Aspect Awareness in the Development of Configurable System Software

—

Aspektgewahrheit bei der Entwicklung konfigurierbarer Systemsoftware

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Daniel Lohmann

Erlangen — 2008

Als Dissertation genehmigt von
der Technischen Fakultät der
Universität Erlangen-Nürnberg

Tag der Einreichung:	27.10.2008
Tag der Promotion:	30.03.2009
Dekan:	Prof. Dr.-Ing. Johannes Huber
Berichterstatter:	Prof. Dr.-Ing. Wolfgang Schröder-Preikschat
	Prof. Dr.-Ing. Olaf Spinczyk
	Prof. Dr.-Ing. Jörg Nolte

Abstract

More than 98 percent of the worldwide annual production of microprocessors ends up in embedded systems – typically employed in goods of mass production, like cars, appliances, or toys. Such embedded systems are subject to an enormous hardware-cost pressure. System software for this domain has to cope not only with a broad variety of requirements and platforms, but especially with strict resource constraints. To compete against proprietary systems (and thereby to allow for reuse), a system-software product line for embedded systems has to be highly configurable and tailorable. However, this flexibility has to be provided in a way that meets the strict resource constraints.

The state of the art for the overhead-free implementation of fine-grained configurability in system software is conditional compilation with the C preprocessor. However, this approach leads to scattered and tangled code and does not scale up. At the same time, the demands on configurability of system software are still increasing. AUTOSAR OS, a new industry standard for automotive operating systems, requires configurability of even fundamental architectural system policies, such as protection and isolation strategies.

This thesis evaluates aspect-oriented programming (AOP) as a first-class concept for implementing configurability in resource-constrained systems. It shows that a well-directed, pragmatic application of AOP leads to a much better separation of concerns in the implementation of configurable system software – without compromising on resource thriftiness. Moreover, the suggested approach of *aspect-aware operating-system development* facilitates providing even fundamental architectural policies as configurable features.

The suitability of AOP is evaluated with state-of-the-art operating systems from the embedded-systems domain. The practicability of aspect-aware operating-system development is validated by the design and development of the CiAO operating-system family, which is the first operating system that has been designed and developed with AOP concepts from the very beginning. CiAO combines a competitive implementation of the AUTOSAR-OS standard with a highly configurable architecture.

Acknowledgments

Many people have supported me in writing this thesis and deserve to be mentioned here. It seems to be common practice to mention them in a strictly hierarchical order (not to say *top-down*) starting with the supervising professors. However, as I shall point out in Chapter 3, this thesis follows a *bottom-up* approach.

So let me start with the domestiques of science – the students who wrote study and diploma theses under my supervision, in which they evaluated some of my crazy ideas: GEORG BLASCHKE, CHRISTOPH ELSNER, WANJA HOFER, FABIAN SCHELER, JOCHEN STREICHER, and REINHARD TARTLER. Boys, it was fun working with you – and still is! Given that five of the six of you have continued with science afterwards (a fact I am admittedly a bit proud of) and have even become colleagues, it can't have been too bad :-)

Speaking of colleagues (and continuing bottom-up): Many thanks go to MEIK FELSER, RÜDIGER KAPITZA, and CHRISTIAN WAWERSICH, who shared with me the good times and the bad times of thesis writing. Christian, I am *really* happy that you finished the thing before me. The party ain't over 'till "The Fat Lady Sings"!

A significant part of the CiAO implementation was done during my time at the University of Victoria, BC, Canada. I wish to thank YVONNE COADY and especially CHRIS MATTHEWS, who supported me in all "aspects" of life during that time, and FIONN YAXLEY and CALEY CAMPBELL, who did the same in all aspects of living.

Many thanks go to FRANK BELLOSA, who taught me the hidden politics of science during his post-doc time in Erlangen, and to JÜRGEN KLEINÖDER, who has been doing the same with respect to the hidden secrets of administration.

So we have passed the post-doc level and eventually arrived at the supervising professors: My special thanks go to OLAF SPINCZYK and WOLFGANG SCHRÖDER-PREIKSCHAT. It was Olaf who invited me to join Wosch's group in Erlangen after we met at AOSD 2003 in Boston. Together, we carried out a lot of great research – which would never have been possible without the open, inspiring and encouraging atmosphere Wosch cultivates within his group.

Finally, I wish to thank those people whose support was neither bottom-up nor top-down, but just there when I needed it. BERNHARD GABLER did a great job of proofreading and language polishing. But above all there was my wife Katja, who took care of me and raised my spirits – again and again.

Erlangen, April 2009

Contents

List of Figures

List of Tables

List of Program Listings

1

Introduction

1.1. Motivation

About 98 percent of the annual world-wide production of microprocessors (around eight billion units in 2000) ends up in an embedded system [Ten00]. Embedded systems are special-purpose computers, often implanted into goods of mass production (such as cars, appliances, or toys) where they run a very specific computing or control application. Such embedded systems are subject to an enourmous cost pressure (a few cents can decide over market success or failure), hence their hardware resources (in terms of CPU and memory) are strictly constrained. An 8-bit microcontroller with a few KiB of memory is a common setup.

The shortage on hardware resources has consequences on the design and implementation of operating systems and other system software for this domain. System software for embedded devices has to be easily tailorable for the very specific application running on top of it. One way to achieve this is to design it as a software product line and ship it together with some configuration tool. The application developer can then choose from a fine-grained set of optional or alternative features to generate a tailored variant of the system software for his specific application.

In the implementation of the system software, this flexibility has to be enforced in a way that copes with the strict resource constraints. The state of the art to enforce fine-grained configurability in the implementation of system software is conditional compilation with the C preprocessor. The implementation of configurable features (for example, a synchronization strategy) is embedded as `#ifdef` – `#endif` blocks into the implementation of other features. However, the scattering of the code makes the implementation hard to comprehend and maintain. Even worse: if several configuration options are implemented this way, we quickly find ourself in "`#ifdef` hell" – a phenomenon describing the situation that the code has become unreadable because of all those `#ifdef` blocks.[1] Listing 1.1 shows a real world example from the eCos operating system [eCo]. Even though we have to deal with only four configuration options in this example, the code is already very hard to comprehend. The enforcement of configurability by means of conditional compilation does not scale up.

Nevertheless the demand on functional variability in embedded operating systems is constantly increasing. A good example is the new embedded operating-system standard specified by AUTOSAR, a consortium founded by all major players in the automotive industry in order to specify a system software stack for car applications. The AUTOSAR OS standard [AUT06b] asks for configurability of all policies regarding temporal and spatial isolation. To achieve this within a single kernel implementation is challenging. The decision about such fundamental operating-system policies (like the question if and how address space protection boundaries should be enforced) is typically made in the early phases of operating-system development and deeply reflect in some of its fundamental

[1] The term "#ifdef hell" is common hacker jargon. Its first documented use can be found in a usenet posting by BRIAN HOOK to `comp.os.opengl` from November 5th, 1993.

```
Cyg_Mutex::Cyg_Mutex() {
  CYG_REPORT_FUNCTION();
  locked      = false;
  owner       = NULL;
#if defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT) && \
  defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DYNAMIC)
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_INHERIT
  protocol    = INHERIT;
#endif
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_CEILING
  protocol    = CEILING;
  ceiling     = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
#endif
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_NONE
  protocol    = NONE;
#endif
#else // not (DYNAMIC and DEFAULT defined)
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY
  // if there is a default priority ceiling defined, use that to initialize
  // the ceiling.
  ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
#else
  // Otherwise set it to zero.
  ceiling = 0;
#endif
#endif
#endif // DYNAMIC and DEFAULT defined
  CYG_REPORT_RETURN();
}
```

Listing 1.1: Constructor of the eCos mutex class.
The mutex class `Cyg_Mutex` from the eCos operating system [eCo] is configurable for different prevention strategies against priority inversion, which leads to "`#ifdef` hell" in the implementation.

"architectural" design decisions, which in turn have an impact on many other parts of the kernel implementation. But now they shall be implemented as configurable features.

We need a better way to implement configurability in system software product lines.

Aspect-Oriented Programming (AOP) [KLM+97] is a promising candidate here. AOP provides extra language means for a finer-grained separation of concerns, which could help to escape the "`#ifdef` hell". A specific strength of AOP is the separation of so called "crosscutting concerns" – concerns, which affect the implementation of many other concerns (Figure 1.1). With the help of AOP, it might become feasible to implement even fundamental operating-system policies as configurable features.

1.2. Purpose of This Thesis

In this thesis, I evaluate the suitability of AOP as a *first-class mechanism* for the implementation of configurability in operating-system product lines for resource-constrained

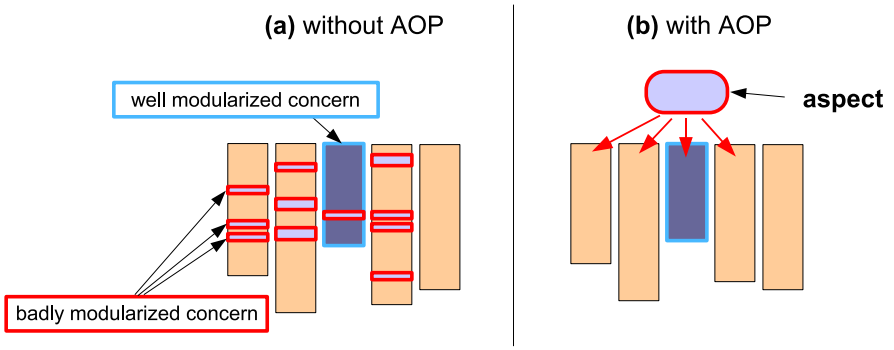


Figure 1.1.: Implementation of a "crosscutting concern" without and with AOP.
The bars represent source artifacts (e.g., classes); a concern is well modularized if it can be implemented by its own, dedicated source artifacts. **(a)** Without AOP, the implementation of "crosscutting concerns" is spread over the implementation artifacts of other concerns. **(b)** With AOP, the implementation of "crosscutting concerns" can be separated into their own source artifacts, the *aspects*.

embedded systems. The overall goal is to show that a well-directed, broad-scale application of AOP does significantly improve on the state of the art to implement configurability in operating systems without disadvantages on the hardware cost side.

A positive effect on the maintainability and evolvability of operating system code by AOP has already been shown on the examples of FreeBSD [CK03] and Linux [ÅLS+03, FGCW05]. A previous thesis [Spi02] and several workshop papers [MSGSP02, SL04] from my own group could furthermore show that AOP improves on the configurability in the PURE embedded operating system [BGP+99b].

Despite all these promising results, the question if and how a *broad-scale application* of AOP in the design and implementation of (embedded) system software yields similar benefits by the bottom line is yet not completely answered:

1. The present studies focus on the qualitative effects of using AOP – the effects on maintainability and configurability. Still missing, especially in the face of the envisioned broad-scale application in the resource-thrifty domain of embedded systems, is an in-depth analysis of the quantitative effects of AOP.

   Questions: Which AOP language features induce an overhead with respect to CPU and memory resources? Can we improve on that? Is the expressive power of a general-purpose, feature-rich AOP language an affordable luxury for the domain of resource-constrained embedded systems?

2. In the present studies, AOP was mostly used as a "last resort" to implement configuration options that were otherwise not feasible. Still open is the question how implementing configurability by aspects compares in general – qualitatively and quantitatively – to the existing approaches, such as conditional compilation or object-oriented programming.

Questions: Can we achieve the same or better efficency and flexibility as with the existing approaches? What are the idioms for implementing configurability by aspects?

3. In the present studies, AOP was applied *ex post* to separate out a few concerns from an existing kernel. Still open is the question what we could get if we design and implement an operating-system kernel with aspects from the very beginning.

   Questions: What are good design rules for aspect-oriented kernel design? What are the benefits? Is it thereby possible to implement even fundamental architectural policies as configurable features?

With this thesis I want to extend on the previous work, provide answers for the questions stated above, and advance the current state of the art by *aspect-awareness in the developement of configurable system software.*

1.3. Title and Objectives

The title of this thesis is "Aspect-Awareness in the Development of Configurable System Software". According to the New Oxford American Dictionary [McK05], *awareness* is the noun of the adjective *aware*, which means *"having knowledge or perception of a situation or fact"*. In Merriam-Webster's Online Dictionary[2] we find: *"*AWARE *[...] mean[s] having knowledge of something [...]* AWARE *implies vigilance in observing or alertness in drawing inferences from what one experiences*."

So *awareness* has two major connotations: (hard, fact-based) *knowledge* and (soft, experience-based) *perception*. This thesis aims for a better knowledge *and* perception of the issues of implementing configurability in resource-constrained system software by AOP. This comprises objectives on three different levels, namely *language*, *implementation*, and *design*:

Language level objectives: Show that by a careful design of the aspect language, highly expressive, yet cost-neutral AOP is possible. Figure out which AOP features induce what overhead and if and how this can be avoided.

Implementation level objectives: Show, based on this knowledge, that AOP compares qualitatively *and* quantitatively very well to the state of the art for implementing configurability in system-software product lines. Figure out enabling and preventing factors for configurability by aspects.

Design level objectives: Show, based on this knowledge, that by understanding aspects as first-class design elements in the development of a kernel it becomes possible to implement even architectural operating-system policies as configurable features. Figure out idioms and rules to achieve *aspect-awareness* when developing an operating-system kernel.

[2] `http://www.merriam-webster.com/dictionary/aware`

1.4. Structure

This thesis is structured as follows:

Chapter 2: *Background, Context, and State of the Art* (pp. 9–45)
There are three topics that constitute the pillars and the context of this thesis: *system software for embedded systems*, *software product lines,* and *aspect-oriented programming*. In the second chapter, I provide an introduction into all three of them and discuss the current state of the art.

Chapter 3: *Problem Analysis and Suggested Approach* (pp. 47–66)
State of the art for the overhead-free enforcement of fine-grained configuration options is the preprocessor. However, this approach does not scale. At the same time the demands on operating-system product lines are still increasing. In the third chapter, I analyze the problems (illustrated by examples from eCos and AUTOSAR OS), present my research assumption – that AOP improves on the situation – and discuss my research approach to evaluate this.

Chapter 4: *Language Level – Aspects Demystified: Evaluation and Evolution of AspectC++* (pp. 67–98)
I begin the evaluation on the *language level* of configurability. The fourth chapter provides a detailed look "under the hood" of AOP in general and AspectC++ in particular. This "demystification" is important to understand the qualitative and quantitative impact of AOP language elements on the implementation of configurability in embedded system software. In the context of this thesis it also paved the path to several improvements of AspectC++, including the new *generic advice* concept and more efficient code generation patterns.

Chapter 5: *Implementation Level – Aspects in Action: Practicing Configurability by AOP* (pp. 99–120)
Focus of the fifth chapter is the *implementation level* of configurability. I apply AOP to practice by refactoring some of the problem-causing concerns found in the eCos operating system from preprocessor-based configuration into a much cleaner implementation based on aspects. The goal is to understand the preconditions and circumstances under which the assumed benefits of AOP hold – respectively, can be realized at all – when we use aspects on a larger scale to implement configurability in system software for embedded devices.

Chapter 6: *Design Level – CiAO Aspects: Aspect-Aware Operating-System Development* (pp. 121–169)
The sixth chapter handles the *design level* of configurability. I present the idea of *aspect-aware* kernel development by the example of the CiAO family of operating systems. CiAO – the acronym stands for CiAO is Aspect-Oriented – is a highly configurable operating-system product line I have designed and developed from scratch with aspects as a first-class development concept. Thereby, CiAO combines configurability of even fundamental architectural properties with excellent granularity

and variability.

Chapter 7: *Summary, Conclusions, and Further Ideas* (pp. 171–175)
In the seventh chapter I summarize and review my work and provide an outlook of further ideas.

The seven main chapters are followed by two appendices that contain collateral material closely related to the topics of this thesis.

Appendix A: *AspectC++* (pp. 177–198)
In the first appendix I provide additional information about the AspectC++ language and tools. This includes a general overview, some advanced application examples, and a further analysis of the optimized code generation patterns developed in the context of this thesis.

Appendix B: *Case Study "WeatherMon"* (pp. 199–218)
In the second appendix I describe the results of an additional case study. The goal of the "WeatherMon" study was to evaluate AOP in comparison to OOP for the development of a software product line for very small, resource-thrifty "deeply embedded" systems. For this purpose, I designed, implemented, compared, and evaluated AOP and OOP versions of an embedded weather-station product line based on a small 8-bit microcontroller.

1.5. Typographical Conventions

Boldface indicates the introduction of a new term or, in some few occasions, the accentuation of an important result. In some cases, the introduction of a term is postponed to a following paragraph; in these cases the term yet to be introduced is typeset in *italics*, which is also generally used as a stylistic means for *text emphasis*. Source-code identifiers are typeset in `monospace`, identifiers that refer to concepts or conceptual features are depicted in Sans serif and start with a capital. Tools or commands, such as the used compiler, are depicted by SMALL CAPS.

2

Background, Context, and State of the Art

In short, the goal of this work is to evaluate AOP as a means to increase the achievable configurability in software product lines for embedded system software. Hence, there are three topics that constitute the pillars and the context of this thesis: *system software for embedded systems*, *software product lines,* and *aspect-oriented programming*. This chapter provides a brief introduction into all three of them. It is brief, as I do not claim to be comprehensive here – each of these topics comprises an area of research of its own. It is an introduction (in contrast to a mere overview), as many of the definitions and points under discussion reflect my own point of view – results from several years of work in the field. The goal is to elaborate the principles and terminology that are important for understanding this thesis.

The chapter is structured as follows: Section 2.1 introduces the target domain – *system software for embedded systems*. It describes the typical properties and constraints of embedded devices and their impact on the application of system software, especially operating systems. Section 2.2 contains an overview of the engineering of *software product lines*. In Section 2.3, I then introduce the basic principles and ideas of *aspect-oriented programming*. Finally, the chapter is briefly summarized in Section 2.4.

2.1. System Software for Embedded Systems

We start with a working definition for embedded systems:

> An **embedded system** is a special-purpose system in which the computer is completely dedicated to (and often also physically encapsulated within) the device it controls.[1]

Unlike a general-purpose PC, which is meant to be usable for different tasks by means of user-installable software, an embedded system is aimed at only one predefined tasks (or a relatively small set of variations thereof). This leverages optimization and tailoring of both hard- and software.

Embedded systems are ubiquitous – even though most people are not aware of their omnipresence [Ten00]. We can find dozens of them in a modern car [Bro06]. They control small devices such as MP3 players, cameras and wrist watches, appliances like washing machines and fridges, but also airplanes and nuclear power plants. More than 98 percent of the world-wide annual processor production (in units) ends up in an embedded system [Tur02, Ten00]. What most of us perceive as "computers" – PCs, laptops, servers, and so on – makes up just a mere two percent of the computers in use.

2.1.1. Properties of Embedded Systems

The most notable commonality among embedded systems is, strangely enough, diversity. The functional properties of devices controlled by an embedded system differ a lot – so does the hardware used to implement the functionality. We refer to this as **application diversity** and **platform diversity**.

- Embedded systems are applied to a large number of different application areas, each with its own very specific requirements and constraints.
- There are dozens of different hardware platforms available to run embedded devices. Many hardware platforms can furthermore be adapted and tailored with respect to the specific application.

Nevertheless, many embedded systems share a set of **nonfunctional properties** and requirements that are considered typical constraints for this domain. Important examples are **dependability**, **energy efficiency**, and **cost efficiency**:[2]

- Many embedded systems run safety-critical processes and thus have to be dependable. Airplanes and nuclear power plants are just two examples of systems that may cause disastrous consequences in the event of failure.

[1] compare, e.g., [Coo03, p. 12][Mar06, p. 1].
[2] compare, e.g, [Coo03, Mar06]

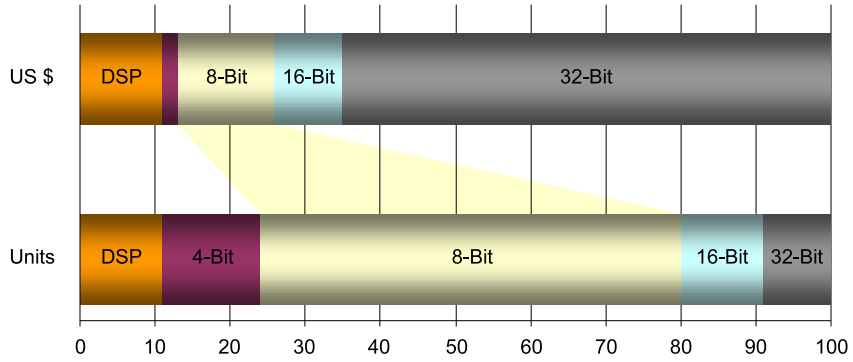


Figure 2.1.: Market share of 4-bit to 32-bit CPUs and DSPs in 2002.
Depicted is the percentage turnover of the different CPU types in units and money. Even though 8-Bit microcontroller make more than 50% of the sold units, they account for less than 15% of the fiscal turnover .
(Graph adopted from [Tur02].)

- Many embedded systems are mobile systems that are powered by batteries. Compared to other pieces of hardware, batteries are relatively expensive, heavy, and bulky. Energy efficiency has a significant impact on the durability, weight, size, and production cost of an embedded device.
- Embedded systems are often used in mass-produced goods – manufactured and sold in millions. This leads to an immense pressure on the per-unit hardware cost; a single cent can decide over market success or failure.[3]

Most relevant in the context of this thesis are *application diversity*, *cost efficiency*, and *platform diversity*. I understand *platform diversity* as a consequence of the other two: Mass-production imposes a demand for as-cheap-as-possible hardware, whereas the different but predefined application-specific requirements facilitate the reduction of per-chip cost by leaving out unnecessary functionality. Hardware vendors cope with the varying demands of scale by offering different CPU types, from small 4-bit microcontrollers up to multi-core 32-bit and 64-bit number crunchers.

2.1.2. Hardware for Embedded Systems

The effects of the hardware cost pressure can be observed in the utilization of the available CPU types. 8-bit and even 4-bit technology dominates CPU production in terms of units (~70%), but yields only a very small portion (~15%) of chip industry revenues (Figure 2.1). Device manufacturers seem to use it especially in areas of high-volume mass production, as, according to a survey published by *embedded.com* [Tur05], 32-bit technology is employed in nearly 60 percent of all embedded system developments.

[3] An expert from the automotive industry told me once that their procurement agents calculate and negotiate per-unit hardware cost by the quarter of a cent.

Type	Flash	SRAM	IO	Timer 8/16	UART	I^2C	AD	Price (€)
ATTINY11	1 KiB		6	1/-	-	-	-	0.31
ATTINY13	1 KiB	64 B	6	1/-	-	-	4*10	0.66
ATTINY2313	2 KiB	128 B	18	1/1	1	1	-	1.06
ATMEGA4820	4 KiB	512 B	23	2/1	2	1	6*10	1.26
ATMEGA8515	8 KiB	512 B	35	1/1	1	-	-	2.04
ATMEGA8535	8 KiB	512 B	32	2/1	1	1	-	2.67
ATMEGA169	16 KiB	1024 B	54	2/1	1	1	8*10	4.03
ATMEGA64	64 KiB	4096 B	53	2/2	2	1	8*10	5.60
ATMEGA128	128 KiB	4096 B	53	2/2	2	1	8*10	7.91

Table 2.1.: Features and prices of the 8-bit AVR ATmega microcontroller family.
Depicted is a selection of the > 70 available variants. Variants differ with respect to memory sizes (*flash*, *SRAM*), number of digital IO pins (*IO*), number and width of hardware timers (*Timer 8/16*), serial interfaces (*UART*, *I²C*), and number and width of analog to digital converters (*AD*).
(Wholesale prices taken from: Digi-Key product catalog, Summer 2006)

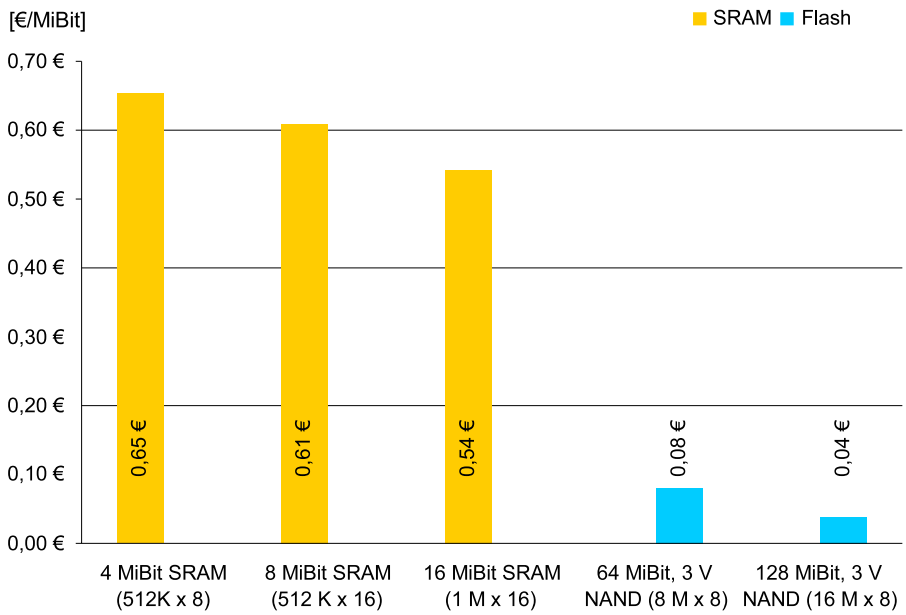


Figure 2.2.: Comparison of per-MiBit cost of NAND flash memory and SRAM.
SRAM is roughly 10 times more expensive than NAND flash memory. Depicted are the prices for Toshiba chips with different capacities, but comparable packaging (TC55NEM208ATGN55L-ND, TC55W800FT55M-ND, TC55VBM416AFTN55-ND, TC58V64BFT-ND, TC58DVM72A1FT-ND).
(Wholesale prices taken from: Digi-Key product catalog, Summer 2006)

Embedded or Not Embedded?

Interestingly, the example for an embedded system mentioned most often is – at least in my experience and among computer scientists – the cell phone. Is the cell phone a good example for a typical embedded system?

Ten years ago, I would not have hesitated to say so. Cell phones were special-purpose devices, aimed to provide a specific predefined task: mobile telephony. Ever since, however, manufacturers have been extending their little gadgets with more and more functionality: photography, gaming, schedule management, ... On top of that, the average cell phone today even provides a Java Virtual Machine (JVM) to run arbitrary user-installable software. In fact, cell phones have turned into general-purpose computers. Telephony is still there, but has merely become a negligibility. Curiously enough, it is still implemented by dedicated hard- and software components – embedded into, but well separated from the surrounding general purpose computer. Network providers insist on this separation because of security and safety concerns.

The issue, however, is not so much that cell phones fail my definition of embedded systems. It is their perception as *typical* embedded systems. Many computer scientists are convinced to do research "for typical embedded systems" when their work targets at devices with "just 64 MiB of RAM" and a "not so powerful JVM" – basically small PCs.

As DAVID TENNENHOUSE already stated: *"Over the past 40 years, computer science has addressed only about 2% of the world's computing requirements"* [Ten00].

Many CPU platforms are furthermore designed as **hardware families** comprising dozens of (binary and pin) compatible microcontroller variants. Table 2.1 shows an excerpt from the AVR ATmega microcontroller family, which offers more than 70 different variants, all based on the same 8-bit RISC core. The amount of peripheral features has quite an impact on the chip price. Especially the amount of SRAM is a significant cost driver.[4] RAM is required for the run-time state of the system (e.g. stack space, global variables), but, as Figure 2.2 shows, SRAM is about 10 times more expensive than flash memory (ROM), which is used for program code and read-only data. Embedded system developers consider RAM requirements as *the* critical source of hardware costs. Hence, software for this domain has to be optimized with respect to low RAM utilization.

2.1.3. The Role of System Software

System software provides application developers with a higher-level interface than that offered by the bare hardware. We understand the interface implemented by some system software as a new *virtual machine layer*, that is, an abstract hardware (abstract processor, abstract devices, ...) that specifies its own data abstractions (types) and instruction set (system calls). This instruction set is internally implemented by programs (compiled or interpreted) that use the instruction set offered by the next lower level. For instance, a

[4] For the sake of access time predictability and low energy consumption, SRAM is used in most embedded systems, which is more expensive than the DRAM used in PC-like computers.

middleware layer is built on top of an *operating system layer*, which in turn is built on an *instruction set architecture layer* (such as Intel's IA32), and so on.[5]

System software provides no business value of its own. Its sole purpose is to ease the development and integration of applications, that is, to serve application developers and integrators with a virtual machine layer that provides the "right" instruction set (abstractions) for their particular problems. Of course, the whole idea of easing the development pays off only if the system software is reusable for more than one application. System software has to be *general-purpose* to create synergies.

System software for embedded systems also has to cope with the specific properties of this domain. Diversity of embedded applications and platforms imposes a large spectrum of functional and nonfunctional requirements on system software. Hardware cost pressure calls for meeting the specific requirements of each application exactly, that is, without any overhead for unneeded functionality. Compared to the domains of "big" computing (PCs, servers, mainframes), we have a broader set of *potential* requirements but smaller sets of *actual* requirements. [6] As a consequence, developers of embedded systems can almost never reuse existing system software "as is", but have to **customize** or **tailor** it:

> **Customizing** or **tailoring** is the activity of modifying existing system software in order to fulfill the requirements of some particular application.

