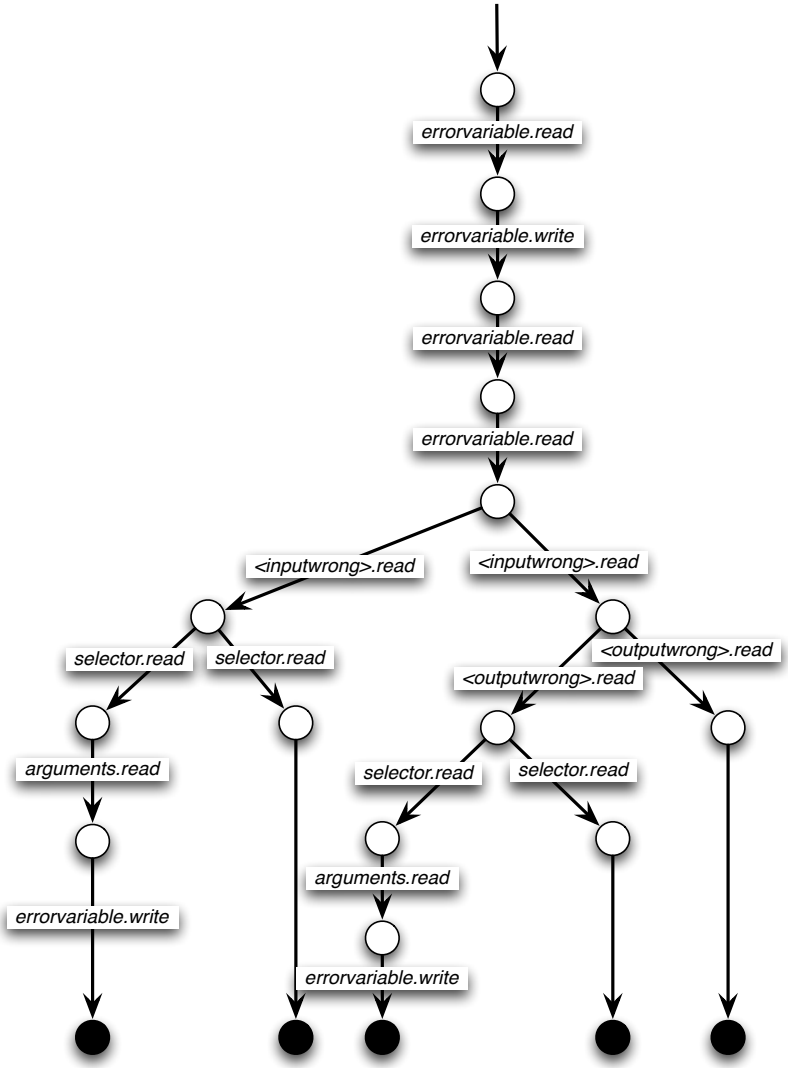
An annotated message flow graph G_{amflow} consists of a set of vertices, edges and a set of resource-operation labels. We constrain set L to only allow resource-operation tuples where the operations for a specific resource are in the alphabet of that particular resource.

Figure 5.18 presents the annotated control flow graph G_{amflow_asml} for the ASML example.

Figure 5.18 has a slightly modified structure than the original message flow graph G_{mflow_asml} in figure 5.17. The first edge is extended to accommodate the four transitions caused by the unfolding of filter action propagate. All filter actions have been rewritten as resource operation tuples, as have the condition and matching expressions.

5.5.3 Output

An annotated message flow graph (G_{amflow_asml}) is the output of this phase and serves as the input for the next phase, called Conflict Detection.



Figuur 5.18: Annotated message flow graph for the ASML example (G_{amflow_asml})

5.6 Conflict Detection Phase

5.6.1 Inputs

5.6.1.1 Annotated message flow graph

One input is an annotated message flow graph, called G_{amflow_asml} from the Abstraction phase.

5.6.1.2 Conflict Rules

The second input is a set of conflict rules. Similar to the previous section, we make a distinction between message property conflicts, control flow conflicts and other conflicts. For each kind of conflicts we have different conflict detection rules.

Message properties-related conflict rules These conflict rules apply to the following resources: sender, server, target, selector, arguments and returnvalue. We use extended regular expressions as defined by IEEE standard 1003.1[GI04] to specify the conflict patterns. These are the message properties-related conflict rules:

- *Conflict*(*data*): *read write*: In this case the read operations obtains a value that is no longer valid, since the resource is changed immediately afterwards. We assume here that the operations are caused by different aspects.
- *Conflict*(*data*): *write write*: In this case the effect of the first write operation is lost due to a subsequent write operation. We assume here that the operations are caused by different aspects.

Control flow-related conflict rules We now show which combinations of operations on resource controlflow yield a conflict:

- *Conflict*(*controlflow*): *return* $.^{+}$: If an advice returns, another advice which should be executed after this advice is never executed, hence if there are one or more other operations after a *return*, this will be signaled as a conflict.
- *Conflict*(*controlflow*): *exit* $.^{+}$: Similarly, if an advice terminates the execution, the advice which should be executed after this advice will never be executed. Hence if an *exit* operation is followed by one or more other operations, this will be signaled as a conflict.

Especially these generic rules are typically conservative, i.e. they aim at detec-

ting *potential* conflicts, and will also identify situations that in reality are not conflicting.

Conflict rules from the ASML example We also have two rules from the ASML example, we summarize these here:

- *Conflict*(*errorvariable*): (*write*)(*write*): The previous value of the error variable is overwritten.
- *Conflict*(*errorvariable*): (*write*)$: A written error variable must be read before the end of the operations sequence.

Once we have defined all the conflict rules we can transform them and match them against the annotated message flow graph.

5.6.2 Transformation

We transform the conflict rules to graphs, which are matched to the message flow graph. A conflict rule graph $G_{conflict}$ is similar to the annotated message flow graph, i.e. a directed acyclic graph and is defined as $< V, E, L >$, where:

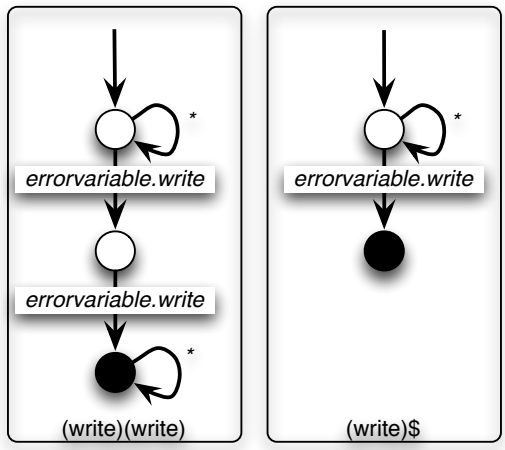
V is a set of vertices,
E : is the set of edges connecting the vertices, such that $E = \{(u, v) \bullet u, v \in V \wedge u \neq v\}$,
L : is the set of resource-operation labels attached to the edges, such that
$L = \{(e, rsrcop) \bullet e \in E \wedge rsrcop \in ResourceOperations\}$
$ResourceOperations = Resource \mapsto Operation$-**set** $\bullet$
$\forall rsrc \in \mathbf{dom}\ ResourceOperations \cdot$
$ResourceOperations(rsrc) \subseteq rsrc.alphabet$

Although this graph is similar to the annotated message flow graph, the semantics of both graphs is very different. Message flow graphs describe control flow behavior and conflict rule graphs describe conflicting situations.

We now show the two conflict graphs for resource errorvariable in the ASML example. We transform these two rules into conflict graphs G_{rule1} and G_{rule2}, as shown in figure 5.19.

Figure 5.19 shows the two conflict rule graph for the example. These graphs are equivalent to the final state automata representations of both regular expressions. We have added self edges that capture everything except the labeled outgoing transitions.

To determine whether a particular conflict rule graph matches a given annotated message flow graph, we intersect both graphs. A message flow graph is conflict-



Figuur 5.19: Conflict rules G_{rule1} (left) and G_{rule2} (right)

free if this intersection is empty. If the intersection is not empty, we have detected a conflict. A shared join point contains a conflict if and only if:

$$\exists g_{rule} \in \mathit{ConflictRules} \bullet g_{rule} \cap G_{amflow} \neq \{\,\} \tag{5.1}$$

For each conflict we have a corresponding path $P_{conflict}$, or a set of paths if there are more paths leading to the same conflicting situation. $P_{conflict}$ is a subgraph of G_{amflow}. To get all conflicting paths we remove $P_{conflict}$ from G_{amflow}, and intersect the new G_{amflow} with g_{rule}, until no more matches are found.

To illustrate how the process works we now transform G_{amflow} to a set of all possible traces through the graph. In the ASML example we have five traces. These five traces are listed here in terms of resource operation tuples.

- $Trace_1$: `errorvariable.read errorvariable.write errorvariable.read errorvariable.read <inputwrong>.read selector.read arguments.read errorvariable.write`
- $Trace_2$: `errorvariable.read errorvariable.write errorvariable.read errorvariable.read <inputwrong>.read selector.read`
- $Trace_3$: `errorvariable.read errorvariable.write errorvariable.read errorvariable.read <inputwrong>.read <outputwrong>.read selector.read arguments.read errorvariable.write`
- $Trace_4$: `errorvariable.read errorvariable.write errorvariable.read errorvariable.read <inputwrong>.read <outputwrong>.read selector.read`
- $Trace_5$: `errorvariable.read errorvariable.write errorvariable.read errorvariable.read`

```
<inputwrong>.read <outputwrong>.read
```

We can organize all operations per resource. The result is presented here:

- $Trace_1$:
  errorvariable : `read write read read write`
  <inputwrong> : `read`
  <outputwrong> :
  selector : `read`
  arguments : `read`
- $Trace_2$:
  errorvariable : `read write read read`
  <inputwrong> : `read`
  <outputwrong> :
  selector : `read`
  arguments :
- $Trace_3$:
  errorvariable : `read write read read write`
  <inputwrong> : `read`
  <outputwrong> : `read`
  selector : `read`
  arguments : `read`
- $Trace_4$:
  errorvariable : `read write read read`
  <inputwrong> : `read`
  <outputwrong> : `read`
  selector : `read`
  arguments :
- $Trace_5$:
  errorvariable : `read write read read`
  <inputwrong> : `read`
  <outputwrong> : `read`
  selector :
  arguments :

We can check whether the two regular expressions represented by G_{rule1} and G_{rule2} accept the traces presented above. There are two traces that are excepted by the rules:

- $P_{conflict1} = Trace_1$
- $P_{conflict2} = Trace_3$

5.6.3 Output

The output of the conflict detection phase is a set of paths through the filters that contain a conflict. Our implementation in Compose* produces an HTML file with the results from the analysis process. Figure 5.20 shows the report for the ASML example.

Figure 5.20 shows the conflicting situation for class CC.CX.FS. It states that for selector compare_data a conflict has been detected on resource errorvariable. It also shows the conflicting sequence of operations and the message related to this conflict rule. In addition, a trace of the filter sequence is produced, in this case errorpropagationfilter followed by paramcheckfilter. SECRET also checks the other possible orderings of filter modules, thus ParameterChecking.check followed by ErrorPropagationConcern.propagate. This does not yield any conflicts, and might be an indication that a filter module ordering specification is required to solve the problem. The current implementation does not report detailed paths: it does report all conflicts.

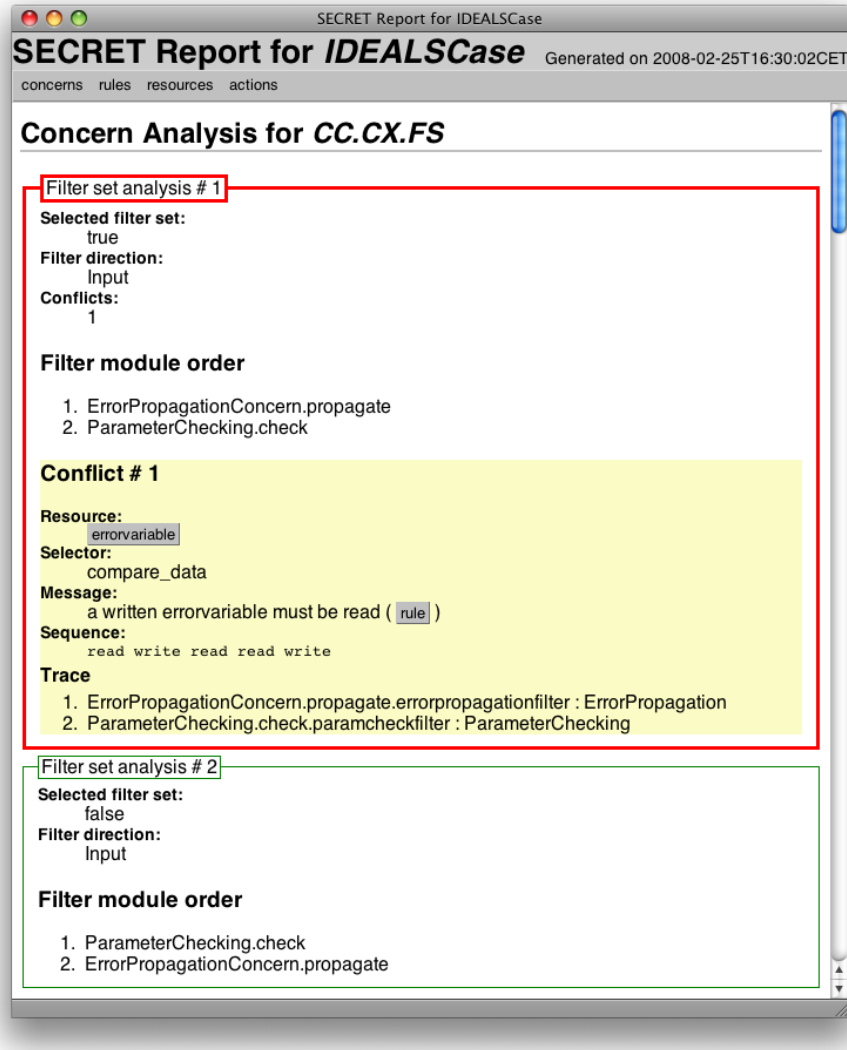
5.7 Discussion

We now discuss several remaining issues and possible extensions.

5.7.1 Generality of the approach and implementation

We can generalize our approach in several ways:

- *Can the approach be applied to other base languages?* The Composition Filters model is language agnostic. It only assumes a message, interaction or event to be present. If such messages or events exist, the Composition Filters approach can be applied. The behavioral conflict detection, as presented here, only deals with conflicts among concerns. Therefore, we do not make any assumptions about a particular base programming language.
- *Can the approach be applied to other aspect languages?* The approach presented in this chapter is Composition Filters specific. However, the general approach presented in the previous chapter can be applied to other aspect languages. This may require different resources, operations and conflict rules. Further, since we use the declarative properties of Composition Filters, we can automatically derive a part of the behavioral specification and accurately determine when conflicting situations occur. This may



Figuur 5.20: Conflict detection report

be more difficult for more expressive and Turing complete languages like AspectJ. One can automatically derive behavior specifications from advices in AspectJ by inspecting the usage of *thisJoinPoint* and of explicit context bindings. This does require advanced control and data flow analysis. For an aspect language like AspectJ, the set of resources, operations and conflict rules will be similar to the set presented here, i.e. the arguments and return value are also present in these languages.

- *Can the approach be applied to detect conflicts between aspect and base code or between composed base code?* The approach presented in this chapter is specific for the detection of behavioral conflicts among aspects. The model for expressing and detecting behavioral conflicts is generic and can be used to detect aspect and base code conflicts or composed base code conflicts. However, this would require behavioral specifications from the base program, and automatically deriving these is hard. However, if behavioral annotations of the program units are available, the approach can be applied.
- *Can the approach be applied to detect conflicts at the architectural level?* Our approach for detecting behavioral conflicts uses a resource-based abstraction of the behavior of aspects. As such, our approach can also be used for behavioral conflicts at an earlier phase in the software development life-cycle. In these earlier phases, we do not have implementation details, but we can specify the behavior abstractly using resource and operations. In this case, we can also use our resource-based approach. In [DBT08], we proposed the usage of our resource-based approach for detecting behavioral conflicts at the architectural level.

5.7.2 Complex behavioral specifications

In this chapter we have expressed the behavioral specifications as a sequence of operations on resources. The control flow of an advice can be more complex than just a sequence. Currently these more complex cases are serialized into worst case sequences. To accommodate more complex control flow behavior we can introduce a description of the behavior using an automaton. Consider for example the behavior specification of filter action PropagateAction. We specified this as:

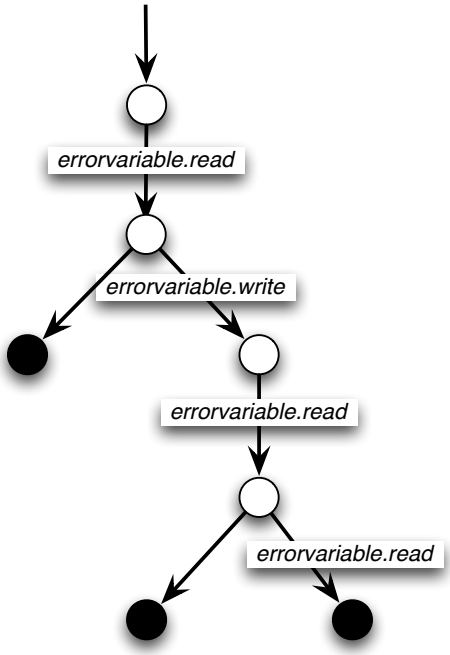
```
errorvariable.read;errorvariable.write;errorvariable.read;errorvariable.read
```

This was derived from the following code extract (listing 4.2).

```
1 ...
2   if (result == OK)
```

```
3   {
4     result = example_action1(...);
5     if(result != OK)
6     {
7       LogError(result);
8     }
9   }
10 ...
```

In reality, parts of this behavior are executed conditionally. To express such behavior, we can use automata. In the example, the automaton expresses a specification of the behavior including control flow (see figure 5.21).



Figuur 5.21: Complex behavior automaton

Figure 5.21 shows that there are three paths through the code. The impact of introducing automata instead of sequential behavior specification on the conflict detection process will be minimal. Consequently, there were will be more nodes and edges in the annotated message flow graph, which increases the computational complexity. However, it does not affect the ability to detect

conflicts.

5.7.3 Alternative conflict rule specifications

The rules we have presented in this chapter have been implemented using extended regular expressions as defined by IEEE standard 1003.1 [GI04]. These expressions are capable of detecting very complex combinations of operations on resources. Alternatively, we can use Linear Temporal Logic(LTL)[Pnu77] to express the conflict rules. LTL expresses properties on the states of a büchi automaton, which represents the execution of a program. LTL is usually used for expressing safety and liveliness properties. For example conflict rule: `(write)(write)`, can be translated to *write* **X** *write*, in words: write **next** write.

The operations on resources are limited to a finite alphabet. LTL formulas can be transformed to deterministic finite automata, as is done with extended regular expressions. As such, we can also use LTL formulas with minimal impact.

Computation Tree Logic (CTL) [CGP99] is used to verify properties along paths in a tree-like structure. CTL can verify properties for all paths ($\forall$) or for a specific one ($\exists$). These two quantifiers can be mapped to conflict and assertion rules. Assertion rules should hold for all paths through the annotated message flow graph. In contrast, there should be no path for which a conflict rule holds. Both CTL and LTL can be used together in CTL*. This could be an alternative rule expression language for detecting conflicts, and again the impact would be minimal.

Another possible extension of the approach is concerned with detecting behavioral interference when multiple resources are involved. It could be the case that the combination of sequences of operations on distinct resources could be conflicting. Currently, we only consider operation patterns on single resources. Detecting interference between resources requires prefixing the operations with the specific resources, and adjusting the conflict rules, to not only reason about operations, but also about the resources on which these operations are executed.

5.7.4 False positives and false negatives

False positives and false negatives can be caused by several reasons.

Firstly, if the behavioral specifications are erroneous. Section 4.8 discussed what behavior can be expressed. In the approach and implementation presented in this chapter, we assume that the behavior of filter actions and advice methods are

adequately described. A detailed analysis by the developer of a filter action or advice method is desired. However, since filters and filter actions are reused, we only need to describe the behavior once. If a developer introduces a new domain or application specific resource or operation, he or she must be aware that there are other resources, operations and conflict rules, and adjust his specifications to cope with the current set of resources, operations and conflict rules. Since we do not analyze the implementation of the filter action or advice method, a developer has to provide this information.