In the domain of embedded systems both terms are used mostly synonymously, so they are in this thesis.

The practice of tailoring system software for a specific application has a history in the area of operating systems. This is not surprising – cost and availability of hardware were major problems of applied computing in the 1970s. In his article *Designing Software for Ease of Extension and Contraction*, DAVID L. PARNAS, one of the pioneers in the design of reusable operating systems, already wrote:

> *Some applications may require only a subset of services or features that other applications need. These 'less demanding' applications should not be forced to pay for the resources consumed by unneeded features.* [Par79]

The ease and extent to which some system software can be customized and tailored for a specific purpose depends mostly on the **variability** and **granularity** offered by its implementation:

> **Variability** of system software is the property that denotes the *range* of functional requirements that can be fulfilled by it.
>
> **Granularity** of system software is the property that denotes the *resolution* of which requirements can be fulfilled by it, in the sense that requirements are fulfilled but not overfulfilled.

[5] compare [Tan06, pp. 2–13], but also [HFC76]
[6] compare [Ten00].

Variability describes scalability with respect to features or functionalities. A piece of system software offers the lowest variability if it fulfills the requirements of only one specific application. A piece of system software offers the highest variability if it fulfills the requirements for any potential application.

Granularity describes scalability with respect to hardware resources. A piece of system software offers the coarsest granularity, if the induced overhead does not depend on the actual application requirements, that is, if the overhead is same for any two applications. A piece of system software offers the finest granularity if for any application the induced overhead is not higher than the overhead induced by a best possible hand-crafted solution for this particular application.

The extreme cases do not exist in real system software. Nevertheless both, variability and granularity, are important properties. Variability reflects the theoretical reusability of system software for different applications – which I described above as the major motivation to develop discrete system software. Granularity reflects the practical usability of system software in a domain with high hardware cost pressure.

General purpose operating systems, for example, offer quite high variability but only coarse granularity. Windows or Linux fulfill all requirements to run a simple control loop of some device. They also fulfill the requirements to concurrently execute a bunch of multi-threaded server applications that are protected from each other by address space boundaries, CPU time warranties and kernel capabilities. The system software overhead, however, is almost the same in both cases; the requirements of the simple control loop application are dramatically overfulfilled. This is the price of using a general-purpose operating system that is meant to be a one-size-fits-all solution for all types of applications.

In the following we take a closer look on the role of operating systems in the embedded systems domain today.

2.1.4. The Role of Operating Systems

The operating system is the lowest layer (and, thus, the fundamental building block) in the system software stack for an embedded system. As a consequence, the demand for application-specific operating systems is particularly high. This can be observed in the diversity of the operating-system market: While the number of general-purpose operating systems for PCs and server-like computers has undergone a strong consolidation over the last two decades (eventually resulting in Windows, Linux, MacOS and a few Unices), embedded application developers can select from a zoo of more than two hundred available operating systems, most of which are real-time operating systems. [7]

[7] This is an estimate; it is difficult to find reliably citable numbers for this. Published surveys – such as [FSH+01] – are incomplete, so are the numerous lists that can be found online. However, at the time of this writing (8-April-2007), *Dedicated Systems Encyclopedia*, lists 66 real-time operating systems (`http://www.dedicated-systems.com/encyc/buyersguide/products/Dir1048.html`) – but their list includes only commercial systems. *Wikipedia* lists 51 commercial (denoted as proprietary) and 23 open-source

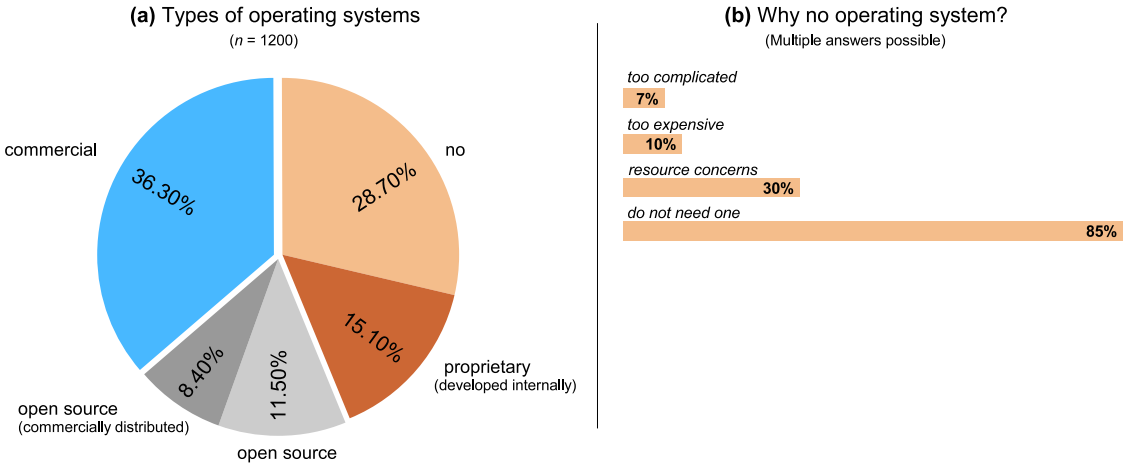


Figure 2.3.: Types of operating systems used in embedded system development – and reasons not to use one. Depicted are the results from a survey performed by *embedded.com* in 2006 [Tur06]: **(a)** Nearly 44 percent of all embedded systems either use no operating system at all (*No OS*, 28.7%) or depend on in-house developed operating-system functionality (*Proprietary OS*, 15.1%). **(b)** An (assumed?) lack of need (85%) and general concerns about the RAM/CPU resource overhead (30%) are the most frequently mentioned reasons not to use an operating system.
(Graphs based on data taken from [Tur06].)

In spite of the broad supply, the utilization of existing operating systems is not yet state of the art and rises only slowly. In his keynote at the *embedded world* exhibition (Nuremberg, 2004), WALLS reported that the operating-system functionality is proprietary or part of the application code in more than 50 percent of all embedded system developments [Wal04]. In the more recent numbers published by TURLEY [Tur06] the proprietary operating system / no operating system fraction has dropped below 44 percent (Figure 2.3.a); a fact from which TURLEY concludes that the use of operating systems is "on the rise".

Nevertheless, the number of embedded projects that do not employ existing operating system functionality is still remarkably high. Why is that the case? According to TURLEY's report, the most mentioned reason was "a simple lack of need", followed by concerns regarding a thereby induced RAM/CPU overhead (Figure 2.3.b). While the second answer emphasizes – once again – the hardware cost pressure, the vastly expressed "lack of need" seems surprising. We can assume that it is, at least partly, expression of the engineers' perception of operating systems. WALLS reported that engineers of embedded systems tend to develop and maintain typical low-level operating-system abstractions (such as hardware resource management or control flow coordination) as part of the *application* instead of taking them from an existing operating system; a habit that presumably causes

real-time operating systems (`http://en.wikipedia.org/wiki/RTOS`). And still, there are many systems missing on both lists: open-source and research systems – such as *TinyOS* [Ber, GLC05], *PURE* [BGP+99b], *EPOS* [FSP99] – but also the various commercial implementations of domain-specific standards, such as OSEK [OSE05], AUTOSAR [AUT06b], or μItron [Sak98]. We should perhaps understand this as another indicator for the fact that the domain of embedded systems is broad and inhomogeneous.

tremendous costs [Wal04]. One can argue whether some simple memory management or the means to coordinate a few interrupt handlers already constitute a reusable operating system. According to HABERMANN, who was another pioneer in the design of customizable operating systems in the 1970s [HFC76], it clearly does. The more recent exokernel idea [EKO95], moreover, explicitly reduces the operating-system's responsibilities to the management of hardware resources and leaves the implementation of strategies to the concrete application.

However, these are academic arguments. In practice, the question of what is and what is not an operating system is probably answered on the RTOS market. If embedded system developers are willing to spend a lot of money on in-house and per-application development of operating-system functionality, we can only conclude that existing operating systems are not tailorable well enough or easy enough.

In the following, we take a closer look at the state of the art in implementing customizable operating systems.

2.1.5. Implementation Techniques for Customizable Operating Systems

The key towards tailorability (and the more general customizability) are design concepts and language means that provide a good support for granularity and variability in the resulting operating-system implementation, that is, a good **separation of concerns**. In their book "Operating System Concepts" SILBERSCHATZ and associates write:

> *Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details (mechanisms). This separation allows maximum flexibility if policy decisions are to be changed later.* [SGG05, p. 72]

Such complete separation between policies and mechanisms is, however, very difficult to achieve – especially for system policies that have to be reflected in the implementation of many mechanisms.

A lot of approaches for the design and implementation of customizable operating systems have been suggested. The surveys published by FRIEDRICH and colleagues, DENYS and associates, and TOURNIER [FSH+01, DPM02, Tou05] provide a good overview on the topic. Most recent work concentrates on the implementation of dynamic adaptation at run time; relevant in the context of this thesis are, however, approaches for *static* customization and tailoring:

- *Libraries* – the operating system is provided as a library of functions that is tailored by the linker at link time.
- *Functional hierarchies and layers* – the operating system is provided as a stack of function layers; each layer presents an optional stage of expansion to the previous layer.

- *Frameworks* – the operating system is provided as an (object-oriented) framework; its mechanisms and policies are to be customized by subtyping.
- *Components* – the operating system is provided as a set of components that can be aggregated according to the rules of a component model.
- *Preprocessor-based customization* – the operating system is provided as a set of source files that are customized by a preprocessor before compilation.

In the following sections, I discuss these approaches in further detail.

2.1.5.1. Libraries

The operating system is provided as a (linker) **library**. The library bundles a set of symbols (functions and global variables) that represent the services provided by the operating system. When the application is linked against the library, the linker includes only those services in the resulting image that the application uses, either directly or indirectly (via other user symbols); the code and global state that implements unused services are automatically omitted and do not consume memory space in the resulting image. More precisely: Included is the reflexive and transitive closure of referenced symbols calculated from the application's entry point (`main()`).

Libraries provide automatic granularity on the level of *functions*. Variability is possible in the sense that the application developer might choose between different functions that represent variants of the same functionality. The C standard library, for instance, provides with `memmove()` and `memcpy()` two variants of the functionality to copy a block of bytes; the latter one is optimized for cases where the source and destination blocks are known to not overlap. The application developer, however, has no influence on internally used mechanisms and polices of the library; so overall variability is relatively low.

The library approach works well if the library implementation does not exhibit internal coupling, that is, if the functions of the library are self-contained and do not refer to each other. A library should implement only mechanisms, but no policies – which often lead to internal coupling.[8] The very first operating systems were shipped as libraries. These systems typically implemented just hardware abstraction and maybe a compiler. A more recent example of a library operating system is the *exokernel* [EKO95], which intentionally provides only mechanisms to let the application implement the policies. However, in general it is challenging to design and implement a library in a way that does not suffer from internal coupling.

[8] The GNU C library (on Ubuntu Linux 2.6.26-x86_64) is a good (well, bad) example for internal coupling: Even if linked against an *empty* program (`echo 'int main(){}' | gcc - -xc -O6 -static`), more than 470 KiB (!) of library code is included in the resulting image. The reason is the startup code of the C library, which alreads installs the internal cleanup and error handling polices, which contain references to many library functions (`atexit()`, `malloc()`, `printf()`, ...), which in turn contain references to more library functions, which ultimately causes the linker to always include a large part of the library into the final image.

The library approach is still very common in the domain of embedded operating systems, but usually combined with one of the other approaches (especially *preprocessor-based customization* which will be discussed in Section 2.1.5.5) for the sake of better granularity and variability *within* the library. However, internal coupling can also be prevented by design, which is the objective of the following approach.

2.1.5.2. Functional Hierarchies and Layers

The services provided by the operating system are designed as hierarchies of operating-system functionalites bottom up from the hardware. The resulting graph, the **functional hierarchy** [Par76b, HFC76], describes the **functional dependencies** between the different operating-system functionalites – dependencies in the sense of "needs" or "has to be present for". The graph has to be acyclic; if some functionalities (indirectly) depend on each other, this is considered as an indicator for the need of further decomposition [HFC76]. The elements of a functional hierarchy are arranged in a stack of **layers**; each layer represents bottom-up from the hardware a **minimal extension** of the previous layer, that is, a new *virtual machine layer*.

Functional hierarchies and layered systems are a *design approach* to minimize the internal coupling between operating-system functionalities. The approach itself makes no assumption on how functionalites actually get implemented. The goal of an acyclic graph of fine-grained functional dependencies is, however, quite challenging. The specification of all operating-system functionalites has to be already complete; furthermore efficient implementation means to postpone design decisions from lower layers to higher layers are reqiuired. LISTER and EAGER diagnose that "Only a few real life operating systems have actually exhibited this rather neat structure" [LE93, p. 7].

Early examples for layered systems are *T.H.E.* [Dij68, Par76b] and *FAMOS* [HFC76]. Probably the most consequent application of these design principles has been presented in the *PURE* family of operating systems for deeply-embedded devices [SSPSS98, BGP+99a, BGP+99b]. In the most recent PURE implementation, the layers of the functional hierarchy are mapped to C++ classes. Each layer extends the previous layer by means of C++ implementation inheritance from the corresponding class; design decisions that are already needed in lower layers but should not be bound there are postponed to higher layers by means of C++ virtual functions. The Thread concept, for instance, is defined by a 14-level class hierarchy in which each class implements a minimal extension to the previous layer. The lowest layer already depends on a design decision, namely, which code the new thread should execute. This decision is postponed to higher layers by representing it as a virtual function. By its stringent adherence to the *principle of minimal extension* in the design, PURE exhibits excellent granularity.

Functional hierarchies and layers are closely related to the idea of *program families* [Par76a]. Each path through the dependency graph can be understood as one *family member* of a *family of operating systems*.[9] Program families are further elaborated in Section 2.2.2.

[9] *FAMOS* [HFC76] is actually an acronym for *FAMily of Operating Systems*.

2.1.5.3. Object-Oriented Frameworks

The operating system is provided as a **framework**. Compared to libraries, frameworks provide additional means for customization, as they do not only define services that may or may not be invoked by the application, but also a **reference architecture** for interaction between the (operating system) framework and the application [Joh97, Deu89]. Frameworks are often based on object-oriented concepts. In object-oriented frameworks, the application developer can extend the representation of kernel-internal entities by subclassing; kernel-internal behavior can be customized by method overriding.

Frameworks provide granularity on the level of *classes*. This is less fine-grained than with libraries, as a class typically contains several methods and subclassing is an extension mechanism only. An important factor is the number of virtual functions. These functions are bound at run time, so it is usually not possible to detect and remove unused (dead) code at compile time or link time. Hence, for the sake of granularity, a framework should offer as few virtual functions as possible.

Variability is determined by the number of *extensible classes* and overridable *methods*. Even internal state and behavior can be modified by subclassing and method overriding, often in combination with design patterns such as *factory method* [GHJV95]. An important factor is the number of virtual functions, as only in these points operating-system–internal policies can be customized. Hence, for the sake of variability, the framework should offer as many virtual functions as possible.

Object-oriented frameworks have a principle trade-off between *granularity* and *variability,* caused by the fundamental binding and customization mechanism, which is subclassing and overriding of virtual functions. As already mentioned, virtual functions cause a general, nonnegligible overhead [DH96]. More overhead is induced by the fact that all customizable policies and mechanisms have to be instantiated at run time. For these reasons, the object-oriented framework approach has not become very popular in the domain of ressource-constrained embedded systems.[10] The best known example for an object-oriented operating system framework is *Choices* [CIMR93]; the *K42* system [SKW+06] uses the object-oriented framework idea internally, but does not exhibit it via its application interface.

2.1.5.4. Component Frameworks

The operating system is provided as a **set of components** that adhere to some **component model**. Compared to objects and classes, components add another level of syntactical abstraction as they make dependencies (in the sense of *provides* interfaces and *requires* interfaces) more explicit and provide, via the component model, additional means to analyze and customize the binding of and interaction between the units of abstraction.

[10] For small embedded systems, the cost of virtual functions and run-time instantiation of concepts can be dramatical. The “WeatherMon” case study (Appendix B on page 199) contains a detailed analysis of these costs.

The unit of granularity is the *component*. Most component models support the hierachical composition of components (*nesting*), so components can theoretically vary in size from fine-grained (like a queue abstraction) to coarse-grained (like a complete file system). Granularity can furthermore be improved by sophisticated means for intra-component dead code analysis; this, however, depends on the component model.

The *component* is also the unit of variability. Components can be substituted by other components adhering to the same interface. It is usually not possible to modify internal parts of a component. This means that policies have to be externalized into their own components to be customizable.

The component framework approach is often used in operating systems that target a particular application domain. The *Flux OSKit*, for instance, is intended for the quick prototypical construction of x86-based research operating systems [FBB+97, oU]. OSKit focuses on the easy reuse of existing operating-system code, especially device drivers, from Linux and FreeBSD by encapsulating them into components adhering to a COM-based component model. Thus, OSKit components are relatively coarse-grained. Part of the run-time overhead induced by late binding and the COM interface model is later removed by an extra optimization tool [RFS+00].

A COM-like component model is also used in the *THINK* operating system framework [FSLM02]. THINK is intended for the domain of (distributed) telecommunication routers, so in THINK even the *binding* between components is represented by separate components, which facilitates the customization of many policies. It remains open, however, how much overhead is caused by the indirection over extra binding components.

A somewhat opposite strategy is exhibited by *TinyOS*, an operating system for deeply-embedded sensor nodes [GLC05, Ber]. TinyOS exhibits its own component model with its own implementation language called NesC [GLv+03]. For the sake of simplicity of application development, certain policies are hard-coded and part of the component model and the NesC language. Synchronization, for instance, is always done via a global interrupt lock; threads are always nonpreemptable. This knowledge makes it possible to perform very sophisticated dead-code analyses and other optimizations in the NesC compiler. Thereby, TinyOS reaches excellent granularity even though the TinyOS components itself are relatively coarse-grained (if compared to, e.g., PURE classes).

2.1.5.5. Preprocessor-Based Customization

The operating system is provided as a set of annotated source files that are transformed on source-code level by some **preprocessor** before compilation. The source-code annotations are given in their own language, the preprocessor language, and describe textual transformations to be performed by the preprocessor on the annotated source code.

The most commonly used preprocessor is the C preprocessor (`CPP`), a macro processor that is automatically invoked by C, C++, and Objective-C compilers. Depending on macro definitions, source lines can be included or excluded from compilation by `#if`, `#else`,

`#endif` block annotations – a mechanism that is also known as *conditional compilation*. Furthermore, macro references in the source code are textually expanded to their definition by the preprocessor, hence, part of the source code can be generated.

The preprocessor approach is fundamentaly different from the other approaches in two respects:

1. The language to *implement variability and granularity* has no direct relation to the (type system of the) *general implementation language*. Granularity and variability does not depend on the kind and size of *syntactical* entities offered by the implementation language (functions, classes, or components), but can theoretically be implemented across several syntactical entities on the level of source-code lines.
2. The preprocessor approach is mostly *decompositional*, whereas the other approaches are mostly *compositional*. In the compositional approaches we start with a minimal system that represents the *intersection* of all possible variants and add or substitute units *(*functions, classes*,* components) to this system until all requirements are met. With the decompositional preprocessor approach, the source code contains the *union* of all possible variants and we customize the system by filtering out the unnecessary parts.

The decoupling from the type system of the implementation language makes it possible to implement very *fine-grained granularity and variability*. Moreover, the approach is *inherently overhead-free*, as the customization is not based on abstraction mechanisms of the programming language, which can induce some run-time overhead.

On the other hand, the decompositional nature of the preprocessing approach leads to substential disadvantages regarding the *understandability* and *maintainability* of the code. The source code gets bloated, as it has to contain all implementation variants, which are scattered over the source base. The better the provided granularity and variability, the worse are the bloating and scattering effects. Hence, the approach does not scale – it quickly leads to the "`#ifdef`-hell" phenomenon already presented in Listing 1.1. We will see more examples of this in Chapter 3.

Several papers emphasize the problems of preprocessor-based customization of software [SC92, Fav97]. Nevertheless, the C preprocessor has to be considered as state of the art for the implementation of overhead-free, fine-grained customizability [EBN02]. Application examples from the operating systems domain include all variants and flavors of Linux and BSD as well as the majority of embedded operating systems, such as *FreeRTOS* [FRT], *ProOSEK* [Pro], *Contiki* [DGV04], or *eCos* [eCo].

2.1.6. From Customizing to Configuring of System Software

The decompositional nature of preprocessor-based customization has another interesting consequence: All possible means of customization have to be provided *explicitly* by the system-software developer. The set of macros that control the process of conditional

compilation constitutes an interface to customization that predefines the (so far purely declarative) properties of variability and granularity (Section 2.1.3). We understand this as *configurability* of system software:

> **Configurability** is the property that denotes the degree of pre-defined variability and granularity offered by a piece of system software via an explicit configuration interface.

This is fundamentally different from the compositional approaches, where customization is based on the aggregation, substitution, and extension of *implementation entities* – which is theoretically open-ended and, thus, more flexible. However, the compositional approaches require a profound knowledge about the system software and its implementation model. Hence, from the viewpoint of an application developer, the explicit representation of variability and granularity may be preferable.

Preprocessor macros are just one example for a simple configuration interface. The idea of representing customizable system software by more abstract and problem-related *features* gives rise to many questions regarding the identification, representation, and management of configurability on a larger scale. These issues are addressed by the field of *software product lines*.

2.2. Software Product Lines

The general idea of a **product line** is to exploit the commonalities between the different yet similar sets of requirements customers express for products from a particular domain of interest. Instead of developing an individual solution for each specific set of requirements, a product line aims to provide (pre-manufactured) solutions for a whole problem domain. The problem domain is spanned by the set of **features** that describe the **commonalities** and **differences** between individual problems. WITHEY defines:

> *A product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market.* [Wit96]

In **software product lines** the products are software programs. Software product lines offer the opportunity to create significant synergetic effects for the development of similar software products as they drive systematic reuse [NC01]: Problem specifications, development processes, design decisions, and component implementations can often be shared for all features that are common among multiple products; thus, they have to be undertaken only once. Ideally, only the **variation points** – features that distinguish one software product from all others – induce extra development efforts. To achieve these benefits in practice should be the ultimate goal of principles, methods, and tools for software product-line engineering.

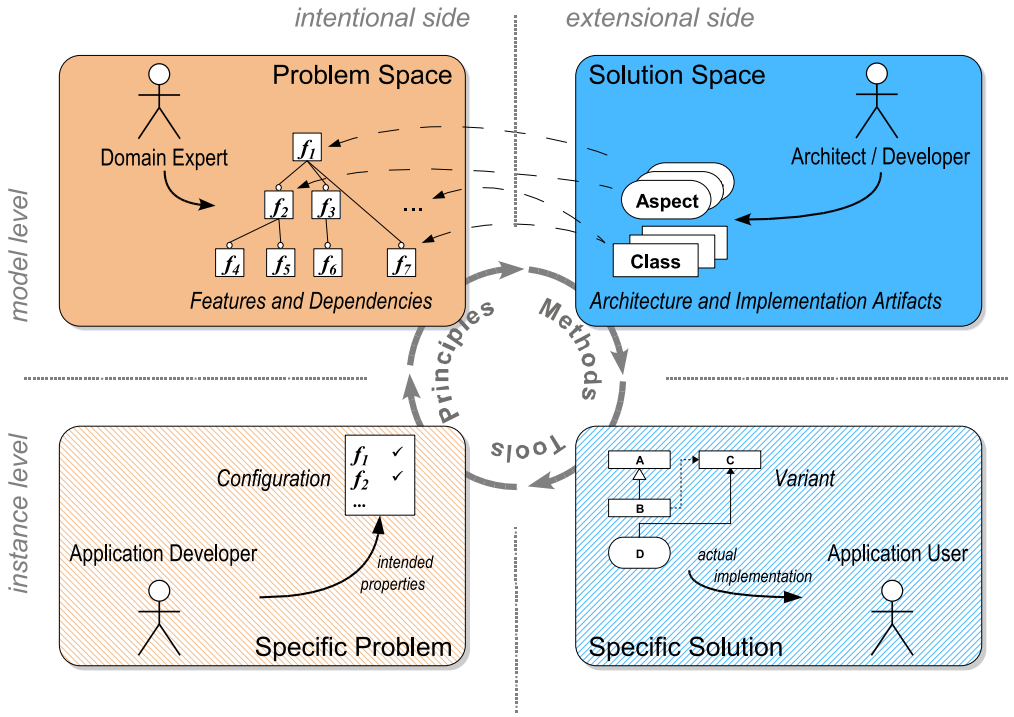


Figure 2.4.: Problem–solution model of software product lines.
A software product line consists of a set of *intentional* problem specifications (the **problem space**), a set of *extensional* solution descriptions, and a *relation* between both sets (the **solution space**). Thereby each instance of the problem space (a **specific problem**) can be mapped to an instance of the solution space (the **specific solution**). On the *model level*, the *domain expert* specifies the variability of the problem space by a formal model of (abstract) *features and dependencies*. The *architect/developer* implements this variability (*architecture and implementation artifacts*) in a solution space and also provides a formal mapping from *implementation elements* to *features*. On the *instance level*, the *application developer* specifies the *intended properties* by a selection of features (a **configuration**). This description is evaluated to derive the *actual implementation* (the **variant**) for the *application user*.

2.2.1. Concepts and Terminology of Software Product Lines

As suggested by SIMOS [Sim95] and CZARNECKI and EISENECKER [CE00], among others, I understand a software product line as a mapping from problems to solutions. A **problem** denotes a set of requirements; a **solution** denotes an implementation that fulfills these requirements. The **problem space** specifies the variability that can be found within a domain of interest (the "selected market"), thus the set of all potential problems. The **solution space** maps these potential problems to pre-manufactured components, generators, or other means to implement them, thus to the actual solutions. Thereby, for each **specific problem** covered by the problem space, a **specific solution** can be found in the solution space.

The variability found within the problem space of a software product line is typically operationalized as a set of selectable and configurable **features**. We call a selection

of features that describes a specific problem a **configuration**. We call the resulting implementation of a specific solution a **variant**.

The problem space is *intentional*; its elements are the domain-specific features (properties, requirements) expressed by human stakeholders.[11] The solution space is *extensional*, respectively; its elements are executable computer programs, instruction sequences for an imperative machine. Figure 2.4 illustrates the problem–solution model of software product lines.

2.2.2. History of Software Product-Line Engineering

It is remarkable that software product-line engineering has its roots in the work on operating systems in the 1970s [Dij68, Par72, Par76a, Par76b, HFC76]. The high complexity of these systems – even in the first decades of electronic computing – was the motivating factor for DIJKSTRA, PARNAS, HABERMANN, and others to develop fundamental concepts of software engineering, such as *modules*, *layers*, *hierarchies*, *levels of abstraction*, and *program families*. Whereas we accept most of these principles as natural today, the design of related software programs as **program families** is still not that common. PARNAS defines:

> *Program families are defined [...] as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members.* [Par76a]

Note that PARNAS' definition is about similarity between *programs,* that is, similarity between *solutions*. In general, the early work focuses on what today we call the *solution space*, that is, composition and decomposition techniques for program code. Program families (also known as *system families* or *product families*) are one design concept to implement the solution space of a software product line; the other techniques discussed in Section 2.1.5 can serve the same purpose; this thesis evaluates AOP in this respects.

In the following decades, many additional methods to analyze and engineer software product lines have been developed. Although additional techniques to implement the solution space have been suggested as well (such as *generators* [Cza98]), there has been a shift of focus from solution space eingineering to problem space engineering. Probably the first method in this direction was *Draco* by FREEMAN, which introduced the idea of *domain analysis* [Fre87]. Later, KANG and colleagues introduced with *Feature-Oriented Domain Analysis* (FODA) [KCH+90] – today an element of the *SEI Framework for Product Line Practice* [NC01] – the notion of *features* as user-visible discriminating elements between variants. Features as entities of the problem space were further generalized by SIMOS with *Organization Domain Modeling* (ODM) [Sim95]. Other examples of more recent methods include (but are not limited to): *Family-Oriented Abstraction Specification and Translation* (FAST) [WL99], *ProdUct Line Software Engineering* (PuLSE)

[11] A more precise definition of the term *feature* follows in Section 2.2.3.