To aid in extracting behavioral specifications and keeping the specifications consistent, one can try to automatically derive these specifications. In [vO06], we explore the possibility to automatically derive behavioral specifications from base code. This relies on detailed control and data flow analysis and implementation details. For example, we analyze the base code for the usage of a context parameter such as JoinPointContext in Compose* and thisJoinPoint in AspectJ. These context parameters are usually the only way to query and manipulate message properties or other context information. As such we can automatically derive a behavioral specification. The work described in [vO06] has been executed by M. van Oudheusden, and directed and supervised by P. Dürr and L. Bergmans.

Secondly, the conflict rules may be too strong or too weak. In section 5.6.1.2, we proposed a set of conflict rules to capture conflicts on message properties within Composition Filters. These have been formulated such that they capture read and write conflicts between aspects. However, some read and write interactions are intended. But we need to information from a developer to determine this. In such a case a developer can exclude a particular rule for a resource or choose to ignore the error, since it is a desired interaction and not a conflict.

Thirdly, we may detect conflicts that cannot occur, since certain paths through the message flow graph are not possible. For example, in the current approach we assume that conditions can change during the evaluation of the filters. If two filters use the same condition, we assume that each filter can accept or reject independent of each other. We do not assume that if the first filter accepts, the second filter accepts as well.

Finally, certain conflicts cannot be detected statically. Filter specifications may have dynamic elements like conditional and dynamic execution and superimposition. Statically, we can detect the worst case. Section 5.8 discusses these dynamic elements and proposes an extension to the static approach.

5.7.5 Computational complexity

We split the discussion about the computational complexity into two separated discussions. The first discussion details the complexity for phases *Composition* and *Abstraction*. The second discussion details the complexity for phase *Conflict Detection*.

Composition and abstraction The described message flow graph (see section 5.5.2.2), is an abstraction of the message flow graph in the implementation. There is no fundamental difference, however the actual implementation has more nodes and edges than the examples presented here. These extra nodes and edges are required and introduced by the graph transformation rules used to build the message flow graph. For example, in the implementation there is a node called True that represents the condition part of filter `errorpropagationfilter : ErrorPropagation = { [*] }`, since the transformation rules use the canonical form: `errorpropagationfilter : ErrorPropagation = { True => [*.*]*.* }`. These nodes have been excluded in the example graph in figure 5.17.

The implementation starts with an Abstract Syntax Tree (AST) representation of the filters and transforms this to a flow chart. Then control flow edges are added between the elements in the AST. The resulting flow chart is simulated to achieve the execution model of the filters. During this simulation the control flow paths of all distinct messages are calculated.

This entire process is executed using the Groove toolset [Ren]. In [dR07], this entire process in detailed. We use graph transformation rules to transform the AST to the flow chart. Each rule results in a set of transformation steps. The transformation steps for each rule are present in table 5.5. The work described in [dR07] has been executed by A. de Roo, and directed and supervised by P. Dürr, L. Bergmans and T. Staijen.

Tabel 5.5: Computational steps when transforming an AST to a flowchart, from [dR07].

Rule	#Steps
Join Point rule	$\#FilterModules + 4$
SI rule	$5 \cdot \#FilterModules$
Filter module rule	$\#Filters + 2 \cdot \#FilterModules$
Filter rule	$2 \cdot \#FilterElements + 10 \cdot \#Filters$
Filter element rule	$9 \cdot \#FilterElements$
Matching pattern rule	$2 \cdot \#MatchingPatterns + 4 \cdot \#FilterElements$
Filter Action rule	$8 \cdot \#Filter$

The total number of transformations is $8 \cdot \#FilterModules + 19 \cdot \#Filters + 15 \cdot \#FilterElements + 2 \cdot \#MatchingPatterns + 4$. All the Composition Filters elements are nodes in the AST. The number of nodes in the AST is linear with the number of filters on a join point. The time complexity of the algorithm to transform the AST into a flowchart is $O(\#Nodes)$.

Next, we have to simulate the execution of the message flow graph. In this simulation all classes of messages are taken as inputs. A so-called message-class is a set of messages (target and selector tuples) that exhibit the same flow behavior through the sequence of filters. The number of message classes is usually a lot smaller that all individual messages, but in the worst case the same. In such a worst case, we can state that the number of states in the resulting execution space of the simulation corresponds the number of nodes in the message flow graph times the number of message classes. As such we can define the time complexity of simulating the execution, as follows: $O(\#States) = O(\#Nodes \cdot \#Message)$.

The number of messages is related to the number of matching parts, and thus related to the size of the flow chart. For the maximum number of messages it holds that $O(\#Message) = O(\#Nodes^2)$, since, the possible messages are constructed by taking the cross product between the set of targets and the set of selectors in the matching and substitution expressions. The total complexity of the simulation is: $O(\#States) = O(\#Nodes \cdot \#Nodes^2) = O(\#Nodes^3)$. This is a worst case situation, since in practice the number of possible message classes is limited and thus the number of flow nodes in the graph is also limited.

The total time complexity of phases Composition and Abstraction is: $O(\#Nodes) + O(\#Nodes^3) = O(\#Nodes^3)$. As such, it can be concluded that the filter reasoning algorithm has polynominal time complexity.

Conflict Detection Each conflict rule is converted to an automaton and is intersected with the automaton that represents the execution model. The time complexity for conflict detection is $O(\#States(FlowChart) \cdot \#States(ConflictRuleAutomaton) \cdot \#PossibleResourceOperationsTuples)$. The number of possible resource-operation tuples are constrained by the finite and limited alphabet of resources. The detection algorithm is thus linear in the size of the execution model as well as linear in the size of the conflict rule.

The message flow analysis is always executed for each join point, since there are other modules that also rely on the message flow graph, e.g. for calculating the signatures of classes on which filter modules are superimposed. We could implement an optimization that only analyzes unique shared join points. Then, even if the number of shared join points is equal to the number of join points, we only have to analyze the number of distinct filter module orderings on those

distinct join points, since we do not consider the base program. As such we only introduce a small linear overhead for the detection of behavioral conflicts. For all the details we refer to chapters 5 and 6 in [dR07].

5.7.6 Output and returning filters

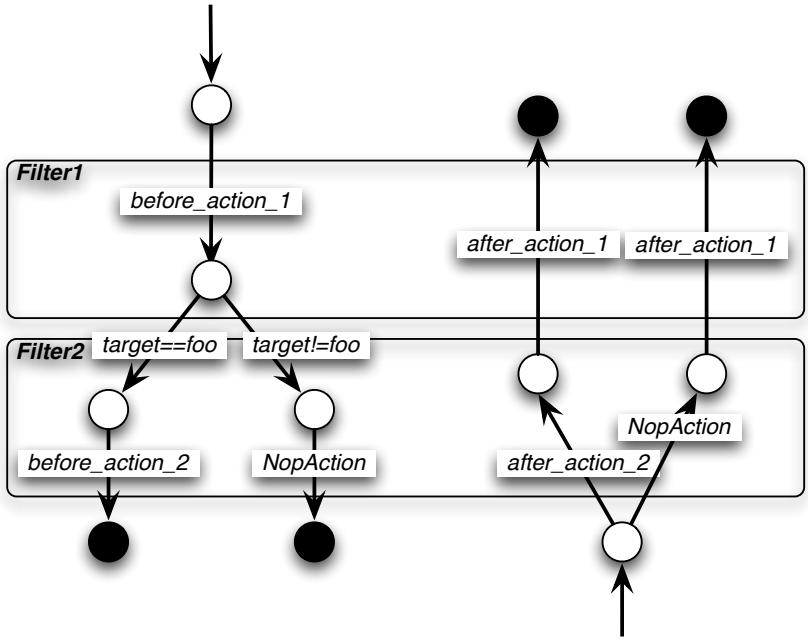
Input or output filters can not only execute behavior before the execution of the join point, but also after the execution of the join point, namely on the return of a message. The filter actions that are executed on the return of messages depend on the condition and matching expressions of the filter. These are evaluated when messages are received. If a filter executes an accept filter action for an incoming message, the accepting return action also executes, when the message returns. The same situation holds for reject actions. Condition and matching expressions are thus not re-evaluated when messages return. Also, the order in which the returning filter actions are executed, is the reverse order of the incoming filter actions. The effect of this choice is that we have a simplified message flow graph on the return of messages, since condition and matching expressions are removed.

Figure 5.22 shows two message flow graphs. one for incoming messages and another for returning messages.

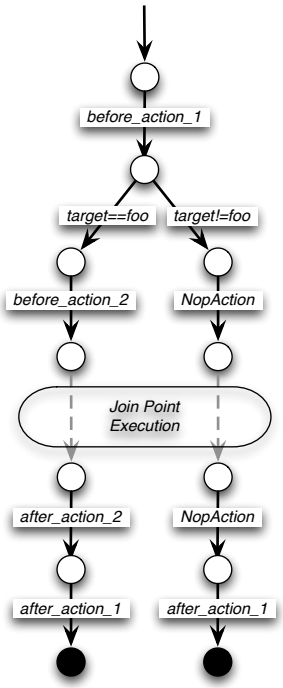
The example that we use here has two filters. The first filter always accepts. We also assume that both filters execute some behavior before and after join point execution. The graph on the left represents the message flow of an incoming message. This has two filter actions before_action_1 and before_action_2. We only execute the second filter action if the target of the message is equal to foo. The graph on the right represents the message flow for returning messages. This is very similar to the incoming graph, but all condition and matching expressions have been removed. Both these graphs can be annotated with behavioral specifications. Since we do not consider conflicts within one filter modules, we can analyze the graphs individually. The complexity of analyzing returning message flow graphs is the same or smaller than the complexity of incoming message flow graphs, since the returning message flow graphs do not have condition and matching expressions.

Our approach could be applied to reason about behavioral conflicts between incoming and returning behavior. In such a case we can create a composed message flow graph (see figure 5.23).

In this case there is a dependency between before join point execution filter actions and after join point execution filter actions. This dependency is clearly



Figuur 5.22: Incoming message flow graph (left) and returning message flow graph (right)



Figuur 5.23: Composed before and after message flow graphs

visible in figure 5.23. If the left branch is taken we execute two returning filter actions after_action_2 and after_action_1. If the right branch is taken, we also execute two returning filter actions, in this case NopAction and after_action_1. The composed graph can be annotated with behavioral specifications and the conflict detection can be applied on this composed graph.

In this chapter we only showed the conflict detection approach for input filters. Our approach can be applied equally well to output filters. The same composition rules for input filters apply to output filters. Therefore, the analysis process does not change.

5.7.7 Conflicts within filter modules

We can distinguish between conflicts caused between filter modules and within a single filter module. We assume that the implementation within a single filter module is correct. Within a filter module all filters are clearly visible, as such we assume that any interaction between these filters is intended. Filter modules on the other hand are independently developed and a developer cannot determine easily whether his filter module conflicts with others. As such any interference detected between filter modules is considered a conflict.

Currently we do not distinguish between behavioral conflicts that occur within a filter module or conflicts that are between different filter modules. We could distinguish between the conflicts and show a *warning* to the user if a conflict rule matches within a filter module, and issues an *error* if a conflict rule matches between filter modules.

5.8 Runtime conflict detection

This section discusses the need for a runtime extension to the described static approach. It also presents a possible implementation approach of such an extension in Compose*. This allows us to reason efficiently about the behavior of aspects at runtime. It also enables us to detect behavioral conflicts with limited overhead at runtime.

5.8.1 An example conflict: Security vs. Logging

Assume that there is a base system which uses a *Protocol* class to interact with other systems. Class *Protocol* has two methods: one for transmitting

data, *sendData(String)* and another one for receiving data, *receiveData(String)*. Now imagine that we would like to improve the security of this protocol. To achieve this, we encrypt all outgoing messages and decrypt all incoming messages. We implement this by superimposing an *encryption* advice on the execution of method *sendData*. Likewise, we superimpose a *decryption* advice on the execution of method *receiveData*. Now imagine, a second aspect which traces all methods including its arguments. The implementation of the tracing aspect uses a condition to dynamically determine if a method should be traced, because tracing all methods is not very efficient.

The two advices are superimposed at the same join point, in this case *Protocol.sendData*[1]. Now assume that we want to ensure that no one accesses the data before it is encrypted. This constraint is violated if the two advices are ordered in such a way that advice *tracing* is executed before advice *encryption*. In this way the log file can contain "sensitive" information.

We can make two observations. The first is that there is an ordering dependency between the aspects. The second observation is that, although this order can be statically determined, we are unsure whether the conflicting situation will even occur at runtime, because advice *trace* is conditionally executed.

We now show result of superimposition of aspects Encryption and Tracing. The result is the following composed filter sequence on method *Protocol.sendData*:

```
1 trace : ParameterTracing = { ShouldTrace => [*.*] };
2 encrypt : Encryption = { [*.sendData] }
```

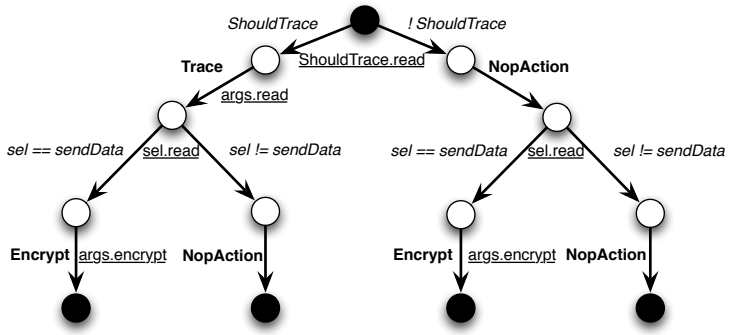
Listing 5.5: Composed filter sequence for join point *Protocol.sendData*.

Filter trace traces all parameters and return value at the start and end of a method execution. Filter encrypt subsequently secures the data being sent. The filter sequence presented in listing 5.5 can be translated to the annotated message flow graph in figure 5.24.

This graph is a simplified version of the actual graph, for readability purposes. The *italic* labels on the transitions are evaluations of the conditions (e.g. *ShouldTrace*), and the message matching, e.g. *message.sel(ector) == sendData*. The bold labels on the transitions show the filter actions. The underlined labels are resource-operations tuples, corresponding to the evaluation of the conditions, matching parts and the filter actions.

In this case we specify the following conflict rule: `Conflict(arguments): (read)(encrypt).` This rule states that it is not allowed to read the arguments before they are

[1]Here, we only focus on join point *Protocol.sendData*, but a similar situation occurs for join point *Protocol.receiveData*.



Figuur 5.24: Annotated message flow graph representation of the filters in listing 5.5

encrypted. In figure 5.24, we observe that the left most path matches the conflict rule.

Now let us elaborate on this conflict a bit more. In the example we use two filters, and one of these filters uses a condition. Condition *ShouldTrace* is used to determine to trace this method or not. We assume that whether this condition is true or false depends on certain runtime information. Statically, we observe that there is a possibility of a conflict, as we consider both the true and false values of the condition. This enhances our ability to detect behavioral conflicts but it also introduces possible false positives. Such a condition can always yield a false value, i.e. no methods should be traced. It thus requires dynamic monitoring to determine whether such a conflict actually occurs at runtime. We prefer to detect these conflicts statically, since that does not impact runtime performance and can guarantee that an application is indeed correct before deploying.

5.8.2 Limitations of static checking in AOP

This section discusses two AOP constructs that limit the ability to statically reason about behavioral conflicts.

5.8.2.1 Dynamic weaving

There are AOP approaches which employ dynamic weaving or proxy-base constructs to instrument an application. Although this provides some unique features over statically based weaving, it does present difficulties when statically

reasoning about behavioral conflicts at shared join points. One such difficulty is that not all shared join points are known statically. As such, it becomes hard to know which advices are imposed at a shared join point.

An example of such a construct is *conditional superimposition* found in Composition Filters, see listing 2.4 in section 2.3.4.6. We can assume a worst case situation for dynamic weaving approaches. We assume that all advices can be composed with any other advice. However, this can result in a large set of possible orders and combinations of advices that have to be checked. This worst case scenario is only an option when all aspects are known.

5.8.2.2 Dynamic advice execution

Most AOP approaches support conditional or dynamic properties in either pointcut or advice language. Examples of such constructs are the *if(...)* pointcut in AspectJ and *conditions* in Composition Filters. Even though all shared join points are known, not all possible combinations of advice may occur at runtime. This can depend on runtime state. In the example, we use condition *ShouldTrace* to determine whether to trace or not. At runtime this condition can be true or false. In our static conflict detection approach, we consider all possible values of conditions.

5.8.3 Conflict detection at runtime

As motivated by the previous section, we would like to extend our work to also capture behavioral conflicts at runtime. A naive implementation would be to simply instrument all advices and monitor all join points dynamically. However, we can reason more efficiently. In section 5.6, we stated that for each possible conflict we get a conflicting path called $P_{conflict}$. Each edge in this path has a set of labels attached to it, which represents the corresponding resource-operation tuples.

The set of conflicting paths is most likely smaller than the set of all possible paths. We only have to monitor those paths which are conflicting for a specific resource, and of these paths only the paths that contain dynamic elements.

To informally outline the runtime extension, we use the example conflict, as presented earlier. In figure 5.24 we saw that the left most path was a conflicting path. This full path is `ShouldTrace.read, args.read, selector.read, args.encrypt.` However, only part of this path is conflicting with our requirement. In this case: $Conflict(args)\colon(read)(encrypt)$. This conflict rule only limits the usage

of operations for resource *args*. We can thus reduce the conflicting path to: `args.read, args.encrypt`. Where `args.read` is caused by the execution of filter action Trace, and `args.encrypt` is caused by filter action Encrypt. We only have to monitor the execution of these two filter actions to determine whether the conflict occurs or not. In this case, even the execution of filter action Trace is sufficient, but this is not true in the general case. There are cases where one has to monitor the evaluation of conditions, message matching expressions and message substitutions.

5.8.3.1 Instrumentation

To be able to monitor the system while running, we have to inject monitoring code inside the advices. We assume that all code will be passed through the Compose* compiler. Our compiler will inject the monitoring code in the appropriate places. This ensures that the executing code will emit updates to the monitor. The next section provides more details about this monitor.

5.8.3.2 Analysis process at runtime

There are multiple steps involved in checking at runtime for a behavioral conflict. Our runtime extension uses a monitor to monitor the execution at runtime. The monitor is instantiated each time a join point is executed. Alternatively, we could have implemented the monitor as dedicated filters, this is considered future work. This monitor contains the following elements:

ConflictingResources is the set of resources which should be monitored, where $ConflictingResources \subseteq Resources$,

OperationSequence(rsrc) is the sequence of operations carried out on resource *rsrc*, where $\forall rsrc \in ConflictingResources \cdot OperationSequence(rsrc) \subseteq Alphabet(rsrc)$,

ConflictRules(rsrc) is the set of conflict rules for resource *rsrc*.

Now that we have defined the monitor, we define three phases to be able to reasoning about behavioral conflicts at a shared join point at runtime.

1. **Initialization**: At the start of the first edge in a conflicting path, we initialize the monitor. This monitor is responsible for keeping the state of the resources during the execution of this join point. It keeps track of all operations that are carried out on the resources. If an operation is carried out on a resource which does not exist, this resource is created. In our running example the initialization is done before the filter action *Trace*

or the first *continue* action is executed. The monitor also initializes the operation sequence for resource *args* to $OperationSequence(args) = [\,]$.

2. **Execution**: As the program execution continues along the edges of the paths, we execute the operations on the conflicting resources. These are carried out on the monitor, and this monitor will update its state accordingly. The execution of the operations has to be done just after the corresponding filter actions and conditions have been executed or evaluated. In the example, the execution step is carried out if the edge with label *args.read* attached is taken. This corresponds to the execution of operation *read* on resource *args*. The result is $OperationSequence(args) = [read]$. The execution step is also done when the edge with label *args.encrypt* attached is taken. This corresponds to the execution of operation *encrypt* on resource *args*, resulting in $OperationSequence(args) = [read\ encrypt]$.
3. **Evaluation**: If we reach the end of the execution path, we have to signal the monitor to verify whether the conflict rules match on the given execution path. If these rules match, we have encountered a conflict and the user will be signaled, e.g. via a message or exception. At the end of a join point we verify that: $\forall conf_{rule} \in ConflictRules(rsrc) \cdot$ $conf_{rule} \cap OperationSequence(rsrc) \neq \{\,\}$. Section 5.6 explained this in more detail.
   In the example case, this will occur after the edge that is labeled *args.encrypt* is taken. It this case, rule $Conflict(args)\colon (read)(encrypt)$ accepts $OperationSequence(args) = [read\ \ encrypt]$, and the conflict is detected.

The above process has to synchronize all conflict path thus, it should monitoring is started at the beginning of the first conflicting path. Similarly, at the end of the execution the evaluation phase has to be performed. To reduce the complexity, we could initialize the monitor at the start of the join point. Similar, we could verify the conflict rules at the end of the execution of the join point. However, this might impose a larger runtime performance hit.

Another option would be to verify the rules continuously. This would provide possibly earlier detection of the conflict. However, the runtime performance might also decrease, due to the abundance of verifications. The runtime extension presented in this chapter has been partially implemented in Compose*.

5.9 Related Work

Aspect interference has been stated as one of the key issues that still remain in the field of aspect-oriented software development [SR06, DFS05].

Our approach to behavioral conflict detection is considered model checking, pioneered by E. Clarke, E. Emerson, J. Queille, and J. Sifakis. Model checking techniques can verify certain properties (usually expressed in a temporal logic language) of programs. They can automatically check if a finite state transition system (modeling the program) conforms to a given state or event property. In our approach we model the behavior of advice as (abstract) resources and operations, and verify that properties, i.e. conflict rules, hold for the composed behavior of advices. In [HD01], Hatcliff and Dwyer, discuss four problems that are preventing model-checking technology from being applied successfully:

The state explosion problem : the exponential increase of the state space as the complexity of the modeled system increases. To prevent this, aggressive abstractions have to be made to produce traceable models. In our approach we implemented an abstraction mechanism, through the use resources and operations. Also, the use of Composition Filters restricts the possible compositions and thus reduces the state space as well.

The model construction problem : the gap between the artifacts produced by software developers and those accepted by the verification tools. This is still a problem in our approach. One has to write the behavioral specification next to writing the implementation of filters and advice. However, we have shown that these can be reused to large extent and can partially be automatically derived. The behavioral specification must be consistent with the implementation for successful conflict detection, keeping the implementation and specification consistent can be hard, but can possible be verified.

The requirements specification problem : the specification form of most temporal specification languages can be difficult to use, read and debug. This is also a problem that is not solved by our approach. However, since we are not tied to any particular conflict detection rule language, an elegant and more user-friendly alternative can be used. The use of regular expressions may appeal to more people than temporal logic.

The output interpretation problem : when a property fails, a model checker usually provides a counter example. This allows the developer to examine the violation in detail. However, these traces can be very long and are expressed in low-level model representations. In our implementation we provide a complete overview of which rules were violated and caused by

which composition of filter modules. Our traces tend to be shorter since we are reasoning locally instead of globally.

A lot of work has been conducted on the categorization of aspects. Categorizations can be used to define which combinations of categories can be considered harmful. These categories have been discussed in the previous chapter, section 4.4. Here we only focus on work that deals with aspect interference.

Krishnamurthi et. al. propose one such approach in [KFG04]. The paper considers the base program and aspects separately. The authors discuss that a set of desired properties, given a pointcut descriptor, can be verified by checking the advice in isolation, thus providing modular reasoning. The paper focuses on ensuring that the desired properties are preserved in the presence of aspects, in other words, the situation where applying aspects causes the desired properties of the base system to be invalidated. The paper only considers aspect-base conflicts and not conflicts between aspects.

In [KK08] and [GK07], Katz et. al. propose an approach to use model checking to verify aspects modularly.

The authors create a generic state machine of the assumptions of an aspect. If the augmented system, this is the state machine with the aspect applied, satisfies certain desired properties, then all base systems satisfying the assumptions of the aspect will satisfy the desired properties. The base system is expressed as a state machine, which represents the execution of the base program. Aspects are also expressed as state machines, if the assumption, i.e. the pointcut, of the aspect matches the base system, the base system is augmented with the state machine representing the execution of advice. Advice can be either *before* or *after*, where *around* advice is split into *before* and *after* advices.

The authors identify three kinds of properties that should hold for a system:

Safety properties describe assertions that should hold for all states of all executions of a program.

Liveness properties describe properties that should hold at the end of all executions of a program.

Existence properties describe properties that should hold for a particular execution of a program.

The authors also map aspects into three categories:

Spectative aspects can only read certain properties of the message or variables in a base system. They cannot alter the control flow, nor can they change the value of message properties or variables in a base system. It is of course allowed to change variables of the aspect itself. An example is the tracing

aspect, as described in section 3.1.

Regulative aspects can alter the control flow a base system. Aspects can choose to delay, restrict or prevent the execution of operations in base system. Examples are synchronization and authorization.

Invasive aspects can change message properties and variables in a underlying base system. There are two types of invasive aspects. *Weakly invasive* aspects return to already existing states in the base program. *Strongly invasive* aspects create new states in the base program.

The authors emphasize that automatically determining whether an aspect is spectative, regulative or invasive is far from trivial, and requires detailed control and data flow analysis of program code to derive such a classification.

In [Kat06], the authors use the kinds of properties and aspect categories to define aspect interference. Spectative aspects can be woven in any order without affecting safety, liveness and existence properties. The order of application of regulative and invasive aspects matters. As such the resulting augmented system might have changed in such a way that the assumptions of another aspect are not fulfilled anymore. For example one aspect could prevent the execution of other aspects or base program.

A major benefit of their approach is that it does not discriminate between conflicts caused by base-aspect and aspect-aspect interaction. Neither do they require the notion of a shared join point. Both assumptions and effects of aspects are expressed as temporal logic formulas. Also, the approach uses tableaus for representing from specific bases system. The authors show that if a tableau satisfies the assumptions of an aspect, then for all base systems that satisfy the assumptions of the tableau, the assumptions of the aspect are satisfied. The authors also prove that aspects only need to be verified pair-wise. If one can proof that the composition of two aspects is sound, then the composition of another aspect with these two other aspect is sound, assuming the pair-wise composition of the three aspects is sound. This is an elegant solution for addressing a possible state explosion.

In [KK08], the authors mention the use of Bandera [HD01] to automatically create the automaton representing the base system and advices. The authors partially address some of the four problems mentioned in [HD01]. The state explosion problem is reduced by the authors, through the use of a tableau. The model construction problem does not seem to be addressed by the authors, since they do not offer any means of abstraction, as we do in our conflict detection approach. The requirement specification problem, the authors use LTL for expressing the assumptions and guarantees of advice, in practice it is difficult for developers to accurately express complex properties. The output interpretation

problem is addressed in section 6 in [KK08]. The authors outline a process for determining which advice is responsible for the interference.

In [SR06], Staijen and Rensink model –part of– the Composition Filters behavior with graph based semantics. This approach also detects aspect-aspect conflicts. One step of the analysis is to construct a state space representation of the execution of the composed filter sequence at a shared join point. The authors propose an interference detection approach based on the ordering of filter modules. The use the following definition of a conflict: the order in which the filter modules are superimposed should not make any observable differences in the resulting state spaces from these different order filter modules. If the order of superimposition matters, an interaction among aspects is detected. There are three key differences with our work.

- The authors only detect conflicts related to control flow. They only model the control flow semantics of filter actions, not the behavior of the filter actions themselves.
- Secondly, they detect interactions rather than interference. If the order of application matters, then the program is considered ambiguous and as such conflicting. However, there might be a harmless or desired interaction between the aspects, there is no way to define what is desired and what isn't. This may lead to a lot of false positives.
- Thirdly, the approach does not allow the introduction of domain or application-specific information, such as specific conflict rules.

In several papers (e.g. [DFS04], [DFS05] and [SDMF+06]), Südholt et. al. present an event-based AOP technique to detect shared join points, based on similarities in the crosscut specification of the aspects involved. The approach does not consider the semantics of the advice that is inserted, it just considers the presence of a shared join point to be an interaction. The authors define an interaction as follows: two distinct aspects are said to interact when they match the same join point. Two aspects are independent if their superimposition specifications never match the same join point. Independence of two aspects ensures that their parallel composition is well-defined: they can be woven in any order. Dependent (i.e. interacting) aspects require the programmer to resolve the interactions by changing aspects or making the composition ordering more precise.

Program slicing is another approach to detect state based *interactions* among aspects. In [BCM05] Balzarotti et. al. present an approach to slice AspectJ woven code. First a slice of the woven program for one aspect is created, and subsequently the slice of the woven program for another aspect. If the slices intersect, then the aspects interact. The approach is not only capable

of detection interaction between aspects, but also between aspects and a base program. It supports the detection of conflicts that are due to side affects of advice. However, the approach is unable to determine whether an interaction is desired or undesired, hence it does not provide interference detection.

In [PDS05], Pawlak, Duchien and Seinturier present a language called *CompAr*, which allows the programmer to specify a set of execution constraints over the advice. The approach also provides an abstraction of the implementation language. This technique also analyzes interactions of aspects at shared join points. The *CompAr* compiler verifies whether the execution constraints hold for that given abstract specification. The work focuses on determining the correct order of composition given the execution constraints.

The notion of using resources and operations on these resources to model dependencies and conflicts has already been applied in several specific fields in software engineering, e.g. for synchronization constraints [Ber66] and for transaction systems [LMWF93].

In [EMK+06], Eichberg et. al., use resources and operations to model the dependencies between static analysis tasks in a build system. The paper also uses resources to represent build artifacts and analysis results. It proposes to model the dependencies between these artifacts as operations, e.g. *reads* and *maintains*. A scheduler is used to create a valid schedule. The goal is to define analysis tasks independently from each other. The goal of the approach is different since the authors are mainly interested in finding a valid order of the analysis tasks and to run these tasks within an incremental build process.

5.10 Conclusions

This chapter presents a detailed instantiation for the resource-based conflict detection approach that was presented in the previous chapter. We have used the Composition Filters approach to automatically derive parts of the behavioral specifications. The parts that are not automatically derived, have to be specified explicitly by the concern or filter developer. Once this information is present we can automatically detect behavioral conflicts and also indicate exactly when such a conflict can occur.

The contributions of this chapter are:

- An analysis of possible behavioral conflicts among filters in Composition Filters. This analysis can also be reused for detecting behavioral conflicts in other AOP approaches.

- A detailed discussion about the instantiation of our approach for detecting behavioral conflicts in Composition Filters.
- Implementation details explaining how the conflict detection approach has been implemented in Compose*.
- A discussion that motivates that through a careful language design with declarative features, automated reasoning about the (composition of) behavior of aspects becomes feasible.
- A discussion of several extensions of the approach, for example, to detect conflicts at runtime.

The current and earlier versions of the conflict detection approach applied to Composition Filters, have been published in [DSBA05], [DBA06], [DBA07a] and [DBT08].

Extending Composition Filters for improved Reasoning

6

This chapter discusses some constructs of Composition Filters that hinder manual and automated reasoning. We explain why these constructs reduce the ability to reason. We present novel extensions to Composition Filters, to improve the ability to reason.

6.1 Splitting Filter sets

The goal of this section is to explain the issues with the current *coarse- grained* filter sets, and to present a more *fine-grained* model that addresses these issues. We first show one possible implementation of concern *Tracing*, as presented in chapter 1. Next, we discuss the limitations of this implementation and propose an alternative implementation. We discuss the limitations with this alternative design. Subsequently, we propose a more fine-grained model and discuss the benefits of this model. We also discuss some remaining issues and show how we addressed them.

6.1.1 Initial Tracing Implementation

The first implementation uses a dedicated filter type to implement tracing. We first present the implementation of concern Tracing in listing 6.1.

```
1  concern Tracing
2  {
3    filtermodule TracingModule
4    {
5      inputfilters
6        tracing : TracingFilter = { [*.*] }
7    }
8
9    superimposition
10   {
11     selectors
12       sel = { Class | isClass(Class) };
13     filtermodules
14       sel <- TracingModule;
15   }
16 }
```

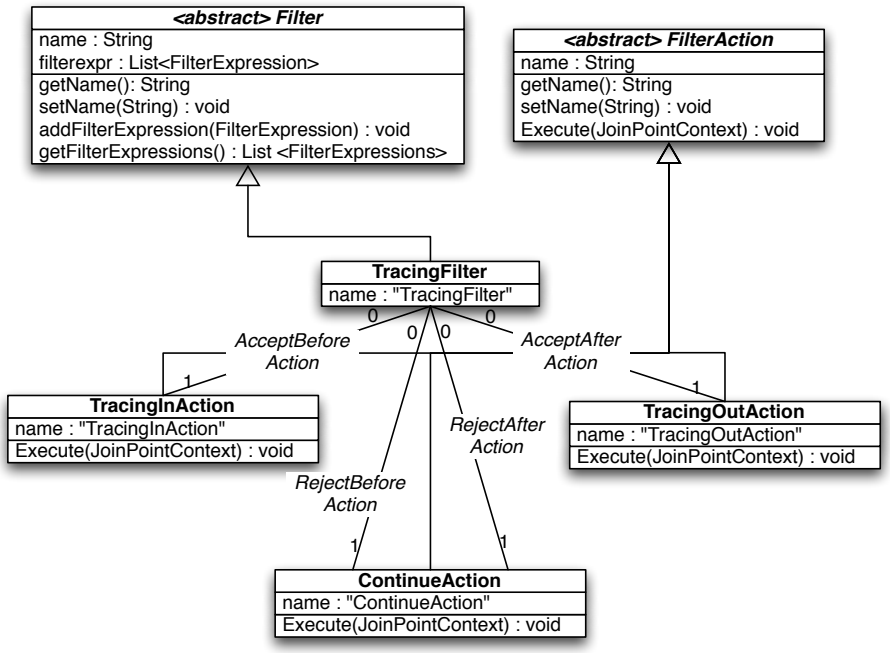
Listing 6.1: Simplified Tracing Concern

Listing 6.1 defines concern Tracing. This concern contains a filtermodule definition (lines 3 to 7) and a superimposition definition (lines 9 to 15). Filtermodule TracingModule defines a single input filter tracing of type TracingFilter. This matches all messages and thus traces all messages. Lines 9 to 15 superimpose filtermodule TracingModule on all classes in the application.

Figure 6.1 presents a class diagram of the implementation of filter TracingFilter.

We have two abstract classes in figure 6.1. Class Filter has a name and a list of filter expressions, which correspond to the condition, matching and substitution expressions. Class FilterAction also has a name but also an abstract method called Execute. Class Filter is subclassed by class TracingFilter. This class has four filter actions attached: filter action TracingInAction, filter action TracingOutAction and twice filter action ContinueAction. These three filter actions are subclasses of class FilterAction. Filter action TracingInAction is executed if an instance of filter TracingFilter accepts *before* the execution of the join point. Filter action TracingOutAction is executed if an instance of filter TracingFilter accepts *after* the execution of the join point. Filter action ContinueAction is executed if an instance of filter TracingFilter rejects *before* or *after* the execution of the join point. The (partial) source of filter TracingFilter and its related filter actions is shown in Appendix B.

Now that we have elaborated on the design of concern tracing and its implementation, we discuss several issues with this design.



Figuur 6.1: Implementation of filter TracingFilter

Observation: No clear separation between behavior and execution time

One of the cornerstone assumptions of Composition Filters is that filter types provide encapsulation of behavior through abstraction. This abstraction provides reusability of the filter types. In essence, the available filters is a so-called aspect library. In Composition Filters we superimpose filter modules that contain filters in filter sets. Superimposition specifies *where* a behavior is applied. Filter modules specify *what* behavior is executed and partially *when* this behavior should be executed, i.e. output filters (method *call*) or input filters (method *execution*). Also, currently filter types encapsulate *what* behavior is executed, and *when* this is executed, i.e. *before* or *after*.

Consider filtermodule TracingModule in listing 6.1, we see a single filter named tracing of type TracingFilter. This filter encapsulates both *what* to trace, i.e. the parameters of a method, and *when* to trace, i.e. at the *start* and *end* of a method execution.

This encapsulation has several drawbacks:

- Manual reasoning is more complicated. A developer has to be aware that different behavior may be executed before or after the execution of the join point.
- Automated reasoning is hindered, since we have to take the moment of execution into account. The compiler needs to know more information to determine what is executed at which point in time.
- There is no way to specify different ordering constraints for returning actions. If a filter accepts a message, also the accepting return action is executed. Conditions are thus only checked when a message enters the incoming filter set.
- Writing a concern that requires a slightly different combination of before and after behavior, requires the definition of a new filter type.

The next section presents an alternative implementation which partly addresses this problem.

6.1.2 An Alternative Tracing Implementation

Consider the implementation of concern Tracing in listing 6.2.

```
1 concern Tracing
2 {
3   filtermodule TracingModule
4   {
```

```
5     externals
6       tracer : TracingLib.Tracer = TracingLib.Tracer.Instance;
7     inputfilters
8       tracingBefore : Before = { [*.*] tracer.StartTrace}
9       tracingAfter : After = { [*.*] tracer.EndTrace}
10  }
11
12  superimposition
13  {
14    selectors
15      sel = { Class | isClass(Class), not(isClassWithName(Class,'TracingLib.Tracer')) };
16    filtermodules
17      sel <- TracingModule;
18  }
19 }
```

Listing 6.2: Alternative Simplified Tracing Concern

Listing 6.2 defines concern Tracing. This concern contains a filtermodule definition (lines 3 to 10) and a superimposition definition (lines 12 to 18). Filtermodule TracingModule has a single external, called tracer of type TracingLib.Tracer. This is a singleton class that can be accessed at every join point. Next, we define two filters; tracingBefore of type Before and tracingAfter of type After. These match all messages and will call method tracer.StartTrace and tracer.EndTrace respectively, to trace messages. Lines 12 to 18, superimpose filtermodule TracingModule on all classes, except for class Tracer itself. Methods StartTrace and EndTrace implement the corresponding trace functionality (not shown here).

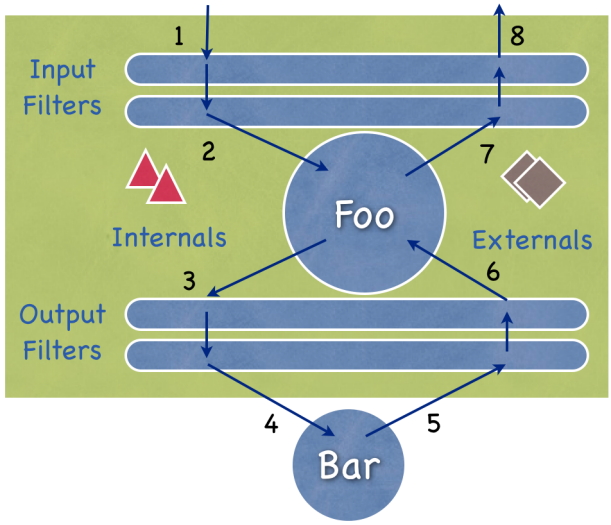
Observation: lack of distinct orderings

The implementation in listing 6.2 uses filter types Before and After to implement concern Tracing. These are built-in filter types. This solves one of the problems mentioned in the previous section, since it is now clear when each filter is executed.

However this solution also introduces some new drawbacks and emphasizes certain other drawbacks.

- We no longer have a single dedicated filter type which implements tracing. The implementation has been split into two filter types.
- The order in which *after* filter actions are executed is the reverse order of *before* filter actions. There is no way to deviate from this.
- In Compose* we can only order filter modules as a whole. There are no means to order filters differently before or after the execution of a join point.
- Similar to the previous tracing implementation, we cannot filter messages on the return of the message, as the filter matching expressions of filter tracingAfter are still evaluated upon entering the filter set.

Figure 6.2 depicts the current situation.



Figuur 6.2: Filters according to the current model

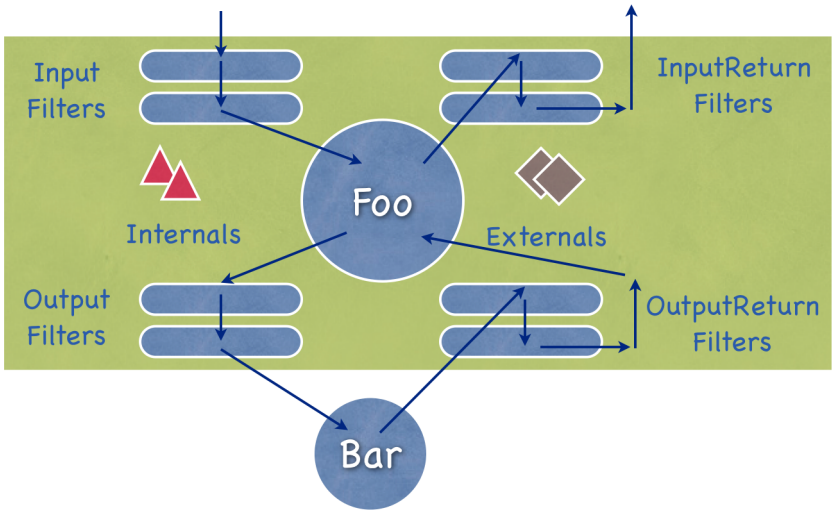
Figure 6.2 shows class Foo that has a set of input and output filters superimposed on it. For simplicity reasons, we assume here that these filters are specified in a single filter module. There are also external and internal objects. The arrows indicate the control flow through the filters. First a call to a method of class Foo is received (1), this message is subjected to the input filters. If the message reaches the end of the filter set, the join point is executed (2). If this method returns (7), all returning filter actions are executed in the reverse order and control is returned to the caller (8).