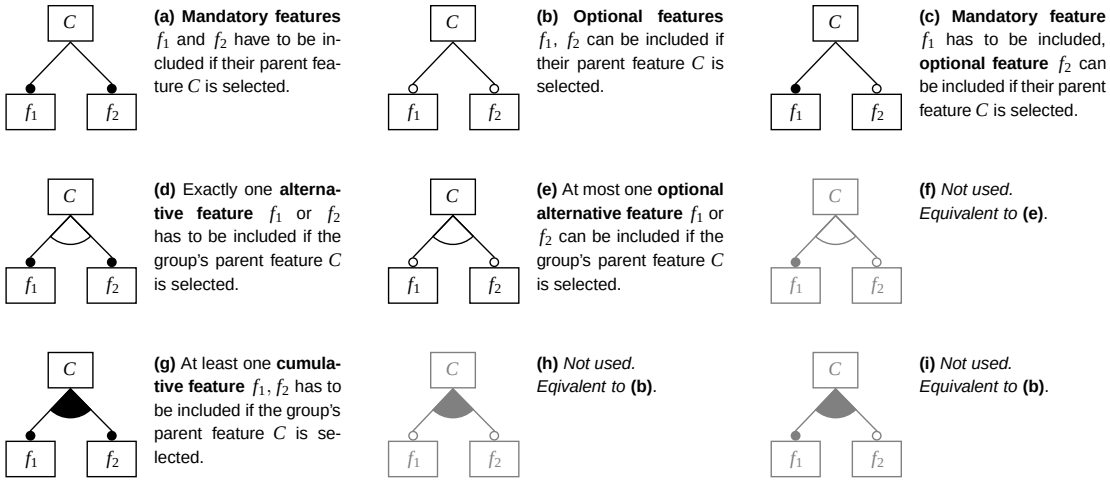


Figure 2.5.: Syntactical elements of feature diagrams.
The *concept C* is defined by its *features* $f_1$and f_2. Features can either be *mandatory* (displayed as nodes with a filled circle) or *optional* (displayed as nodes with an empty circle). Sub-features sharing the same parent node (concept or feature) can either be ungrouped (first row), constitute a group of *alternative features* (displayed by an arc over all edges leading to the group members, second row), or constitute a group of *cumulative features* (displayed by a filled arc, respectively, third row). Some theoretically possible combinations (depicted in gray) are not used as they are semantically equivalent to other constructs.

[BFK+99], *Komponenten-BasieRte Anwendungsentwicklung* (KobrA) [Atk01], *CONfiguration SUpport Library* (CONSUL) [Beu03a], *Process Integrated Modeling Environments* (PRIME) [PBvdL05], *Evolutionary Software Product Line Engineering Process* (ESPLEP) [Gom04, Gom05], and *Product Line Use case modeling for Systems and Software engineering* (PLUSS) [EBB05].

2.2.3. Specifying the Problem Space

As mentioned above, the entities used to specify the problem space are **features**. CZARNECKI and EISENECKER define (in line with earlier approaches, such as ODM [Sim95]) a feature as:

> *A distinguishable characteristic of a concept [...] that is relevant to some stakeholder of the concept.* [CE00, p. 38]

The explicit notion of stakeholders and the (for *them*) relevant characteristics of some concept underlines the abstract and intentional nature of features. Features are given in the terminology of the problem domain. The definition of the stakeholders and their view on the intended products, which ultimately leads to the set of features that span the problem space, is an important step of the domain analysis.

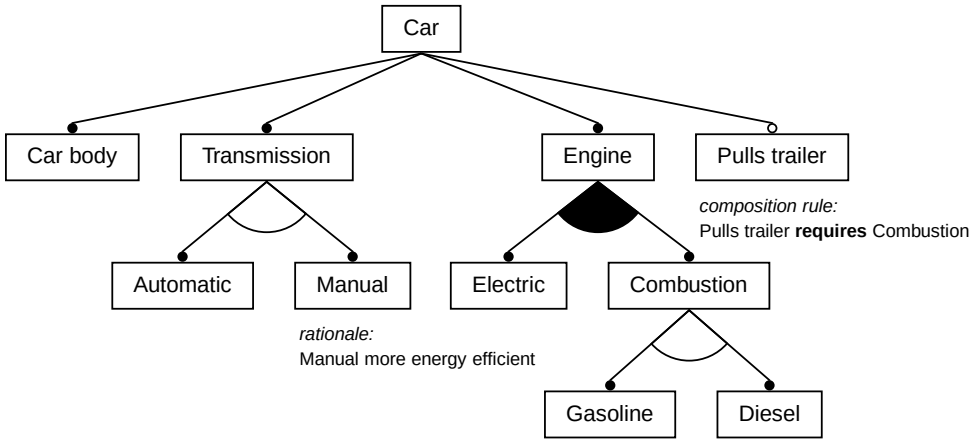


Figure 2.6.: Example for a feature diagram.
The concept Car consists of the mandatory features Car body, Transmission, Engine, and the optional feature Pulls trailer. The Transmission can be either Automatic or Manual; the Engine can be Electric, Combustion (with the alternative sub-features Gasoline or Diesel), or both (effectively a hybrid). If the optional Pulls trailer is chosen, however, an additional composition rule requires to choose a Gasoline engine as well (e.g., to ensure the torque required to pull a trailer).
(Figure adapted from [CE99])

In FODA [KCH+90], the outcome of the domain analysis is the **feature model**. The feature model defines and describes the abstract product, the **concept**, by its features and their relationships. A FODA feature model consists of four key elements:

1. The **feature diagram**, a graphical representation of the hierarchical (tree-like) decomposition of features with the concept as root node.
2. The **feature definitions**, which describe all available features.
3. The **composition rules**, which constrain feature combinations (and thereby possible configurations). The feature diagram syntax already offers syntactical means to specify certain composition rules, namely the indication whether or not some feature is optional, alternative, or mandatory. Additional composition rules (those that cannot be described by the feature diagram syntax) are given explicitly.
4. The **rationales for features**, annotations that indicate when to choose some feature and when not.

The central element is the feature diagram. Figure 2.6 shows an example of a feature diagram; Figure 2.5 explains the notation and the thereby expressible composition rules.[12]

Feature diagrams offer a compact and easy to comprehend presentation of possible configurations, that is, of the problem space. We will see more examples for feature diagrams in the subsequent chapters of this thesis.

[12] Throughout this thesis I do not use the original FODA notation [KCH+90], but the more pleasing notation introduced by CZARNECKI in his PhD thesis [Cza98].

2.2.4. The Problem Space of Configurable Operating Systems

Understanding the embedded operating system as a software product line can pave the path towards an easier and more problem-oriented way for customizing and tailoring of operating-system functionality. There are, however, some particularities that have to be considered.

In the literature, software product lines are mostly understood as a *process model* for the development of reusable software. Even though every author has his own take on the subject, there is a common agreement on what is called the *reference process of software product line engineering* [BKPS04, CE00, GS04]. This process describes software product line engineering by the two major activities of *domain engineering* and *application engineering*. Essentially, domain engineering is "developing *for* reuse" whereas application engineering is "developing *with* reuse" [CE00] around a common *software platform*. Software product-line development is generally understood as an *in-house* development process where both, domain engineers and application engineers, are on the same payroll [GS04, p. 354] and feedback from the application engineers triggers evolution of the software platform.

With system-software product lines the situation is different. In most cases an embedded operating system is sold by some vendor to *external* customers, which use it for their embedded applications and products. In this scenario, it is unrealistic that deriving concrete variants (the "application engineering") can be done on side of the operating-system vendor. Instead, the customer – the developer of the actual embedded product – has to become the "application engineer" of the operating-system product line.[13] The operating-system product line, or at least a part of it, is sold by the vendor and bought by the customer as a whole – it is itself a product.

This has some consequences:

1. The "application engineers" (customers) can not be expected to have expertise in operating-system development. They need extra guidance and tool support to derive the concrete variants without having to deal directly with the operating system plattform.
2. Because of the organizational distance between the vendor and the customers, the operating-system engineers have to anticipate more of the potential requirements application engineers might have.

The first issue can be addressed by representing *all* available options for customizing and tailoring as features in the problem space. In the sense of the problem-solution model of software product lines (Figure 2.4), an "application engineer" (customer) can then customize and tailor the system by configuring it – that is, by selecting features from

[13] The requirements often change during the develoment of the embedded product. Furthermore, customers want to protect their intellectual properties; if the vendor did the tailoring of the operating system, too much information about the product had to be passed around.

a representation of the feature model in some configuration editor. According to the resulting configuration, an accompanying tool generates the customized and tailored operating-system variant. This tool-based configuration approach is implemented, for instance, by PURE (configured with PURE::VARIANTS), ProOSEK (TRESOS), and eCos (ECOSCONFIG); also, the Linux kernel can be understood as a software product line with a feature-based configuration in this sense [SSSPS07] (multiple configuration tools).

The second issue can be addressed by understanding the set of requirement-motivated features only as a *feature starter set* [CE00] that is to be extended by additional features – the more, the better. An important source for additional features is the design and implementation process of the operating system itself. Once the core requirements have been analyzed and set (*top down*), operating systems are usually designed and developed *bottom up* – from hardware abstractions over operating-system abstractions up to the kernel interface. Alongside this bottom-up process, additional options for variability and granularity can sytematically be examined and, whenever feasible, made explicit as *additional* features in the feature model.

Hence, in operating-system product lines we have features from two different sources:

1. Requirement-motivated features that are the outcome of a *top-down domain analysis*. These features are motivated by *concrete requirements* (intentions), for instance compliance to some operating system standard such as *OSEK* [OSE05], *µITRON* [Sak98], or *POSIX*.
2. Implementation-motivated features are the outcome of a *bottom-up design and implementation process*. These features represent additional configuration options. They are motivated by common sense (of the implementing engineer who considers them as useful) and the *technical feasibility* to provide them with no extra run-time overhead and little implementation effort as extra configuration options.

Implementation-motivated features are often sub-features of requirement-motivated features. A requirement-motivated context-switch feature could, for instance, relatively easily be extended by additional configuration options for a further tailoring of the amount of context information to be saved and restored (e.g., if floating point registers should be included or not).

The result can be seen in the number of features offered by typical operating-system software product lines. The PURE feature model contains more than 250 features, most of which represent minimal extensions or variants of other features – fine granularity. eCos offers alltogether more than 750 configurable features and compatibility to different standards (µITRON, POSIX) – high variablity. The Linux kernel specifies even more than 3000 configuration options.

2.2.5. Implementing the Solution Space

The aim of the solution space is to provide application engineers with an actual implementation (variant) for any valid selection of features (configuration). The intentions

expressed by the features selected in the configuration have to be transformed somehow into an extensional representation, that is, some executable piece of software.

We implement the solution space by a repository of *implementation assets* and a *family model*.

- The **implementation assets** are the elementary building blocks to implement variants. Implementation assets may be complete variants (programs), combinable software units, or even "active elements", such as meta-programs or generators that have to be executed to generate configuration-dependent components or programs.
- The **family model** provides a mapping from each configuration to the set of implementation assets that together constitute the respective variant.

If the implementation assets are complete programs – one for each configuration – the family model is a simple one-to-one mapping from *configurations* to *programs*. This is feasible only if the product line offers very few configurations.

In the best case, the implementation assets are therefore arbitrarily combinable implementation units. Each unit implements exactly one feature; the family model is a one-to-one mapping from *features* to *units*. A configuration thereby leads to a set of units which have to be glued together (e.g., compiled and linked) into the respective variant.

Such an implementation of the problem space would be beneficial in many ways:

- It would lead to best possible synergies in the implementation of variants. The reuse of feature implementations is maximized.
- It should lead to a good variability and granularity. The implementation itself does not impair combining of features; variants do not contain any code related to unneeded features.[14]
- It would lead to an implementation with good software-engineering–related properties, such as good maintainability, extensibility, traceability of requirements, and so on. The one-to-one mapping from features (requirements) to implementation artifacts leads to an optimal separation of concerns.

Again, **separation of concerns** is the key here – it is not the result, but the prerequisite to achieve these benefits. Separation of concerns in software product lines means *feature cohesion,* that is, decomposition of mechanisms and policies into loosely coupled, composable program modules.

Hence, in the end we are back to implementation techniques for customizability, some of which have been discussed in Section 2.1.5 – with the small, but important difference that these techniques are now transparent for the application developer.

Nevertheless, the implementation approach for configurability has a significant influence on what economically *can* become a configurable feature in the problem space. With

[14] If the technique used for component glueing induces an extra overhead, as late binding in OOP does, it could as well have a negative impact on granularity.

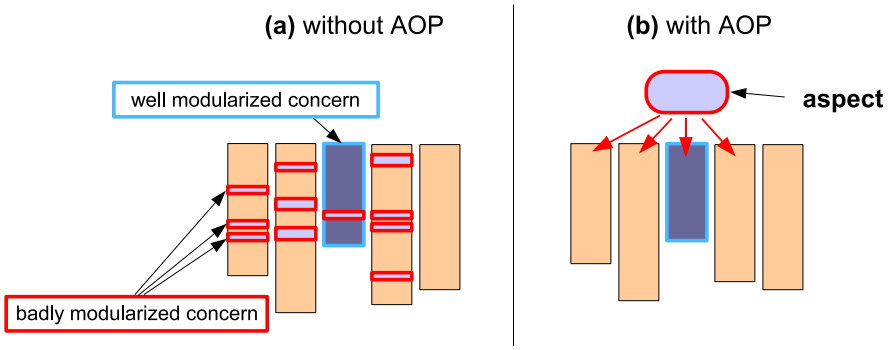


Figure 2.7.: Implementation of a "crosscutting concern" without and with AOP.
The bars represent source artifacts (e.g., classes); a concern is well modularized if it can be implemented by its own, dedicated source artifacts. **(a)** Without AOP, the implementation of "crosscutting concerns" is spread over the implementation artifacts of other concerns. **(b)** With AOP, the implementation of "crosscutting concerns" can be separated into their own source artifacts, called **aspects**.

the exception of PURE, the before-mentioned operating-system product lines all follow a two-level approach. Coarse-grained configuration decisions (e.g., regarding the inclusion of some device driver or other subsystem) are mapped to the abstractions of the implementation language – functions, classes, modules – and, hence, well separated in the code. Fine-grained configuration options, however, are enforced with the preprocessor – partly for efficiency reasons, partly because the configuration decision has an effect across multiple implementation entities. This is often the case for policy decisions.

With the concept of *aspects,* the relatively new paradigm of *aspect-oriented programming* aims to overcome some of these issues by offering extra language support for the clear separation of such "problematical" concerns.

2.3. Aspect-Oriented Programming

Aspect-oriented programming (AOP) aims to improve the separation of concerns in software by providing better means for the decomposition of software concerns into independent modules and the composition of complex software systems from such modules [KLM+97]. In particular, AOP tries to solve one of the most severe defects of object-oriented programming (OOP): The inability to decompose concerns in a way that they do not *crosscut* each other in their implementation. Often mentioned examples for concerns that tend to crosscut the implementation of other concerns are *synchronization*, *error handling*, and *tracing*. AOP offers means to encapsulate such **"crosscutting concerns"** into their own modules, called **aspects** (Figure 2.7).

2.3.1. The Problem of "Crosscutting Concerns"

The crosscutting between the implementation of two or more concerns usually becomes manifest as **scattered** and **tangled** code in the source code base.

> A concern implementation is **scattered** over the implementation of other concerns if it is not encapsulated by its own artifacts, but its implementing language entities (such as methods or fields) are distributed over the modules or source artifacts of the other concerns.
>
> A concern implementation is **tangled** with the implementation of some other concern if they share the same language entities (such as being implemented within the same set of methods).

Code scattering and code tangling basically describe the same phenomenon from different perspectives. A concern that is implemented by adding one or two members to a large number of classes has a scattered implementation. From the perspective of the affected classes, the concern implementation is tangled with the implementation of the other concerns implemented by these classes.

Tangling and scattering of feature implementations can become a major issue in the development of highly configurable software product lines. As pointed out in Section 2.2.5, separation of concerns – ideally a one-to-one–mapping from features to implementation components – is the key to the efficient assembly of variants. However, especially fine-grained configuration options are often difficult to implement by own, separated components.

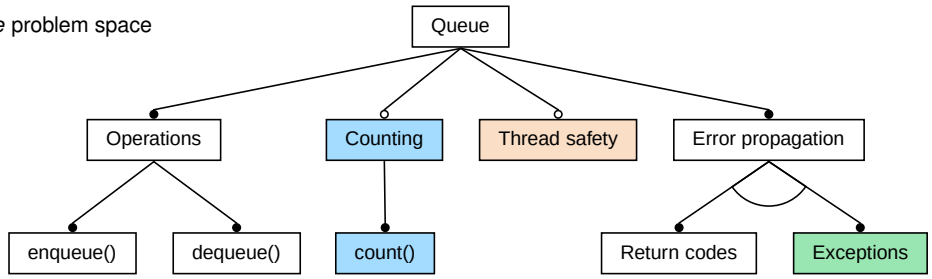
2.3.2. Queue Example: Scattering and Tangling in a Simple Product Line

Figure 2.8 demonstrates this by the example of a (very simple) product line for a FIFO data structure. The concept Queue (with mandatory operations enqueue() and dequeue()) can be configured with up to three optional or alternative features, namely Counting, Thread safety, and Exceptions.
Albeit their simplicity, the implementation of these features overlaps with the basic Queue implementation in a nontrivial way. It is difficult to separate them.

To implement the feature Counting, for instance, a new member variable `int counter` has to be added to class `Queue` (line 8); it has to be initialized (line 14), incremented as part of the `enqueue()` operation (line 25), and decremented after a successful `dequeue()` operation (lines 45ff.). Furthermore, a new `count()` method has to be added to class `Queue` to provide access to the counter value (lines 59ff.).

For the Thread safety and Exceptions features, the situation is similar: Their implementation crosscuts the implementation of the basic Queue functionality (Figure 2.8.b).

(a) *Queue* problem space



(b) Queue.h (C++ implementation)

```
1  #include "os/Mutex.h"
2  struct Item {
3    Item* next;
4    Item() : next(0){}
5  };
6  class Queue {
7    Item *first, *last;
8    int counter;
9    os::Mutex lock;
10 public:
11   struct InvalidItemError{};
12   struct EmptyError{};
13   Queue () : first(0), last(0) {
14     counter = 0;
15   }
16   void enqueue(Item* item) {
17     lock.enter();
18     try {
19       if (item == 0)
20         throw InvalidItemError();
21       if (last) {
22         last->next = item;
23         last = item;
24       } else { last = first = item; }
25       ++counter;
26     }
27     catch (...) {
28       lock.leave();
29       throw;
30     }
31     lock.leave();
32   }
33   Item* dequeue() {
34
35     Item* res;
36     lock.enter();
37     try {
38       res = first;
39       if (first == last) {
40         first = last = 0;
41       }
42       else {
43         first = first->next;
44       }
45       if (counter > 0) {
46         --counter;
47       }
48       if (res == 0) {
49         throw EmptyError();
50       }
51     }
52     catch (...) {
53       lock.leave();
54       throw;
55     }
56     lock.leave();
57     return res;
58   }
59   int count() {
60     return counter;
61   }
62
63 }; // class Queue
```

Figure 2.8.: Scattered and tangled code in the implementation of the Queue product line.
Depicted is a feature diagram describing the variability of a simple FIFO data structure and a corresponding C++ implementation in the solution space. Feature nodes and source lines have been colored for the sake of traceability; each feature accounts for the source lines shaded in the corresponding color. **(a)** The concept Queue can optionally be extended by three features: Counting (bookkeeping of the number of enqueued elements), Thread safety (execution of methods is serialized between independent threads), and Exceptions (error propagation by throwing an exception; this is an alternative to the standard behavior of indicating fault situations by a return code.) **(b)** The C++ class `Queue` implements the "deluxe variant" (configured with all optional features) of the Queue concept. The implementation of Thread safety, Counting, and Exceptions is scattered over and tangled within the implementation of the basic queue. Affected are all operations of the basic queue implementation (`enqueue()`, `dequeue()`) as well as the static structure (elements) of the class `Queue` itself.

2.3.3. Dimensions of Crosscutting

We can categorize crosscutting between concern implementations by two dimensions. The first dimension describes the *space* in which the crosscutting arises (structural space vs. behavioral space). The second dimension the *distribution* of crosscutting (regular overlapping vs. irregular overlapping).

2.3.3.1. Space Dimension

The space dimension describes *where* some concern implementations overlap. We distinguish between **static crosscutting** and **dynamic crosscutting**:

Static Crosscutting. The concern implementations overlap in the structural space of the program; they share some of the entities that constitute its static structure.

In most languages, the static structure of a program is constituted by the types and their relationships, so static crosscutting effectively comes down to *sharing of types*.

Dynamic Crosscutting. The concern implementations overlap in the behavioral space of the program; they share events in the run-time control-flow that are augmented with their behavior.

In most languages, the run-time control flow is specified by sequences of statements that constitute the control-flow graph of the program. Dynamic crosscutting effectively comes down to sharing edges in the control-flow graph (although this is a somewhat simplistic description).

Static and dynamic crosscutting often go together. The implementation of each of the optional or alternative features from our example (Figure 2.8.b), namely Counting, Thread safety, and Exceptions, statically and dynamically crosscuts the basic Queue implementation. The implementation of Counting, for instance, statically crosscuts with the implementation of Queue by contributing members (member variable `counter`, method `count()`) to the same type (class `Queue`). It furthermore dynamically crosscuts with the implementation of Queue by augmenting the run-time behavior associated with the events "execution of `Queue:enqueue()`" and "execution of `Queue::dequeue()`".

2.3.3.2. Distribution Dimension

The distribution dimension describes *how* the overlapping between concern implementations is distributed over the space dimension. We distinguish between **homogeneous crosscutting** and **inhomogeneous crosscutting**:

Homogeneous Crosscutting. There is a recurring and regular pattern in the overlapping of concerns; some concern implementation *template* is effectively instantiated many times and distributed over the space dimension where it overlaps with other concern implementations.

Inhomogeneous Crosscutting. There is no regular pattern in the overlapping of concerns; specific parts of some concern implementation overlap with other concern implementations in specific entites of the static structure (static crosscutting) or specific run-time events (dynamic crosscutting).

The canonical example for a homogeneously (and dynamically) "crosscutting concern" in the AOP literature is Tracing. Its intended behavior (print the method name at the begin and end of any method execution) typically results in an implementation where the same line of code is inserted at the begin and end of every single method.

In our Queue example (Figure 2.8.b), the implementation of Thread safety homogeneously and dynamically crosscuts the implementation of Queue. The intended synchronization by a mutex object makes it necessary to augment the execution of every method of class `Queue`.[15] If Thread safety should not only protect our class `Queue`, but all classes in the system, it would also statically crosscut the implementations of other concerns. In this case, the `os::Mutex lock` member would have to be inserted into every single class of the system.

The implementations of Exceptions and Counting inhomogeneously crosscut the implementation of Queue. Their intended semantics makes it necessary to augment method-specific behavior (how to test and report failures, if to increment or decrement the counter) to the execution of each method of class `Queue`.

2.3.4. Concepts and Terminology of Aspect-Oriented Programming

The fundamental idea behind AOP to overcome the issues described in the previous section is to separate the *what* from the *where* of concern implementations. Separation of *what* and *where* means that the implementation of some concern (the *what*) does not have to be specified at the (physical) place it augments the program in the space and distribution dimensions. This is often referred to as **quantification** and **obliviousness** [EFB01] and considered a fundamental property of AOP. FILMAN and FRIEDMAN write that:

> *the distinguishing characteristic of Aspect-Oriented Programming systems (qua programming systems) is that they provide quantification and obliviousness.* Quantification *is the idea that one can write unitary and separate statements that have effect in many, nonlocal places in a programming system;* obliviousness*, that the places these quantifications applied did not have to be specifically prepared to receive these enhancements.* [Fil01]

It is the quantification property that facilitates decoupling from the distribution dimension and the obliviousness property that facilitates decoupling from the space dimension of crosscutting.

[15] In fact we also have an overlapping between the implementations of Thread safety and Counting. The presented implementation of `Queue::count()` is correct only under the assumption that loading and storing of an `int` is performed by the CPU as an atomic operation. Otherwise, `Queue::count()` would have to be synchronized by the mutex as well.

"Crosscutting Concerns"

You may have noticed that I use the phrase "crosscutting concern" only in quoted form. In my opinion "crosscutting concerns" do not really exist, despite the fact that they are mentioned over and over again in the AOP literature.
First and foremost, concerns are intentional constructs. As such, we do not deal with them directly, but only with some *model* (or *representation*) of them in some *modeling language*. Hence, crosscutting is not a *property* of a concern, but has to do with some specific model of it. In fact, crosscutting is a *relationship* between concern models. More precisely:

> **Crosscutting** is a *symmetrical relationship* between the *models* of at least two concerns that indicates an overlapping between these models.

In theory, there might always be another modeling language in which the concerns would not overlap. If, for instance, the object-oriented models of two concerns crosscut each other, it might well be the case that this would not happen if using a functional programming paradigm or a modeling based on state automata. Note, moreover, the word *symmetrical* in the above definition: If A and B are models of concerns and A crosscuts B, then this implies also that B crosscuts A.
Nevertheless, developers *perceive* crosscutting as an inherent property of *certain* concerns: Tracing, for example, is said to be "a typical crosscutting concern". This is probably a heritage of the AOP model introduced with AspectJ [KHH+01b, KHH+01a]. By selling AspectJ as a language *extension* to Java and aspects as an *additional* syntactic means to model concerns, the relationship between concern models became asymmetrical: Aspects affect classes – and not vice versa. Most concerns get still implemented as Java classes. Only if, for whatever reason, this does not work, the new `aspect` construct is there as a back-up. This is substantially different from the original idea of AOP, which considered all concern implementations as equally important aspects [KLM+97], or the attempts around the Hyperspace-Approach, which was intended to overcome the fixation on decomposing into classes – nicely described by OSSHER and TARR as the "tyranny of the dominant decomposition" [TOHS99, OT00].
However, it might also be a reason that in fact we often have a "natural order" between concerns: The core functionality and the business logic are considered as "first order concerns", issues like tracing and monitoring are understood as "second order concerns". So even if the crosscutting between their implementations (models) is symmetrical in principle, we perceive only the "second-order concerns" as causing the trouble, hence, attribute the crosscutting to them. It seems just natural to use the `class` construct to implement the "first order concerns" and the `aspect` construct for the "problem-causing crosscutting second order concerns".
After all, it is striking that the AspectJ model of AOP with its (conceptually not favorable) asymmetrical relationship between classes and aspects has been so much more successful than the other models. It is, as GREGOR KICZALES called it, " more programmer compatible" [KHH+01b].

2.3.4.1. Language Concepts

The most relevant AOP language concepts for this purpose are *join point* and *advice*. An **advice** definition describes a transformation to be performed at specific positions in the space dimension (the *what*). A **join point** denotes such a specific position in the space dimension of the target program (the *where*). Advice, however, is not given to single

join points, but to *sets of join points* called **pointcuts**. A pointcut describes a specific distribution in the distribution dimension. Pointcuts are given by **pointcut expressions**, sentences of a declarative **join point description language** that yield sets of join points by reasoning and quantifying over the static structure or the dynamic control flow of the program. Advice and pointcuts can be grouped (together with other elements, such as state variables and methods) in class-like structures called **aspects**. An aspect thereby implements a "crosscutting concern" in a well encapsulated manner.

2.3.4.2. Implementation Concepts

Eventually, the separated and encapsulated concern implementations somehow have to be composed into the final program. This composition process is called **aspect weaving**, the corresponding tool is the **aspect weaver**. Technically, the aspect weaver actually applies the transformations specified as advice.

The process of aspect weaving can be compared to the process of (object-file) linking. As with linking, a set of separated implementation modules is bound together into a single module. A linker, however, collects and resolves symbol references, whereas much of the composition work of an aspect weaver takes place on a sub-symbol level. This explains the *weaving* metaphor.

Aspect weaving, as linking, can be performed at various stages in the life cycle of a program. We distinguish between **static weaving**, which means weaving at build time, and **dynamic weaving**, which stands for weaving after build time, respectively. Whereas static weaving can (such as static linking) be applied as an overhead-free mechanism,[16] dynamic weaving is in general much more expensive than dynamic linking. The reason is the very fine level on which an aspect weaver performs its composition; a much larger set of potential junction points (symbol references in the case of linking, join points in the case of weaving) has to be maintained at load time or run time [LGS04]. It is possible to reduce this overhead by a strict tailoring of the potential join points [SPLG+06], however, this also limits the flexibility of aspects and still induces severe costs. I consider static weaving as the more suitable approach for the domain of resource-constrained embedded systems.

2.3.5. Queue Example: Solution Space Implementation with AspectC++

Listing 2.1 demonstrates the application of these concepts to overcome the issues in the implementation of the Queue product line discussed in Section 2.3.2. The AOP-based implementation uses **AspectC++** [SGSP02, SLU05a, SL07, AC+], an AOP language extension to the C++ language. In the following, I give only a short overview of AspectC++ (you can find a detailed introduction into AspectC++ and the complete language reference in Appendix A). The example under discussion is the aspect `LockingMutex` from Listing 2.1.c, which implements the Thread safety feature.

[16] We will see this in Chapter 4 of this thesis.