If a method within class Foo calls a method of, for instance, class Bar (3), the corresponding message is subjected to output filters. If the message reaches the end of the output filter set, the join point is executed (4). When this call returns (5), all returning filter actions are executed and control is returned to the caller, in this case class Foo (6).

6.1.3 Proposal: Distinct Filter Sets

To address some of the drawbacks exhibited by both implementations of concern Tracing in sections 6.1.1 and 6.1.2, we propose a refinement to the Composition

Filters model. This extension involves the addition of two filter sets, namely; InputReturnFilters and OutputReturnFilters. Graphically, the proposal looks like figure 6.3.



Figuur 6.3: Filters according to the proposed model

Figure 6.3 shows a similar situation as figure 6.2. However, we now have four distinct filter sets:

InputFilters : contains filters that filter incoming messages.
InputReturnFilters : contains filters that filter the return of incoming messages.
OutputFilters : contains filters that filter outgoing messages.
OutputReturnFilters : contains filters that filter the return of outgoing messages.

All filters in such a set can be ordered individually, determined by the order in which these filters are written. This is already possible in the current Composition Filters approach, as is the ordering of filter modules at shared join points. Filter modules can be ordered at shared join point using an ordering language, which is elaborated in more detail later. Also, all filters in the four filter sets are evaluated top to bottom, as opposed to bottom-up for the returning filter actions, in the original model. Finally, all filter matching expressions are always evaluated, which includes condition and message matching patterns. This happens now also for retuning messages, which was not the case in the original model.

To summarize, this design has several benefits over the original model:

- Separation between *what* a filter does and *when* it is executed.
- Same top to bottom control flow for all filter sets.
- Condition and message matching pattern are evaluated both before and after the execution of the join point, for the corresponding filter sets.

Concern Tracing can now be expressed as follows:

```
1  concern Tracing
2  {
3    filtermodule TracingModule
4    {
5      inputfilters
6        tracingBefore : StartTracingFilter = { [*.*]}
7      inputreturnfilters
8        tracingAfter : EndTracingFilter = { [*.*]}
9    }
10
11   superimposition
12   {
13     selectors
14       sel = { Class | isClass(Class) };
15     filtermodules
16       sel <- TracingModule;
17   }
18 }
```

Listing 6.3: Updated Tracing Concern

Listing 6.3 defines concern Tracing. This concern contains a filtermodule definition (lines 3 to 9) and a superimposition definition (lines 11 to 17). Filtermodule TracingModule defines a single input filter tracingBefore of type StartTracingFilter and a single input return filter, called tracingAfter of type EndTracingFilter. Both these filters match all messages and thus trace all messages. Lines 11 to 17 superimpose filtermodule TracingModule on all classes.

Filter types StartTracingFilter and EndTracingFilter are only slightly different from filter type TracingFilter (in appendix B). Filter StartTracingFilter has three properties: a name, an accept action TracingInAction and a reject action ContinueAction. Filter EndTracingFilter also has three properties: a name, an accept action TracingOutAction and a reject action ContinueAction. The filter actions have not been changed.

6.1.4 Discussion

In this section we discuss several remaining drawbacks and how to address them.

6.1.4.1 Reusable Filter Modules

As mentioned in section 6.1.2, one of the drawbacks of splitting the filters in a *before* and *after* part is that we no longer have a single filter that implements for instance tracing. This issue is not solved by our proposal. However, we can change the way we look at the reusability of the filter types. This reusability is somewhat strange as we have no *direct* means of superimposing a filter, but we can only superimpose filter modules. Filter modules are a primary unit of reuse in Composition Filters, since they can be superimposed directly, and filter modules encapsulate all the behavior involved, including internals, externals and conditions.

One of the key benefits of filters is that they are parametrized by the condition, matching and substitution patterns. One occurrence of a filter may have a very complex matching and condition pattern, while another occurrence may have a simple expression. This strongly increases the reusability of filters, since application-specific information is filled in when instantiating a filter. Filter modules on the other hand do not have such kind of parametrization. In [Doo06], Doornenbal added parametrization to filter modules, to enable reuse of filter modules. The author describes and proposes such an extension to filter modules using superimposition.

We now illustrate the usage of the proposed filter module parameters mechanism. We use the superimposition language of Compose* to extract elements from the program model, e.g. a class or a method. While superimposing a filter module, one can pass such information to the filter module. Listing 6.4 illustrates how to create a reusable filter module definition for tracing.

```
1  concern Tracing
2  {
3    filtermodule TracingModule(?tracer_class, ?get_trace_instance, ?tracer_start_method,
4      ?tracer_end_method)
5    {
6      externals
7        tracer : ?tracer_class = ?get_trace_instance;
8      inputfilters
9        tracingBefore : StartTracingFilter = { [*.*] tracer.?tracer_start_method}
10     inputreturnfilters
11       tracingAfter : EndTracingFilter = { [*.*] tracer.?tracer_end_method}
12   }
13
14   superimposition
15   {
16     selectors
17       sel = { Class | isClass(Class), not(isClassWithName(Class,'TracingLib.Tracer')) };
18       tracer_class = { Class | isClassWithName(Class,'TracingLib.Tracer') };
19       get_trace_instance = { Method | isClassWithName(Class,'TracingLib.Tracer'),
             isMethodWithName(Method,'TracingLib.Tracer.Instance'), classHasMethod(
             Class,Method)};
```

```
20      tracer_start_method = { Method | isClassWithName(Class,'TracingLib.Tracer'),
              isMethodWithName(Method,'TracingLib.Tracer.StartTrace'), classHasMethod(
              Class,Method)};
21      tracer_end_method = { Method | isClassWithName(Class,'TracingLib.Tracer'),
              isMethodWithName(Method,'TracingLib.Tracer.EndTrace'), classHasMethod(
              Class,Method)};
22    filtermodules
23      sel <- TracingModule(tracer_class, get_trace_instance, tracer_start_method,
              tracer_end_method);
24  }
25 }
```

Listing 6.4: Parameterized Tracing Filter Module Concern

Listing 6.4 defines a filter module called TracingModule (lines 3-12) and a more complex superimposition specification (lines 14-24). Filter module TracingModule is parametrized with four parameters, where each parameter is identified by a preceding question mark. In listing 6.4, these parameters are:

?tracer_class : is the type of the external.
?get_trace_instance : is the method used to get an instance of this external type.
?tracer_start_method : is the method called on the external to create a start trace.
?tracer_end_method : is the method called on the external to create an end trace.

We have now created a parametrized filter module for tracing (lines 3 to 12). To use this filter module we have to provide actual values for these parameters. This is carried out using the superimposition specification at lines 14 to 24. In this superimposition specification, but we not only select the classes we want to superimpose on, we also select the appropriate tracing class (TracingLib.Tracer), a method for instantiation this tracing class (TracingLib.Tracer.Instance), start trace method (TracingLib.Tracer.StartTrace) and end trace method (TracingLib.Tracer.EndTrace). For simplicity reasons, we omitted the selection of an interface to guarantee that the methods for instantiation, start trace and end trace are present with the right signature.

Finally, at line 23 we superimpose filter module TracingModule and provide the selected class and methods. The Compose* compiler verifies that none of the selectors yields an empty set. The compiler also checks that there is only a single element in the result sets of the parameters, since filter module TracingModule expects single elements, indicated by a single question mark, for each of the arguments, and not a list of elements, which would be indicated by a double question mark.

In the example, we superimpose filter module TracingModule immediately after

declaring it. If we include this filter module in a library, we can write a superimposition specification at a later time, in a different concern, thus reusing the same filter module multiple times within the same or different applications.

The presented approach provides the same modularity as the original Composition Filters model. However, filters can no longer execute behavior on both incoming messages and the return of these messages. To achieve this one can use parametrized filter modules.

6.1.4.2 Ability to reason

As mentioned previously, one of the drawbacks of not splitting the filter sets, was that manual and automated reasoning was hindered. As all four filter sets now exhibit similar behavior, we can assume that manual reasoning is indeed improved. However, we still need to assess the impact on automated reasoning, especially for those modules which reason about the composed sequence of filters in Compose*. All modules that reason about the composed sequence of filters rely on a module called FIRE (Filter Reasoning Engine).

The impact of our proposal on FIRE is limited. Our approach even improves reasoning power, while preserving the original design of FIRE. In its current incarnation, FIRE only reasons about the incoming messages of the input filter set and output filter set. The latter reasoning is currently limited, since Compose* does not determine the call sites of a class. FIRE analyzes the filter expression, which includes condition, message matching and substitution expressions, to determine how messages flow through a filter set. This analysis does not have to be adapted, since:

- all four filter sets are evaluated in the same way, namely top-down,
- no new composition operators between filters are introduced,
- conditions are evaluated each time the filter is evaluated, in contrast with the current design that only evaluates conditions before a join point is executed,
- similar to conditions, message matching expressions are also evaluated each time the filter is evaluated, in contrast with the current design that only evaluates message matching before a join point is executed,
- substitution expressions are still allowed for returning messages. However this can only affect further message matching and does not alter the object to which the message is returned.

We assume that the execution of the join point does not alter properties of the message, except for the return value. The execution of the join point can only

cause a change in the state of the system, thus it *can* affect conditions. However, as conditions are independently evaluated, FIRE is not affected by this.

6.1.4.3 Around Constructs

Most AOP languages support some sort of *around* advice construction. In Composition Filters this is usually done via Meta filters. There are two reasons for using around advice. Either to alter the control flow, or to share data between the before and after parts of an around advice in a convenient localized manner.

Affecting control flow - In Compose*, there is currently no way to affect the control flow in the code of a filter action, short of throwing an exception. Originally, *Meta* filters could be used to also affect the control flow. Currently, the only way to affect the control flow is using an annotation attached to a filter action. Listing 1 in Appendix B shows this annotation at lines 9 and 18. There we explicitly state that the execution of these filter actions, does not affect the control flow through the filter set. This design does not need to be changed and can be used in the new filter sets as well. The semantics of using either continue, exit or return actions differs in the incoming and returning filter sets (see figure 5.14). Again, this is also the case in the current situation.

Sharing data - In the proposed design, we no longer have a direct way to share data between before and after parts of a filter. However, this sharing can still be implemented in several different ways:

- One can define and use custom message properties that can be queried and altered. These properties have the same scope as a message, and as such cannot be used for sharing data among join points that are not in the same control flow.
- Sharing data at the same join point can also be achieved through the use of internals. Internals are instantiated for each application of a filter module. This thus provides a scope of the join point and not for an individual message.
- In case one needs to share data between join points, once can use externals. Externals are objects that already exist in the system and are aliased in a filter module. Therefore, the scope of externals is the same as the life span of the objects they refer to.
- Finally, one can share state through the implementation of the filter and filter actions. Here one can use the facilities of the programming language to implement the sharing of data. For example, one can use static variables in a filter or filter action class.

The four presented ways of sharing data shows that the expressiveness of the original design has remained. Some solutions do introduce a slight overhead when sharing data. The exact implementation of these solutions might differ slightly.

6.1.4.4 Fine-grained ordering language

To address the different ordering issues, we need a more fine-grained ordering model. In [Nag05], Nagy presents the design of an ordering and execution constraint language for aspects, in particular of filter modules in Composition Filters. Only the ordering part of the language proposal is currently implemented in Compose*. An example of such an ordering specification is presented in listing 6.5, lines 10 and 11.

```
1  concern ExampleOrderingConcern
2  {
3    superimposition
4    {
5      selectors
6        sel = { Class | isClassWithName(Class,'Foo.Bar') };
7      filtermodules
8        sel <- TracingConcern :: TracingModule;
9        sel <- ProfilingConcern :: ProfilingModule;
10     orderings
11       TracingConcern :: TracingModule pre ProfilingConcern :: ProfilingModule;
12   }
13 }
```

Listing 6.5: Current Ordering Specification

Concern ExampleOrderingConcern superimposes filter module TracingModule from concern TracingConcern, and filter module ProfilingModule from concern ProfilingConcern on class Foo.Bar. Lines 10 and 11 state that filter module TracingModule must be executed before ProfilingModule, if present. In the current implementation of Compose* this would yield the following execution order:

1. Before execution of the join point input filters from TracingModule.
2. Before execution of the join point input filters from ProfilingModule.
3. Execution of the join point.
4. After execution of the join point input filters from ProfilingModule.
5. After execution of the join point input filters from TracingModule.

Now, imagine that we would like to have a different ordering before and after the execution of the join point, i.e. not the inverse. This cannot be achieved using the current implementation of Compose*. Our proposal is to extend the ordering language to explicitly refer to the incoming and returning filter sets. The result would be as in listing 6.6.

```
1  concern ExampleOrderingConcern
2  {
3    superimposition
4    {
5      selectors
6        sel = { Class | isClassWithName(Class,'Foo.Bar') };
7      filtermodules
8        sel <- TracingConcern :: TracingModule;
9        sel <- ProfilingConcern :: ProfilingModule;
10     orderings
11       TracingConcern :: TracingModule.inputfilters pre ProfilingConcern :: ProfilingModule.
             inputfilters;
12       TracingConcern :: TracingModule.inputreturnfilters pre ProfilingConcern ::
             ProfilingModule.inputreturnfilters;
13   }
14 }
```

Listing 6.6: Proposed Ordering Specification

Lines 10 to 12 in listing 6.6, show one example of the proposed new ordering language. The proposed ordering language has several assumptions:

- Operations pre and post both require the types of both operands to be the same, i.e. both must be of type *inputfilters*, *inputreturnfilters*, *outputfilters* or *outputreturnfilters*.
- Only referring to entire filter modules instead of filter sets results in the same orderings at all four filter sets. This provides language backwards compatibility. However, if the order of the returning filter actions mattered in the original design, the behavior is different since the returning filter actions are executed in the reverse order.

There is no need for a more fine-grained filter ordering language that refers to individual filters, as this would break the design and encapsulation of the enclosing filter module.

Another extension to the filter ordering language is the possibility to select different orderings for different join points. For one join point, a user may want a particular order, while for another the user may require a different order, e.g. the reverse. We can extend the ordering language to accommodate this. An example of such an extension is presented in listing 6.7.

```
1  concern ExampleOrderingConcern
2  {
3    superimposition
4    {
5      selectors
6        sel1 = { Class | isClassWithName(Class,'Foo') };
7        sel2 = { Class | isClassWithName(Class,'Bar') };
8      filtermodules
9        sel1 <- TracingConcern :: TracingModule;
10       sel2 <- TracingConcern :: TracingModule;
```

```
11      sel1 <- ProfilingConcern :: ProfilingModule;
12      sel2 <- ProfilingConcern :: ProfilingModule;
13    orderings
14      sel1 = TracingConcern :: TracingModule pre ProfilingConcern :: ProfilingModule;
15      sel2 = ProfilingConcern :: ProfilingModule pre TracingConcern :: TracingModule;
16  }
17 }
```

Listing 6.7: Revision of the proposed Ordering Specification

Lines 14 and 15 in listing 6.7, state that filter module TracingModule must be executed before filter module ProfilingModule, for class Foo, using selector sel1, and that filter module ProfilingModule must be executed before filter module TracingModule, for class Bar, using selector sel2.

The proposals of splitting the filter sets, updating the affected modules and the updated filter ordering language have not yet been implemented in Compose*.

6.1.4.5 Generality

The idea of splitting the behavior of an advice and when this behavior is executed, is not new to AOP languages. For example, most languages use *before*, *after* and *around* advice. However, such an execution time specification is usually tightly coupled with the advice specification. In essence, these languages have a similar situation as in the alternative implementation of concern Tracing (listing 6.2). They thus suffer from part of the same problems as described in this section.

One can imagine extending, for instance AspectJ, with a way to specify advice separately from a specific execution time. This is already partially implemented using pointcut languages, e.g. the difference between `call` and `execution`. An extension similar to the one proposed here to such a pointcut language would simplify this and would create more reusable advice.

Also, we have proposed a fine-grained ordering language. Most AOP languages only order at the granularity of aspects. In some languages the order in which *before*, *after* and *around* advices are declared in the same aspect impacts the selected ordering, similar to the declarative composition of filters within a filter set. Languages like AspectJ, could implement a more fine-grained and elaborate aspect ordering language than `declare precedence` or declaration order of advice within a file.

6.2 Atomic Filters

A lot of heterogeneous crosscutting concerns implement behavior that is usually not application-specific. Concerns like *Profiling*, *Parameter Tracing* and *Parameter Checking*, can be used in numerous applications. One of the benefits of separation of concerns is that one is able to create reusable modules. Preferably, we would like to have aspects in a library that addresses these common crosscutting concerns, while providing enough parametrization to be useful in multiple applications and environments. Ultimately, such an aspect library could also contain "atomic" aspects that can be used to build up other (more complex) aspects. We use the term atomic to indicate something that cannot be split. In this section we demonstrate the benefits of having such atomic aspects. We also create atomic filters for the Composition Filters model.

6.2.1 Delegation Example

Consider the example in listing 6.8.

```
1  concern Delegation
2  {
3    filtermodule DelegationModule
4    {
5      externals
6        delegator : Delegator;
7      outputfilters
8        send : Send = { [*.foo] delegator.newfoo }
9    }
10
11   superimposition
12   {
13     selectors
14       sels = { Class | isClassWithName(Class, 'Bar') };
15     filtermodules
16       sels <- DelegationModule;
17   }
18 }
```

Listing 6.8: Concern Delegation

Listing 6.8 defines concern Delegation. This concern contains filtermodule DelegationModule (lines 3 to 9) and a superimposition specification (lines 11 to 17). Filtermodule DelegationModule declares a single external named delegator of type Delegator. The filtermodule also has one output filter named send of type Send. This filter matches all messages with a selector named foo, it is not interested in a specific target. If such a message is encountered it is sent to method newfoo of external delegator. Filtermodule DelegationModule is superimposed on class Bar.

Now, imagine that a call from class Bar to Foo.foo is intercepted and this message is thus passed to the send filter. The following execution steps are carried out:

1. The *target* is matched against the wildcard character, which always matches.
2. The *selector* of the message is matched against the selector matching pattern, which is equal to foo, thus the filter accepts the message.
3. The *target* of the message is substituted with the target from the substitution expression, in this case delegator.
4. Similar to the *target*, the *selector* is also substituted with the selector from the substitution expression, in this case newfoo.
5. The accept action of filter type Send is executed. This results in:
   - changing the *sender* of the message to the current *server* of the message, in this case class Bar,
   - changing of the *server* of the message to the current *target* of the message, in this case class Foo,
   - setting the *target* of the message to the specified *target*, in this case to external delegator,
   - setting the *selector* of the message to the specified selector, in this case method newfoo,
   - sending the modified message to the current target and selector,
   - ending the evaluation of filter set OutputFilters, and directing control to filter set OutputReturningFilters once this filter returns.

This example shows that even using a simple filter like Send results in a complex series of steps. Steps 1 to 4 are inherent to the filtering mechanism, and are always executed for each filter. However, if we consider step 5 as the key action of this filter, we see that the effect of this action is threefold. First, it changes several properties of the message. Second, it calls another method. Third, it changes the control flow. In Composition Filters, the goal of a filter type is to encapsulate such behavior.

Encapsulation is a desirable property, however it does hinder automated and manual reasoning and reduces the reusability of these kind of filters. Automated and manual reasoning is impaired, since a tool or developer has to know the exact semantics of this filter. The semantics of a filter cannot be derived from its name. Reusability is impaired since it is hard to build other filters from a complex filter, only if there is an exact match can we reuse a filter. Atomic filters would increase not only manual and automated reasoning, since each filter only performs one specific action. This would also increase reuse since more complex filters can be built from these atomic filters.

We can perform the same kind of analysis for the other filters. We present the current filter actions and their effect on properties of the message or control flow. We do this on the basis of the resource-operation model introduced in chapter 4. The result is depicted in table 6.1.

Tabel 6.1: Effect of the current filter actions

Filter Action	**Sender**	**Server**	**Target**	**Selector**	**Args**	**Return Value**	**Control Flow**
DispatchAct.			write	write			return
SendAction	write	read write	read write	write			return
ErrorAction							exit
NopAction							continue
SubstitutionAct.							continue
MetaAction	read write	read write	read write	read write	read write	read write	continue return exit
BeforeAction	read	read	read write	read write	read write		continue
AfterAction	read	read	read write	read write	read write	read write	continue

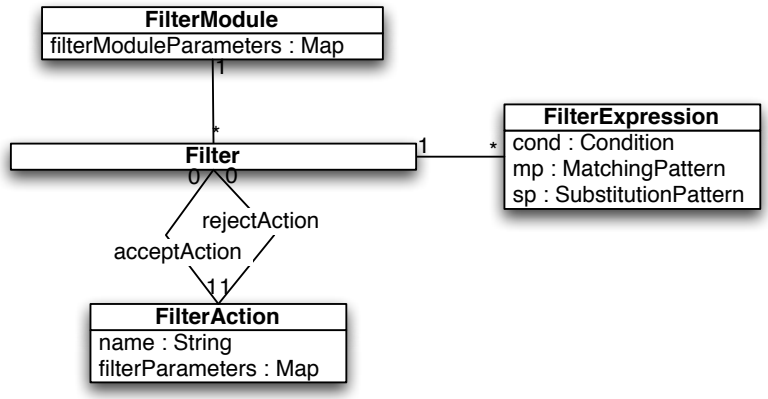
We have excluded all read and write operations on the target, selector and conditions that are caused by filter expressions, since these are filter instance-specific and have to be derived from the filter specification. Also we have opted here to only show the filter actions. Filters consist of an accept and reject filter action and all original filters have a NopAction either as an accepting or rejecting action. Therefore, we can map these actions one-to-one to a filter. We have excluded those features from the MetaAction that introduce concurrency, for example to execute an advice in parallel with the filter evaluation. One can create a dedicated filter action which introduces this concurrency, if required.

Table 6.1 illustrates that the current filters are not atomic. As explained, this is a desirable property because it enables easier building of new functionality. In atomic filters, there are no filters that have the same or overlapping behavior. Also, each atomic filter should perform only one basic functional action. The atomic filters should be orthogonal in the sense that all (existing) filters can be build from a linear combination of the atomic filters. This enables better manual and automated reasoning about the composition of filters.

6.2.2 Filter Parametrization

Before we present the atomic filters, we first explain a construct in Composition Filters that is used by several filters. This construct is called filter parameters, and enables the passing of parameters to the filter.

Filter parameters should not be mistaken with either filter module parameters or filter expressions. We clarify the distinction and show the relationship between these three kinds of parametrization using figure 6.4. We have removed all details that do not contribute to the parametrization.



Figuur 6.4: Simplified UML model of filter modules and filters

At the top of figure 6.4, class FilterModule is defined. This class has a map of filter module parameters. In section 6.1.4.1, we discussed the details of filter module parameters. These parameters can either refer to a single element (indicated with a single question mark) or a list of elements (indicated with a double question mark).

Each Filter (in the middle of figure 6.4) has an accept and reject filter action, as well as a sequence of FilterExpressions. These filter expressions are of the form `condition => [target.selector]target.selector`. We assume here a canonical form of filter expressions. All filter expressions can be rewritten to a sequence of these canonical filter expressions. We ignore the difference between name and signature matching, since this has no impact on this discussion. Filter expressions offer a way to parametrize the accept or reject criteria for a filter. Each class Filter has two associate FilterActions. These are, respectively, the accept and reject actions

for this Filter.

Filter parameters provide a way to pass properties to the filter, other than the acceptance criteria. Imagine the following code snippet:

```
1 error : Error ( exception:ContractViolationException ) = { isContractValid => [*,*] }
```

In this case we provide a custom exception class called ContractViolationException that should be thrown if the filter rejects. The filter action can query the list of filter parameters and decide what to do with each of the filter parameters. The filter parameters are type-value tuples that are comma separated in case there are more than one. The type of a filter parameter is always a String. The value of a filter parameter can either be an Object or a String (enclosed within a double quotation marks). Listing 6.9 shows the implementation of class ErrorFilterAction.

```
1  [FilterActionAttribute("ErrorFilterAction", FilterActionAttribute.FilterFlowBehavior.Exit,
       FilterActionAttribute.MessageSubstitutionBehavior.Original)]
2  public class ErrorAction : FilterAction
3  {
4    public override void Execute(JoinPointContext context)
5    {
6      String exceptionClass = context.GetFilterParameter("exception");
7      if(exceptionClass != null)
8      {
9        Exception e = // Using reflection create a new instance of specified Exception class.
10       throw e;
11     }
12     else
13     {
14       throw new RuntimeException("Error Action");
15     }
16   }
17 }
```

Listing 6.9: ErrorFilterAction in C#

One can see that filter action ErrorAction (listing 6.9) queries (line 6) the context for a filter parameter called exception. If this exists, a new instance of the exception class is created and thrown (lines 7 to 11). If this does not exist, a generic RuntimeException is instantiated and thrown (lines 12 to 15).

All filter module parameters are available in the map of filter parameters. As such, a filter action can use the reflective properties of the superimposition language to gather information about the application. Filter module parameters can be accessed through the name of the specific filter module parameter. We will demonstrate this mechanism using a simplified version of the example presented in listing 6.4. Consider the example concern in listing 6.10.

```
1  concern Parametrization
2  {
3    filtermodule TracingModule(?tracer_class, ?get_trace_instance, ?tracer_start_method,
4      ?tracer_end_method)
5    {
6      externals
7        tracer : ?tracer_class = ?get_trace_instance;
8      inputfilters
9        tracingBefore : StartTracingFilter (recurse:"false") = { [*.*] tracer.?
             tracer_start_method}
10     inputreturnfilters
11       tracingAfter : EndTracingFilter (recurse:"false") = { [*.*] tracer.?tracer_end_method}
12   }
13 }
```

Listing 6.10: Parameterization Example

In listing 6.10, we have omitted the superimposition specification. Lines 9 and 11 show two filter definitions that are parametrized. In this case we use a parameter to indicate that we do not want to trace recursive method calls. The filter actions that implement start and end tracing, can access this parameter by querying the context in the accepting filter actions. The filter parameter map of these filters not only contains tuple `(recurse:false)`, but also tuples: `(?tracer_class :TracingLib.Tracer)`, `(?get_trace_instance:TracingLib.Tracer.Instance)`, `(?tracer_start_method: TracingLib.Tracer.StartTrace)`, and `(?tracer_end_method:TracingLib.Tracer.StopTrace)`. The latter four parameters can only be read, i.e. they are not allowed to be changed.

In its current state, Composition Filters implements only rudimentary support for filter and filter module arguments. The extension presented here is thus a novel extension. The need for this extension will become more apparent in the next section.

6.2.3 Proposal: Atomic Filters

In this section we propose atomic filters. One goal of these filters is to be able to more easily create complex filters from these atomic filters. A second goal of these filters is to improve automated and manual reasoning.

We use the analysis performed in chapter 5, to derive these filters. We distinguish between behavior that accesses the properties of a message from behavior that affects the control flow.

6.2.3.1 Message Properties Filter

In this section we only discuss filters, the behavior of a filter is implemented by an accept and reject filter action. In this section this distinction is not important as we map filter actions one-to-one to filters, see section 6.2.4.1. This distinction does not affect the definition of the filters, as presented here.

We introduce one filter that can affect the properties of the message. This filter is called SetProperty. Filter SetProperty can be used to set properties of the message or custom properties. If a property does not exists, it is created. Filter SetProperty can be used in the following three forms:

- SetProperty(property:prop1,property1:prop2): sets the value of property prop1 to the value of property prop2. Property prop2 must exist, else an error is reported.
  Example: SetProperty(property:sender, property1:server), this sets the sender of the message to the server of the message.
- SetProperty(property:prop,variable:!var): sets the value of property prop to the value of variable var. Variables will be explained in more detail shortly. Variable !var must exist, else an error is reported.
  Example: SetProperty(property:sender, var:!newsender), this sets the sender of the message to the value contained in variable !newsender.
- SetProperty(property:prop,value:"xxx"): sets the value of property prop to a specified value, in this case "xxx".
  Example: SetProperty(property:selector, value:ïnstance"), this sets the selector of the message to "instance".
  Example: SetProperty(property:authenticated, value:"true"), this sets the custom property authenticated to true.

Composition Filters declare the following set of properties:

Sender: the object that initiated the message
Server: the object that initially received the message
Target: the object that received the message, in most cases this is equivalent to the server.
Selector: the method that is called.
Arguments: the arguments of the message.
ReturnValue: the return value of the message.

The above presented properties are all readable and writable.

There are also properties that provide reflection about the specific matching and

substitution expression. Equation 6.1 shows the details of these expressions.

$$err\text{: }Error = \{[\ \overbrace{target1}^{MatchingTarget}\ .\ \underbrace{selector1}_{MatchingSelector}\ ]\ \overbrace{target2}^{SubstitutionTarget}\ .\ \underbrace{selector2}_{SubstitutionSelector}\ \} \tag{6.1}$$

MatchingTarget: the object that matched the target of the message.
MatchingSelector: the text string that matched the selector of the message.
SubstitutionTarget: the object for which the target of the message is substituted.
SubstitutionSelector: the text string for which the selector of the message is substituted.

The above stated properties are read-only; they can not be changed by filters. Related to behavior conflicts, we assume that there can be no conflicts for these properties, since they can not be changed and reading the same property multiple times is not considered conflicting.

Some filters like Dispatch use the target and selector as specified in the substitution expression of a filter expression. However, other filters like Before and After only use this to determine which advice method to execute, and the target and selector of the message remains unchanged. Whether the target and selector of the message is changed, depends on the filter action. We now state that filters have to substitute to these properties explicitly, using the appropriate SetProperty(property:target, ...) and SetProperty(property:selector, ...). Once can use the previously mentioned properties SubstitutionTarget and SubstitutionSelector to get specific values or use filter parameters.

Scope of properties The previously presented properties have a different scope. Some properties are only present during the evaluation of one filter, while others are present during the evaluation of join point. We now present the scope of each of the properties:

Sender: always available during the execution of the thread, but may change at each join point.
Server: always available during the execution of the thread, but may change at each join point.
Target: always available during the execution of the thread, but may change at each join point.
Selector: always available during the execution of the thread, but may change at each join point.

Arguments: always available during the execution of the thread, but may change at each join point.
ReturnValue: always available during the execution of the thread, but may change at each join point.
MatchingTarget: evaluation of the enclosing filter.
MatchingSelector: evaluation of the enclosing filter.
SubstitutionTarget: evaluation of the enclosing filter.
SubstitutionSelector: evaluation of the enclosing filter.
Filter Module Parameters: evaluation of the enclosing filter module.
Custom: custom properties are retained during the execution of the thread.

Typing of properties The presented properties are typed. We define the following types and show which properties below to which type.

Object : Sender, Server, Target, ReturnValue, MatchingTarget, Substitution-Target
String : Selector, MatchingSelector, SubstitutionSelector
List : Arguments

Custom properties can be any type, depending on the last assignment of this custom property. For example assigning the arguments of a message to the target, is not allowed. Tables 6.2 and 6.3 show which properties or values can be assigned to other properties.

Tabel 6.2: Types of properties - 1

	Sender Server Target	**Selector**	**Args**	**Return Value**
Sender Server Target	V			V
Selector		V		
Args			V	
Return Value	V			V
Custom	V	V	V	V

Most properties can be assigned only to similar properties. Selectors can be assigned the value of another selector or a string value. A property can only be assigned a custom property if their types are equivalent. A custom property can be assigned any type. Properties that are inherited from the filter module are only allowed to be queried, as are the matching and substitution target and selectors. The compiler should enforce these typing rules.

Tabel 6.3: Types of properties - 2

	Matching & Substitution Target	Matching & Substitution Selector	Custom	Value
Sender Server Target	V		V	
Selector		V	V	V
Args			V	
Return Value	V		V	
Custom	V	V	V	V

6.2.3.2 Variables

We introduce the notion of variables to offer a simple, light-weight mechanism to transfer data between filters. Like properties, variables can be assigned the value of a property, the value of another variable, or a string value. We introduce a filter that manipulates variables, a so-called SetVariable filter. Variables must be proceeded with a exclamation mark, e.g. !foo. This makes the difference between properties and variables clear. The key difference between variables and properties is that fact variables have a scope that is limited to a filter module, whereas properties can have a longer life span. Filter SetVariable is similar to filter SetProperty, the difference is that the first argument now is a variable instead of a property. Filter SetVariable can be used in the following forms:

- SetVariable(variable:!var1,variable1:!var2): sets the value of variable var1 to the value of variable var2. Variable var2 must exist, else an error is reported.
  Example: SetVariable(variable:!x, variable1:!y), this sets the value of variable x to the value of variable y.
- SetVariable(variable:!var, property:prop): sets the value of variable var to the value of property prop. Property prop must exist, else an error is reported.
  Example: SetVariable(var:!newsender, property:sender,), this sets the value of variable !newsender to the sender of the message.
- SetVariable(variable:!var,value:"xxx"): sets the value of variable var to a specified value in this case "xxx".
  Example: SetVariable(var:!authenticated, value:"true"), this sets the value of variable authenticated to "true".

A key difference between properties and variables, is that properties are only accessible in the filter specifications. An advice that wants to manipulate the

properties of the message need to do this via variables. Section 6.2.4.2 provides more information about this.

Scope of variables The scope of variables is limited to a filter module. If a larger scope is required, one must use properties.

Typing of variables The same typing rules that apply to properties, also apply to variables. The compiler can enforce these typing rules.

6.2.3.3 Control Flow Filters

The following filters affect the control flow of message evaluation:

Return : This filter returns the control flow to the start of the returning filters or to the caller.

Exit : This filter exits or terminates the evaluation of the filters completely. For example, in case of an exception, in such a situation control is transferred to the exception handler, and the filter evaluation is terminated.

In most cases we are not interested in changing the control flow. We execute some behavior and continue to the next filter. This continuing behavior is encapsulated in the composition operator between filters, this is indicated by a semicolon at the end of filter specification. Also, sequential composition is the default composition mechanism to compose two or more filter modules at the same join point. Currently, Composition Filters only offer sequential composition between filters.

6.2.3.4 Call and Advice Filters

There are filters that execute a method in the base system in order to implement some functionality, typical examples are *before* and *after* filters. To accommodate these filters, we introduce one filter called Advice and one called Call. Filter Advice calls a method of the base code, with a context object, see appendix A for more details about this context object. This context only exposes those properties that have been explicitly queried using SetVariable filters. Setting properties will have no effect, unless they are committed to the message using SetProperty filters. The context it self only provides read-only access to certain properties, like MatchingTarget. Filter Advice is similar to filters Before and After. However, in the first section of this chapter, we proposed an extension of

Composition Filters, where we introduced filter sets for incoming and returning messages. As such we do not need to distinguish between before and after, we can unify this to a single filter. This filter has to be placed in the appropriate filter set.

We immediately return to the filter set after executing the advice. All control flow changes must be implemented using appropriate control flow filters. Filter Advice refers to two properties to determine which method it should execute. It first checks for the existence of filter parameters target and selector, if these do not exist, the filter queries properties SubstitutionTarget and SubstitutionSelector. To illustrate the usage of this filter, an example of filter Advice is presented here:

```
1 adv : Advice = { shouldTrace => [*.*]tracer.dotrace }
```

Listing 6.11: Example of filter Advice

The above filter definition states that, if a condition shouldTrace is true, method dotrace of internal or external object tracer, is called. We now present part of the implementation of filter Advice. We only show the accepting filter action, called AdviceAction. The reject action does not execute anything.

```
1  [FilterActionAttribute("AdviceAction", FilterActionAttribute.FilterFlowBehavior.Continue,
       FilterActionAttribute.MessageSubstitutionBehavior.Original)]
2  public class AdviceAction : FilterAction
3  {
4    public override void Execute(JoinPointContext context)
5    {
6      object target = context.GetFilterParameter("target");
7      string selector = context.GetFilterParameter("selector");
8      if(target != null && selector != null)
9      {
10       // Lookup the target and call the selector method with the context
11     }
12     else
13     {
14       target = context.GetFilterProperty("SubstitutionTarget");
15       selector = context.GetFilterProperty("SubstitutionSelector");
16       if(target != null && selector != null)
17       {
18         // Lookup the target and call the selector method with the context
19       }
20       else
21       {
22         throw new RuntimeException("Advice Invocation Error");
23       }
24     }
25   }
26 }
```

Lines 6 and 7 retrieve filter parameters target and selector. These are filter parameters that have to be explicitly passed to this filter, thus overriding properties SubstitutionTarget and SubstitutionSelector. If none of these parameters

is empty (line 8), we lookup the target. This target object can be a reference to an internal, external or a class, in the example this is an internal or external called tracer, and we call a method of this object. In the example a method called dotrace. This method has a single parameter representing the join point context. If filter parameters target and selector are empty, we retrieve the target and selector from properties SubstitutionTarget and SubstitutionSelector (lines 14 and 15). If these are not empty (line 16), we call the designated target, with the context. If both the filter parameters and properties are empty, we throw an exception. We could have implemented filter Advice in listing 6.11, using filter parameters, the result would look like:

```
1 adv : Advice(target:tracer, selector:"dotrace") = { shouldTrace => [*.*] }
```

Listing 6.12: Alternative example of filter Advice

The result is equivalent to the implementation in listing 6.11.

Filter Call redirects the message to the current target and selector. This filter is similar to filter Dispatch, except now we always return to the filter set. Filter Call is similar to filter Advice, except that the advice methods are called with different arguments. Methods called using filter Advice expect a JoinPointContext object as a parameter, whereas methods that are called using filter Call, get the current arguments as parameters. We now show an example of a Call filter:

```
1 call : Call = { [*.*]delegator.delegate }
```

Filter call of type Call ensures that all messages are redirected to method delegate of external delegator. As with filter Advice, we can override the specified target and selector using the filter parameters. The previous Call filter can be rewritten to the following equivalent form:

```
1 call : Call(target: delegator, selector:"delegate") = { [*.*] }
```

We now map the previously described filters to the resource model, as shown in tables 6.4 and 6.5.

Filters SetTarget and SetSelector are shorthand forms for SetProperty(property:target,property1:SubstitutionTarget) and SetProperty(property:selector,property1:SubstitutionSelector), respectively. Implicitly querying the target and selector using the matching expressions in filter expressions is still possible. As previously mentioned, matching on either target or selector results in a read operation on the target or selector.

We have not included filter SetVariable in tables 6.4 and 6.5. Variables are not resources in our model, since the scope of variables is limited to a filtermodule.

Tabel 6.4: Effect of the proposed filters - Part 1

Filter	Sender	Server	Target	Selector
SetProperty(..., property:sender)	read			
SetProperty(property:sender, ...)	write			
SetProperty(..., property:server)		read		
SetProperty(property:server, ...)		write		
SetProperty(..., property:target)			read	
SetProperty(property:target, ...)			write	
SetTarget			write	
SetProperty(..., property:selector)				read
SetProperty(property:selector, ...)				write
SetSelector				write

Tabel 6.5: Effect of the proposed filters - Part 2

Filter	Arguments	Return Value	Control Flow
SetProperty(..., property:args)	read		
SetProperty(property:args, ...)	write		
SetProperty(..., property:retval)		read	
SetProperty(property:retval, ...)		write	
Return			return
Exit			exit
Advice			
Call			

Since the design choice has been made to assume that the implementation of a filtermodule itself is correct, there is no need to reason about variables, hence to model variables as properties.

Tables 6.4 and 6.5 show that each filter can be specified by a single operation. Filters Call and Advice neither affect the properties of the message, nor the control flow, as all properties can only be manipulated or queried using the appropriate filters. Section 6.2.4.2 discussion the motivation behind this.

6.2.4 Discussion

6.2.4.1 Filter Action versus Filter

Until now, we only presented atomic filters. Filters can be mapped to filter actions. For completeness, we show all filters with their corresponding accept and reject actions, in table 6.6.

Tabel 6.6: The new filters

Filter	Accept Filter Action	Reject Filter Action
SetProperty	SetPropertyAction	NopAction
SetVariable	SetVariableAction	NopAction
SetTarget	SetTargetAction	NopAction
SetSelector	SetSelectorAction	NopAction
Return	ReturnAction	NopAction
Exit	ExitAction	NopAction
Advice	AdviceAction	NopAction
Call	CallAction	NopAction

All filters use filter action NopAction, in case of rejection. This action executes nothing, and the result is that the next filter is evaluated, if any. This provides consistent behavior, as opposed to the current filters. Where, for example, filter Error throws an exception if it rejects.