(a) Queue.h

```
struct Item {
  Item* next;
  Item() : next(0){}
};
class Queue {
  Item *first, *last;
public:
  Queue () : first(0), last(0) {
  }
  void enqueue(Item* item) {
    if (last) {
      last->next = item;
      last = item;
    }
    else
      last = first = item;
  }
  Item* dequeue() {
    Item* res;
    res = first;
    if (first == last)
      first = last = 0;
    else
      first = first->next;
    return res;
  }
}; // class Queue
```

(b) Counting.ah

```
#include "queue.h"
aspect ElementCounter {
  advice "Queue" : slice class {
    int counter;
  public:
    int count() const { return counter; }
  };

  advice construction("Queue")
    && that(queue)
  : before (Queue& queue) {
    queue.counter = 0;
  }

  advice execution("% Queue::enqueue(...)")
    && that(queue)
  : after( Queue& queue ) {
    ++queue.counter;
  }

  advice execution("% Queue::dequeue(...)")
    && that(queue)
  : after( Queue& queue ) {
    if( queue.counter > 0 )
      --queue.counter;
  }
};  // aspect ElementCounter
```

(c) Threading.ah

```
#include "mutex.h"
#include "queue.h"
aspect LockingMutex {
  advice "Queue" : slice class {
    os::Mutex lock;
  };
  advice execution("% Queue::%queue(...)")
    && that(queue)
  : around( Queue& queue ) {
    queue.lock.enter();
    try {
      tjp->proceed();
    }
    catch(...) {
      queue.lock.leave();
      throw;
    }
    queue.lock.leave();
  }
}; // aspect LockingMutex
```

(d) Exceptions.ah

```
#include "queue.h"
aspect ErrorException {
  advice "Queue" : slice struct {
    struct QueueInvalidItemError {};
    struct QueueEmptyError {};
  };
  advice execution("% Queue::enqueue(...)")
    && args(item)
  : before(Item* item) {
    if( item == 0 || item->next != 0 )
      throw QueueInvalidItemError();
  }
  advice execution("% Queue::dequeue(...)")
    && result(item)
  : after(Item* item) {
    if( item == 0 )
      throw QueueEmptyError();
    item->next = 0;
  }
}; // aspect ErrorException
```

Listing 2.1: Implementation of the Queue product line using AspectC++.
Depicted is an alternative implementation of the solution space for the Queue product line from Figure 2.8. By means of AOP, feature implementations could be separated into distinct artifacts. **(a)** Class `Queue` implements the basic Queue concept with no optional features. **(b, c, d)** The optional features Thread safety, Counting, and Exceptions are implemented by the *aspects* `LockingMutex`, `ElementCounter`, and `ErrorException`, respectively. Each of these aspects gives *advice* to superimpose the implementation of the particular feature into class `Queue`.

As mentioned in the previous section, *advice* and *join point* are the fundamental language concepts of AOP. Advice is given to sets of join points called *pointcuts*, which in turn are specified by *pointcut expressions*. In AspectC++, pointcut expressions are made from **match expressions** and **pointcut functions**. Match expressions are already primitive pointcut expressions and yield a set of **name join points**. Name join points represent elements of the structural space, such as classes or functions. Technically, match expressions are given as quoted strings that are evaluated against the identifiers of a C++ program. The expression `"Queue"` in line 4, for instance, yields a name pointcut containing just the class `Queue`. The expression `"% Queue::%queue(...)"` in line 7 yields a name pointcut containing any overload of any function in the scope of the class or namespace `Queue` whose identifier ends in `queue`. The percent sign (`%`) and the ellipsis (`...`) serve as wildcards for any name/type and any number of arguments, respectively. In other words, the expression evaluates to a name pointcut containing the methods `Queue::enqueue()` and `Queue::dequeue()`.

Events in the behavioral space are represented by **code join points**. Code pointcuts are retrieved by feeding name pointcuts into certain pointcut functions such as `call()` or `execution()`. The pointcut expression `execution("% Queue::%queue(...)")` in line 7, for instance, yields all the events in the dynamic control flow where any overload of `Queue::enqueue()` or `Queue::dequeue()` is about to be executed.

Name pointcuts are used to give advice to the *structural space* of the program. In AspectC++, structural extensions are called **introductions**; syntactically, an introduction is expressed as a `slice` of elements to be introduced into the classes specified by the name pointcut. In lines 4–6, the `LockingMutex` aspect uses this mechanism to introduce a member `os::Mutex::lock` into the class `Queue`.

Code pointcuts are used to give advice to the *behavioral space* of the program. In AspectC++, behavioral modifications are given as advice code to be triggered `before`, `after`, or `around` (instead of)[17] the code join points. In lines 7–19, the `LockingMutex` aspect uses `around` advice to ensure that the introduced mutex is acquired and released around the execution of `Queue::enqueue()` or `Queue::dequeue()`. For the sake of exception-safety, the original event behavior is triggered inside a `try-catch` block in line 12. In AspectC++, the original behavior can be invoked using the `proceed()` method from the **join point API**. This API is transparently available inside an advice body and provides the advice code with access to join-point–specific context or behavior.

As Listing 2.1 shows, not only the Thread safety feature, but also Counting and Exceptions could be decomposed into their own, independent aspects. By means of AOP and AspectC++, the solution space of the Queue product line now offers a one-to-one mapping from features to implementation components – an optimal separation of concerns.

[17] *Around* is more than *before* plus *after* as it possibly replaces the original behavior associated to the code join point.

2.3.6. History of AOP Languages

AspectC++ is just one example for an AOP language. The term *Aspect-Oriented Programming,* as well as the notion of *join points, advice*, and *aspects,* was coined by the group around CRISTINA VIDEIRA LOPES and GREGOR KICZALES. They originally introduced the AOP idea independently from some specific language [KLM+97, Lop97]. However, AOP became particularly popular after the introduction of *AspectJ* [KHH+01b, KHH+01a], an AOP-extended version of the Java programming language.

AOP was not the first attempt to deal with the problem of crosscutting between concern models. Nevertheless it is understood today as the general term for any such approach, including earlier attempts. Newer attempts, moreover, mostly adapt or extend the AOP model introduced by AspectJ. Hence, the history of AOP can be divided in a *pre*-AspectJ and a *post*-AspectJ era.

2.3.6.1. Early AOP Languages

Most notable among the early approaches, which today are understood as the roots of AOP, are *Composition Filters, Subject-Oriented Programming*, *Adaptive Programming*, and the *D Framework*.

The *Composition Filters* model by AKSIT and BERGMANS was probably the first approach that addressed the problem of dynamic crosscutting with declarative means. Declaratively described, composable message interception filters made it possible to alter the message passing between objects [ABV92, Ber94]. A little later, *Subject-oriented Programming (SOP)* was suggested by HARRISON and OSSHER to address the problem of crosscutting in software composition [HO93]. By the concept of *subjects* (which are basically sets of type slices), SOP pioneered the idea of language support for static, inhomogeneous crosscutting; some support to deal with dynamic crosscutting was also available by means of method composition. SOP was later generalized by TARR, OSSHER and colleagues to the *Hyperspace Approach* and the *Hyper/J* language, which advocate the idea of a real *multi-dimensional separation of concerns* [TOHS99, OT00, OT01]. Approaches to deal with specific kinds of crosscutting were suggested by LIEBERHERR, LOPES and colleagues. *Adaptive Programming* with the *Demeter* method was introduced to decouple the implementation of object traversals from the actual class structures in complex class graphs [Lie96], which can be understood as a specific form of inhomogeneous static crosscutting. Later, the Demeter method was also applied with AspectJ [LLW03]. The goal of the *D Framework* was to decouple synchronization and distribution concerns from component implementations [Lop97]; for this purpose it particularly dealt with problems of quantification and homogeneous dynamic crosscutting.

2.3.6.2. Current AOP Languages

The introduction of AspectJ denoted a philosophical shift in AOP towards general purpose aspect languages. An important goal of AspectJ was to increase *"programmer compatibility"*, that is, to make it easier for the average programmer to adopt AOP [KHH+01a]. So AOP was now understood and sold as a language *extension* to some broadly used **base language**. AspectJ is an upward-compatible super-set of Java; it extends Java (the base language) by additional syntactic means for aspects, pointcuts and advice.

Current AOP approaches, that is, approaches introduced after AspectJ, mostly follow the AspectJ philosophy, semantics, and terminology. Many of them adapted the concepts of AspectJ to other (nonJava) language domains. AspectC++, which is used in this thesis, is just one example. Besides many other Java-based approaches, the AOP community has suggested AspectJ-like language extensions for C [CKFS01, GJ, NvGvdP08, Ada08], Smalltalk [Hir03], C# and .NET [SP02, SSW02], PHP [Yu], Ruby [BF], Cobol [LS05a], and many more. However, only few aspect languages are suitable for the domain of resource-constrained embedded systems. I shall discuss these languages more extensively in Chapter 4.

2.3.7. AOP in Operating Systems

Among the first who applied aspects to the domain of operating systems were COADY and colleagues. In the *α-kernel* project, they retroactively analyzed the evolution of four scattered OS concern implementations (namely: *prefetching*, *disk quotas*, *blocking*, and *page daemon activation*) between version 2 and version 4 of the FreeBSD kernel [CK03]. Their results showed that an aspect-oriented implementation would have led to significantly better evolvability of these concerns.

Around the same time, MAHRENHOLZ, SPINCZYK and colleagues experimented with AspectC++ in the PURE OS product line [MSGSP02, Spi02, SL04]. By application of AOP, they could implement two previously hard-wired OS concerns (namely: *interrupt synchronization*, *driver execution model*) as configurable features.

Not a general-purpose AOP language but an AOP-inspired language of temporal logic was used by ÅBERG and associates to integrate the *Bossa* scheduler framework into the Linux kernel [ÅLS+03]. Another example for a special-purpose AOP-inspired language is *C4* by FIUCZYNSKY and colleagues, which is intended for the application of kernel patches in Linux [FGCW05].

Other related work concentrates on *dynamic* aspect weaving as a means for run-time adaptation of operating-system kernels: *TOSKANA* by ENGEL and FREISLEBEN provides an infrastructure for the dynamic extension of the FreeBSD kernel by aspects [EF05, EF06]; YANAGISAWA and colleagues presented an approach for aspect-based dynamic instrumentation in Linux [YKCI06].

All these studies demonstrate that there are good cases for aspects in system software. However, for a broader application to the resource-thrifty domain of embedded systems, an in-depth analysis of the hardware cost impact of aspects is required. This is still missing.

Also remarkable is that all these studies are based on *existing* kernels. So far no study exists that analyzes the effects of using AOP for the *development* of an operating-system kernel from the very beginning. We can assume that AOP leads to even higher benefits in such case, especially with respect to application-specific configurability of such a kernel.

2.3.8. AOP Critique

With its rise, AOP and the related language concepts have also been meeting with more and more criticism that, for the time being, culminated in a widely noticed essay by F. STEIMANN: "The Paradoxical Success of Aspect-Oriented Programming" [Ste06]. In general, the critical examination of AOP shows two major movements:

1. *It is not powerful enough* – the "practitioners" point of view, which has been expressed by researchers from various disciplines who experimented with AOP for the separation of concerns in existing software systems [ÅLS+03, HG04, SWK06, FCF+06, EA06, HE07].
2. *It is too powerful (that is, dangerous)* – the "software engineers" point of view, which has been expressed by researchers from the software engineering community who analyzed the impact of AOP to modularity and modular reasoning [CL03, CSS04, SGS+05, Ste06, KAB07, KAK08].

Even though seemingly contradictory (and, admittedly, a bit simplisticly summarized above), both stances on AOP are valid from their perspective. They are mostly the result of a different operationalization of the terms *modularity* and *separation of concerns*.

2.3.8.1. The "Practitioners" Perspective

The "practitioners" operationalize modularity primarily as *source code modularity*; separation of concerns comes down to keeping things *separated in the code*. Hence, they concentrate their critique on (the limitations of) the AOP *mechanisms*. The general issue in the above cited papers was that because of missing potential join points the aspect code could not interact as densely as necessary with the base code to achieve the intended separation of concerns. As a consequence, some of the "practitioners" also propose new AOP mechanisms to overcome the limitations for their particular situation, such as *temporal logic* [ÅLS+03], *statement annotations* [EA06], or *explicit join points* [HE07].

2.3.8.2. The "Software Engineers" Perspective

For the "software engineers", modularity is a much more fundamental concept, based on PARNAS' ideas of data *encapsulation*, *information hiding*, and *well-defined interfaces*

[Par72].[18] Separation of concerns means the ability to *separately design* modules during the software development process. Hence, their critique concentrates on the *methodology* behind AOP (but also attacks the AOP mechanisms for breaking encapsulation in an uncontrolled manner – "even worse than `goto`" [CSS04] – and for providing an expressiveness that is rarely needed [KAB07]).

Especially the *oblivousness principle* postulated by FILMAN and FRIEDMAN ("Just program like you always do, and we'll be able to add the aspects later." [FF00], see also Section 2.3.4) is a frequent point of critique, as it has a negative impact on modular reasoning [CL03, SGS+05, Ste06] and leads to interfaces between classes and aspects (the potential join points that aspects can bind to) that are "implicit at best" [Ste06]. As all attempts to make components *more* aware of aspects (to restore modular reasoning in aspect-oriented programs) also lead to a loss of obliviousness, STEIMANN concludes that "the problems of AOP cannot be fixed without giving up its distinguishing characteristics" [Ste06]. The *quantification principle* (also postulated by FILMAN and FRIEDMAN [FF00], compare Section 2.3.4) is generally less criticized, even though considered as "rarely applicable" by some authors [KAB07].

Another point of critique is the irrelevance of aspects as an early design concept. In another paper, STEIMANN points out that "literally all aspects discussed in the literature are technical in nature: authentication, caching, distribution, logging, persistence, synchronization, transaction management, etc.", from which he concludes that aspects are (in contrast to roles or classes) not a domain concept but "aspects *of programming*" [Ste05].

2.3.8.3. The Perspective Taken in This Thesis

The goal of my thesis is to evaluate the suitability of AOP for the implementation of fine-grained configurability in the solution space of software product lines targeted a specific domain – system software for resource-constrained embedded systems. Hence, this thesis takes more the "practitioners" perspective. It is first and foremost about AOP *mechanisms* – understanding their cost, limitations, and benefits:

> *Understanding something involves both understanding how it works (mechanism) and what it's good for (methodology). In computer science, we're rarely shy about grandiose methodological claims (see, for example, the literature of AI or the Internet). But mechanism is important – appreciating mechanisms leads to improved mechanisms, recognition of commonalities and isomorphisms, and plain old clarity about what's actually happening.* (FILMAN et al in [FECA05])

I agree with STEIMANN's opinion that aspects are generally not a (problem!) domain concept, but "technical in nature". However, that does not render AOP *per se* useless –

[18] It is a curiosum that this article from PARNAS ("On the criteria to be used in decomposing systems into modules") is also universally cited by the AOP community to *motivate* the need for aspects; a fact that STEIMANN commented with: "My problem with citing Parnas's work is that in my eyes it does not accommodate the AOP form of modularity; if anything, it forbids it." [Ste06]

system software *is* very technical in nature, too; the above mentioned “technical” aspects are dominant concerns of system software development! In the *specification* of AUTOSAR OS [AUT06a], for instance, we can find the *requirement* OS093:

> *If interrupts are disabled and any OS services, excluding the interrupt services, are called outside of hook routines, then the operating system shall return* `E_OS_DISABLEDINT`. [AUT06b, p. 40]

This requirement translates almost “literally” to an AspectC++ aspect:

```
aspect DisabledIntCheck {
  advice call( pcOSServices() && !pcInterruptServices() )
  && !within( pcHookRoutines() ) : around() {
    if( interruptsDisabled() )
      *tjp->result() = E_OS_DISABLEDINT;
    else
      tjp->proceed();
} };
```

Nevertheless, my thesis also involves facets of AOP methodology. The idea of total obliviousness, for instance, will be given up – however, we will see that the mechanisms behind obliviousness remain *nevertheless* useful. The mechanisms for quantification will be improved – and shape up as *broadly* applicable.

2.4. Chapter Summary

Embedded systems are special-purpose systems. They are ubiquitous, we can find them in many different shapes in our everyday lives. They are typically applied to areas of mass production – domains, in which the hardware cost pressure calls for a strict reduction of functionality to what is actually needed. Hence, system software for embedded systems has to cope with a huge set of potential requirements, but also with relatively small and specific sets of actual requirements. It has to be easily customizable and tailorable. This is especially true for the operating system. State of the art to implement fine-grained customizability in operating system code is the C preprocessor.

Software product lines are a promising approach towards a higher-level understanding and representation of customizability in embedded operating systems. The general idea is to provide customers with a mapping from an intentional problem space describing potential configurations to an extensional solution space that contains the actual software variants. In practise, all these variants have to be implemented somehow, which ideally would be done by a good separation of concerns in the implementation of features.

AOP aims at improving separation of concerns by extending other programming paradigms, such as OOP, by another dimension of decomposition. It specifically provides developers with a clear separation of the *what* and the *where* of concern implementation.

3

Problem Analysis and Suggested Approach

Embedded system software should be easy to customize and tailor for different sets of requirements. The notion of configurable features in software product lines paves a way towards an intuitive, problem-centric view on fine-grained customization and tailoring. However, as domain engineers of the system-software product line we eventually have to implement all this configurability – while still maintaining run-time and memory efficiency in the resulting code.

In this chapter I analyze the problems, present my research assumption – that AOP improves on the situation – and discuss my approach to evaluate this.

I begin in Section 3.1 with the problem analysis, which is twofold. In the first part, I analyze the limits of configurability by conditional compilation on the base of a real operating system that reflects the current state of the art of operating-system product lines. In the second part, I then motivate that we need even better configurability – configurability of architectural system policies. I discuss in Section 3.2 why and how AOP could help here and what are the open issues. This is followed in Section 3.3 by the presentation of my research approach – the bottom-up evaluation of aspect-aware operating-system development. Finally, I summarize the chapter in Section 3.4.

Related Publications

The ideas and results presented in this chapter have partly also been published as:

[LS03] Daniel Lohmann and Olaf Spinczyk. Architecture-neutral operating system components. *23rd ACM Symposium on Operating Systems Principles (SOSP '03)*, October 2003. WiP presentation.

[LSPS05] Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS '05)*, pages 413–420, Sedona, AZ, USA, February 2005.

[LSSP05] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, pages 19–25, Chicago, IL, USA, March 2005. Northeastern University, Boston (NU-CCIS-05-03).

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM Press.

3.1. Problem Analysis

State of the art to implement overhead-free and fine-grained configurability in system-software product lines is the C preprocessor. As already mentioned in Section 2.1.5.5, the major disadvantage of this approach is that the variant generation process is *decompositional* – the feature representation in the source code has to encompass *all* variants; concrete variants are generated by filtering out the unneeded parts. Instead of *separation* we have *commingling* of concerns. The resulting source code is bloated, difficult to read, understand, and maintain. In the following, I illustrate what this actually means on the example of eCos, a well-known operating-system product line for embedded devices.

3.1.1. How Configurability Becomes Manifest in the Code – The eCos Case

eCos, the *embedded Configurable operating system* [eCo, Mas02] is an operating-system product line for the domain of embedded systems. As such, eCos is available for a broad variety of 16- and 32-bit microprocessor architectures (PPC, x86, H8/300, ARM7, ARM9, ...) and used in many different application domains. eCos reflects the current state of the art in many respects:

- more than 750 configurable features
- feature-based configuration process and tool
- kernel implementation in C++
- production system (as opposed to most research operating systems)
- well-known and broadly accepted in the embedded-systems community

In a survey performed by *embedded.com* in 2006, eCos takes position 12 on the list of the most popular embedded operating systems [Tur06], which is the highest position among open-source operating systems for small embedded devices.[1]

3.1.1.1. The eCos Configuration Process

eCos is shipped as a repository of components (the assets) and a feature-based configuration tool called ECOSCONFIG (Figure 3.1). From a valid configuration, ECOSCONFIG generates the variant of the operating system as set of headers and makefiles, which are then used to compile the *eCos-library*. Against this library the final applications will be linked.

Configuration decisions are enforced by a two-level approach: Each *package* selected in ECOSCONFIG defines a set of source artifacts to be compiled into the eCos-library. This

[1] with "small" being defined as "without MMU". Linux was ranked higher, but addresses a quite different domain.

is the first level of configuration; only selected packages become part of the eCos-library (the actual variant). Fine-grained inter-artifact and intra-artifact *configuration options* are then enforced by the preprocessor. Each of these options is represented as a preprocessor macro. This is the second level of configuration. The macro `CYGFUN_KERNEL_THREADS_DATA`, for example, enables thread-local storage support in the *eCos kernel* package. Overall, more than 60 packages (*TCP/IP*, *disk IO*, *µItron*, *POSIX*, ...) offer a total of more than 750 configuration options; the kernel can be configured by about 100 configuration options.

3.1.1.2. Overview of the eCos Kernel

The eCos kernel is implemented by 5000 lines of C++ code and provides a rich set of abstractions to develop and maintain multi-threaded, event-triggered applications. This includes *thread management*, *thread synchronization*, and *interrupt handling*:

Thread management. At configuration time, the user can select among several priority-based scheduling algorithms. All threads are preemptable; preemption can temporarily be disabled by acquiring a global scheduler lock. Threads can optionally be equipped with support for thread-local storage, exit handlers, stack validation, and much more.

Thread synchronization. The eCos kernel offers all typical abstractions for thread synchronization, such as *mutex*, *semaphore*, *event*, and *message*. Optional features include, for instance, several protocols to prevent priority inversion problems.

Interrupt handling. eCos uses a classical two-level interrupt handling scheme: Low-level *interrupt service routines* (*ISRs*) have the highest priority among all control flows. ISRs are noninterruptable, nonpreemptable, and must not access kernel state. Optionally, an ISR can trigger the delayed execution of a *delayed service routine* (*DSR*). DSRs run under the control of the kernel; they may access kernel state and also invoke (nonblocking) kernel services. DSRs are nonpreemptable, but interruptable by ISRs. Technically, they are implemented as a facility of the thread scheduler, which dispatches all pending DSRs before the control flow returns to thread level. Hence, DSRs always have priority over threads.

Furthermore, the kernel implements several *central kernel policies*, among them *tracing*, *kernel instrumentation*, and *interrupt synchronization*:

Tracing. To observe the control flow through the system, entrances to and exits from system functions are recorded. Furthermore, it is possible to track values of function arguments, local variables and function results. Tracing is an *optional* and *configurable* kernel policy; it can be disabled completely and the amount of context provided in the output can be restricted.

(Kernel) Instrumentation. For monitoring and optimization purposes, the kernel employs means to log occurrences of various events, such as thread creation or mutex locking. Instrumentation is an *optional* and *configurable* kernel policy; it can individually be enabled for several classes of kernel events.

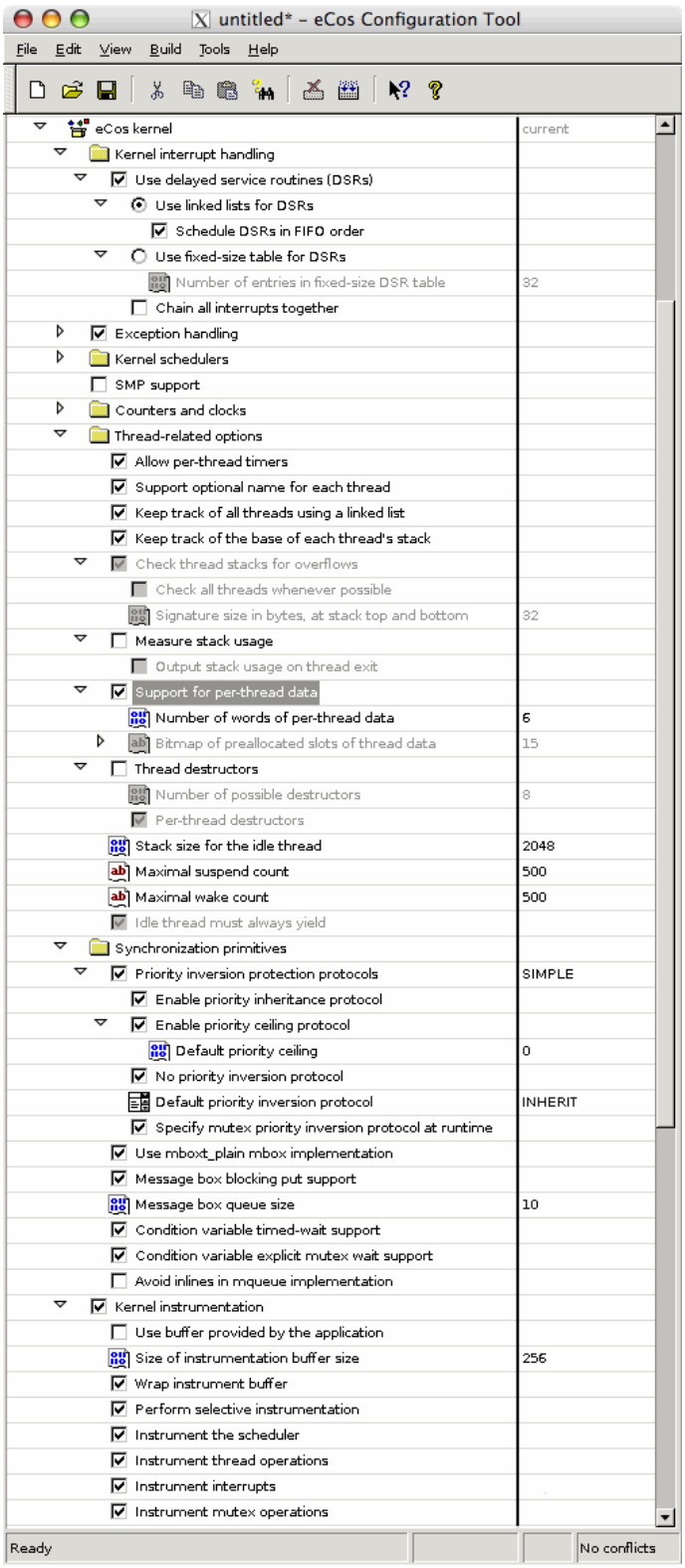


Figure 3.1.: Representation of Features in the ECOSCONFIG configuration tool

(Kernel) Synchronization. In order to guarantee the consistency of operating system data structures, most kernel functions must run mutually exclusive to DSRs. Synchronization is a *mandatory* and *nonconfigurable* kernel policy.

3.1.1.3. Analysis of Configuration Options and Kernel Policies

As already mentioned, eCos offers a high number of configuration options (features) that may or may not be selected in ECOSCONFIG for the configuration of a concrete eCos variant. In the code, these options are represented as preprocessor macros and enforced by means of conditional compilation. Table 3.1 lists a selection of the available options (thread and mutex configuration options) with their respective preprocessor macros.

Most configuration options affect multiple code locations. This holds true for all of the 12 analyzed options from Table 3.1, for which a total of 73 `#ifdef` blocks can be found in the eCos source base: 39 for *thread options* and 34 for *mutex options*. In average, each configuration option is spread over 6 places in the source code.

Furthermore, this spreading is not local to the scope of single classes, functions, or even files. Figure 3.2 visualizes the distribution of the mutex configuration options over the kernel source base. The implementation is scattered over the implementation of other concerns, in this case the scheduler.

Even more problematic is how these configuration options actually become manifest in the code. A typical example is the implementation of the `Cyg_Mutex` constructor. The constructor implementation without any of the optional features would look as follows:

```
Cyg_Mutex::Cyg_Mutex() {
  locked   = false;
  owner    = NULL;
}
```

However, because of the decompositional nature of the preprocessor approach, the actual constructor code had to be "enriched" by the implementation of the optional mutex configuration options. Compare the easy-to-understand four lines of code with the real implementation in Listing 3.1.a – the differences are dramatical. Besides all those `#ifdef` blocks for the mutex configuration options, we can also find two lines of code related to the configurable Tracing policy.

Optional kernel policies, namely Tracing and Instrumentation, are enforced by invoking special preprocessor macros, such as `CYG_REPORT_FUNCTION()` and `CYG_REPORT_RETURN()`, which have to be added by the developers to the implementation of each function. Depending on the actual configuration, these macros may be defined as empty statements. The Synchronization policy, which is not configurable in eCos, is enforced by explicit acquisition and releasing of a global kernel lock using the functions `Cyg_Schedulder::lock()` and `Cyg_Scheduler::unlock()`.

The `Cyg_Mutex` constructor implementation is special in so far as it is affected by only one of the central kernel policies. The implementation of the `Cyg_Mutex::unlock()` function in

	configuration option	#ifdef blocks
thread options (8)	CYGVAR_KERNEL_THREADS_NAME	*3*
	CYGVAR_KERNEL_THREADS_LIST	*4*
	CYGVAR_KERNEL_THREADS_STACK_LIMIT	*7*
	CYGVAR_KERNEL_THREADS_STACK_CHECKING	*6*
	CYGVAR_KERNEL_THREADS_STACK_MEASUREMENT	*2*
	CYGVAR_KERNEL_THREADS_DATA	*3*
	CYGVAR_KERNEL_THREADS_DESTRUCTORS	*3*
	CYGVAR_KERNEL_THREADS_DESTRUCTORS_PER_THREAD	*11*
	thread configuration options total	*39*
mutex options (4)	CYGSEM_KERNEL_ SYNC_MUTEX_PROTOCOL	*14*
	CYGSEM_KERNEL_ SYNC_MUTEX_PROTOCOL_INHERIT	*5*
	CYGSEM_KERNEL_ SYNC_MUTEX_PROTOCOL_CEILING	*10*
	CYGSEM_KERNEL_ SYNC_MUTEX_PROTOCOL_DYNAMIC	*5*
	mutex configuration options total	*34*

Table 3.1.: Source code statistics for configuration option enforcement in eCos.
Listed is the number of `#ifdef` blocks in the C++ parts of the kernel code base (`.cxx`, `.hxx`, `.inl` files).

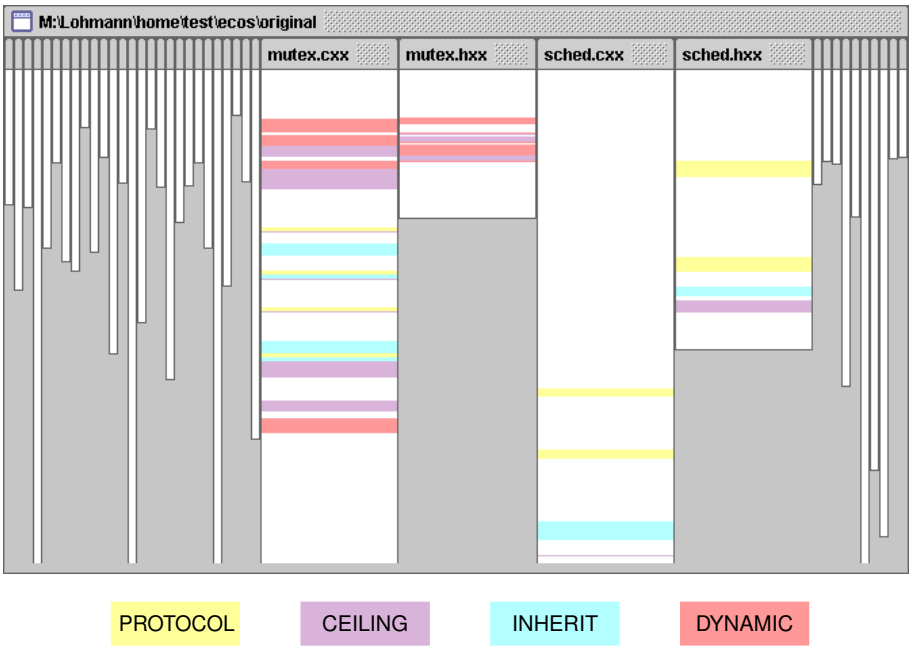


Figure 3.2.: Distribution of mutex configuration options in the eCos kernel source base.
Each bar represents a single C++ file (`.hxx`, `.cxx`, `.inl`) from the kernel source base. The height of a bar represents the relative size of the file in lines of code; the colored areas represent lines of code taken by `#ifdef` blocks for the respective configuration option. The slender bars represent source files which do not contain code related to the analyzed configuration options.
[Bar graph created with *AspectBrowser for Eclipse* [Gri].]

	lines of code	
policy	**#**	**%**
Tracing	*336*	*6.5*
Instrumentation	*162*	*3.1*
Synchronization	*187*	*3.6*
policy total	*685*	*13.2*
nonpolicy total	*4520*	*86.8*
kernel total	*5205*	*100*

Table 3.2.: Source code statistics for kernel policy enforcement in eCos.
Listed is the number and percentage of policy-related macro and function invocations in the C++ parts of the kernel code base (`.cxx`, `.hxx`, `.inl` files).
[Lines of code counted with CCCC [Lit].]

Listing 3.1.b demonstrates that the situation can get even worse if a function is affected by multiple configuration options *and* multiple kernel policies. Eight concerns (the plain `Cyg_Mutex` implementation counts as one concern as well) are tangled with each other in only 31 lines of code. The eCos developer employed even additional preprocessor macros here to prevent the code from becoming more complicated due to further `#ifdef` nesting.

Whereas the mutex configuration options still affect *relatively* few functions and files, the enforcement of the central kernel policies is distributed over the whole kernel source base. Many functions in the kernel source base look like the functions `Cyg_Counting_Semaphore::post()` (shown in Listing 3.1.c) or `Cyg_Thread::sleep()` (shown in Listing 3.1.d). The global distribution of the central kernel policies is visualized in Figure 3.3. Together, the three analyzed policies account for more than 680 lines (more than 13 percent) of the kernel source, spread over 23 source files. Table 3.2 provides a detailed breakdown of these numbers.

3.1.1.4. Discussion of Results

We diagnose that in eCos the implementation of configuration options and central kernel policies is heavily scattered over the implementation of other kernel concerns. The source code is bloated and hardly understandable; the examples from Listing 3.1 speak for themselves. Especially the effects of implementing fine-grained configuration options by means of conditional compilation seem to be disastrous for the source code quality. A developer who has to maintain this code, or – heaven forbid! – even add a *new* configuration option, literally finds herself in "`#ifdef` hell".

So why did the eCos developers opt for this mechanism? We can only speculate on their reasons. In my experience, however, embedded systems engineers frequently mention two major reasons:

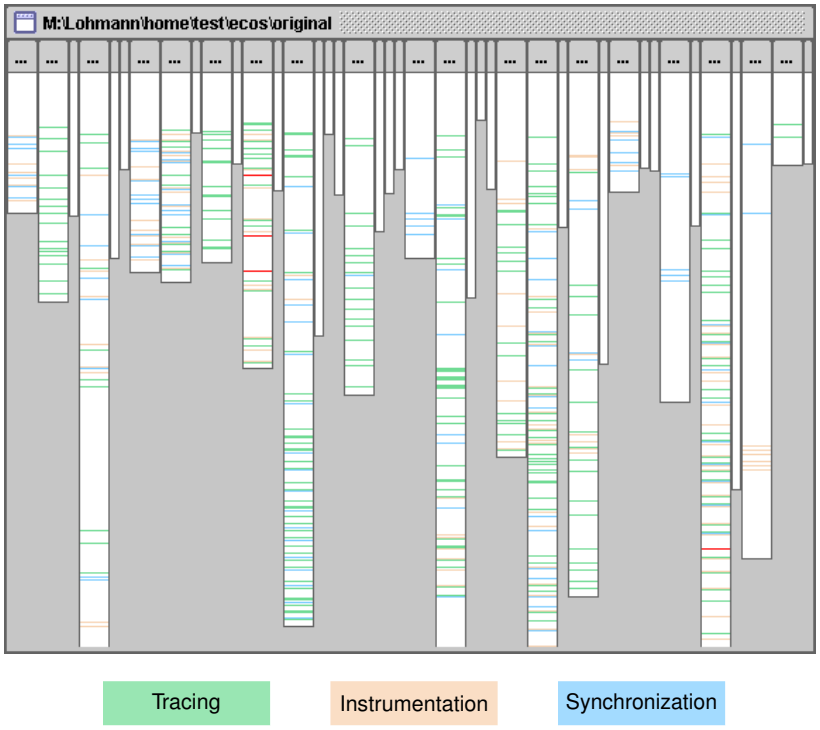


Figure 3.3.: Distribution of policy enforcement in the eCos kernel source base.
Each bar represents a single C++ file (`.hxx`, `.cxx`, `.inl`) from the kernel source base. The height of a bar represents the relative size of the file in lines of code; lines that account to the enforcement of some kernel policy (i.e., contain a Tracing or Instrumentation macro invocation or a call to the `lock()` / `unlock()` Synchronization methods) are depicted in the respective color. Due to scaling issues the image is not a hundred percent exact: Occasionally, source lines that are close together, but contribute to different policies are mapped on the same pixmap line, which is indicated by red color.
[Bar graph created with *AspectBrowser for Eclipse* [Gri].]

Flexibility. Configuration options tend to be crosscutting in their implementation with other concerns on a very fine level of granularity. All of the analyzed thread and mutex configuration options affect just one or two statements in each function; however, are distributed over several functions, classes, and files (Table 3.1). Such features are often not decomposable with the abstractions offered by the programming language. With the preprocessor, developers can escape the expressiveness limits of the language by going down to the level of text processing. The resulting solutions are not nice – but nevertheless effective solutions.

Efficiency. Especially in the domain of embedded systems, the approach to enforce configurability itself must not induce an extra overhead with respect to hardware resources. Conditional compilation is inherently free of any overhead. So even if the decomposition of some feature into, for example, classes, functions, and object-oriented interfaces is possible, this might not be adequate because of the additional overhead induced by these extra indirections.

(a)

```
Cyg_Mutex::Cyg_Mutex() {
  CYG_REPORT_FUNCTION();
  locked      = false;
  owner       = NULL;
#if defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT) && \
  defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DYNAMIC)
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_INHERIT
  protocol    = INHERIT;
#endif
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_CEILING
  protocol    = CEILING;
  ceiling     = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
#endif
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_NONE
  protocol    = NONE;
#endif
#else // not (DYNAMIC and DEFAULT defined)
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY
  // if there is a default priority ceiling defined, use that to initialize
  // the ceiling.
  ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
#else
  // Otherwise set it to zero.
  ceiling = 0;
#endif
#endif
#endif // DYNAMIC and DEFAULT defined
  CYG_REPORT_RETURN();
}
```

(b)

```
void Cyg_Mutex::unlock(void) {
    CYG_REPORT_FUNCTION();
    Cyg_Scheduler::lock();
    CYG_INSTRUMENT_MUTEX(UNLOCK, this, 0);
    if( !queue.empty() ) {
        Cyg_Thread *thread = queue.dequeue();
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_INHERIT
        IF_PROTOCOL_INHERIT
            thread->relay_priority(owner, &queue);
#endif
        thread->set_wake_reason( Cyg_Thread::DONE );
        thread->wake();
        CYG_INSTRUMENT_MUTEX(WAKE, this, thread);
    }
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL
    IF_PROTOCOL_ACTIVE
        owner->uncount_mutex();
#endif
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_INHERIT
    IF_PROTOCOL_INHERIT
        owner->disinherit_priority();
#endif
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING
    IF_PROTOCOL_CEILING
        owner->clear_priority_ceiling();
#endif
    locked      = false;
    owner       = NULL;
    Cyg_Scheduler::unlock();
    CYG_REPORT_RETURN();
}
```

Mutex configuration options **(c)**

PROTOCOL

CEILING

INHERIT

DYNAMIC

```
1  void Cyg_Counting_Semaphore::post() {
2      // Prevent preemption
3      Cyg_Scheduler::lock();
5      CYG_INSTRUMENT_CNTSEM( POST, this, 0 );
6
7      count++;
8      if( !queue.empty() ) {
9          // The queue is nonempty, so grab the next
10         // thread from it and wake it up. The waiter
11         // will decrement the count when he is awakened.
12         Cyg_Thread *thread = queue.dequeue();
13         thread->set_wake_reason( Cyg_Thread::DONE );
14         thread->wake();
15         CYG_INSTRUMENT_CNTSEM( WAKE, this, thread );
16     }
18     // Unlock the scheduler and maybe switch threads
19     Cyg_Scheduler::unlock();
20 }
```

eCos kernel policies **(d)**

Tracing

Synchronization

Instrumentation

```
1  void Cyg_Thread::sleep() {
2    CYG_REPORT_FUNCTION();
3    Cyg_Thread *current = Cyg_Scheduler::get_current_thread();
4    CYG_INSTRUMENT_THREAD(SLEEP,current,0);
5    // Prevent preemption
6    Cyg_Scheduler::lock();
7    // If running, remove from run qs
8    if ( current->state == RUNNING )
9      Cyg_Scheduler::scheduler.rem_thread(current);
10   // Set the state
11   current->state |= SLEEPING;
12   // Unlock the scheduler and switch threads
13   Cyg_Scheduler::unlock();
14   CYG_REPORT_RETURN();
15 }
```

Listing 3.1 (both pages): Source code examples from eCos.
Depicted are four functions from the kernel and their pollution by optional mutex configuration options (PROTOCOL, CEILING, INHERIT, DYNAMIC) and kernel policy enforcement (Tracing, Synchronization, Instrumentation). **(a)** The default constructor of the `Cyg_Mutex` class is heavily tangled with the code of optional features and the enforcement of the Tracing kernel policy; only four of the 30 lines of code are taken by the plain `Cyg_Mutex::CygMutex()` implementation. Note that the code is even more complicated than is indicated by the coloring. Technically, some lines have to be accounted to more than one configuration option because of `#ifdef` nesting – which is, however, not displayable with the applied coloring scheme. **(b)** Similar situation in the implementation of `Cyg_Mutex::unlock()`, which additionally contains code for the enforcement of the kernel policies Synchronization and Instrumentation; only nine out of 31 lines of code belong to the plain mutex implementation; the eCos developers prevented nesting of `#ifdef` blocks in this function by extra `IF_PROTOCOL_...` macros, which expand to specific `if` statement if the feature DYNAMIC is chosen. **(c, d)** Examples for the enforcement of kernel policies in eCos. Depicted are the implementations of `Cyg_Counting_Semaphore::post()` and `Cyg_Thread::sleep()`, which are tangled with the implementation of the Tracing, Instrumentation, and Synchronization kernel policies.

So we have *qualitative* (flexibility, expressiveness) and *quantitative* (efficiency) reasons that generally speak for preprocessor-based enforcement of configuration options. In this cost-sensitive domain, quantitative arguments even dominate qualitative arguments; preprocessor-based configuration is even applied if some other solution were possible, but less run-time or memory efficient – despite all the problems this causes.

We need a better way to implement configurability!

3.1.2. Going Ahead – The Case for Configurable Architecture

The issues identified in the preceding section are caused by the aim of the eCos designers to provide a highly configurable operating system. Yet it is remarkable what remains *not* configurable. Tracing and Instrumentation are implemented as configurable kernel policies, whereas the enforcement of Synchronization is hard coded and mandatory.

On first sight this seems to be a sensible and plausible design decision. Tracing and Instrumentation are "truly optional" features. Tracing is related to development and debugging activities, and rarely needed in production systems; Instrumentation is only occasionally useful. Synchronization, in contrast, is a "most fundamental responsibility" of any operating system kernel [LE93, SGG05, Tan07, Sta08] – a feature that is always needed.

However, as PARNAS already stated:

> *I know of no feature that is always needed. When we say that two functions are almost always used together, we should remember that "almost" is a euphemism for "not".* [Par79]

In fact, there are good cases for understanding Synchronization as an optional (and even configurable) feature. In eCos, it is employed to ensure consistency of kernel state that is accessed from thread level as well as from interrupts (DSRs). However, in the embedded-systems domain we have application cases in which only interrupts but no threads are used – or vice versa. In these cases, Synchronization would not be required – a clear case for nonoptimal granularity.

Variability with respect to Synchronization can be important as well. An impressive example for this lesson was the extension of the Linux kernel with support for symmetric multi processing (SMP). Originally, Linux implemented a simple synchronization strategy similar to eCos;[2] consistency of kernel state was ensured by what later became known as the "big kernel lock" [BC01]. The first kernel release that supported SMP hardware was version 2.0. However, because of the coarse-grained kernel synchronization scheme it performed badly in SMP environments. To improve the performance, a switch towards a fine-grained synchronization strategy was unavoidable. Hundreds of device drivers, file systems, and other kernel components had to be adapted – a process that took three kernel releases and several years to complete. The current 2.6 kernel series now applies fine-grained

[2] The `runrun` variable in "Lions' Commentary on UNIX 6th Edition with Source Code" from 1976 is probably the earliest documented reference of this synchronization strategy.

locking in all parts of the system if running on SMP hardware and corse-grained locking if running on a single processor machine. Synchronization has become a configurable policy.

3.1.2.1. Nonfunctional Properties

In both examples, eCos and Linux, the nonoptimal synchronization strategy has no direct influence on the *functionality* or correctness of the operating system. All applications worked well on kernel-2.0 SMP machines – only that the *perfomance* could have been better. An application that does not employ threads works perfectly on eCos – only that the *event latency* is higher than necessary. How a kernel implements Synchronization is completely transparent to the application – but can have quite an effect on *nonfunctional properties*:

> **Nonfunctional properties** of a software system are those properties that do not describe the principal task or functionality of the software, but can be observed by end users in its run-time behavior.

Performance and *latency* are just two examples for nonfunctional properties. Other important examples in the domain of embedded operating systems are *memory requirements* and *robustness;* this list is open and also depends on the viewpoint of the stakeholder.[3] Independently from this viewpoint, we can, however, say:

1. Nonfunctional properties are **emergent** properties. They are neither visible in the code nor in the structure of certain components, but emerge from the orchestration of system components and applications into a complete system.
   (*Performance* and *latency* are obviously not only determined by the chosen synchronization strategy; they are the result of many design decisions as well as the specific characteristics of the application.)
2. Nonfunctional properties can be **mission-critical**.
   (A system that provides perfect functionality but works terribly slow is not accepted by users; a software update for some embedded device that would provide great new functions, but does not fit in the available memory is just unusable.)

In Section 2.2.3, I introduced the concept of a *feature* in a software product line with the definition from CZARNECKI and EISENECKER, who define a feature as [CE00, p. 38] *"a distinguishable characteristic of a concept [...] that is relevant to some stakeholder of the concept."* Many nonfunctional properties would match this definition. Nevertheless, we cannot provide them as features in the problem space of an operating-system product line – it is not possible to find an extensional representation for them in the solution space. Emergent properties are inherently indecomposable. The consequence is that they can only be influenced *indirectly*.

[3] We have discussed some more hardware-related nonfunctional properties in Section 2.1.1.

3.1.2.2. The Role of Architecture

If nonfunctional properties emerge from the orchestration of system components and application characteristics, we can theoretically influence them indirectly via the feature selection. Theoretically, as the feature selection is usually determined by the concrete functional requirements. If the application requires preemptive multithreading, we cannot configure our operating system for nonpreemptable threads to improve the memory footprint and context switch performance.

To influence nonfunctional properties, we need features that are like Synchronization. That is, we are looking for features that:

1. are *transparent* to the application and
2. have a *significant influence* on certain nonfunctional properties.

These two conditions are met by many of the fundamental policies that define how an operating-system kernel is *internally* structured and organized. The set of these **architectural policies** constitutes what is generally called the **architecture** of an operating system [LE93]. Synchronization can be considered as an architectural policy. Other important policies are the applied mechanisms for Isolation and Interaction. The following list describes these policies with some of their variants and the nonfunctional properties they are expected to have an effect on:

Synchronization. If the kernel supports concurrent/parallel execution of control flows, concurrent data access must not lead to race conditions. Synchronized access to data may be implemented by blocking, nonblocking, or wait-free algorithms. The implementation may be based on special hardware support such as atomic CPU operations, transactional memory, or software algorithms. Locks may be allocated on a coarse-grained or fine-grained base. The chosen kernel synchronization strategy can have a noticeable influence on *latency* and *performance*.

Isolation. The different components of an operating system may have access to the whole system state or to well-isolated subsets only. Components may be isolated by design through type-safe programming languages, by hardware support (segmentation or address spaces via memory-protection units (MPUs), memory-management units (MMUs), or translation lookaside buffers (TLBs)) – or even by distributing them across hardware boundaries. Isolation may cause additional requirements on data alignment, sharing and interaction. The chosen isolation strategy can have a significant effect on *memory requirements*, *robustness*, and *performance*.

Interaction. System services may be invoked and interact with each other by plain procedure calls, inter-process procedure calls (LPCs), remote procedure calls (RPCs), or a generic message passing mechanism. Interaction may imply implicit synchronization, data duplication or (in the case of RPCs) even fail on occasion. The chosen interaction strategy often goes in line with Isolation. The chosen interaction strategy can have a significant influence on *latency*, *memory requirements*, *performance*, and *robustness*.

Common architectures and philosophies for the design of operating systems, such as *microkernel* or *monolith*, can essentially be broken down to these three policies. Typical for a *microkernel-based system* is the fine-grained isolation of all system components into their own address spaces, interaction between these components via messages, and implicit synchronization due to the sequential processing of messages; a *monolith* instead applies a single address space for all system components, which interact with each other via plain procedure calls, and have to be synchronized explicitly via some synchronization mechanism; characteristical for a *reflexive kernel* is that interaction between system components is routed over an extra mediator; and so on. From the functional perspective of the application, however, all this makes no difference [LN79]. Architecture is an all-embracing, transparent policy.

What we can say, however, is that architectural policies generally reflect a *trade-off* with respect to different nonfunctional properties. It is, for instance, commonly accepted that the isolation of every system component into its own address space (as in microkernel operating systems) improves robustness – at the price of higher memory requirements and reduced performance. Even though these effects may be more or less distinctive [HHL+97] – there always is a trade-off. If the penalties are significant, acceptable, or observable at all depends solely on the application and its particular workload.

Hence, there is no such thing like "*the* best policy" for all cases. Even the fine-grained implementation of Synchronization that Linux now applies on SMP systems may not be optimal in all cases. The additional locks reduce the probability of lock contention at the price of a (small) memory and run-time overhead. This leads to great performance benefits on massive-parallel workloads that lead to many input–output operations, which otherwise would suffer badly on lock contention. However, lock contention would not be a problem with a CPU-intensive application that employs only one thread per processor and performs only very few input–output operations. In such cases a coarse-grained locking scheme would perform better – even on an SMP system.

Architectural policies should become configurable features!

3.1.2.3. The AUTOSAR Case

Recent developments in the domain of automotive system software present us another motivating case for the understanding of architectural policies as configurable features. The automotive industry is in the beginning of a transition phase from the OSEK operating system standard (OSEK OS) to its successor AUTOSAR OS. The AUTOSAR OS standard prescribes features that essentially require the configurability of Isolation and Interaction.

The OSEK consortium[4] was founded in 1993 by the German automotive industry to develop what later became the leading standard for system software in automotive applications [OSE]. Central element is the OSEK OS standard [OSE05], a detailed

[4] OSEK is an abbrevation for "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug", which can be translated as "open systems and their interfaces for automotive electronics".

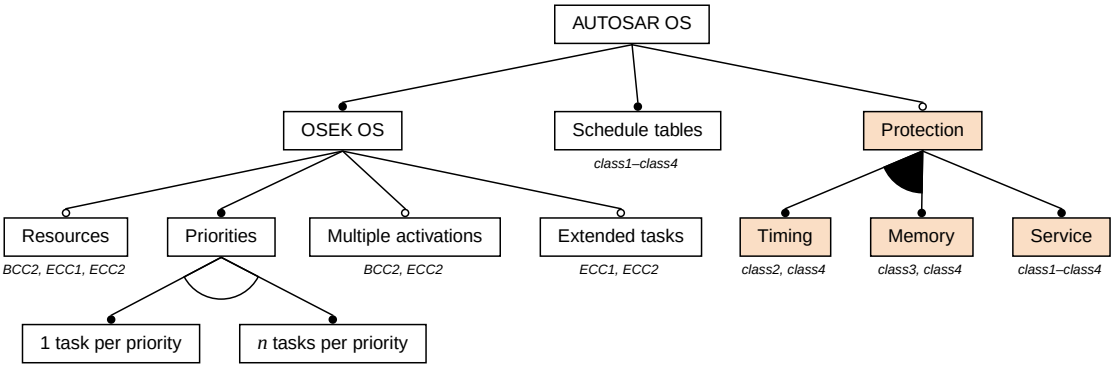


Figure 3.4.: AUTOSAR OS scalability classes and OSEK OS conformance classes.
The feature diagram depicts the variability defined by the AUTOSAR OS scalability classes, respective OSEK OS conformance classes. OSEK OS (left branch) predefines variability and granularity by two *basic conformance classes* (*BCC1* and *BCC2*, only support for basic tasks) and two *extended conformance classes* (*ECC1 and ECC2,* support for basic and extended (=preemptable) tasks) [OSE05, p.14]. AUTOSAR OS extends on the variability of OSEK OS by additional *scalability classes* [AUT06a, p. 21]; *class1* is the basic configuration, consisting of OSEK OS + Schedule tables + Service protection; scalability classes 2–4 describe extension stages regarding the protection policies Timing protection and Memory protection.

specification of an event-triggered real-time operating system. OSEK OS provides a standardized API with abstractions for control flows (*tasks*), synchronization (*events*, *resources*), interrupt handling (*ISRs*), and so on. OSEK OS aims at best-possible resource frugalness and is inteded for small 8-bit to 16-bit microcontrollers. For the sake of granularity, the standard requires an OSEK OS implementation to support four different extension stages of the operating system called *conformance classes*, which differ from each other in the support of Extended tasks (preemptable tasks), Resources, Multiple activations (of tasks), and different priority models. These conformance classes can be understood as different configurations of an operating-system product line (Figure 3.4, left branch).

The great success of OSEK triggered the foundation of AUTOSAR[5], a new and international consortium of car manufacturers that aims at the development of a successor of the OSEK standards [aut]. Central element is again the operating system specification, AUTOSAR OS [AUT06a, AUT06b], which extends OSEK OS by new features and abstractions. Besides support for time-triggered task activations (Schedule tables), AUTOSAR OS specifically adds new means for the isolation of faults, which are subsummized under the term *protection facilities* [AUT06a, pp. 13ff]. Besides parameter and context checking for operating-system services (Service protection), this particularly includes measures for the temporal and spacial isolation of tasks and ISRs (Timing protection and Memory protection).

The motivation behind this is the idea of *microcontroller consolidation*; software from different vendors, which so far has always been running on dedicated 8-bit and 16-bit hardware, should be integrated on fewer, but more powerful 32-bit microcontrollers. With respect to robustness, safety, and – ultimately – liability, this requires means for strong

[5] AUTOSAR stands for AUTomotive Open System ARchitecture

isolation in the hardware and the operating system. At the same time, however, AUTOSAR OS should remain usable for small stand-alone systems. To reflect this extra variability and granularity, AUTOSAR OS extends OSEK OS by four additional extension stages called *scalability classes*, which differ from each other in the support of Timing protection and Memory protection (Figure 3.4).

Essentially, the AUTOSAR-OS specification thereby suggests configurable architectural policies as a means for the configurability of the trade-off between the nonfunctional properties *memory requirements*, *performance*, *latency,* and *robustness*.

3.1.3. Problem Summary

State of the art to implement configurability of fine-grained features and kernel policies in operating-system product lines for embedded systems is the C preprocessor. However, this approach does not scale up:

Problem 1: The implementation of fine-grained configuration options by means of conditional compilation results in commingling of concerns; the scattered `#ifdef` cascades make the code hard to read, understand, and maintain. The situation gets worse the more configuration options are provided.

The "`#ifdef` hell" in the source code of eCos is an impressive example here – it clearly demonstrates how implementing configurability solely by means of conditional compilation hits its limits. At the same time, however, even eCos is yet not configurable enough:

Problem 2: For the sake of variability regarding nonfunctional properties, architectural policies should be provided as configurable features.

The developments around AUTOSAR OS show us that – besides all fundamental advantages – there already is concrete need for configurability of architectural kernel policies in embedded operating systems. This, however, is challenging to implement. Architectural policies have to be reflected in basically every part of a kernel implementation; they crosscut with the implementation of all other kernel concerns. The distribution of Synchronization in the eCos source base is a good example – and yet it reflects just *one* variant of this policy. Other variants, such as fine-grained or waiting-free synchronization, would probably affect other parts of the kernel.

The goal of a configurable architecture makes it neccessary to push the current limits for the implementation of configurability in embedded system-software product lines.

3.2. Is AOP the Solution?

Both problems are essentially problems of crosscutting concern implementations. AOP provides extra language support for these kind of problems; hence, AOP is a promising candidate for a solution that pushes the current limits of configurability:

1. Fine-grained configuration options typically crosscut *inhomogeniously* with the implementation of other concerns in the structural and behavioral space. A clear separation of the *what* and the *where* of a concern's implementation should make it possible to implement them using aspects instead of conditional compilation – and thereby reach separation of concerns.
2. Architectural policies typically crosscut *homogeniously* with the implementation of many other concerns in the behavioral space. The flexible match mechanism and the quantification principle should make it possible to separate them into aspects – and thereby reach their configurability.

With respect to the first point we have, however, to be perfectly aware of the reasons *why* conditional compilation is the state of the art for the implementation of configurability: *flexibility* and *efficiency*. It is yet open if there is an aspect language that is expressive and efficient enough for the implementation of fine-grained configuration options in such a cost-sensitive domain.

With respect to the second point there are many open issues. Architectural policies do not only become manifest in a crosscutting implementation, but tend to subtly influence the whole development process of operating-system components. They lead to implicit assumptions about side effects, execution contexts, and more. It is totally open if – with or without AOP – operating-system components can be implemented in an *architecture-transparent* manner.

Hence, I have to answer questions regarding the *qualitative* and *quantitative* dimension of implementing configurability by AOP:

> The **qualitative dimension** of configurability describes what can be made configurable at all – theoretically (with respect to the expressiveness of the underlying implementation mechanism) and practically (with respect to development efforts).
>
> The **quantitative dimension** of configurability describes the cost of the configuration mechanism with respect to hardware resources.

The goal of **aspect-awareness in the development of configurable system software** is to show to what extend a well-directed, broad-scale application of AOP does improve on the state of the art to implement configurability in embedded operating systems (qualitative dimension) without disadvantages on the hardware cost side (quantitative dimension).

3.3. Suggested Approach

Analyze the quantitative and qualitative effects of a broad-scale application of AOP for the implementation of configurability in operating-system product lines for resource-constrained embedded systems. Evaluate thereby, if AOP is suitable as a first-class

mechanism for the implementation of configurability in the domain of embedded system software.