6.2.4.2 Prescriptive Filters

In order to improve reasoning about filters, we want to guarantee that all operations on the properties of messages are written explicitly in the filter expressions. To achieve this, we have to ensure that the filters not only execute the specified behavior but we also have to prevent manipulations of the properties in alternative ways. We can only automatically reason about filters in the filter

language. We do not analyze the source code of advice and call filters, however these can be annotated with a semantic annotation that specifies the behavior. Only filter Advice could potential change the properties of the message in a non declarative manner. For this filter we have to ensure that all queries on and changes to the properties of the message are known. This is ensured by the design and usage of variables and properties. Only variables are directly accessible in an advice method. To change a property of a message, one first has to use filter SetVariable. Next one must use filter Advice that manipulates a variable, variables are accessible through the context object. Finally, filter SetProperty is used to set the property to the changed variable. Setting read-only properties properties within the filters will yield an error.

6.2.4.3 Expressiveness

We have created atomic filters and filter actions. To proof that this is indeed correct we now implement the current filters using the new atomic filters.

Dispatch Filter
Filter Dispatch can be created using filter Call followed by filter Return.

Consider the following example:

```
1 disp : Dispatch = { [*.foo] delegator.newfoo }
```

This can be translated in the following atomic filters:

```
1 f1 : Call = { [*.foo] delegator.newfoo };
2 f2 : Return = { [*.foo] delegator.newfoo }
```

Send Filter
Filter Send can be created using two SetProperty filters and filter Call followed by filter Return.

This can be translated to the following atomic filters:

```
1 f1 : SetProperty(property:sender, property1:server) = { [*.foo] delegator.newfoo };
2 f2 : SetProperty(property:server, property1:target) = { [*.foo] delegator.newfoo };
3 f3 : Call() = { [*.foo] delegator.newfoo };
4 f4 : Return() = { [*.foo] delegator.newfoo }
```

Error Filter
Filter Error can be created using filter Advice and filter Exit. Filter Advice is responsible for throwing the required exception. Filter Exit is not really needed, as it is never reached at run time, since the exception prevents this. However,

from a reasoning perspective it is desired to explicitly mention that the filter set is no longer evaluated.

Consider the following example:

```
1 err : Error = { !isContractValid => [*.*] }
```

This can be translated to the following atomic filters:

```
1 f1 : Advice = { isContractValid => [*.*] };
2 f2 : Exit = { isContractValid => [*.*] }
```

Substitution Filter
Filter substitution can be created using filters SetTarget, SetSelector.

Let us consider the following example:

```
1 subs : Substitution = { [*.foo] main.bar }
```

This can be translated to the following atomic filters:

```
1 f1 : SetTarget = { [*.foo] main.bar };
2 f2 : SetSelector = { [*.foo] main.bar }
```

Meta Filter
The functionality of filter Meta has mostly been superseded by filters Before and After. As such most meta filters can be rewritten to a combination of Advice and control flow manipulation filters. Meta filters can also introduce concurrency in the filter set, for example executing the advice in parallel with the filter set evaluation. As previously mentioned, we do not consider concurrency here. This functionality can be added by creating an appropriate filter type that introduces this concurrency.

Before Filter
Filter Before is equivalent to an Advice filter that is located in either InputFilters or OutputFilters, see section 6.1.

Consider the following example:

```
1 inputfilters
2   bef : Before = { [*.*] checker.checkArguments }
```

This can be translated in the following atomic filter:

```
1 inputfilters
2   f1 : Advice = { [*.*] checker.checkArguments }
```

After Filter

Filter After is equivalent to an Advice filter that is located in either InputReturnFilters or OutputReturnFilters, see section 6.1.

Consider the following example:

```
inputfilters
  aft : After = { [*.*] checker.checkReturnValue }
```

This can be translated in the following atomic filter:

```
inputreturnfilters
  f1 : Advice = { [*.*] checker.checkReturnValue }
```

TracingIn Filter

We show how one can rewrite filter TracingIn using four atomic filters. These filters are: three SetVariable filters and a single Advice filter. The first three filters ensure that the target, selector and arguments are accessible in the advice method of the fourth filter.

Consider the following example:

```
inputfilters
  tracingIn : TracingIn = { shouldTrace => [*.*] tracer.traceIn }
```

This can be translated in the following atomic filters:

```
inputfilters
  f1 : SetVariable(variable:!thetarget, property:target) = { shouldTrace => [*.*] };
  f2 : SetVariable(variable:!theselector, property:selector) = { shouldTrace => [*.*] };
  f3 : SetVariable(variable:!theargs, property:arguments) = { shouldTrace => [*.*] };
  f4 : Advice(target:tracer, selector:"traceIn", tracetarget:!thetarget,
    traceselector:!theselector, args:!theargs) = { shouldTrace => [*.*] }
```

Listing 6.13: TracingIn Filter using atomic filters

The four filters in listing 6.13 all use the same condition, matching and substitution pattern, in this case: `shouldTrace => [*.*]`. This redundancy is addressed in section 6.3.

TracingOut Filter

We now show how to implement filter TracingOut using five atomic filters. These filters are: four SetVariable filters and a single Advice filter. The first four filters ensure that the target, selector, arguments and return value are accessible in the advice method of the fifth filter.

Consider the following example:

```
inputfilters
  tracingOut : TracingOut = { shouldTrace => [*.*] tracer.traceOut }
```

This can be translated in the following atomic filters:

```
inputreturnfilters
  f1 : SetVariable(variable:!thetarget, property:target) = { shouldTrace => [*.*] };
  f2 : SetVariable(variable:!theselector, property:selector) = { shouldTrace => [*.*] };
  f3 : SetVariable(variable:!theargs, property:arguments) = { shouldTrace => [*.*] };
  f4 : SetVariable(variable:!retval, property:returnvalue) = { shouldTrace => [*.*] };
  f5 : Advice(target:tracer, selector:"traceOut", tracetarget:!thetarget,
    traceselector:!theselector, args:!theargs, retval:!retval) = { shouldTrace => [*.*] }
```

Listing 6.14: TracingOut Filter using atomic filters

Alternatively, we can use the substitution pattern to define which advice method to executed. In such a case we remove the specific target and selector from the filter parameter list of filter f5, the result is as follows:

```
inputreturnfilters
  f1 : SetVariable(variable:!thetarget, property:target) = { shouldTrace => [*.*] };
  f2 : SetVariable(variable:!theselector, property:selector) = { shouldTrace => [*.*] };
  f3 : SetVariable(variable:!theargs, property:arguments) = { shouldTrace => [*.*] };
  f4 : SetVariable(variable:!retval, property:returnvalue) = { shouldTrace => [*.*] };
  f5 : Advice(tracetarget:!thetarget, traceselector:!theselector, args:!theargs, retval:!
       retval) = { shouldTrace => [*.*] tracer.traceOut}
```

In recent years many other filters have been proposed. For example *Wait*, *Realtime*, *Atomic Dispatch* and *Multiple Dispatch*. We have not included these filters since they impact the filter processing dramatically or require virtual machine support, and are not supported by any of the current implementations.

6.2.4.4 Substitution Behavior of Filters

In the original filters the substitution behavior differed from filter to filter. For example, a substitution expression in a Dispatch filter, ensures that if the filter matches, we dispatch the message to the specified target and selector. However, the substitution expression in filters Before and After only indicate which method should be executed. Once the control flow is returned to the filter set, we still have the original target and selector. In the new design, one has to explicitly substitute the target and selector, using filters SetTarget and SetSelector or the equivalent SetProperty filters. By default, the original target and selector is used, after executing the filter action.

6.2.4.5 Replicated filter expressions

As illustrated by the translation of filters TracingIn and TracingOut, the newly introduced atomic filter result in replicated filter expressions. We have to specify

the same matching pattern in every filter which we translate to. To address this issue we propose a new extension to the Composition Filters language called *Filter Composition*. This enables declarative composition of filters and filter actions. Section 6.3 elaborates on this.

6.2.4.6 Key Characteristics

We summarize the key characteristics:

- Atomic filters that accommodates all common operations on the properties of a message.
- Atomic filters that encapsulates all common operations on the control flow.
- The introduction of variables as a way to enforce the prescriptive behavior of the filters, and a lightweight mechanism for sharing information between filters.
- Substitution expressions are not executed by default, only using filters SetTarget and SetSelector or the equivalent SetProperty filters.
- Variables are scoped within a filter module. The scope of properties depend on the specific kind of properties, but most properties are scoped with a thread.

6.2.4.7 Generality of the approach

The proposed atomic filters are Composition Filters specific. However, the functionality that most of these filters implement is not. Almost all aspect-oriented programming languages offer a means to query and manipulate properties of the message, or allow changing the control flow. Creating atomic advices for aspect-oriented programming languages not only enables easier constructing of more complex advices from these atomic advices, but also improves reasoning of an aspect-oriented programming language. Atomic advices can then be combined into an aspect library, for everyone to use and compose. The atomic filters presented in this chapter serve as a step towards reusable advice libraries. The next section details how one can construct filters from these advice libraries, using a declarative language.

6.3 Filter Composition Language

The previous section presented a set of atomic filters and showed how we can construct more complex filters from this set. Currently, Composition Filters does not offer a declarative way to compose these complex filters, only through the use of filter sets in filter modules. Also, section 6.2.4.5 discussed that the usage of the new atomic filters resulted in replicated filter expressions. In this section we introduce a new language extension to Composition Filters that enables the declarative composition of filters. First, consider the following example from the previous section:

```
1 tracingOut : TracingOut = { shouldTrace => [*.*] tracer.traceOut }
```

Listing 6.15: TracingOut Filter

We translated this into the following sequence of canonical filters:

```
1  f1 : SetVariable(variable:!thetarget, property:target) = { shouldTrace => [*.*] };
2  f2 : SetVariable(variable:!theselector, property:selector) = { shouldTrace => [*.*] };
3  f3 : SetVariable(variable:!theargs, property:arguments) = { shouldTrace => [*.*] };
4  f4 : SetVariable(variable:!retval, property:returnvalue) = { shouldTrace => [*.*] };
5  f5 : Advice(target:tracer, selector:"traceOut", thetarget:!thetarget, theselector:!
       theselector, args:!theargs, retval:!retval) = { shouldTrace => [*.*] }
```

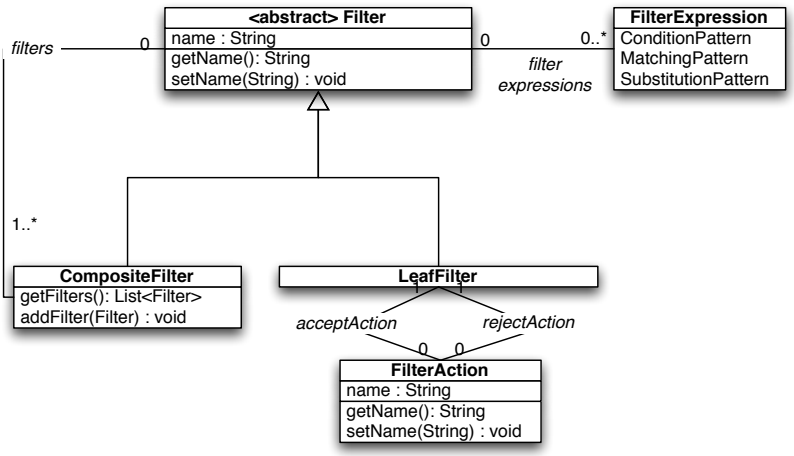
Listing 6.16: TracingOut Filter using Atomic Filters

This expresses a complex filter in five atomic filters. However, this introduces the problem that the same matching pattern, in this case `shouldTrace => [*.*]`, has to be repeated five times. We would like to specify this pattern only once. This replication is inherent to using a set of (smaller) atomic filters.

To prevent repetitive filter expression specifications, we propose a filter composition language. This language enables declarative composition of composite filters using a sequence of other filters or using accept and reject filter actions.

We use the composite pattern [GHJV95] to illustrate the filter composition language. Figure 6.5 shows the instantiated composite pattern for our language.

Figure 6.5 defines one abstract super class, called Filter. This class has two properties, a name and a list of FilterExpressions, with the appropriate get and set methods. This class is sub classed to either class CompositeFilter or class LeafFilter. Class CompositeFilter has a list of filters that it is composed of. There has to be at least one filter in this list. Class LeafFilter has two actions associated with it, either an accept action or a reject action. Each leaf filter has to have these two actions.



Figuur 6.5: Class diagram of new filter structure

6.3.1 Semantics

There are two ways to create a filter, either as a sequential composition of other filters or as a composition of two filter actions.

6.3.1.1 Leaf Filters

We can construct a filter through the composition of two filter actions. For example, consider the construction of filter SetProperty from filter actions SetPropertyAction and NopAction:

```
1 compositions
2 {
3   compose SetProperty of acceptaction SetPropertyAction and rejectaction NopAction;
4 }
```

This creates a new filter called SetProperty using two filter actions. If this filter accepts, filter action SetPropertyAction is executed. If this filter rejects, filter action NopAction is executed.

6.3.1.2 Composite Filters

We can expand an occurrence of the composite filter to the specified sequence of filters. All filter parameters that are specified in the super filter are passed to all filters in the sequence. Also, all filters inherit the condition, matching and substitution patterns from the super filter.

Consider the following definition of filter TracingOut:

```
1 compositions
2 {
3   compose TracingOut of filters: SetVariable(variable:!thetarget, property:target),
4     SetVariable(variable:!theselector, property:selector),
5     SetVariable(variable:!theargs, property:arguments),
6     SetVariable(variable:!retval, property:returnvalue),
7     Advice(target:"tracer", selector:"traceOut", tracetarget:!thetarget, traceselector:!
          theselector, args:!theargs, retval:!retval);
8 }
```

Section 6.2.4.3 discussed the details of filter TracingOut. Once we have defined filter TracingOut we can use this filter in either InputReturnFilter or OutputReturnFilter sets. The following specification states that all messages should be traced if condition shouldTrace is true:

```
1 inputreturnfilters
2   tracingOut : TracingOut (tracefile:"trace.log") = { shouldTrace => [*.*] tracer.trace }
```

This filter will be translated into the following sequence of filters:

```
1 inputreturnfilters
2   f1 : SetVariable(tracefile:"trace.log", variable:!thetarget, property:target) = {
         shouldTrace => [*.*] tracer.trace };
3   f2 : SetVariable(tracefile:"trace.log", variable:!theselector, property:selector) = {
         shouldTrace => [*.*] tracer.trace };
4   f3 : SetVariable(tracefile:"trace.log", variable:!theargs, property:arguments) = {
         shouldTrace => [*.*] tracer.trace };
5   f4 : SetVariable(tracefile:"trace.log", variable:!retval, property:returnvalue) = {
         shouldTrace => [*.*] tracer.trace };
6   f5 : Advice(tracefile:"trace.log", tracetarget:!thetarget, traceselector:!theselector,
         args:!theargs, retval:!retval) = { shouldTrace => [*.*] tracer.trace }
```

All of the filters have the same condition, matching and substitution pattern. Also, all filters have an filter parameter, called tracefile, in the parameter list. Filter parameters that are specified in the composite filter are propagated to all sub filters.

6.3.2 Constraints

We now explain the requirements of each CompositionSpecifier. We can create two types of filters, either a composite filter or a leaf filter. In case of a leaf filter, a sentence in the grammar would look like:

```
1 compose FILTER_NAME of acceptaction ACCEPTING_FILTER_ACTION and rejectaction
      REJECTING_FILTER_ACTION;
```

The elements in this sentence are:

FILTER_NAME : the name of the filter we are defining.
ACCEPTING_FILTER_ACTION : the name of the filter action that is executed if this filter accepts.
REJECTING_FILTER_ACTION : the name of the filter action that is executed if this filter rejects.

There are some constraints on these elements:

FILTER_NAME : this must be a unique name and not a keyword in the language.
ACCEPTING_FILTER_ACTION : this must be the fully qualified name of a subclass of class FilterAction. This must be an existing class.
REJECTING_FILTER_ACTION : similar to the accepting action, this must be the fully qualified name of a subclass of class FilterAction. This must be an existing class.

The compiler can enforce all these constraints. Only if all the constraints are met, can we use this filter.

In case of a composite filter, a sentence in the grammar would look like:

```
1 compose FILTER_NAME of filters FILTER_1, FILTER_2, ... ,FILTER_N ;
```

The elements in this sentence are:

FILTER_NAME : the name of the filter we are defining.
FILTER_1...FILTER_N : the name of the filter that is part of this filter definition.

Again, there are some constraints on these elements:

FILTER_NAME : this must be a unique name and not be a keyword in the language.
Filter_* : this filter must be present in the system after inspecting all sources, including Compose* concerns.

The compiler can enforce all these constraints. A filter can only be used if all the constraints are met. We have chosen here to only use a sequential composition operator: ‘,‘. One can imagine other composition operator, e.g. a parallel operation: ‘||‘.

We now present an example that uses the new language to create two new filters.

6.3.3 An example

Consider the four, respectively five filters, in listings 6.13 and 6.14, we can now create a single TracingIn and TracingOut filter using these four, respectively five filters.

```
concern TracingConcern
{
  filtermodule TracingModule
  {
    externals
      tracer : TracingLib.Tracer = TracingLib.Tracer.Instance;
    conditions
      shouldTrace : tracer.shouldTrace;
    inputfilters
      tracingIn : TracingIn = { shouldTrace => [*.*] tracer.traceIn }
    inputreturnfilters
      tracingOut : TracingOut = { shouldTrace => [*.*] tracer.traceOut }
  }

  compositions
  {
    compose TracingIn of filters:
      SetVariable(variable:!thetarget, property:target),
      SetVariable(variable:!theselector, property:selector),
      SetVariable(variable:!theargs, property:arguments),
      Advice(target:"tracer", selector:"traceOut", tracetarget:!thetarget,
        traceselector:!theselector, args:!theargs);
    compose TracingOut of filters:
      SetVariable(variable:!thetarget, property:target),
      SetVariable(variable:!theselector, property:selector),
      SetVariable(variable:!theargs, property:arguments),
      SetVariable(variable:!retval, property:returnvalue),
      Advice(target:"tracer", selector:"traceOut", tracetarget:!thetarget,
        traceselector:!theselector, args:!theargs, retval:!retval);
  }
}
```

Listing 6.17: TracingOut Filter using Canonical Set

Section 6.3.4.1 presents an example of the filter composition language using filter actions.

6.3.4 Discussion

6.3.4.1 Bootstrapping Filters

We can now define filters as a composition of either a sequence of other filters or of two filter actions. As such we can create the atomic filters in the Composition Filters language itself. We can define concern AtomicFilters as shown in listing 6.18.

```
concern CanonicalFilters
{
  compositions
  {
    compose SetProperty of acceptaction SetPropertyAction and rejectaction NopAction;
    compose SetVariable of acceptaction SetVariableAction and rejectaction NopAction;
    compose SetTarget of acceptaction SetTargetAction and rejectaction NopAction;
    compose SetSelector of acceptaction SetSelectorAction and rejectaction NopAction;
    compose Return of acceptaction ReturnAction and rejectaction NopAction;
    compose Exit of acceptaction ExitAction and rejectaction NopAction;
    compose Advice of acceptaction AdviceAction and rejectaction NopAction;
    compose Call of acceptaction CallAction and rejectaction NopAction;
  }
}
```

Listing 6.18: Concern CanonicalFilters

In addition, we can create the predefined filters, as discussed in section 6.2.4.3.

```
concern OriginalFilters
{
  compositions
  {
    compose Dispatch of filters Call, Return;
    compose Send of filters SetProperty(property:sender, property1:server),
      SetProperty(property:server, property1:target), Call Return;
    compose Error of filters Advice, Exit;
    compose Substitution of filters SetTarget, SetSelector;
    compose Before of filters Advice;
    compose After of filters Advice;
  }
}
```

Listing 6.19: Concern OriginalFilters

6.3.4.2 Propagation of Filter Expressions

The filter developer has to be aware while creating such composite filters that the condition, matching and substitution patterns are propagated to the filters in the composite filters. For the condition and matching pattern this is not a real problem, as in most case, the developer wants to execute all filters under

the same conditions and for the same set of messages. However, the substitution expression can pose a problem. Imagine the case where a filter substitutes the target or selector halfway in the filter set. In such a case, the developer might only execute the filters before this substitute filter, since filters after this filter might not match the modified message anymore. Also, in case the developer uses two Call filters to call two different methods, he or she can no longer use the substitution expression. In this case he or she has to use filter parameters to achieve this.