Handle the topic *bottom-up* on three different levels: *language*, *implementation* (of configurability), and *design* (for configurability). We require an overall benefit with respect to the qualitative and quantitative properties on each of these levels, as higher levels depend on the success in the lower levels.

3.3.1. Language Level

In order not to run into similar problems as with OOP, it is necessary to think about the cost effects of AOP *before* applying it to the domain of embedded system software. First and foremost, this has to be reflected in the design of the AOP language itself. An AOP language for this domain has to provide expressiveness *and* run-time cost efficiency. Ideally, using AOP language concepts to separate otherwise tangled and scattered concern implementations does not lead to an overhead.

Questions: Which AOP language features induce an overhead with respect to CPU and memory resources? Can we improve on that? Is the expressive power of a general-purpose, feature-rich AOP language an affordable luxury for the domain of resource-constrained embedded systems?

Objectives: Show that by a careful design of the aspect language, highly expressive, yet cost-neutral AOP is possible. Figure out which AOP features induce what overhead and if and how this can be avoided.

3.3.2. Implementation Level

State of the art to *implement* configurability is the C preprocessor – which is flexible and efficient, but does not at all provide separation of concerns. Ideally, AOP provides better means for the separation of concerns without disadvantages regarding flexibility and efficiency.

Questions: Can we achieve the same or better flexibility and efficiency as with the existing approaches? What are the idioms for implementing configurability by aspects?

Objectives: Show that AOP compares qualitatively *and* quantitatively very well to other approaches for implementing configurability in software product lines for embedded systems. Figure out the idioms and patterns to achieve configurability by aspects in such product lines.

3.3.3. Design Level

Compared to procedural or object-oriented programming, AOP offers new and very different means for encapsulation and separation of concerns into software modules.

Being a relatively young paradigm, there is still only few experience on *how* and *when* to use this extra expressive power. So far, attempts to apply AOP to the domain of operating systems have always been based on existing operating system kernels. This led to a perception of AOP as a "better patch technology" to bring configurability or extensions into obstructed subsystems. The benefits would possibly be much higher if aspects were used as a *primary design concept* from the very beginning.

To evaluate and assess the chances offered by AOP towards a higher quality of configurability in system software, it is therefore necessary to think about configurability by aspects from the very beginning. The goal is an *aspect-aware design* that takes real advantage of AOP by achieving configurability of even architectural policies.

Questions: What are good design rules for *aspect-aware* kernel design? What are the benefits? Is it thereby possible to implement even architectural system policies as configurable features?

Objectives: Show that AOP can lead to the better configurability of policies in an operating-system kernel. Show that configurability is even possible for architectural operating-system policies. Figure out methods and rules to achieve *aspect-awareness* when designing an operating-system kernel.

3.4. Chapter Summary

The goal of this chapter was to analyze the problems of *implementing* configurability in system-software product lines for embedded applications. The problem is twofold.

Firstly, conditional compilation – the state of the art for the implementation of fine-grained configurability – leads to commingling instead of separation of concerns. This does not scale, takes us to "`#ifdef` hell", and has already hit its limits. We need a better approach.

Secondly, configurable operating systems are yet not configurable enough – they do not provide configurability of architectural policies. Even though these policies are transparent to the application, they influence important nonfunctional properties and, thus, should be understood as configurable features. This makes it necessary to push the limits for the implementation of configurability in embedded system software even further.

Both problems are essentially problems of crosscutting, so aspect-oriented programming is a promising candidate for a remedy. It is, however, still open if AOP can compete qualitatively – but especially quantitatively – with the existing approaches. It is furthermore open if it thereby becomes really possible to implement architectural policies as configurable features.

To answer these questions, I suggest to analyze and evaluate the suitability of AOP for the implementation of configurability *bottom-up* on three levels of increasing abstraction: *language level*, *implementation level*, and *design level*.

The remaining parts of my thesis reflect this bottom-up approach. We start in the next chapter, Chapter 4, on the language level.

4

Language Level – Aspects Demystified: Evaluation and Evolution of AspectC++

The expressive power of a programming language has a significant influence on the productivity of developers and the quality of the resulting software. This has already been stated by DIJKSTRA in the early 1970s:

> *"[...] the language or notation we are using to express or record our thoughts are the major factors determining what we can think or express at all!"* [Dij72]

High-level languages that offer more and better abstractions help to concentrate on the actual problem to solve [McC04, p. 62ff]. System developers, however, have always taken an ambivalent stance on high-level programming paradigms. Even though they embrace extra expressive power, system developers generally are concerned about the cost of such luxury: memory footprint, performance impacts, and a general loss of control over the resulting code. An in-depth analysis of such effects is crucial to assess the trade-off between cost and benefits of a new approach such as AOP.

In this chapter I provide a look "under the hood" of AOP in general and AspectC++ in particular. This "demystification" – basically a technical operationalization of the terms *obliviousness* and *quantification* – is an important contribution to understand the impact of the *language level* to the qualitative and quantitative properties of configurability. In the context of this thesis it also paved the path to several improvements of AspectC++.

The chapter opens in Section 4.1 with a general discussion on aspect languages for our domain: basic requirements, costs to expect, and criteria for choosing an aspect language. For AspectC++ these requirements triggered the development of the *generic*

advice concept, which I introduce in Section 4.2. This is followed in Section 4.3 by an in-depth *cost analysis* of all AspectC++ language constructs, which also led to significant code generation improvements. All language level results are then discussed and related to the dimensions of configurability in Section 4.4. Finally, I give an overview on further related work in Section 4.5 and briefly summarize the chapter in Section 4.6.

Related Publications

The ideas and results presented in this chapter have partly also been published as:

[LBS04] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 55–74. Springer-Verlag, October 2004.

[LS05b] Daniel Lohmann and Olaf Spinczyk. On typesafe aspect implementations in C++. In F. Geschwind, U. Assmann, and O. Nierstrasz, editors, *Proceedings of Software Composition 2005 (SC '05)*, volume 3628 of *Lecture Notes in Computer Science*, pages 135–149, Edinburgh, UK, April 2005. Springer-Verlag.

[SL07] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

4.1. General Considerations

The goal of this thesis is to analyze the applicability of AOP towards an increased configurability of features in system software product lines for embedded devices. In Chapter 2 we learned that our target domain entails some specific constraints – constraints that should be taken into consideration when thinking about aspect languages and tools for this domain.

4.1.1. AOP Fundamentals: A Recap

The major goal of AOP is to separate the *what* from the *where* of concern implementations (Section 2.3.4). This is motivated by the problem of *crosscutting*, which can be observed in the fact that the implementations of many concerns (*what*) are not well localized, but spread over (*where*) the implementations of other concerns (Section 2.3.1).

Crosscutting is a two-fold phenomenon with a *space dimension* and a *distribution dimension* (Section 2.3.3). An AOP language has to provide *obliviousness* and *quantification* (Section 2.3.4) to facilitate the decoupling of an actual concern's implementation (*what*) from its application alongside these dimensions (*where*).

On the syntax level the *what* is given by *advice*; the *where* is reflected by the concept of a *pointcut*, which is a set of *join points* given as a *pointcut expression* in a *join point description language* (Section 2.3.4). The join point description language provides declarative means to reason over the space and distribution dimensions of the program.

Most AOP languages follow the philosophy, terminology and semantics of AspectJ with its explicit notion of *aspects* – special implementation entities that are superimposed into the entities (e.g., classes) of the *base program*. Like AspectJ, these languages are usually implemented as an (upward-compatible) *extension* to some existing *base language* (Section 2.3.6).

Aspects are technically applied by a process called *aspect weaving*; the corresponding tool is the *aspect weaver*. We distinguish between *static weaving* (at built-time) and *dynamic weaving* (at run-time). Because of the nonnegligible overhead of dynamic weaving, this thesis focuses on static weaving (Section 2.3.4.2).

4.1.2. Requirements on an Aspect Language for System Software

An aspect language for system software has to reflect the following general requirements:

Support for obliviousness and quantification, as the AOP-style of separation of concerns is the motivation to use AOP anyway. Hence, the language should support the typical AOP features that are required to achieve obliviousness and quantification.

Platform independence, as platform variety is a major criterion of our target domain (Section 2.1.2). This is, strictly speaking, not a language issue, but a question of tools, especially compiler back-ends.

Minimal overhead in the resulting code, as thriftiness with respect to hardware resources is another major criterion of our target domain (Section 2.1.1). The overhead, if any, must be acceptable even for small systems with only few KiB of memory.

We have already learned that, for practical reasons, most AOP languages are implemented as an extension to some existing base language. In these cases, the general requirements have to be reflected in both the base language and its AOP extension. This is obvious for the latter two (*platform independence* and *minimal overhead*). In the following we will see, however, that the base language also has a significant influence on the achievable expressiveness of its AOP extension, that is, on *obliviousness* and on *quantification*.

4.1.2.1. Support for Obliviousness

Conceptually speaking, **obliviousness** denotes the fact that concerns implemented by the base program can be oblivious of the concerns implemented by aspects – even though they overlap in the space dimension. Technically speaking, obliviousness suggests that the base code does not have to be prepared in any way to be (later) affected by aspects.

First and foremost, obliviousness requires a **declarative join point description language** that provides means for the aspect developer to specify the relevant join points by reasoning over the base program in the space dimension. Ideally, we would be able to reason with this language over the *semantics* of the base program. Technically, however, the reasoning can only be based on syntactic entities or run-time states of the base program to which we, the programmers, attribute some semantics. Hence, an intended set of join points with respect to the program's semantics (such as, "activate advice in aspect `ConnectionMonitor` when a client connects") has to be mapped by the aspect developer to an expression over concrete identifiers and run-time states of the base program (such as, "activate advice in aspect `ConnectionMonitor` when the constructor of class `ClientConnection` is invoked").

Effectively, the result is an *inversion* in the way control-flows are specified. I consider this **inversion of control-flow specifications by advice** as the fundamental mechanism behind obliviousness. Figure 4.1 illustrates it by the example of two components P and C. With function or method calls, control-flow relationships can only be established in the *direction of knowledge* (P *invokes method in* C) $\Longrightarrow$ (P *knows* C); hence, P can not be oblivious of C.[1] With advice, it becomes possible to specify control-flow relationships in the *opposite direction of knowledge* (P *invokes advice in* C) $\Longrightarrow$ (C *knows* P).[2]

[1] If using late binding, P does not have to know the concrete C – but nevertheless that there is some C.

[2] Because of quantification (see Figure 4.2.b) it is even not necessary that there is *some* P – advice-based binding is inherently loose. Precisely spoken: (P *invokes advice in* C) $\Longrightarrow$ (C *knows* P) && (P *does exist*).

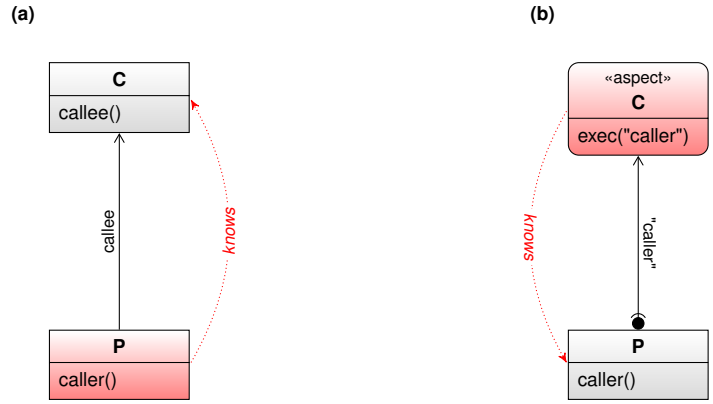


Figure 4.1.: The mechanism behind obliviousness: inversion of control-flow specifications by advice. At run time, event producer P shall invoke event consumer C. **(a)** With function calls, the control-flow relationships is in the direction of knowledge; it has to be established by *event producer* P, for which P has to *know* C. **(b)** With advice, the control-flow relationship is in the opposite direction of knowledge; it can be established by the *event consumer* C, for which C has to *know* P. (The diagram notation is further explained in Section 6.3.2 on page 129.)

The fact that join points are yielded by reasoning over the base program means that obliviousness is influenced by the base program itself. Even though the base program (in principle) does not have to be aware of aspects, it nevertheless implicitly exhibits – by its structure and nature – the set of *potential join points* that are available to the aspects.[3] The better the concerns of the base program are reflected in its syntactic entities, the stronger is the semantics of these potential join points and, thus, the expressiveness of pointcut expressions.

This is where the demand for a **syntactically rich base language** with a powerful type system comes into consideration. Programs written in a multi-paradigmatic language, such as Ada or C++, usually provide more structure – and thereby more potential join points – than those written in a simpler language, such as C. The additional syntactic concepts of "rich" languages enable the programmer to express more of the concerns and their relationships in the program structure. The above mentioned client connection concern, for instance, would usually be implemented in C++ or Java as a class `ClientConnection`, derived from a more general `Connection` class. Thereby functions, state variables and relationships related to the concern are implicitly grouped. They are visible and verifiable on the type level and pointcuts can be used to reason about this knowledge. In C, the programmer *might* use explicit naming conventions to indicate an equally strong semantic relationship between a set of functions and a set of state variables. However, such idiomatic conventions are difficult to develop, often not followed thoroughly, and, according to a recent study by BRUNTINK and colleagues, a fragile and imprecise base for pointcut expressions [BvDDT07].

[3] In AOP literature, exhibited join points are often called *join point shadows* [HH04, MKD04]; however, I prefer to denote them as potential join points.

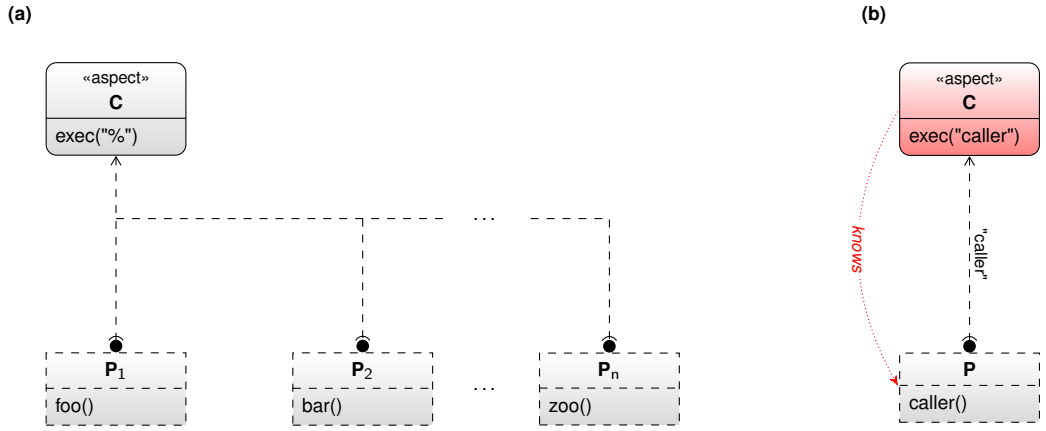


Figure 4.2.: The mechanism behind quantification: implict per–join-point instantiation of advice.
(a) The aspect weaver implicitly instantiates the advice per join point yielded by the wildcard pointcut expression over the unknown (and possibly empty) set of components $P_{1...n}$. **(b)** Loose coupling by inversion (cf. Figure 4.1). Implicit instantiation per–join-point takes place even if the pointcut expression specifies exactly one join-point; it is not an error if P is not present. (The diagram notation is further explained in Section 6.3.2 on page 129.)

4.1.2.2. Support for Quantification

Conceptually speaking, **quantification** denotes the fact that concerns implemented by aspects can be oblivious of the distribution dimension, that is, the number of join points in which they overlap with the base program. Technically speaking, quantification suggests that advice code does not have to be prepared with respect to the *number* of join points it affects.

First and foremost, quantification requires a concept for **quantors in pointcut expressions** to specify a (potentially open) set of related join points. Most join point description languages support wildcard symbols in identifier signatures for this purpose. In AspectC++, for instance, the match expression `"% ClientConnection::%(...)"` would match all member functions of the class `ClientConnection`, with `%` being an *identifier wildcard* symbol, and the triple dots `...` being a *list wildcard* symbol. This match expression might be used in a tracing aspect that logs transactions with clients on method-level. If a method is added to, changed in, or removed from class `ClientConnection`, the tracing aspect would not be affected; it always transparently quantifies over all methods of class `ClientConnection`.

Advice quantification over a set of join points can be understood as an *implicit instantiation* of the advice execution, performed by the aspect weaver for each join point. I consider this **implicit per–join-point instantiation of advice** as the fundamental mechanism behind quantification. Figure 4.2.a illustrates it by the example of an aspect C that gives advice to an (unknown) set of components $P_{1...n}$ by using a wildcard match expression. The weaver actually instantiates this piece of advice for every matching join point – which in turn depends on the actual *presence* of the components $P_{1...n}$ in the base program.

Note that implicit per–join-point instantiation of advice does even apply when we do *not* use quantors in pointcut expressions. This is depicted in Figure 4.2.b on another example for the inversion of control-flow specifications by advice. Essentially it means that advice-based control-flow specifications are always loosely coupled.

In most cases, the advice code depends in some way on the affected join point's context to implement the intended concern. From a tracing aspect, for instance, we would expect that it does not just print out "I was here", but some useful contextual information, such as the name and signature of the method and its actual parameter values. However, at the same time, the advice implementation has to be oblivious of this kind of context information – otherwise it would not be quantifiable over methods with different signatures. This means that the advice definition of our tracing aspect has to be polymorphic; it has to use, but be invariant with respect to, join-point–specific context. Support for **advice polymorphism** is crucial for quantification.

Advice polymorphism involves two requirements, one of which has to be reflected in the aspect language whereas the other affects the base language. The aspect language has to provide language means for **join-point–specific context retrieval** in advice code. This can be support for free *context variables* in the join point description language. Another common approach is to provide a **join-point API** that is implicitly available in advice definitions and can be used to request contextual information. Both approaches have their pros and cons; consequently many aspect languages implement both.

The *computations* performed by the advice code have to be invariant with respect to the join point context as well. This requires **type polymorphism in the base language**. To print out the actual parameters of method invocations in our tracing aspect, for instance, an interface to transform values of *any* type into their string representation is needed. In Java this is trivial: `toString()` is part of the polymorphic root interface `java.lang.Object`; every potential argument type of a function can be converted into its string representation via `toString()`. C, on the other hand, offers only very limited means for polymorphism – and therefore only limited support for quantification.[4] C++ is somewhat in-between as it provides compile-time type polymorphism, which is – as we will see in Section 4.2 – a technically challenging, but sufficient mechanism to implement advice polymorphism.

4.1.2.3. Support for Platform Independence

Conceptually speaking, **platform independence** of a programming language means that programs written in this language should finally be executable on any instruction set architecture (ISA). If the ISA is computationally complete, this is the case as a matter of principle. Hence, we understand platform independence as a *technical* measure; it denotes for how many ISAs the necessary tools (weaver, compiler, ...) are *actually* available.

[4] Note that the "generic" `sprintf()` would not come as a rescue. Neither is its format specifier language extensible for new types, nor does C provide any mechanism to deduce the correct format specifier from the static or dynamic type of an expression.

From the viewpoint of an aspect language designer, the goal of platform independence suggests reusing existing tools as much as possible. Most aspect weavers transform the programs written in the aspect language into a platform-independent intermediate representation for which broad tool support is already available. This transformation could either take place on the level of source code (*source-to-source weaving*) or, in the case of virtual platforms such as Java or .NET, on the level of byte code (*byte-code weaving*).

For the (nonvirtual) platforms used in the domain of embedded systems, **source-to-source weaving** to some *commonly used base language*, such as Ada, C, or C++ would ensure good platform independence.

4.1.2.4. Support for Minimal Overhead

Conceptually speaking, **minimal overhead** of a programming language means that the hardware resource requirements of any program written in this language should be "as low as possible". In the embedded-systems domain a minimal overhead is especially important with respect to RAM utilization (Section 2.1.2).

Minimal overhead is a theoretical measure that is hardly possible to assess. However, some language and compiler properties are known to be beneficial with respect to hardware resource utilization and, thus, a general suitability for the embedded systems domain. This includes the general demand for a **low base overhead** by applying a strict **pay-as-you-use** semantics with respect to high-level language constructs and run-time library support. Another such property is to do whatever can be done at compile-time instead of run-time, which suggests **static weaving and compilation** and a language that focuses on static typing.

4.1.3. The Expected Cost of Aspects

Having understood the general requirements on an AOP language for system software development, we can now elaborate on the cost one would expect for the extra expressiveness offered by AOP. In the following, I take a look on the cost of AOP from an *"ideal weaver" perspective*, that is, under the assumption of having an ideal static weaver for a statically typed and compiled base language. The goal of this assumption is to increase our understanding on how an AOP language for our domain should be designed and how its features should be used.

4.1.3.1. Static Crosscutting

For advice given to join points in the structural space (**introductions**), we do not expect any overhead. In a statically typed and compiled language, advice-induced transformations of the static program structure have to be performed at build-time anyway. It does not matter if transformations like adding a field to some class have been performed manually by a human developer or automatically by an aspect weaver.

4.1.3.2. Dynamic Crosscutting

For advice given to join points in the behavioral space of the program (**code advice**), it depends on the type of the join points. Conceptually, join points in the behavioral space are always evaluated at run-time, as they describe events in the running control flow. Technically, however, many of them can be evaluated at weave-time, as the event occurrences are unambiguously connected to specific positions in the code. Hence, we have to distinguish (from the technical point of view) *statically evaluable pointcuts* from *not-statically evaluable pointcuts*:

Statically evaluable pointcuts. Obviously evaluable at weave-time are **execution pointcuts**, which describe the execution of some function and can be connected to the first and last statement of a function. Many aspect weavers also evaluate **call pointcuts**, which represent a call to some function on the caller's side, at weave-time.[5] Also evaluable at weave-time is any kind of scoping based on static type information, such as **within pointcuts**, which describe all join points in the lexical scope of some language entity.

Not statically evaluable pointcuts. Besides statically evaluable pointcuts there are, however, pointcuts that a static weaver cannot (always) evaluate at weave-time. A common example are **cflow pointcuts**, which describe the event of being in the control flow of some function. On the implementation side, this requires some extra effort, such as maintaining a "cflow-counter" in the running program that has to be incremented, decremented, and tested at various positions. Obviously not evaluable at weave-time is any kind of filtering that is based on the run-time type, such as **target pointcuts**, which filter call join points by the run-time type of the callee.

For advice given to *statically evaluable pointcuts* we expect no run-time overhead – an ideal aspect weaver should be able to completely inline the advice functionality into the target program. For advice given to *not-statically evaluable pointcuts* we expect some run-time overhead – even an ideal aspect weaver could not avoid this overhead in all cases.

4.1.3.3. Join-Point–Specific Context Retrieval and Advice Polymorphism

In all but trivial cases, advice needs access to some join-point–specific context to fulfill its purpose. Examples for join-point–specific context are the parameter values passed to the intercepted function, or a human-readable representation of the function's name and signature. Access to join-point–specific context might lead to additional overhead. This is obvious if the advice implementation uses context information that is specifically generated by the aspect weaver, such as the string representation of the affected function's name and signature. It is less obvious if the advice code accesses information that should be available in the current context anyway, such as the arguments passed to the

[5] This has implications on the semantics of the aspect language itself, namely that calls through function pointers are not considered as call join points.

affected function. For the sake of polymorphic advice, this information has to be provided through a generic interface – an additional level of abstraction that might induce an overhead. If advice polymorphism is based on static typing, an ideal aspect weaver should be able to optimize this in the woven code by substituting invocations of the generic interface with direct context access. By analyzing the advice code, an ideal aspect weaver would furthermore tailor the amount of provided context information to what is actually requested ("pay as you use").

4.1.4. Aspect Languages for Embedded System Development

Dozens of aspect languages have been proposed by researchers and practitioners (compare Section 2.3.6). So, after all these considerations: Which one is "best" for our domain of embedded systems and the goal to increase configurability of system-software product lines? There is, of course, no silver bullet. However, the discussion in the previous sections should have made two points evident:

1. Our major requirements on an aspect language – *obliviousness, quantification*, *platform independence*, and *minimal overhead* – are mostly determined by properties of the underlying *base language*. Hence, the base language is an ideal first-line discriminator for the suitability of an aspect language for the domain of embedded systems.
2. An ideal *static* aspect weaver could implement most AOP features without any extra overhead. Hence, tool support for static weaving is another good discriminator for the suitability of an aspect language for our domain.

In the following, I use the base language perspective to discuss the applicability of aspect languages to the domain of embedded systems, and for each base language I give a brief overview on the available AOP extensions with static weaving support.

4.1.4.1. Java-based Aspect Languages

Java as a base language offers very good support for obliviousness and quantification in AOP extensions, but only limited platform independence and not at all a minimal overhead:

- Due to its strong type system and the OO approach, Java programs generally offer many potential join points with strong semantics – which supports *obliviousness*.
- Advice polymorphism – and hence *quantification* – is supported by a polymorphic type system. All types are inherited from the common `java.lang.Object` base interface. Polymorphic behavior beyond the services offered by `java.lang.Object` is supported by Java's run-time type reflection capabilities.

- *Platform independence*, however, an often-cited strength of Java programs ("write once, run anywhere"), is actually very limited. For many platforms used in the embedded systems domain a JVM is not available.
- This is mainly caused by the fact that Java's run-time support induces an *unacceptably high base overhead*. Even the more tailorable J2ME (Java 2 Micro Edition) is way too big for most 8- and 16-bit microcontroller platforms. As J2ME drops run-time type reflection – which is required to reach quantification – it is furthermore not as a good base for AOP as is Java's Standard Edition.

Overall, Java-based AOP approaches are inapplicable to the domain of small, embedded systems, because Java induces too high an overhead. This is a pity from the AOP perspective – the most mature AOP approaches are available for the Java domain. *AspectJ* [KHH+01b, KHH+01a] clearly is the reference for AOP with respect to language and tools; and besides AspectJ many other AOP language extensions have been proposed for Java, such as *Hyper/J* [OT00, OT01], *DemeterJ* [LLW03], *CaesarJ* [AGMO06], *Josh* [CN04], and *LogicAJ* [KRH04, KR06]. This list is by no means complete; however, as Java-based approaches can be ruled out for our target domain I shall not discuss them in further detail.

4.1.4.2. C-Based Aspect Languages

C as a base language offers minimal overhead and excellent platform independence, but lacks the necessary properties to achieve obliviousness and quantification in AOP extensions:

- C is well known as the language of choice if it comes to *minimal overhead*.
- For this very reason it is by far the most common language in the domain of embedded systems. Mature tool support is *available for every hardware platform*.
- However, compared to other languages, C programs tend to have a relatively coarse-grained structure. Even the module concept is not part of the language, but idiomatically emulated outside of it with a preprocessor and the file system. All this remarkably reduces the amount of syntactical information that determines the initial set of potential join points, which limits *obliviousness*.
- As a relatively low-level language, C has a rather primitive type system. Polymorphic types or reflection are not supported, hence *quantification* is technically difficult to reach.

Overall, the situation with C is exactly the opposite as with Java: C fits well into the specific requirements of the embedded systems domain, but limits obliviousness and quantification too much to consider it as a good base language for AOP.

Nevertheless, several approaches have been suggested to bring aspects into the C language domain [Ada06, Ada08]. Billions of lines of existing C code are a strong argument

for AOP with C – even if the resulting expressiveness remains behind the Java-based approaches. *AspectC* by COADY and colleagues, for instance, neither supports introductions nor offers the required language means for quantification, but has successfully been used to factor out four scattered operating-system concerns (prefetching, disk quotas, blocking, and page daemon activation) from the FreeBSD kernel to improve their evolvability [CKFS01, CK03].

The fact that a weaver for AspectC is still missing has recently been the driving factor for the development of other general purpose AOP languages for C with static weaving support: *Mirjam* and *WeaveC* by DURR, NAGY, and colleagues [Dur, NvGvdP08], *Aspicere* and *Aspicere2* by ADAMS [Ada, Ada08], and *ACC* by GONG et al [GJ, GJ08]. At the time of this writing (June 2008) most of these languages are still at an early yet rapidly evolving state, maybe with the exception of Mirjam and the corresponding WeaveC weaver back-end, which successfully have been applied to industrial-strength projects [NvGvdP08].

Interesting are the different strategies to overcome C's inherent limitation with respect to quantification: ACC attempts to mimic AspectJ as far as possible and provides a run-time type reflection mechanism for function arguments in its join-point API. However, run-time type reflection in an inherently not run-time–typed language has only limited value with respect to quantification; I will elaborate on this in more detail in Section 4.2 on the example of AspectC++. Mirjam and Aspicere, on the other hand, escape to an external logic language. In these languages the match and advice expansion mechanism is built on Prolog. The resulting aspect languages are less "AspectJ-like" and very different from their base language. This probably makes it more difficult to teach and use them, however, effectively overcomes the C deficiencies with respect to polymorphic advice.