For example, consider *composed* filter TracingOut: `tracingOut : TracingOut = { shouldTrace => [*.*] tracer.traceOut }`. We see that the decomposition of this filter is:

```
1  f1 : SetVariable(variable:!thetarget, property:target) =
2    { shouldTrace => [*.*] tracer.tracer.traceOut };
3  f2 : SetVariable(variable:!theselector, property:selector) =
4    { shouldTrace => [*.*] tracer.tracer.traceOut };
5  f3 : SetVariable(variable:!theargs, property:arguments) =
6    { shouldTrace => [*.*] tracer.tracer.traceOut };
7  f4 : SetVariable(variable:!retval, property:returnvalue) =
8    { shouldTrace => [*.*] tracer.tracer.traceOut };
9  f5 : Advice(tracetarget:!thetarget, traceselector:!theselector, args:!theargs,
10   retval:!retval) = { shouldTrace => [*.*] tracer.traceOut }
```

Listing 6.20: Decomposed Filter TracingOut

Here each filter has the same filter expression: `shouldTrace => [*.*] tracer.traceOut`. In this case this is not a problem as the substitution expression is ignored by all filters except filter Advice and there is no explicit substitute filter. This may not always be the case, one has to be aware of this when defining composed filters.

Also, the example is not entirely the same as the original single filter implementation. We now check condition shouldTrace for each filter. If this condition changes during the evaluation of these filters, some part of the filters may accept while other reject. This might have unintended effects. One way to solve this is to use the original assumption of Composition Filters. This assumption states that the conditions are not changed during the evaluation of the filter set. A snapshot of all conditions is taken at the start of each of the four filter sets. This prevents this issue, however the condition is still checked multiple times, as opposed to a single filter implementation.

6.3.4.3 Related Work

There are some aspect-oriented programming languages that offer features that enable the building of aspects libraries. One example is to use an abstract aspects and pointcut. The user of the aspect, instantiates the aspect and fills in

the pointcut. However, this construct does not enable composition of advices from other advices. This construct is convenient for building aspect libraries. An example of an aspect library is the Spring framework [Fra] that contains an aspect library which contains aspects like Caching, Exception Handling, Logging and Transactions.

The hyperspaces approach developed by IBM Watson Research Center [TOHS05], also provides a composition language. The hyperspaces approach has evolved to Concern Manipulation Environment [IBM]. Hyperspaces assume a fully symmetric model for composition. One defines hyperslices that are a set of conventional modules, written in any formalism. Hyperslices are intended to encapsulate concerns in dimensions other than the dominant dimension. One can then define hypermodules that are a set of hyperslices, together with a composition rule that specifies how the hyperslices must be composed to form a single, new hyperslice that combines the hyperslices. This is similar to the way one can compose filters from other filters. However in our composition language, the composition rules are limited to sequential composition of other filters or of two filter actions.

6.3.4.4 Generality

The extension we proposed here is unique for Composition Filters and other AOP languages. To our knowledge, there is no other AOP language which offers a declarative way of creating complex filters from other filters and filter actions, or complex advices from other advices. In languages like AspectJ, one can only compose advices using shared join points or using standard object-oriented techniques like method calls. However, this is by no means declarative and does not provide a reusable entity. Composition Filters offers a set of atomic filters, this set in combination with the proposed filter composition language, enables building reusable aspect libraries.

6.4 Conclusions

This chapter proposed three contributions to the Composition Filters model. For each of these contributions, we explained the problem in detail, proposed a solution and discussed the limitations of the proposal.

Filter Sets: We proposed the separation of *what* behavior is to be executed and *when* that behavior is to be executed. This separation was not fully existent in Composition Filters. We showed how the current filters can be mapped to

the new model, as well as the requirements and impact on other parts of the Composition Filters language. Especially parametrization of filter modules and an extension of the filter module ordering language, are required to implement this extension.

We proposed the introduction of two additional filter sets, called *inputreturnfilter* and *outputreturnfilters*, that filter on the return of messages. These are similar to filter sets *inputfilters* and *outputfilters*, that filter on the incoming and outgoing messages. Effectively, separating *what* behavior is to be executed and *when* that behavior is to be executed. We unified the matching model for the four filter sets. All condition, matching and substitution expressions in returning filters are evaluated on the return of messages. Finally, we proposed an extension to the filter module ordering language, to enable the declaration of different orderings of filter modules for incoming and returning message.

Atomic Filters: We also proposed atomic filters. Atomic filters can be used to build complex filters or advices. These filters specify in a declarative and explicit manner the precise actions of filters, such that they are easier to understand and to reason (automatically) about and have well defined semantics. The atomic filters were derived from the analysis of the behavior of filters conducted in chapter 5. We also proofed that we could decompose the current filters into a sequence of atomic filters.

We introduced one filter *SetProperty(...)* that is able to manipulate the properties of a message. Next we introduce the notion of variables and filter *SetVariable(...)*, as a lightweight mechanism for sharing state between the filters in the filter set and advice methods. These advice methods can be called using either *Call* or *Advice* filters. We also introduced two filters that explicitly affect the control flow within the filter set, i.e. filter *Return* and *Exit*. We verified that these filters were indeed atomic, using the resource-operation model and by expressing the original filters as a sequence of atomic filters. The introduction of these atomic filters aids in a better understanding of the filters, and can pave the way to create truly reusable and composable filter or advice libraries.

Filter Composition Language: To enable the declarative composition of filters and to address the replication filter expressions, we introduced a novel filter composition language. This language enables the declarative composition of complex filters from either a sequence of filters or from two filter actions. We showed that one could compose the current and new atomic filters using the language.

The contributions of this chapter are:

- A separation between *what* behavior is executed from *when* this behavior

is executed. This is currently encapsulated in the filter definitions, and thus hinders reasoning.
- Atomic filters for querying and manipulating properties of a message or control flow. These filters can be used to create other (more complex) filters. This also increases reasoning, since the semantics of filters are no longer hidden inside the filter type.
- A declarative language for composing filters from filter actions or a sequence of other filters, in order to create complex filter and advice libraries.

All three contributions together create a more explicit and declarative model that supports much more (automated) reasoning about the behavior of aspects and contribute to the formal semantics of Composition Filters.

The extensions to the Composition Filters model proposed in this chapter have not been implemented. We consider this as a topic for future work.

Conclusions and Contributions

7

In chapter 1, we presented a table that provides an overview of the structure of thesis. We now use the same table to summarize the solutions presented in this thesis (see table 7.1).

Tabel 7.1: Structure of this thesis

Language	**Problem**	**Solution**	**Chap.**
Object-Oriented and Imperative	Crosscutting Concerns	Aspect-Oriented Programming	1
Aspect-Oriented	How to introduce AOP in Industry?	Experiences in introducing AOP at ASML	2
Aspect-Oriented	Does AOP reduce the Software Development Effort?	An Assessment of an Aspect-based Approach to Tracing	3
Aspect-Oriented	Behavioral Conflicts among Aspects	Behavioral Conflict Detection Tools	4 & 5
Composition Filters	Limitations of Automated Reasoning	Improved Composition Filters Design	6

7.1 Experiences in introducing AOP at ASML

In the Ideals project, we conducted two case studies for transferring an aspect-oriented solution to ASML. These case studies were concerned with two distinct systems, which had widely different development methods. Chapter 2 reports on our experience introducing AOP at ASML. That chapter also presents a process that consists of several steps that aim at providing a solution fitting into the context of the company. This context consists of the programming language, design methods, software development process, etc. Understanding this context is in our view imperative for a successful adoption of AOP. Knowing the context also helps in better expressing the benefits and drawbacks of the solution. Finally, we have to elicit and address key worries, before an aspect-oriented solution can be transferred.

The proposed process does not guarantee the acceptance of an aspect-oriented solution, as one casestudy demonstrated. However, it provides guidelines to create the optimal conditions for technology transfer. One of the described casestudies resulted in transferring the proposed technology and prototype to the company. Parts of the chapter have been published in [DGB+06].

7.2 A Controlled Experiment for the Assessment of Aspects

The goal of chapter 3 is to assess whether aspect-oriented programming reduces the development effort of programs that include of crosscutting concerns. Chapter 3, presents a controlled experiment for the assessment of an aspect-base approach to tracing. Tracing is sometimes dismissed as a trivial aspect. However, section 3.1 shows that this is certainly not the case in the actual realization of tracing at ASML. Additionally, this "simple" aspect can serve as an excellent driver to adopt AOP, as also suggested by Colyer et. al. [CC04]. Aspects can be used to address a wide range of crosscutting concerns, but the benefits of adopting AOP are not as obvious for heterogeneous or more complex concerns, in comparison with homogeneous concerns like tracing and profiling. These more "straightforward" aspects should not be overlooked, but rather be embraced as effective examples for introducing AOP in industry.

This thesis provides the design of a controlled experiment that can be used to quantify the benefits of AOP in other organizations. The design can easily be adopted for other aspects, other base code, and other scenarios.

We conducted the experiment with 20 ASML developers. The developers had to execute five simple tracing-related change scenarios twice. First, the developers had to implement the scenarios manually. After wards the subjects had to implement the scenarios in a different program using an aspect-oriented approach to tracing (using an AOP language called *WeaveC*). Section 3.4 offers a detailed discussion of validity threats and how we have tried to reduce validity threats through careful design of the experiment.

The results from this experiment show that, overall, the subjects were able to execute the scenarios 6% faster using the AOP solution. More important however, was the reduction in severity of errors when using the AOP solution by 77%. Concluding, in the experiment, a substantial reduction of errors was achieved when using AOP, with even slightly less effort.

Section 3.5 shows the results of a survey which was conducted amongst 26 users of *WeaveC*. The results from this survey confirm that the users "experience" the benefits of AOP.

The contributions of this chapter are:

- A detailed discussion of a real-world aspect, namely *Tracing*. Although this is usually considered trivial, we show that this aspect is complex. In addition, aspect tracing can serve as a key driver for introducing AOP, because it provides a lot of advantages typically in many places.
- A design of a controlled experiment that can be used to quantify the benefits of using an aspect-based approach to tracing in an industrial setting.
- The results of a controlled experiment with 20 professional software developers, using an aspect-based approach to tracing. The results show that overall, the subjects were able to execute the scenario 6% faster and introduced 77% less (severe) errors.

The design and results of the experiment are in principle specific to tracing, WeaveC and ASML: it is not scientifically justifiable to generalize the results. However, as we discuss in section 3.7 there are many indications that this experiment would yield similar results for other concerns, base and aspect languages and organizations.

In conclusion, we believe that the results are a clear indication that AOP can help to reduce the development costs of software, both in terms of effort and error reduction.

7.3 Behavioral Conflict Detection Tools

In chapters 4 and 5, we present the problem of behavioral conflicts among aspects. We explain the problem of behavioral conflicts using an example that we encountered at ASML, this illustrates the relevance of the problem. We scoped our work using a discussion about composition conflicts. Our scope is to detect both specific and generic behavioral conflicts between aspects at shared join points.

The contributions of these chapters are:

- A novel approach for detecting behavioral conflicts, based on an abstraction mechanism of advice behavior in terms of resources and operations. This mechanism has several advantages:
  - It allows for expressing behavior (and conflicts) without involving (needless) implementation-level details.
  - It allows to express not only generic, or universal, conflicts, but also domain- and application-specific conflicts.
  - It reduces the computational complexity of the conflict detection analysis.
- A concrete and detailed application of the approach for detecting behavioral conflicts in Composition Filters.
- An analysis of possible behavioral conflicts among filters in Composition Filters. This analysis can also be reused for detecting behavioral conflicts in other AOP approaches.
- An novel technique for detecting behavioral conflicts at runtime, adopting our resource-based approach.

We have demonstrated that behavioral conflicts are important to address, as illustrated by an example from ASML and other work in this area. The implementation of our approach in Compose* demonstrates that behavioral conflict detection in practice is feasible. The implementation also demonstrates that a carefully designed language with declarative features, increases the ability to automated reasoning about the (composition of) behavior of aspects.

The current and earlier versions of the conflict detection approach have been published in [DSBA05], [DBA06] and [DBA07a]. The proposed runtime extension has been published in [DBA07b].

7.4 Improved Composition Filters Design

Chapter 6 proposes three contributions to the Composition Filters model. These three contributions address limitations on the ability to automatically reason about filters. For each of these contributions, we explained the problem in detail, proposed a solution and discussed the limitations of the proposal.

We proposed the separation of *what* behavior is to be executed and *when* that behavior is to be executed. This separation did not fully exist in Composition Filters. We have shown how the current filters can be mapped to the new model, as well as the requirements and impact on other parts of the Composition Filters language. Especially parametrization of filter modules and an extension of the filter module ordering language are required to implement this extension.

We also proposed atomic filters. Atomic filters can be used to build more complex filters or advices. These filters specify in a declarative and explicit manner the precise actions of filters, such that these actions and filters are easier to understand and to reason (automatically) about. The atomic filters was derived from the analysis of the behavior of filters conducted in chapter 5. The introduction of atomic filters contributes to a better understanding of the filters and to the formal semantics of Composition Filters. The filters can be used for creating reusable and composable filter or advice libraries. We also showed that we can decompose the current filters into a sequence of atomic filters.

Finally, we presented a new filter composition language. This language enables the declarative composition of complex filters from the atomic filters. One cannot only compose filters using filter actions, but also compose filters from (a sequence of) other filters. The current filters can be defined as a composition of the new atomic filters.

The contributions of this chapter are:

- An improvement of the Composition Filters model, which separates *what* behavior is executed from *when* this behavior is executed, thus improving the ability to reason.
- A proposal for *atomic filters* which can be used for querying and manipulating properties of a message or control flow. These filters can be used to create other (more complex) filters. This also improves reasoning, since a part of the semantics is made explicit.
- A newly proposed declarative language for composing filters from filter actions or a sequence of other filters, to create complex filters and advice libraries.

All three contributions together create a more explicit and declarative model that supports (automated) reasoning about the behavior of aspects. Using the proposed extensions we can build complex and reusable aspect libraries.

Samenvatting

Aspect-georiënteerde software ontwikkeling is geïntroduceerd als een oplossing om de modulariteit van software te verbeteren, wanneer deze crosscutting functionaliteit bevat. In tegenstelling tot object-georiënteerde of procedurele technieken, is aspect-georiënteerd programeren (AOP) nog niet op grote schaal toegepast in de praktijk. In deze dissertatie onderzoeken we de toepasbaarheid van AOP in de praktijk en stellen een nieuwe aanpak voor om conflicten in het gedrag tussen aspecten te detecteren.

We beschrijven onze ervaring met het overdragen van een aspect-georiënteerde oplossing naar een bedrijf genaamd ASML (Advanced Semi-conductor Material Lithography). We onderzoeken de criteria om zo'n oplossing te accepteren. Dit doen we gebaseerd op twee case-studies die we bij ASML hebben uitgevoerd. We laten een proces zien dat onder andere de voordelen van AOP laat zien en eventuele zorgen van de industriële partners expliciet maakt en aanpakt.

We hebben een gecontroleerd experiment uitgevoerd om te bepalen wat de voor en nadelen zijn van een aspect-geörienteerde implementatie van *tracing*. Twintig ontwikkelaars van ASML hebben deelgenomen aan het experiment, deze twintig ontwikkelaars moesten 5 simpele onderhoudsactiviteiten uitvoeren. De resultaten van het experiment laten zien dat, vergeleken met de originele procedurele implementatie, bij de aspect georiënteerde oplossing de ontwikkeltijd met gemiddeld 6% is gereduceerd en het effect van fouten met gemiddeld 77% is gereduceerd. Voor een deel van de ontwikkel scenario's hebben we een statistisch

significant resultaat op een betrouwbaarheidsinterval van 95% gehaald.

Het zogenaamde aspect interferentie probleem kan gezien worden als een van de belangrijkste belemmeringen om AOP te accepteren in de industrie. Aspecten kunnen onafhankelijk van elkaar ontwikkeld worden en zich afzonderlijk correct gedragen. Echter, door verwachte of onverwachte compositie van aspecten kan ongewenst gedrag naar voren komen. In deze dissertatie behandelen we het probleem van gedrag conflicten tussen aspecten op het zelfde punt in het programma. We illustreren dit soort conflicten aan de hand van een voorbeeld gebaseerd op aspecten bij ASML. We tonen een aanpak om conflicten in het gedrag tussen aspecten te detecteren. Deze aanpak is gebaseerd op een nieuwe abstractie van het gedrag van aspecten in termen van (gedeelde) entiteiten en acties op deze entiteiten. Deze abstractie stelt ons in staat om het complexe gedrag van aspecten op een simpele manier uit te drukken, die bruikbaar is voor automatische conflict detectie. De aanpak behelst conflict-detectie regels die niet alleen generieke conflicten kunnen detecteren maar ook gebruikt kunnen worden om meer specifieke conflicten te detecteren.

Onze aanpak voor de detectie van conflicten in het gedrag is generiek voor AOP talen, in deze dissertatie laten we de toepasbaarheid van onze aanpak zien voor een specifieke AOP taal, genaamd Composition Filters. De toepassing op Composition Filters laat zien dat het gebruik van een declaratieve taal kan worden benut voor automatische conflict detectie. We geven een gedetailleerd overzicht van het analyseproces en geven aan welke informatie er nodig is van de aspect ontwikkelaar om de analyse te kunnen uitvoeren. We bespreken ook wanneer statische analyse niet voldoende is en hoe we dynamische conflict detectie kunnen uitvoeren. Deze dynamische conflict detectie gebruikt de resultaten van de statische analyse om op een efficiënte manier dynamische conflict detectie mogelijk te maken.

Tot slot stellen we drie verbeteringen van Composition Filters voor om nog beter automatisch en manueel te kunnen redeneren over de filters. De eerste verbetering scheidt *welk* gedrag wordt uitgevoerd van *wanneer* dit gedrag wordt uitgevoerd. Ten tweede introduceren we filters die gebruikt kunnen worden om meer complexe filters te bouwen. De semantiek van deze filters is eenduidig gedefinieerd. De introductie van deze filters heeft duidelijke voordelen om automatisch te kunnen redeneren, echter resulteert dit wel in het herhaaldelijk specificeren van meerdere filters. Om dit op te lossen, introduceren we een derde verbetering die ontwikkelaars in staat stelt om op een declaratieve manier (complexe) filters te maken van andere filters, door middel van een filter compositie taal.