Not a general-purpose AOP language extension for C, but an AOP-inspired language of temporal logic was used by ÅBERG and colleagues to integrate the *Bossa* scheduler framework into the Linux kernel [ÅLS+03]. The *C4* approach by FIUCZYNSKI and associates uses AOP concepts to extend C with language means for a "semantic patch system" for the application of patches to the Linux kernel [FGCW05]. The idea of "semantic patches" in Linux device drivers is also an objective of the *Coccinelle* project and its *Semantic Patch Language* (*SmPL*) by PADIOLEAU et al [LMU05, PLM06, PLMH08]. The focus of SmPL is on specifying context-sensitive patches for collateral changes in Linux, such as a modifications of the kernel API that otherwise had to be caught up manually in hundreds of device drivers. In a sense, SmPL thereby provides obliviousness and quantification for source code transformations, even though the authors do not directly relate their approach to AOP.

4.1.4.3. C++-Based Aspect Languages

C++ as a base language offers minimal overhead, good platform independence, as well as good support for obliviousness and quantification in AOP language extensions:

- C++ offers a C-like *minimal overhead*. Even though certain language features (such as virtual functions, exception handling, and run-time type information) do induce

an overhead, C++ applies strict "pay as you use" with respect to these extended features.

- Tool support is *available for most platforms*, including many 8- and 16-bit microcontroller architectures.
- Compared to C programs, C++ programs tend to be structured more fine-grained and to generally provide a better separation of concerns. By classes, namespaces, templates, and inline functions, C++ offers many additional syntactical concepts for this purpose; the type system is sophisticated and strict. This promises good support for *obliviousness*.
- Run-time polymorphism and type reflection in C++ is less sophisticated than in Java, as there is no common root class and only class types with virtual functions are polymorphic at run-time. However, C++ offers compile-time polymorphism over all types, including the built-in primitive types, by templates and overloading. With *traits* [Mye96, Ale00], it furthermore supports compile-time reflection. Thereby, all means required for advice polymorphism, and hence, *quantification*, are available.

Overall, C++ combines the benefits of C – general applicability to our domain – with the most benefits of Java regarding the achievable level of obliviousness and quantification.

Nevertheless, only very few aspect languages based on C++ and with static weaving support have been proposed. Most of them actually belong to the group of early AOP approaches from the pre-Java (and pre-AspectJ) time (compare Section 2.3.6.1): *Composition Filters* [ABV92], *Subject-Oriented Programming* [HO93], and *Adaptive Programming* [Lie96] all were originally applied to C++. However, they have barely been maintained since then and do not reflect the fundamental improvements the C++ language has experienced in the following decade, eventually resulting in the ISO/IEC 14882:2003 standard (ISO-C++) [Ins03].

Besides AspectC++, which is used in this thesis, there is only one other approach that claims to offer AOP with static weaving support for C++: *XWeaver* by ROHLIK and colleagues [RPCB04, XWe] is a (conceptually base-language–independent) approach for AOP-style source code transformations. Aspect and pointcut definitions are given in their own XML-based dialect, the *AspectX* language. Pointcuts do not yield points in the behavioral or structural space of a program, but nodes in an XML-based representation of its source code; advice describes transformations to be performed in these points; it is even possible to introduce new comments by advice. The authors motivate this unusual take on AOP with the specific demands of safety-critical systems. The software development process for safety-critical systems often requires formal reviews of the (woven) source code; hence, aspect developers should have full control over the resulting source code, including all "aspects" of its readability, which includes formatting and commenting. In fact the source-code–centric take of XWeaver has conceptually more in common with the "semantic patch" approaches for C discussed in the previous section than with general-purpose AOP. Technically, it works similar to *frame processors* (such as XVCL [JZ01] or Angie [Del05]).

	Java	C	C++	Ada
support for obliviousness	++	−	++	++
support for quantification	++	−−	++	+
platform independence	○	++	+	+
minimal overhead	−−	++	++	+
AOP tool support	++	○	+	−−

Table 4.1.: Comparison of Base Languages for AOP in System Software.
Depicted is the suitability of the analyzed base language alongside the requirements from Section 4.1.2. [Scale: (very bad) −− , −, ○, +, ++ (very good)]

4.1.4.4. Ada-Based Aspect Languages

Ada as a base language offers good platform independence, minimal overhead, as well as good support for obliviousness and quantification:

- Even though compiler support for Ada is not as broadly available as is for C++ or even C, Ada is highly *accepted in certain domains* of embedded system development, especially avionics.
- The language's focus on static typing supports *minimal overhead*.
- Ada has an even stricter type system and module concept than C++, thus, generally supports *obliviousness*.
- The expressive power of Ada is comparable to C++, including means for (static) polymorphism, which helps with respect to *quantification*.

However, even though Ada is an interesting candidate for AOP, aspect languages based on Ada are just not available. In fact, I am not aware of even a single attempt to bring AOP ideas into the Ada domain.

4.1.5. Summary

The base language has a most significant impact on the *obliviousness, quantification*, *platform independence*, and *minimal overhead* properties of an aspect language for our domain. Table 4.1 depicts the results from the evaluation of Java, C, C++, and Ada with respect to these criteria: C++ is the most promising candidate. Compared to C, C++ provides significantly more expressiveness – which is necessary to reach obliviousness and quantification. Compared to Java, C++ provides the efficiency and platform independence required for our domain. In combination with an ideal static weaver, AOP based on C++ should be implementable as a (mostly) overhead-free mechanism. Hence, AspectC++ should be an ideal candidate to evaluate AOP for the development of configurable system-software product lines for resource-constrained embedded devices.

Essentially, AspectC++ was choosen for this thesis as it is the one and only available AOP language extension for C++ with static weaving support. However, before applying it to

my research objectives (cf. Section 3.2), I had to further extend, evaluate, and improve the AspectC++ language and code generation. This will be detailed in the following two sections (Section 4.2 and Section 4.3).

4.2. Generic Advice

In Section 4.1.2.2, we learned that support for *polymorphic advice* is a crucial property to achieve quantification with an aspect language. In the context of this thesis, I could significantly improve AspectC++'s support for advice polymorphism by extending the join-point API with additional compile-time context information. Beginning with AC++-0.9, polymorphic advice can be implemented with compile-time genericity on the base of C++ templates and static overloading. Such *generic advice* facilitates highly reusable and efficient aspect implementations.

4.2.1. Generic Advice – Motivation

Compared to languages like Java and C#, the C++ language has a less powerful run-time type system, but a more powerful compile-time (static) type system. C#, while still being a statically typed language, implements a unified type system where even primitive value types offer the interface of the one and only root class `System.object`. In Java all class types derive from `Java.lang.Object`. Due to auto-boxing it is possible in both languages to pass value type instances as object references. Basically, Java and C# allow the programmer to treat "everything as an object" at run-time.[6] This facilitates the development of "type-generic code", in the sense that such code can deal with objects of any type *at run-time*.

In C++ there is no such common root class and the C++ run-time type information (RTTI) system offers only a very limited set of run-time services. On the other hand, C++ implements a static type system that offers static genericity by operator and function overloading, argument-dependent name look-up, and C++ templates. In general, the C++ philosophy is to use *genericity at compile-time*, while Java and C# advise *genericity at run-time*.[7]

4.2.1.1. Advice Polymorphism with Dynamic Typing

Type genericity is particularly important for the development of aspects, which typically are intended to be broadly reusable and applicable. In Section 4.1.2.2, we had the example of a tracing aspect that should log all actual parameter and result values of

[6] Actually, it is the Smalltalk language that carried the "everything is an object" idea to the extremes. However, Smalltalk does not offer a static type system.

[7] This is even true with Java generics introduced in the Java 5, which are basically a syntactic wrapper around the "treat everything as an object" philosophy.

method invocations – independently of the actual method's signature. In AspectJ this is supported by a *run-time join-point API* for advice implementations, which offers a unified interface to access a join point's context information. This information includes the number of parameters, the argument and return values (as `Object`), and (via the interface of `Object` and Java's reflection capabilities) their run-time types.

AspectC++ offers a similar mechanism to retrieve the number, types, and values of an affected function's arguments at run-time. The AspectC++ join-point API is accessible from advice code via the pointer `JoinPoint* tjp` and provides some methods for this purpose: `tjp->args()` returns the number of arguments, `tjp->arg(`i`)` returns the memory position (`void*`) holding the value of an argument, and `tjp->argtype(`i`)` a C++-ABI-V3 conforming string representation of an argument's type.[8]

4.2.1.2. The Issue

The string-based run-time representation of argument types serves informational purpose, but is otherwise of only limited value. It cannot be used as a type in the sense of language syntax. It does not help, for instance, to print out an actual parameter value to `std::cout`. For this purpose, the `void` pointer returned by `tjp->arg(`i`)` would have to be cast to the corresponding *static* C++ type of the parameter:

```
advice methods() : before() {
  std::cout << "before " << tjp->signature() << ", called with:";
  for( int i = 0; i < tjp->args(); ++i ) {
    const char* type = tjp->argtype(i);
    if( std::strcmp( type, "i" ) == 0 )
      std::cout << " " << *static_cast< int* >( tjp->arg( i ) );
    else if( std::strcmp( type, "d" ) == 0 )
      std::cout << " " << *static_cast< double* >( tjp->arg( i ) );
    else if( std::strcmp( type, "RK3Foo" ) == 0 )
      std::cout << " " << *static_cast< const Foo* >( tjp->arg( i ) );
    ...
    else std::cout << " <unknown type>";
  }
  std::cout << std::endl;
}
```

The necessary mapping from run-time string representations to the corresponding compile-time C++ type casts requires a huge `if`-cascade – and will even fail if some argument type was forgotten or simply unknown when the aspect was developed. Even though there had been attempts to extend C++ by a more sophisticated meta-object protocol [Chi95]: The C++ language is just not designed for run-time reflection; it does not fit into the C++ philosophy.

[8] This representation of types is known from the mangled symbol names emitted by many C++ compilers (see `http://www.codesourcery.com/cxx-abi/abi.html#mangling`).

To cope with the C++ philosophy, advice polymorphism in AspectC++ should better be based on static typing. This leads to the concept of **generic advice**:

> *"We call advice a* generic advice, *if its implementation depends on join-point–specific static type information."* [LBS04]

The key to generic advice is a *compile-time join-point API*.

4.2.2. Extending the Join-Point API for Generic and Generative Programming

To support the implementation of generic advice code, the AspectC++ join point API had to be extended. Table 4.2 lists an excerpt from the join-point API with those parts and extensions that are relevant for the implementation of generic advice.[9]

4.2.2.1. Support for Generic Programming

The upper part in Table 4.2 (*compile-time join-point API*) provides compile-time type information, which can be used to instantiate generic code by advice. The lower part (*run-time join-point API*) uses these types to provide a type-safe interface to the dynamic join point context. These methods are bound at compile-time, but called at run-time.

The most important extension is the function `Arg<i>::ReferredType *arg()`, which offers a type-safe way to access argument values:

```
advice methods() : before() {
  ...
  std::cout << " " << *tjp->arg< 0 >(); // print 1st paramter value
  std::cout << " " << *tjp->arg< 1 >(); // print 2nd paramter value
  ...
}
```

As the call to `tjp->arg< 0 >()` returns a *typed* pointer to the argument value of the first argument, no casting is required. The compiler determines (and can even inline) the correct output operator by overload resolution and argument-dependent name look-up. If no compatible version of the operator can be found, a compile-time error is thrown. The result is an efficient and type-safe generic advice implementation.

However, for this type-safe access the argument index has to be bound (hence known) at compile-time. It is no longer possible to use a simple `for`-loop to iterate over the actual sequence of arguments at run-time. In fact, the advice implementation sketched above is still not very generic: Even though we are now flexible with respect to the actual argument types, we can apply the advice only to functions with a specific number of arguments. To become more flexible, we have to move the process of iterating over the sequence of arguments from run-time to compile-time as well. This can be achieved by means of C++ template meta-programming.

[9] You can find the complete join-point API in the AspectC++ quick language reference in Appendix section A.3.7.

(a) compile-time join-point API

element	description	kind
`That`	object type (object initiating a call)	*typedef*
`Target`	target object type (target object of a call)	*typedef*
`Arg<i>::Type` `Arg<i>::ReferredType`	type of the i^{th} argument of the affected function (with $0 \leq i < ARGS$)	*typedef*
`Result`	result type of the affected function	*typedef*
`ARGS`	number of arguments	*enum value*
`JPID`	unique numeric identifier for this join point	*enum value*
`JPTYPE`	numeric identifier describing the type of this join point	*enum value*

(b) run-time join-point API

element	description	kind
`That *that()`	returns a pointer to the object initiating a call or 0 if it is a static method or a global function	*method*
`Target *target()`	returns a pointer to the object that is the target of a call or 0 if it is a static method or a global function	*method*
`Arg<i>::ReferredType *arg()`	returns a typed pointer to the argument value with compile-time index i	*template method*
`Result *result()`	returns a typed pointer to the result value or 0 if the function has no result value	*method*

Table 4.2.: Extensions to the AspectC++ join-point API for generic advice.
Depicted is an excerpt from the AspectC++ join-point API with all elements relevant for generic advice. **(a)** Elements of the *compile-time join-point API* reflect the complete signature of a function or method call or execution; they are provided to advice code as members of the join-point–specific type `JoinPoint`. **(b)** Elements of the run-time join-point API provide type-safe access to the actual values related to a function or method call or execution; they are provided as methods to be invoked for the `JoinPoint` instance pointer `tjp`.

Excursus: C++ Template Meta-Programming

C++ templates are a Turing-complete functional language of its own which can be exploited for static meta-programming [Vel95, CE00, p. 407]. A C++ template meta-program works on types and constants and is executed by the compiler at compile-time.

A language is Turing-complete if it provides a *case discrimination* and a *loop* construct. In the C++ template language, case discrimination is realized by *template specialization*. Loops are implemented by *recursive instantiation* of templates. These language features, in conjunction with nontype (`int`) template parameters, are the building blocks of template

meta-programs.[10] This is demonstrated in the most popular (and simple) example for a template meta-program:

```
// recursive definition of fac; termination by specialization for 0
template< int N > struct fac { enum{ res = N * fac< N - 1 >::res }; };
template<> struct fac< 0 > { enum{ res = 1 }; };

// calculate factorial at compile-time
const fac_5 = fac< 5 >::res;
```

The `fac<`*N*`>` template calculates the factorial of the passed integer constant *N* as an `enum` value `res` at compile-time – by recursive instantiation of itself. The recursion is terminated by a template specialization for `fac<0>`.

4.2.2.2. Support for Generative Programming

The extended compile-time join-point API provides adequate support to iterate in a similar way with template meta-programs over the sequence of argument types related to some join point. This facilitates *generative programming* techniques based on C++ templates [CE99, CE00, Ale01] to *generate* the statically typed advice code according to the affected function's signature at compile-time. Such **generative advice** can be understood as an advanced form of generic advice.

The key is to provide the sequence of argument types in a "meta-program–friendly" way. For this, the argument types are not represented as explicit elements (`Arg1`, ..., `ArgN`) in the join-point API, but indirectly via the `Arg<`*i*`>` template. The AC++ weaver has to generate this template as member of the `JoinPoint` type and specialize it for each parameter. The following shows an excerpt of the `JoinPoint` type generated by AC++-0.9 for a call from `main()` to a function that receives a `bool` and an `int&` value as parameters:[11]

```
struct TJP_main0 { // typedef'ed to JoinPoint inside advice body
  ...
  enum { ARGS = 2 };
  template <int I> struct Arg {};
  template <> struct Arg<0> {
    typedef bool Type; typedef bool ReferredType;
  };
  template <> struct Arg<1> {
    typedef int & Type; typedef int ReferredType;
  };
  ...
};
```

[10] Strictly speaking, nontype template parameters are not mandatory for the Turing-completeness of the C++ template language. They "just" make template meta-programming practicable; we thereby can use the built-in C++ support for integer arithmetics.

[11] This code was generated for VISUALC++ as a back-end compiler. Because of subtle differences in the interpretation of what the C++ standard says about template specializations in class or namespace scope, slightly different code must be generated for G++. You can find a G++ example in Listing 4.2.

Each specialization of the `Arg<i>` template defines type aliases to the actual parameter's type "as is" (`Type`) and with all reference modifiers removed (`ReferredType`). Additionally, the number of arguments (`ARGS`) is provided as a compile-time constant. The default (not specialized) implementation of the `Arg<i>` template remains empty. This ensures that a compile-time error is thrown if an advice implementation tries to use a nonexistent argument (not within the range of $0 \leq i < ARGS$).

Because of this "meta-program–friendly" structure, it is now possible to move the iteration over the function arguments to a compile-time loop and thereby make the advice code truly generic. The result is a reusable tracing aspect that supports quantification over arbitrary join points:

```
#include <iostream.h>
template< class TJP, int N > struct printer {
  static void print( TJP* tjp ) {
    std::cout << " " << *tjp->template arg< TJP::ARGS - N >();
    printer< TJP, N - 1 >::print( tjp );
  }
};
template< class TJP > struct printer< TJP, 0 > {
  static void print( TJP* tjp ) { /* do nothing */}
};
aspect Tracing {
  pointcut virtual methods() = 0;
  advice methods() : before() {
    std::cout << "before " << tjp->signature() << ", called with:";
    printer< JoinPoint, JoinPoint::ARGS >::print( tjp );
    std::cout << std::endl;
  }
};
```

Of course, tracing of parameter values is not the only useful application of generative advice – it is just an example. The caching example in Appendix section A.2.2 demonstrates a sophisticated application of generative advice for the implementation of a reusable and efficient caching aspect.

4.2.3. Example: Checking for Invalid Object Identifiers in AUTOSAR OS

The aspect `ServiceProtection_InvalidObjectCheck` in Listing 4.1 implements a part of the architectural policy Service protection in AUTOSAR OS (see Section 3.1.2.3), namely the test for an invalid system object identifier.[12] AUTOSAR-OS services (API functions) take the system object they should work on as first parameter and shall return with the `E_OS_ID` error code if the object identifier does not refer to a valid instance. However, depending on the service, the parameter refers to either a *task*, or *resource*, an *application*

[12] The example has been taken from the implementation of the CiAO operating system, which we will further discuss in Chapter 6.

```
1  aspect ServiceProtection_InvalidObjectCheck {
2    pointcut affectedServices() = "% AS::ActivateTask(...)" || "% AS::ReleaseResource(...)" …;
3    advice execution( affectedServices ()) : around () {
4      if( *tjp->arg<0>() < (OSObjectTraits< Arg< 0 >::ReferredType >::MAX) ) {
5        tjp->proceed();             // valid object id, proceed
6      }
7      else {
8        *tjp->result() = E_OS_ID;   // invalid object id, return error code
9      }
10   }
11 };

   // default implementation (raises a compile-time error if no specialization can be found)
   template< typename T > struct OSObjectTraits      { /* MAX *not* defined */ };

   // type-specific specializations (examples)
   template <> struct OSObjectTraits< TaskType >     { enum{ MAX = cfOS_NUMBER_OF_TASKS }; };
   template <> struct OSObjectTraits< ResourceType > { enum{ MAX = cfOS_NUMBER_OF_RESOURCES }; };
   …
```

Listing 4.1: Test for invalid system objects in AUTOSAR OS with generic advice.
The aspect `InvalidObjectCheck` employs generic advice to be independent from the actual object type; all type-dependent code (the maximum value to check for) has been externalized into traits classes.

mode, *alarm*, or *ISR*, a *counter* or *schedule table*, or an *OS application*. `ActivateTask()`, for instance, takes a task (`TaskType`) as first parameter, whereas `ReleaseResource()` takes a resource (`ResourceType`). Withouth generic advice it would be necessary to define one piece of advice per object type for the validation, as the test depends on the parameter type. With generic advice it is possible to deal with all object types by a single piece of advice. Only the *actually* type-dependent part has to be implemented on a per-type basis (the maximum identifier value, which is here provided by *traits classes* [Mye96, Ale00]). Depending on the type of the first argument, the compiler selects the fitting specialization of the `OSObjectTraits<` *type* `>` template (line 4).

A further advantage of the solution with generic advice is robustness. If the API gets extended and the advice affects a service that uses a so far *unknown* object type a compile-time error is thrown. A solution without generic advice (one distinct piece of advice per object type) always bears the danger that this situation is silently missed.

4.2.4. Summary

With generic advice, AspectC++ now supports a concept for advice polymorphism that fits well into the philosophy of the C++ language. Generic advice facilitates reusable, type-safe, and efficient aspect implementations on the base of quantification over static types. We will see more examples in Chapter 6.

4.3. AspectC++ Overhead

Cost efficiency in the generated code is a major goal of AspectC++. Therefore, the AspectC++ weaver AC++ follows a source-to-source weaving approach with generation of code patterns that (1) do not use "expensive" C++ language elements (such as RTTI or exceptions), and (2) can be well optimized by current C++ compilers. But how well?

In this section, I present the results of a detailed cost evaluation of code generated by AC++-1.0PRE1. My goal with this evaluation was to get an in-depth understanding of the cost induced by every AspectC++ language feature, and to set this in relation to what we would expect from an "ideal weaver" (Section 4.1.3).

The evaluation was originally performed with AC++-0.9. It revealed some unexpected overhead, mostly caused by deficiencies in the optimization capabilities of current back-end compilers. Partly, however, I was able to overcome these deficiencies by suggesting alternative, more "optimizer-friendly" code generation patterns. The outcome of the first evaluation eventually led to the implementation of AC++-1.0PRE1, which applies the new code-generation patterns.

4.3.1. Code Generation of AC++

The static AspectC++ weaver AC++ transforms AspectC++ code (C/C++ code with AspectC++ language elements) into C++. Advice is transformed into member functions of the corresponding aspect, which in turn is transformed into C++ class. Listing 4.2 demonstrates these transformations for a simple aspect that gives before-advice.

At first sight, the output of AC++ seems to be surprisingly huge and over-complicated. The generated code is, however, not intended for reading by a human developer, but tweaked for flexibility and for an optimizing C++ back-end compiler.[13] An advice method, for example, is not called directly, but routed over an extra forward-declared `invoke_xxx()` inline-function (line 67). This detour makes it possible to place the aspect definition *behind* all class definitions in the translation unit. Only thereby the aspect itself can use the classes affected by it, for instance use them with member variables or method parameters.

4.3.2. Benchmarks

To evaluate the overhead induced by the AC++-generated code for various AspectC++ language features, I conducted a study with a large series of microbenchmarks. In these microbenchmarks the overhead of applying some functionality (incrementation of a global variable) by advice is compared to a hand-written "tangled" solution. The condensed results are depicted in Tables 4.3, 4.4, 4.5, and 4.6. All benchmarks were performed

[13] The most relevant optimizations the C++ compiler has to support are: (1) function embedding for functions that are explicitly marked as *inline* and (2) local alias analysis for structure and array parameters, which is required to prevent unnecessary copies of the join-point context (`tjp`).

(a) translation unit *Test.cpp*

```
volatile int global;
struct B {
  int bar(int a, int b) {...}
};
struct A {
  int foo(B &b) {
    return b.bar(47, 11);
} };
```

(b) aspect header *jpapi.ah*

```
extern volatile int global;

aspect jpapi {
  advice call("int B::bar(int, int)")
  : before () {
    global = *tjp->arg<1>();
  }
};
```

(c) woven translation unit *Test.acc*

```
1  class jpapi;
2  template <class JoinPoint>
3  inline void invoke_jpapi_a0_before (
4    JoinPoint *tjp);
5  volatile int global;
6  struct B {
7    int bar(int a, int b) {...};
8  };
9  struct A {
10   int foo (B &b) {
11     return __call_foo_0_0(
12       this, &b, 47, 11);
13   }
14   struct TJP_foo_0_0 {
15     typedef int Result;
16     typedef ::A That;
17     typedef ::B Target;
18     ...
19     template <int I, int DUMMY = 0>
20     struct Arg {...};
21     template <int DUMMY>
22     struct Arg<0, DUMMY> {
23       typedef int Type;
24       typedef int ReferredType;
25     };
26     template <int DUMMY>
27     struct Arg<1, DUMMY> {...}
28
29     void **_args;
30     template <int I>
31     typename Arg<I>::ReferredType *arg(){
32       return (typename
33         Arg<I>::ReferredType*) _args[I];
34   } };
35   static inline int __call_foo_0_0 (
36     ::A *srcthis, ::B *dstthis,
37     int arg0, int arg1)
38   {
39     AC::ResultBuffer< int  > result;
40     void *args_foo_0[] = { (void*)&arg0,
41                            (void*)&arg1 };
42     TJP_foo_0_0 tjp;
43     tjp._args = args_foo_0;
44     invoke_jpapi_a0_before<TJP_foo_0_0>(
45       &tjp);
46     ::new (&result) int  (dstthis->bar(
47                           arg0, arg1));
48     return (int &)result;
49   }
50 };  // struct A
51
52 class jpapi {
53 public:
54   static jpapi *aspectof () {
55     static jpapi __instance;
56     return &__instance;
57   }
58   template<class JoinPoint>
59   void __a0_before (JoinPoint *tjp) {
60     global=(int)tjp->template arg<1>();
61 } };
62
63 template <class JoinPoint>
64 inline void invoke_jpapi_a0_before (
65   JoinPoint *tjp)
66 {
67   ::jpapi::aspectof()->__a0_before (tjp);
68 }
```

Listing 4.2: Code transformation performed by `ac++-1.0pre1` (example).

(a) Translation unit *Test.cpp* with classes `A` and `B`; `A::foo()` calls `B::bar()`. **(b)** Aspect header *jpapi.ah*. The aspect `jpapi` gives before advice for all calls to `B::bar()` and uses the join-point API to retrieve the value of the second parameter passed to `B::bar()`. **(c)** Woven translation unit *Test.acc*. The aspect `jpapi` has been transformed into a C++ class `jpapi` (lines 52–61), the advice into a member function `jpapi::_a0_before()` (line 59). The original call to `B::bar()` has been replaced by a call to the generated wrapper function `A::_call_foo_0_0()` (line 11). Furthermore, a corresponding join point class `TJP_foo_0_0` has been generated that encapsulates the join-point–specific static and dynamic context. In the wrapper function (lines 35–49), an instance `tjp` of `TJP_foo_0_0` is created, initialized (lines 40–42), and passed as template argument to the before-advice invocation function `invoke_jpapi_a0_before()` (line 44), which retrieves the aspect instance and calls the generated advice method (line 67). Finally, the call to the original `B::bar()` is performed (line 46).

(a)	G++-3.3.5	cycles		stack		code	
	advice type	**abs**	Δ	**abs**	Δ	**abs**	Δ
	tangled	*4*		*0*		*4128*	
execution	before	*6*	*2*	*0*	*0*	*4128*	*0*
	after	*6*	*2*	*0*	*0*	*4128*	*0*
	around	*6*	*2*	*0*	*0*	*4128*	*0*
	around-0.9	*23*	*19*	*56*	*56*	*4192*	*64*
call	before	*6*	*2*	*0*	*0*	*4128*	*0*
	after	*6*	*2*	*0*	*0*	*4128*	*0*
	around	*6*	*2*	*0*	*0*	*4128*	*0*
	around-0.9	*23*	*19*	*56*	*56*	*4192*	*64*

(b)	ICC-9.0	cycles		stack		code	
	advice type	**abs**	Δ	**abs**	Δ	**abs**	Δ
	tangled	*0*		*0*		*6372*	
execution	before	*0*	*0*	*0*	*0*	*6372*	*0*
	after	*0*	*0*	*0*	*0*	*6372*	*0*
	around	*0*	*0*	*0*	*0*	*6372*	*0*
call	before	*3*	*3*	*0*	*0*	*6356*	*0*
	after	*3*	*3*	*0*	*0*	*6356*	*0*
	around	*3*	*3*	*0*	*0*	*6356*	*0*

Table 4.3.: AspectC++ microbenchmark *incrementer*.
Cost (clock *cycles*, *stack* and *code* bytes) of incrementing a global `int` variable either in the body of a function `void f()` (*tangled*) or by giving advice (*before/after/around* for *call/execution* join points) to the same function. Δ denotes the difference to *tangled*. **(a)** Results with G++-3.3.5 and AC++-1.0PRE1 (AC++-0.9 for case *around-0.9*). **(b)** Results with ICC-9.0 and AC++1.0PRE1.
[Measurements were performed on an Intel PIII E ("Coppermine") machine running at 600 MHz under Linux 2.6 in single user mode. Cycles were measured with `rdtsc` for 1000 iterations and averaged over 100 series ($\sigma < 0.1\%$). Used G++ optimization flags: -O3 -mpreferred-stack-boundary=2 -fno-align-functions -fno-align-jumps -fno-align-loops -fno-align-labels -fno-reorder-blocks -fno-prefetch-loop-arrays. Used ICC optimization flags: -O3]

with AC++-1.0PRE as weaver and G++-3.3.5 as back-end compiler. The G++ compiler from the GNU compiler collection (gcc) is of particular interest for our domain, as it is available for a large number of hardware platforms, including many 8-bit and 16-bit "embedded platforms". To get an impression of the "gcc impact" in the results, however, most benchmarks were repeated with Intel's ICC-9.0 as back-end compiler. Even though ICC is not available for any typical "embedded platform", the results with this compiler are of a theoretical interest, as ICC is known to be one of the best optimizing compilers available today.