Bibliography

[ABV92] M. Akşit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In O. Lehrmann Madsen, editor, *Proc. 7th European Conf. Object-Oriented Programming*, pages 372–395. Springer-Verlag Lecture Notes in Computer Science, 1992. 9

[ASM] ASML. ASML. http://www.asml.com. 1

[Asp] AspectJ project. http://www.eclipse.org/aspectj/. 86

[AT88] M. Akşit and A. Tripathi. Data Abstraction Mechanisms in SINA/st. In *Proceedings of the conference Object-Oriented Systems, Languages and Applications*, volume 23 of *ACM Sigplan Notices*, pages 267–275, 1988. 9

[BA05] L. Bergmans and M. Akşit. Principles and Design Rationale of Composition Filters. In Filman et al. [FECA05], pages 63–95. 9, 133

[BC00] C. Y. Baldwin and K. B. Clark. *Design Rules - The Power of Modularity*, volume 1. MIT Press, March 2000. 74

[BCM05] D. Balzarotti, A. Castaldo, and M. Monga. Slicing AspectJ Woven Code. In *FOAL '05: The 4th Workshop on Foundations of Aspect-Oriented Languages*, Chicago, USA, March, 14 2005. 169

[BCMS03] R. Bodkin, A. Colyer, J. Memmert, and A. Schmidmeier, editors. *AOSD Workshop on Commercialization of AOSD Technology*, March 2003. 10, 44, 73

[Ber66] A. J. Bernstein. Program Analysis for Parallel Processing. *IEEE Trans. on Electronic Computers*, EC-15:757–762, 1966. 95, 170

[BF06] R. Bodkin and J. Furlong. Gathering Feedback on User Behaviour using AspectJ. In *Proceedings of the Industrial Track of the fifth International Conference on Aspect-Oriented Software Development*, 2006. 44, 73

[BvDDT07] 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. 28, 47

[BvDT06] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering Faults in Idiom-Based Exception Handling. In *Proceedings of the International Conference on Software Engineering (ICSE'06)*, pages 242–251. ACM Press, 2006. 4, 19

[BvDvET05] M. Bruntink, A. van Deursen, R. van Engelen, and T. Tourwé. On the Use of Clone Detection for Identifying Cross Cutting Concern Code. *IEEE Transactions on Software Engineering*, 31(10):804–818, 2005. 18

[CC79] T. D. Cook and D. T. Campbell. *Quasi-Experimentation: Design and Analysis Issues for Field Settings*. Houghton Mifflin Company, 1979. 68

[CC04] A. Colyer and A. Clement. Large-scale AOSD for middleware. In Lieberherr [Lie04], pages 56–65. 10, 45, 73, 77, 220

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press: Cambridge, 1999. 153

[CKLM08] C. Clifton, S. Katz, G. T. Leavens, and M. Mezini, editors. *FOAL: Foundations Of Aspect-Oriented Languages*, March 2008. 232

[CL02] C. Clifton and G. T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In Ron Cytron and

Gary T. Leavens, editors, *FOAL 2002: Foundations of Aspect-Oriented Languages (AOSD-2002)*, pages 33–44, March 2002. 90

[CLL06] C. Clifton, R. Lämmel, and G. Leavens, editors. *FOAL: Foundations Of Aspect-Oriented Languages*, March 2006. 231

[CSF+06] N. Cacho, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development*, pages 109–121, New York, NY, USA, 2006. ACM. 74, 75

[Daw91] J. Dawes. *The VDM-SL reference guide*. Pitmann, 1991. 112

[DBA05] P. Durr, L. Bergmans, and M. Aksit. A Formal Model for SECRET. Technical report, University of Twente, 2005. 133

[DBA06] P. Durr, L. Bergmans, and M. Aksit. Reasoning about Semantic Conflicts between Aspects. In R.Chitchyan, J. Fabry, L. Bergmans, A. Nedos, and A. Rensink, editors, *Proceedings of ADI'06 Aspect, Dependencies, and Interactions Workshop*, pages 10–18. Lancaster University, Lancaster University, Jul 2006. 110, 171, 222

[DBA07a] P. Durr, L. Bergmans, and M. Aksit. Detecting Behavioral Conflicts among Crosscutting Concerns. In van Engelen and Voeten [vEV07]. 171, 222

[DBA07b] P. Durr, L. Bergmans, and M. Aksit. Static and Dynamic Detection of Behavioral Conflicts between Aspects. In O. Sokolsky and S Tasiran, editors, *Proceedings of the 7th Workshop on Runtime Verification*, number 4839 in LNCS, pages 38–50, Vancouver, Canada, March 2007. Springer Verlag. 222

[DBT08] P. Durr, L. Bergmans, and B. Tekinerdogan. Semantics of Architectural Aspects. Milestone Document M6.17, AOSD Network of Excellence, February 2008. 151, 171

[DFS02] R. Douence, P. Fradet, and M. Südholt. A Framework for the Detection and Resolution of Aspect Interactions. In *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002*, Lecture Notes in Computer Science, Pittsburgh, US, October,6 2002. Springer-Verglag. 92

[DFS04] R. Douence, P. Fradet, and M. Südholt. Composition, Reuse and Interaction Analysis of Stateful Aspects. In Lieberherr [Lie04], pages 141–150. 169

[DFS05] R. Douence, P. Fradet, and M. Südholt. Trace-Based Aspects. In Filman et al. [FECA05], pages 201–217. 166, 169

[DGB+06] P. Durr, G. Gulesir, L. Bergmans, M. Aksit, and R. van Engelen. Applying AOP in an Industrial Context. In *Workshop on Best Practices in Applying Aspect-Oriented Software Development*, Mar 2006. 45, 47, 220

[Dij82] E. Dijkstra. On the Role of Scientific Thought. In *Selected writings on Computing: A Personal Perspective*, pages 60–66. Springer-Verlag, 1982. 7

[Doo06] D. Doornenbal. Analysis and Redesign of the Compose* Language. Master's thesis, University of Twente, November 2006. 181

[dR07] A. de Roo. Towards More Robust Advice: Message Flow Analysis for Composition Filters and its Application. Master's thesis, University of Twente, March 2007. 138, 141, 155, 157

[DSBA05] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about Semantic Conflicts between Aspects. In *EIWAS '05: The 2nd European Interactive Workshop on Aspects in Software*, Brussel, Belgium, September, 1-2 2005. 110, 171, 222

[EMK+06] M. Eichberg, M. Mezini, S. Kloppenburg, K. Ostermann, and B. Rank. Integrating and Scheduling an Open Set of Static Analyses. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Science, 2006. 170

[Eva] J. Evain. CECIL. http://www.mono-project.com/Cecil. 41

[FECA05] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2005. 227, 230, 235

[FGR07] F. Filho, A. Garcia, and C. Rubira. Extracting Error Handling to Aspects: A Cookbook. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance*. IEEE, Oct 2007. 84

[Fra] Spring Framework. Spring framework. http://www.springframework.org/. 215

[GBF+07] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dósea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In Erik Ernst, editor, *Proceedings of the 21st European Conference on Object-Oriented Programming*, volume 4609 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2007. 36, 75

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: elements of reusable object-oriented software*. Addison Wesley, 1995. 74, 208

[GI04] The Open Group and IEEE. Regular expressions. *The Open Group Base Specifications, IEEE Std 1003.1*, 1(6), 2004. 98, 145, 153

[GK06] M. Goldman and S. Katz. Modular Generic Verification of LTL Properties for Aspects. In Clifton et al. [CLL06], pages 11–19. 90

[GK07] M. Goldman and S. Katz. MAVEN: Modular Aspect Verification. In O. Grumberg and M. Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4424/2007 of *LNCS*, pages 308–322. Springer Berlin / Heidelberg, 2007. 167

[Gra] GrammaTech. CodeSurfer. http://www.grammatech.com/. 26, 109

[GSE+07] A. Garcia, S. Soares, M. Eaddy, M. Bartsch, and M. Akşit. Informal Workshop Report on Assessment of Aspect-Oriented Technologies. Technical report, AOSD, 2007. 10, 44, 73

[GSF+05] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. Modularizing Design Patterns with Aspects: a Quantitative Study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14, New York, NY, USA, 2005. ACM. 36, 74

[Hav05] W. Havinga. Designating Join Points in Composestar - a predicate-based superimposition language for Compose*. Master's thesis, University of Twente, May 2005. 126

[HD01] J. Hatcliff and M. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *CONCUR '01: Proceedings of the 12th International Conference on Concurrency Theory*, pages 39–58, London, UK, 2001. Springer-Verlag. 166, 168

[IBM] IBM Research. Concern Manipulation Environment. http://www.research.ibm.com/cme/index.html. 215

[IDE] IDEALS. IDEALS project page. http://www.esi.nl/site/projects/ideals/. 1, 82

[JM97] J. Jazequel and B. Meyer. Design by contract: the lessons of ariane. *The Computer Journal*, 30(1):129–130, January 1997. 22, 82

[Jon90] C. Jones. *Systematic Software Development using VDM, Second Edition*. Prentice Hall, 1990. 112

[Jon92] C. Jones. VDM Specification Language. *ISO publications*, 1992. Document N-246(I-9), BSI IST/5/-/19 and ISO/IEC JTC1/SC22/WG19, December 1992. 112

[Kat93] S. Katz. A Superimposition Control Construct for Distributed Systems. *ACM Trans. Program. Lang. Syst.*, 15(2):337–356, 1993. 90

[Kat06] S. Katz. Aspect Categories and Classes of Temporal Properties. In *Transactions on Aspect-Oriented Software Development*, 3880, pages 106–134. LNCS, 2006. 168

[KFG04] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying Aspect Advice Modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA, 2004. ACM Press. 167

[KG99] S. Katz and J. Gil. Aspects and Superimpositions. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 308–309, London, UK, 1999. Springer-Verlag. 90

[KK08] E. Katz and S. Katz. Incremental Analysis of Interference Among Aspects. In Clifton et al. [CKLM08], pages 28–38. 167, 168, 169

[KM80] D. Kapur and S. Mandayam. Expressiveness of the Operation Set of a Data Abstraction. In *POPL '80: Proceedings of the 7th ACM*

SIGPLAN-SIGACT symposium on Principles of programming languages, pages 139–153, New York, NY, USA, 1980. ACM Press. 95

[KPP+02] B. Kitchenham, S. Pfleeger, M. Pickard, P. Jones, and D. Hoaglin J. Emam J. Rosenberg. Preliminairy Guidelines for Empirical Research in Software Engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, March 2002. 51, 68

[Lad06] R. Laddad. AOP@Work: Myths about AOP, 2006. http://www-128.ibm.com/developerworks/java/library/j-aopwork15. 47

[LB05] C. Lopes and S. Bajracharya. An Analysis of Modularity in Aspect-Oriented Design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2005. ACM. 36, 74

[Lie04] K. Lieberherr, editor. *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004)*. ACM Press, March 2004. 228, 230

[LMWF93] N. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions : In Concurrent and Distributed Systems*. Morgan Kaufmann, 1993. 95, 170

[Lop97] C. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997. 93

[McC76] T. McCabe. A Complexity Measure. *IEEE Transactions on Software Engineering*, 2(2):308–320, 1976. 56

[Mica] Microsoft. C# Language. http://msdn2.microsoft.com/en-us/vcsharp/aa336809.aspx. 36

[Micb] Microsoft. Microsoft .NET Framework. http://www.microsoft.com/net/. 36

[Micc] Microsoft. Microsoft Visual Studio. http://msdn.microsoft.com/en-us/vstudio/default.aspx. 37

[Micd] Microsoft. MSDN - The MSBuild Reference. http://msdn2.microsoft.com/en-us/library/0k6kkbsd.aspx. 37

[MKL97] A. Mendhekar, G. Kiczales, and J. Lamping. RG: A Case-Study for Aspect-Oriented Programming. Technical Report SPL-97-009, Palo Alto Research Center, 1997. 45, 74

[MWB99] G. Murphy, R. Walker, and E. Baniassad. Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-oriented Programming. *IEEE Transactions on Software Engineering*, 25(4):438–455, 1999. 75, 76

[Nag05] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, 2005. 127, 185

[NBA05] I. Nagy, L. Bergmans, and M. Akşit. Composing Aspects at Shared Join Points. In *Proceedings of International Conference NetObjectDays, NODe2005*, Lecture Notes in Computer Science, Erfurt, Gergmany, 2005. Springer-Verglag. 92, 127

[NvEvdP07] I. Nagy, R. van Engelen, and D. van der Ploeg. An overview of Mirjam and WeaveC. In van Engelen and Voeten [vEV07]. 50

[oC] University of California. AspectBrowser. http://www-cse.ucsd.edu/users/wgg/Software/AB/. 5, 18, 22, 23

[PDS05] R. Pawlak, L. Duchien, and L. Seinturier. CompAr: Ensuring Safe Around Advice Composition. In *Proceedings of Formal Methods for Open Object-Based Distributed Systems*, Athens, Greece, June 2005. 170

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *Proceedings of the 18th IEEE Symposium Foundations of Computer Sience*, pages 46–57, 1977. 98, 153

[Ren] A. Rensink. GRaphs for Object-Oriented VErification (Groove). http://groove.sourceforge.net. 155

[RSB04] M. Rinard, A. Salcianu, and S. Bugrara. A Classification System and Analysis for Interactions in Aspect-Oriented Programs. In *Foundations of Software Engineering (FOSE)*. ACM, October 2004. 91

[S. 04] S. McConnell. *Code Complete - 2nd edition*. Microsoft Press, 2004. 17

[SDMF+06] M. Segura-Devillechaise, J. Menaud, T. Fritz, N. Loriant, R. Douence, and M. Sudholt. An Expressive Aspect Language for System Applications with Arachne. *Transactions on Aspect-Oriented Software Development*, 1(1):174–213, March 2006. 169

[SHH+05] D. Sjoberg, J. Hanney, O. Hansen, V. By Kampenes, A. Karahasanovic, N. Liborg, and A. Rekdal. A Survey of Controlled Experiments in Software Engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, September 2005. 73

[SPS] SPSS. SPSS Version 13. for Windows. http://www.spss.com/spss/. 62, 70, 72

[SR06] T. Staijen and A. Rensink. A Graph-Transformation-Based Semantics for Analysing Aspect Interference. In *Workshop on Graph Computation Models*, Natal, Brazil, 2006. 166, 169

[Sud97] T. Sudkamp. *Languages and Machines - An Introduction to the Theory of Computer Science, 2nd edition.* Addison-Wesley, 1997. 99

[TOHS05] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In Filman et al. [FECA05], pages 37–61. 215

[Unia] University of Twente. Compose*. http://composestar.sourceforge.net. 9, 42, 112

[Unib] University of Twente. WeaveC. http://weavec.sourceforge.net. 31

[vEV07] R. van Engelen and J. Voeten, editors. *IDEALS: Evolvability of Software-Intensive High-Tech Systems.* Embedded Systems Institute, Eindhoven, 2007. 229, 234

[vO06] M. van Oudheusen. Automatic Derivation of Semantic Properties in .NET. Master's thesis, University of Twente, August 2006. 154

[WBM99] R. Walker, E. Baniassad, and G. Murphy. An Initial Assessment of Aspect-Oriented Programming. In *Proc. 21st Int'l Conf. Software Engineering (ICSE '99)*, pages 120–130, 1999. 75

[WBM05] R. Walker, E. Baniassad, and G. Murphy. An Initial Assessment of Aspect-Oriented Programming. In Filman et al. [FECA05], pages 531–556. 75

[WHM07] D. Wiese, U. Hohenstein, and R. Meunier. How to Convince Industry of AOP. In *Proceedings of Industry Track of the 6th Conf. on Aspect-Oriented Software Development*, 2007. 10, 44, 73

[WRH+00] C. Wohlin, P. Runeson, M. Hörst, M. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation In Software Engineering* . Kluwer Academic Publishers, 2000. 51

[Xer] Xerox Corporation. The AspectJ Programming Guide. http://www.eclipse.org/aspectj/doc/released/progguide/index.html. 92

Appendix A - *Join Point Context API*

The interface of class JoinPointContext is:

String : getMatchingTarget() : returns the textual representation of the target in the matching expression.

String : getMatchingSelector() : returns the textual representation of the selector in the matching expression.

String : getSubstitutionTarget() : returns the textual representation of the target in the substitution expression.

String : getSubstitutionSelector() : returns the textual representation of the selector in the substitution expression.

Object : getFilterParameter(String) : returns the object corresponding to a key in the map of filter module parameters. This can be used for retrieving the target, server or sender of a message.

String : getFilterParameter(String) : returns the string corresponding to a key in the map of filter module parameters. This can be used for retrieving the selector of a message.

List< *Object* > **: getFilterParameter(String)** : returns a list of objects corresponding to a key in the map of filter module parameters. This is used for retrieving the arguments of a message.

void : setFilterParameter(String, Object) : sets an object corresponding to a key in the map of filter module parameters. This can be used for setting the target, server or sender of a message.

void : setFilterParameter(String, String) : sets a string corresponding to a key in the map of filter module parameters. This can be used for setting the selector of a message.

void : setFilterParameter(String, List< *Object* >**)** : sets a list of objects corresponding to a key in the map of filter module parameters. This can be used for setting the arguments of a message.

Map< *String*, *Object* > **: getInternals()** : returns a map with tuples representing the name of an internal and the instantiation of the internal.

Object : getInternal(String) : returns a specific instantiation of an internal, given the name of that internal.

Map< *String*, *Object* > **: getExternals()** : returns a map with tuples representing the name of an external and the instantiation of the external.

Object : getExternal(String) : returns a specific instantiation of an external, given the name of that external.

All entries in the interface are straightforward to understand, except for the last four entries which provide access to the internals and externals. Note that properties of the message are only available or can be set, if the appropriate atomic filters are used, see chapter 6.

Appendix B - *Implementation of the Tracing filter*

Listing 1 presents the implementation of filter type TracingFilter.

```
1  using Composestar.StarLight.ContextInfo;
2  using Composestar.StarLight.Filters.FilterTypes;
3
4  namespace TracingLib
5  {
6    [FilterTypeAttribute("TracingFilter", "TracingInAction", FilterAction.ContinueAction, "
         TracingOutAction", FilterAction.ContinueAction)]
7    public class TracingFilterType : FilterType { }
8
9    [FilterActionAttribute("TracingInAction", FilterActionAttribute.FilterFlowBehavior.
         Continue, FilterActionAttribute.MessageSubstitutionBehavior.Original)]
10   public class TracingInAction : FilterAction
11   {
12     public override void Execute(JoinPointContext context)
13     {
14       // Create the start trace
15     }
16   }
17
18   [FilterActionAttribute("TracingOutAction", FilterActionAttribute.FilterFlowBehavior.
         Continue, FilterActionAttribute.MessageSubstitutionBehavior.Original)]
19   public class TracingOutAction : FilterAction
```

```
20   {
21     public override void Execute(JoinPointContext context)
22     {
23       // Create the end trace
24     }
25   }
26 }
```

Listing 1: Implementation of filter type TracingFilter in C#

Listing 1 defines three classes:

TracingFilterType (lines 6 to 7): is an empty class that is used by the Compose* compiler to determine the available filter types. The compiler searches for classes which are a subclass of class FilterType. To class TracingFilterType an annotation, of type FilterTypeAttribute, is attached that provides the necessary meta information to the compiler. The annotation has several arguments:

"TracingFilter" : the name of the filter, an aspect developer uses this name to instantiate the filter type.

"TracingInAction" : reference to the class name of the filter action that is executed before the join point is executed, if this filter accepts a message. In this case this refers to class TracingInAction that is defined below.

FilterAction.ContinueAction : reference to the class name of the filter action that is executed before the join point is executed, if this filter rejects a message. In this case this refers to an class that simply passes the message to the next filter. If there is no next filter, the join point is executed.

"TracingOutAction" : reference to the class name of the filter action that is executed after the join point is executed, if this filter accepts a message. In this case this refers to class TracingOutAction that is defined below.

FilterAction.ContinueAction : reference to the class name of the filter action that is executed after the join point is executed, if this filter rejects. In this case this refers to an class that simple passes the message to the next filter. If there is not next filter, the message is returned to the caller.

TracingInAction (lines 9 to 16): is a class which has to override method Execute from superclass FilterAction. A call to this method is inserted at the start of every method in the application. This method has only a single argument: JoinPointContext. This object provides access to the properties of the message, e.g. the arguments. Class TracingInAction has an annotation attached that provides meta information to the compiler.

The arguments of this annotation are:
"TracingInAction" : the name of this filter action.
FilterActionAttribute.FilterFlowBehavior.Continue : this states that this action does not alter the control flow through the filter set.
FilterActionAttribute.MessageSubstitutionBehavior.Original : this states that this action does not change the target and selector of the message.

TracingOutAction (lines 18 to 25): similar to the previous action, class TracingOutAction has an annotation attached that provides meta information to the compiler. The arguments of this annotation are equal to the previous filter action, except for the name of the filter action.

Appendix C - *Updated Compose* Grammar*

We extend the grammar of Compose*, to accept the proposed filter composition language extension, see chapter 6. First we add a non-terminal called Composition to a concern definition.

⟨*Concern*⟩ ::= '`concern`' ⟨*ConcernName*⟩
['`(`'⟨*FormalConcernParameter-SEQ*⟩'`)`']
['`in`' ⟨*PackageReference*⟩] '`{`'
(⟨*FilterModule*⟩)* [⟨*Superimposition*⟩]
[⟨*Composition*⟩] [⟨*Implementation*⟩] '`}`'

We introduced non-terminal Composition before non-terminal section Implementation.

Next we state that there must be at least one non-terminal CompositionSpecifier inside section Composition.

⟨*Composition*⟩ ::= '`compositions`' '`{`' (⟨*CompositionSpecifier*⟩)+ '`}`'

This non-terminal starts with the string literal: compositions.

Subsequently, we define non-terminal CompositionSpecifier as follows:

⟨*CompositionSpecifier*⟩ ::= '`compose`' ⟨*Name*⟩ '`of`'
('`filters`' ⟨*FilterList*⟩) | ('`acceptaction`' ⟨*Name*⟩ '`and rejectaction`' ⟨*Name*⟩)

A CompositionSpecifier must start with the string literal compose followed by a Name and the string literal of. Next, we either expect string literal filters followed by non-terminal FilterList. Or, we expect string literal actions followed by a Name, string literal and and another Name.

Finally, we declare non-terminal FilterList to be a comma separated list of Names.

⟨*FilterList*⟩ ::= ⟨*Name*⟩ ('`(`' ⟨*FilterParameterList*⟩ '`)`')?
('`,`' ⟨*Name*⟩ ('`(`' ⟨*FilterParameterList*⟩ '`)`')?)*

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 En-

gineering, 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 Architectures.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 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, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code.* Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems.* Fac-

ulty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates.* Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines.* Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras.* Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit.* Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks.* Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments.* Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15