For each benchmark, the consumed CPU time (clock *cycles*), dynamic memory consumption (*stack*, bytes), and static memory consumption (*code/data*) were measured.[14]

4.3.2.1. Simple Advice for Parameterless Functions

The base overhead of applying advice to a parameter-less function is low (benchmark *incrementer,* Table 4.3). With G++-3.3.5 and AC++-1.0PRE1, advice invocation takes only 2 cycles, independent from the type of advice (*before, after, around*), the join point type (*call/execution*), and even the number of aspects giving advice to the join point

[14]The *data* numbers were measured, but are omitted in Tables 4.3, 4.4, and 4.5, as the AspectC++ features under observation in these benchmarks do not contribute to *data*.

(a)	G++-3.3.5	cycles		stack		code	
	# aspects	abs	Δ	abs	Δ	abs	Δ
execution	1	*6*		*0*		*4080*	
	2	*5*	*-1*	*0*	*0*	*4080*	*0*
	3	*6*	*1*	*0*	*0*	*4096*	*16*
call	1	*6*		*0*	*0*	*4096*	
	2	*5*	*-1*	*0*	*0*	*4096*	*0*
	3	*6*	*1*	*0*	*0*	*4096*	*0*

(b)	ICC-9.0	cycles		stack		code	
	# aspects	abs	Δ	abs	Δ	abs	Δ
execution	1	*3*		*4*		*6424*	
	2	*2*	*-1*	*4*	*0*	*6340*	*-84*
	3	*3*	*1*	*4*	*0*	*6340*	*0*
call	1	*4*		*0*	*0*	*6324*	
	2	*5*	*1*	*0*	*0*	*6340*	*16*
	3	*9*	*4*	*0*	*0*	*6340*	*0*

Table 4.4.: AspectC++ microbenchmark *multiaspect*.
Scaling of cost (clock *cycles*, *stack* and *code* bytes) if *1–3* aspects give around-advice to the same *call* or *execution* join point. Δ denotes the difference to the previous line. **(a)** Results with G++-3.3.5. **(b)** Results with ICC-9.0. For methodical details see Table 4.3.

(benchmark *multiaspect*, Table 4.4). The size of the text segment (*code*) also remains stable, the increase by 16 bytes in one case was caused by linker alignment of the affected section.

The benefit of the improved code generation patterns for around-advice in AC++-1.0PRE1 become visible when comparing these numbers to the *around-0.9* line in Table 4.3.a. With AC++-0.9, around-advice is noticeably more expensive than with AC++-1.0PRE1. This cost is caused by the call of the original function in `tjp->proceed()`: In the AC++-0.9 implementation, the context is passed via a stack-allocated data structure containing all parameters and a function pointer to invoke the original function later. This explains the extra stack overhead. Moreover, calls via function pointers do not get inlined by most back-end compilers, which explains the CPU overhead. In AC++1.0PRE1, the implementation of `tjp->proceed()` has been improved so that the original function is called directly, which better supports inlining by the back-end compiler. You can find a detailed comparison of the code generation differences in Appendix section A.4.

The ultimate result is that around-advice now induces exactly the same – very low – overhead as before- or after-advice. These results fit well with our expectations regarding the overhead of simple advice for statically evaluable pointcuts, as yielded by the `call()` and `execution()` pointcut functions (Section 4.1.3). The good results for around-advice are nevertheless notable – in AspectJ the cost of around-advice is significantly higher than for before-/after-advice [DGH+04].

With ICC as back-end compiler, the variation in the results is higher, but in most cases they are even better than the G++ results. For example, in the simple *incrementer* test scenario there are cases in which there is no or sometimes even a “negative overhead”. The compiler was able to generate semantically equivalent code with fewer or better-ordered instructions in these cases.

(a)	G++-3.3.5	cycles		stack		code	
	jpapi	**abs**	**Δ**	**abs**	**Δ**	**abs**	**Δ**
	plain	*5*		*16*		*3968*	
	that()	*7*	*2*	*20*	*4*	*3968*	*0*
	target()	*8*	*3*	*20*	*4*	*3968*	*0*
	result()	*11*	*6*	*16*	*0*	*3968*	*0*
$n=1$	*plain*	*13*		*24*		*3968*	
$n=1$	arg<0>()	*13*	*0*	*24*	*0*	*3968*	*0*
$n=2$	*plain*	*13*		*32*		*3984*	
$n=2$	arg<1>()	*13*	*0*	*32*	*0*	*3984*	*0*

(b)	ICC-9.0	cycles		stack		code	
	jpapi	**abs**	**Δ**	**abs**	**Δ**	**abs**	**Δ**
	plain	*9*		16		*6372*	
$n=0$	that()	*8*	*2*	*20*	*4*	*6372*	*0*
$n=0$	target()	*11*	*3*	*20*	*4*	*6372*	*0*
	result()	*7*	*6*	*16*	*0*	*6372*	*0*
$n=1$	*plain*	*14*		*24*		*6404*	
$n=1$	arg<0>()	*14*	*0*	*24*	*0*	*6404*	*0*
$n=2$	*plain*	*17*		*32*		*6420*	
$n=2$	arg<1>()	*17*	*0*	*32*	*0*	*6420*	*0*

Table 4.5.: AspectC++ microbenchmark *jpapi*.
Cost (clock *cycles*, *stack* and *code* bytes) of a member function call to some member function `void C::f(`n`)` (with $n :=$ number of `int` arguments) for which some advice is given which either does not use the join-point API (*plain*) or calls a join-point API function (*that()*, *target()*, ...). Δ denotes the difference to the corresponding *plain* line. **(a)** Results with G++-3.3.5. **(b)** Results with ICC-9.0. For methodical details see Table 4.3.

4.3.2.2. Simple Advice for Parameterized Functions

For advice given to functions with parameters, the stack space allocated by the compiler to pass call-by-value parameters is actually doubled. This becomes evident from the *plain* lines in Table 4.5, which represent the cost of a member function call with 0–2 `int` parameters for which some advice was given.[15] Each additional *4*-byte `int` parameter increases the absolute stack cost by *8* bytes with G++ and, because of a more agressive stack frame aligning, even *8–16* bytes with ICC. The reason turned out to be a limitation of the interactions between the compiler's inliner and optimizer: Whenever a function is inlined, the compiler ensures call-by-value semantics by pushing an extra copy of all function parameters on the stack. In most cases, the optimizer later replaces the parameter passing code with direct register access, but the (now completely useless) stack reservations for the extra copy remain in the code nevertheless. Exactly this happens with the AC++-generated wrapper functions, whose purpose was explained in Listing 4.2.

4.3.2.3. Context Retrieval via the Join-Point API

The overhead to retrieve join-point–specific context is also relatively low (benchmark *jpapi*, Table 4.5). Compared to *plain* advice, only 0–6 extra cycles are consumed to provide access to context with G++, and even less with ICC. Accessing context that is implicitly available at the join point (such as argument and result values) does furthermore not induce any additional stack cost. The optimizer replaces the required join-point–context data generated for this purpose, such as the array of argument references, with direct access

[15] Table 4.5 numbers are not directly comparable with those from Table 4.3 and Table 4.4, as they include the cost of the method call itself.

G++-3.3.5				
pointcut function	**Δ cycles**	**Δ stack**	**Δ code**	**Δ data**
cflow()				*4*
enter + leave	*6*	*16*	*8*	
test	*12*	*52*	*56*	
that()	*10*	*24*	*128*	*50*
target()	*10*	*24*	*128*	*50*

Table 4.6.: AspectC++ microbenchmark *dynamic*.
Cost (clock *cycles*, *stack*, *code*, and *data* bytes) of the not-statically evaluable pointcut functions *cflow()*, *that()*, and *target()*. Δ denotes the difference to a plain *execution()* pointcut. Results with G++-3.3.5. For methodical details see Table 4.3.

to the referenced parameters. Only "extra" context, such as the pointer to the affected instance (returned by `tjp->that()`) or the pointer to the target of the call (returned by `tjp->target()`), requires additional stack space (4 bytes each).
The results show the benefits of the "pay as you use" context tailoring performed by AC++. The additional 4 stack bytes to store the pointer returned by `tjp->that()` are only consumed if the advice code actually uses `tjp->that()`.

4.3.2.4. Not-Statically Evaluable Pointcuts

Compared to these numbers, the overhead of dynamic pointcut functions is relatively high (benchmark *dynamic,* Table 4.6). As pointed out in Section 4.1.3, not-statically evaluable pointcuts induce an overhead on principle, because they can not be resolved completely at weave-time.

For the `cflow()` pointcut function, a run-time counter has to be maintained, which takes 4 additional bytes of data. For every pass through the observed control flow (*enter+leave)*, this counter has to be incremented and decremented, which takes 6 CPU cycles altogether. The pointcut evaluation itself, which basically is a null-test against the counter (*test*), takes 12 CPU cycles at run-time. For both cases, the stack overhead of 16 respectively 52 bytes is higher than necessary. Again this is a problem of the G++ inliner/optimizer; superfluous stack reservations can be found in the object code.

The pointcut functions `that()` and `target()` (not to be mixed up with the equally named join point API methods) yield all join points in the dynamic control flow where the *run-time type* of an object instance affected by advice (`that()`), respectively receiving a call (`target()`), matches a given type. They require a dynamic type test for evaluation, for which AC++ inserts a virtual function into all relevant classes. The additional 10 cycles of CPU and 24 bytes of stack are basically the cost of calling this virtual function. The more than 120 extra bytes of code and 50 additional bytes of data can be considered a "worst case scenario" if `that()` and `target()` are applied to classes that neither contain a virtual function table nor a constructor. If applied to a class hierarchy that already uses

virtual functions, the cost would be lower, as the compiler would not have to generate support for a new virtual function table, but only extend an existing table.

4.4. Discussion of Results

In the following, I discuss the achieved results and insights of the language level with respect to their greater effects on the qualitative and quantitative dimensions of configurability.

4.4.1. Qualitative Effects

The qualitative effects towards a better configurability in system-software product lines are difficult to extrapolate from the *language level*. However, what can be concluded is that AOP extends on the expressiveness of existing programming paradigms, namely object-oriented programming (OOP) and generic or generative programming (GP) by the mechanisms behind obliviousness and quantification (*inversion of control flow specification by advice* and *implicit per–join-point instantation of advice*).

Inversion is promising for the implementation of granularity and variability in the solution space of a software product line. Components thereby can integrate *themselves* by advice, which may reduce internal coupling. We will see examples for this idiom, which is called *advice-based binding*, in Chapter 6, where it is applied extensively in the development of the CiAO operating-system product line.

Implicit instantiation is also promising for loose coupling, but even more important for the implementation of the architectural policies of an operating system. These polices typically crosscut homogeneously with the implementation of many other system components. We will see application examples for this in the "eCos" study in Chapter 5 and in the chapter about the CiAO operating-system product line in Chapter 6.

4.4.2. Quantitative Effects

A major motivation for the in-depth analysis of the *language level* was to get a detailed understanding of the cost induced by AOP. The results are, overall, promising:

The inherent cost of aspects. AOP can be applied as a "mostly statical" approach. Given an "ideal weaver" as described in Section 4.1.3, the cost of an AOP-based, separated implementation of some concern should be identical to the cost induced by an "in-place" solution. In most cases, the extra expressiveness of AOP does not come at the price of a run-time and memory overhead.

Special care, however, has to be taken with AOP features that are not resolvable statically and, hence, induce some overhead at run-time. For not-statically evaluable

pointcuts, based, for instance, on pointcut functions for dynamic control flow or type filtering (`cflow`, `target`, `that`), an overhead is inevitable. The same holds for join-point–specific context that is not just provided, but explicitly generated by the aspect weaver.

Even though these overheads are AOP-related: We should not consider them as AOP-caused. Any "in-place" implementation of similar functionality would obviously induce similar cost. The AOP features, however, imply a danger of hiding this cost; aspect developers have to be aware of these effects.

The actual cost of aspects. The AspectC++ weaver AC++ comes very close to an "ideal weaver". This is especially true for AC++1.0PRE1, which overcomes the deficiences of previous versions when applying around-advice. In the microbenchmarks, advice given to statically evaluable pointcuts now induces only a very small overhead. The remaining overhead could be traced back to unnecessary stack allocations by the back-end compiler. There is little that can be done on the weaver side here, we can only hope that compiler vendors improve on this situation in future versions of their compilers.[16]

The not-statically evaluable pointcut functions `cflow()`, `that()`, and `target()` induce notable CPU and memory cost. This cost is pay-as-you-use; it remains difficult, though, to assess how well AC++ compares in these cases to an "ideal weaver". Nevertheless, the conclusion is clear: In a resource-critical domain, such as embedded systems, these features should be avoided.

As expected, the usage of join-point–specific "extra" context induces some cost as well. Again, AC++ behaves like an ideal weaver in these cases and offers a strict "pay-as-you-use" tailoring of the context.

The cost of generic advice and advice inlining. Generic advice, the AspectC++ mechanism for advice polymorphism, in principle implies a danger of code bloating. The reason is that generic advice is instantiated per join point, which might result in a high number of (similar) template instantiations, each being compiled separately into the machine code. As advice code is instantiated per–join-point anyway, it is furthermore generated as inline functions.

Code bloating by an (accidentally) large number of template instantiations is a general and well-known issue in the C++ domain [Han97]. There have been attempts to address this issue by better compiler and linker technology (e.g., [SBB02]), however, such technology is not yet state of the art. It is difficult to judge the effects that code bloating by generic advice has on real applications as this depends on many other properties, especially the size of the compiled advice code and the different contexts it is instantiated for. The generative tracing aspect from Section 4.2.2.2, for example, instantiates relatively expensive streaming code. If this aspect is quantified

[16] Getting rid of stack reservations that are no longer necessary *after* optimization seems to be a tough problem, though. All analyzed back-end compilers suffer from this problem – and it is still present in recent G++ versions (tested with G++-4.2).

over a large number of join points, code bloat effects will arise.[17] Tracing, however, is an extreme case. Most aspects, such as the observer and caching examples, do neither embed as much external code nor are they quantified over a similar extent of join points.

Whether there is a danger of code bloat or not – we have no alternative to static instantiation. Static genericity is part of the C++ philosophy, and it is for good reasons – it leads to very flexible and efficient code. By advice-code inlining and static instead of dynamic genericity, we basically trade data, stack, and CPU efficiency for code overhead. For the embedded systems domain, this is not too bad a deal: As shown in Section 2.1.2, SRAM is roughly 10 times more expensive than flash memory.

To conclude: AOP with AspectC++ seems to be feasible even in highly resource-constrained environments. Most AOP features do not induce an overhead. Nevertheless, some care has to be taken with respect to not-statically evaluable pointcuts, extensive usage of join-point context, and potential advice code bloat effects. These insights, as well as the significant improvement in the implementation of around-advice, underline the value of the extensive cost analysis.

4.5. Further Related Work

In the following, I discuss some work that is not particularly related to the topic of this thesis, but to specific sub-topics of this chapter.

4.5.1. AOP in Pure C++

Several attempts have been published that aim to "simulate" AOP concepts in pure C++ using advanced template techniques or macro programming [CDE01, Ale02, Dig04]. In these publications it is frequently claimed that in the case of C++ a dedicated aspect language is not necessary: The instantiation of advice code (according to a specific join point at weave time) and the instantiation of a template or macros (according to a set of parameters at compile time) are similar processes. However, the "code instantiation focus" is a too operational view on the ideas of AOP that neither provides obliviousness nor quantification.

[17] This very much depends on the actually involved types and the implementation of the C++ run-time library. To give a ball-park figure: With VisualC++ 2003, the tracing aspect induces around 200 bytes of extra code for a function with two parameters and up to 1000 bytes of extra code for a function with 11 parameters.

4.5.2. Generic Advice in Other Aspect Languages

The AOSD community has suggested several approaches to increase the static genericity of AspectJ and other aspect languages. KNIESEL and RHO provide an excellent overview of the (big) topic of "generic aspect languages" [KR06]. Most "generic aspect languages" are based on an integration of logic meta-programming into the join-point–description and advice-definition language [DJ04, GB03, ZHS04, HU03, KRH04]. In these languages, logic meta-variables can be (either explicitly or implicitly) bound to matching program entities and then be used for generic advice code or introductions. *Sally* [HU03] focuses on genericity for structural aspects and proposes *parametric introductions* as an extension of the inter-type declaration mechanism in AspectJ and Hyper/J. *LogicAJ* [KRH04] supports a similar mechanism called *generic introductions*, together with generic advice and reasoning over nontype program entities, such as methods, fields, or even expressions. The C++ template mechanism, which is used for generic advice in AspectC++, supports "meta variables" for types and (compile-time) integer expressions only. Hence, the LogicAJ approach is a lot more powerful than the generic advice mechanism in AspectC++. However, despite all advantages, the use of logic meta-programming also leads to a very high level of complexity. For C++, which already is a fairly rich language, such an approach implies the risk of introducing redundant language concepts. The goal of AspectC++ is a *careful integration* of AOP concepts with the *existing* idioms and the philosophy of the C++ language.

4.5.3. Aspect Language Overhead

Some related work regarding *cost of AOP* has been conducted in the AspectJ domain. DUFOUR and colleagues have presented a benchmark suite to measure the dynamic behavior of AspectJ programs [DGH+04]. Their work focuses on a novel measuring approach; however, it also shows that several AspectJ features induce significant overhead. Based on these insights, AVGUSTINOV and associates have suggested some improvements for the AspectJ code generation [ACH+05] that would specifically reduce the overhead caused by *cflow* and *around* in AspectJ programs. JOHANSEN and colleagues describe an approach for "zero-overhead composable aspects" for C# and .NET by intentionally supporting only statically evaluable pointcuts in ther *Yiihaw* weaver [JSS07].

4.6. Chapter Summary

The goal of this chapter was to get a deep understanding of the technical issues of AOP in general and AspectC++ in particular. The results from the *language level* show that AOP does not induce an inherent overhead that makes it *per se* unacceptable for the domain of efficient system software. Key to success, however, is to bind advice to join points at compile-time whenever possible, that is, to avoid not-statically evaluable pointcuts.

With the introduction of *generic advice*, AspectC++ now provides the efficiency of compile-time binding in conjunction with aspect genericity. AspectC++ thereby fulfills an important precondition for consideration as a technology towards better separation of concerns, decomposability, and configurability in system software.

While these micro-benchmark effects are promising, they provide still limited insight into the effects of a larger application of aspects. Still open is the question if aspects do really lead to qualitative benefits such as better encapsulation and configurability of concern implementations in system software. Are the quantitative effects still negligible if AOP provides such benefits and is applied in a larger scale? To answer these questions, studies with real software product lines are necessary.

5

Implementation Level – Aspects in Action: Practicing Configurability by AOP

On the *language level,* AOP with AspectC++ provides qualitative benefits (an increased expressiveness we assume to be beneficial for configurability) without significant quantitative disadvantages (overhead compared to an identical "tangled" implementation). We also learned, however, that some language features do induce costs, and therefore should be avoided in the development of software for resource-thrifty embedded systems.

In this chapter, we will lift up this knowledge to the *implementation level* of configurability. The goal is to understand under which circumstances the assumed benefits hold – respectively, can be realized at all – when we use aspects on a larger scale to implement configurability in system software for embedded devices. Are the quantitative effects still negligible if we affect hundreds or thousands of join points? Is the expressiveness of AOP really sufficient to replace conditional compilation as the dominant implementation technique for configurability? Can we really do without the "expensive" AOP features? To answer these questions, I conducted a larger case study with the well-known eCos operating system; in this chapter I describe and discuss the results.

The chapter is structured as follows: I start with an overview on the design of the "eCos" study in Section 5.1, which is followed by the presentation of the study itself in Section 5.2 and Section 5.3. I discuss the results and their relevance for the aspect-aware development of system software in Section 5.4; finally, the chapter is summarized in Section 5.5.

Related Publications

The ideas and results presented in this chapter have partly also been published as:

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM Press.

5.1. Case Study "eCos" – Objectives and Study Design

The overall goal of the "eCos" study was to evaluate and understand the effects and suitability of AOP for the implementation of configurability in operating-system product lines for embedded devices. In other words, the goal was to increase the *awareness* regarding the application of aspects in such systems:

Objective 1: Analyze how kernel policies manifest themselves in the source code. Evaluate if we can implement them by aspects.

Objective 2: Analyze how fine-grained configuration options manifest themselves in the source code. Evaluate if we can implement them by aspects.

Objective 3: Analyze the quantitative effects of a large-scale utilization of aspects to implement configurability. Evaluate if AOP can compete with the dominant technique of conditional compilation.

Objective 4: Evaluate if kernel policies actually become *more* configurable by aspects. Analyze the factors that hinder or enable configurability by aspects.

Each of the above objectives required some analysis and some evaluation work to be carried out. The evaluation work is actually *constructive* – in the sense that the question is operationalized by "trying to implement it by AOP". Hence, for each objective, an *analytical* part and a *constructive* part had to be reflected in the actual study design. This led to the "eCos" study.

The "eCos" study is an *analytical and constructive* case study. It is primarily analytical, as it is based on an existing software product line, namely the eCos embedded operating system. We have already seen and discussed the first part – the analysis – of this study in Chapter 3, where we figured out how the configuration of central kernel policies and fine-grained configuration options becomes manifest in the kernel source code (objectives 1, 2; analytical part). In the following second part of the study (Section 5.2), we now evaluate if and how this configurability can be refactored into aspects (objectives 1, 2; constructive part). The resulting *AspeCos* is functionally identical to the original eCos kernel, but uses aspects instead of conditional compilation to implement the refactored concerns. This permits a fair comparison between AOP and conditional compilation as a means to implement configurability (objective 3). For the third part of the study, AspeCos has been extended by extra configuration options regarding kernel policies (objective 4); these extensions and their implementation issues are discussed in Section 5.3.

5.2. Aspectizing the eCos Kernel

eCos, the *embedded Configurable operating system* [eCo, Mas02], is a highly configurable open-source operating-system product line targeted for the market of embedded systems. We have learned about eCos, its configuration approach, and its kernel in Section 3.1.1.

We have also analyzed the scattering of 12 configuration options (8 thread options plus 4 mutex options) and three central kernel policies (Tracing, Instrumentation, Synchronization) over the kernel source base – and found an "`#ifdef` hell" (Listing 3.1).

With respect to these problems the next logical step was the constructive part of objectives 1 and 2 (Section 5.1) – the refactoring of the identified policies and configuration options as far as possible into aspects. This resulted in the *AspeCos* system, an "aspectized" version of the eCos system in which the enforcement of system policies and configuration options is implemented by means of AOP. In the following sections, I describe these refactorings and their qualitative and quantitative effects on the eCos kernel.

5.2.1. Refactoring Concerns into Aspects

AspeCos was derived from eCos by means of refactoring. With respect to objective 3 – the quantitative comparison of eCos and AspeCos – only *gentle* refactorings were permitted in this process: only the *where* (the distribution of concerns) was to be improved; the actual concern implementations (the *what*) were not to be changed. Basically, refactoring was done by mechanically carrying out the following three steps for each identified policy enforcement and `#ifdef` block:

1. Cut the code related to the concern from the affected function.
2. If some advice for this concern does already exist, then extend the pointcut the advice is given to.
3. Otherwise create a new pointcut and an advice definition for this concern, and
   a) paste the code related to the concern into the advice body
   b) substitute any context access (e.g., to actual parameter values or the object instance) with calls to the join point API.

This self-imposed restriction regarding the refactoring in some cases had the consequence that a concern could not entirely be factored out.

5.2.1.1. Refactoring of Policies

The above holds for all of the identified configuration options and kernel policies, but Tracing. As a general-purpose and I/O-intensive development concern, I did not consider Tracing as a reasonable subject for overhead comparisons in an operating-system kernel. Hence, Tracing was not only refactored, but also improved. The AspeCos implementation of tracing (due to the flexible match mechanism of AspectC++) provides a much higher coverage together with a better tailorability. This can be seen from the (maximal) number of affected join points in Table 5.1, which is way above the number of tracing macro invocations in the original eCos.

In the following, I therefore focus on the more performance-critical and kernel-related Synchronization and Instrumentation concerns:

	(a) eCos		(b) AspeCos			
	lines of code		lines of code		join points	
policy	**#**	**%**	**#**	**%**	**#**	**%**
Tracing	*336*	*6.5*	*4*	*0.1*	*632*	*67.9*
Instrumentation	*162*	*3.1*	*3*	*0.1*	*139*	*14.9*
Synchronization	*187*	*3.6*	*0*	*0.0*	*160*	*17.2*
policy total	*685*	*13.2*	*7*	*0.2*	*931*	*100*
nonpolicy total	*4520*	*86.8*	*4520*	*97.7*	-	-
kernel total	*5205*	*100*	*4527*	*100*	*931*	*100*

Table 5.1.: Comparison of kernel policy enforcement in eCos and AspeCos.
Listed is the number and percentage of policy-related macro and function invocations in the C++ parts of the kernel code base (`.cxx`, `.hxx`, `.inl` files). **(a)** Results from the original eCos code base (reprinted from Table 3.2). **(b)** Results from the AspeCos code base, together with the number of affected code join points. Macro or function invocations from within the aspects (`.ah` files) are considered to be well separated and were not counted.
[Lines of code counted with CCCC [Lit], numbers of join points obtained from the AC++–generated join point repository]

(Kernel) Synchronization. Refactoring Synchronization was relatively simple. Synchronization homogeneously crosscuts the implementation of other concerns; hence, the same piece of advice can be applied to all join points:

```
aspect int_sync {
  pointcut sync() = execution(...) // kernel calls to sync
           || construction(...)
           || destruction(...);

  // advise kernel code to invoke lock() and unlock()
  advice sync() : before() {
    Cyg_Scheduler::lock();
  }
  advice sync() : after() {
    Cyg_Scheduler::unlock();
  }
  // In eCos, a new thread always starts with a lock value of 0
  advice execution(
  "%Cyg_HardwareThread::thread_entry(...)") : before() {
    Cyg_Scheduler::zero_sched_lock();
  }
  ...
};
```

Before- and after-advice is used to superimpose the invocation of `Cyg_Scheduler::lock()` and `Cyg_Scheduler::unlock()` into the execution[1] of

[1] Some kernel functions are actually class constructors or destructors, for which the respective pointcut functions are used instead of `execution()`.

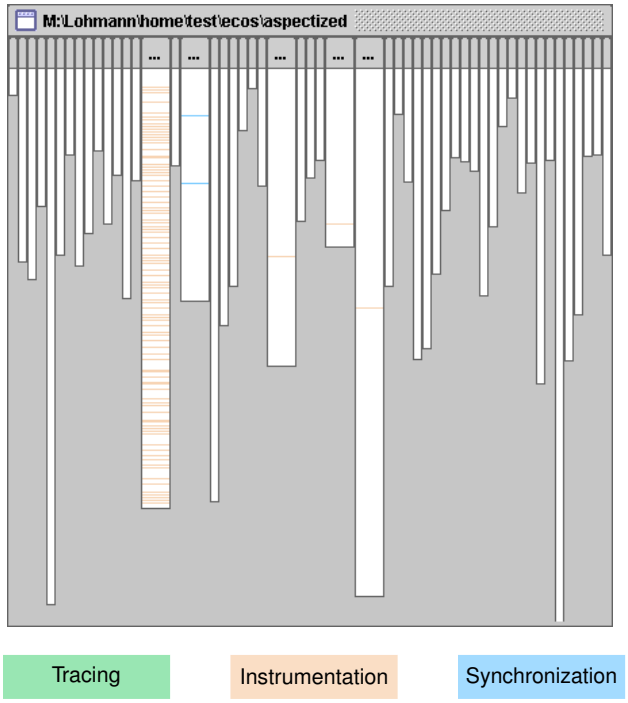


Figure 5.1.: Distribution of policy enforcement in the AspeCos kernel source base.
Each bar represents a single AspectC++ file (`.hxx`, `.cxx`, `.inl`, `.ah`) from the kernel source base. Compared to the original eCos (Figure 3.3), policy enforcement is much better localized. Instrumentation and Synchronization are separated (almost) completely in the new aspect headers `instrumentation_kernel.ah` and `sched.ah` (the two leftmost broad bars). For Instrumentation, three invocations of a `CYG_INSTRUMENT_...` macro could not be factored out from the original code base (the other three broad bars).

all kernel functions that require synchronization. Overall, 160 code join points are affected (Table 5.1). Note that the number of join points in AspeCos is below the number of the corresponding function calls in eCos. In the original, some functions contained more than one exit point, each with a call to `Cyg_Scheduler::unlock()`. After-advice, however, implicitly affects all exit points of a function.

Additionally (not shown above), the functions that implement locking itself have been refactored from the scheduler class into a set of introductions; the `int_sync` aspect now provides a hundred percent encapsulation of Synchronization.

(Kernel) Instrumentation. The refactoring of Instrumentation required a bit more work. Instrumentation inhomogeneously crosscuts the implementation of other concerns. For each kernel abstraction, eCos defines its own set of instrumentation macros. In the refactored version, the invocation of the particular macro is given as advice to the affected kernel functions such as follows:

```
aspect kernel_instrument_mutex {
  ...
  advice execution("% Cyg_Mutex::lock(...)") : after() {
```

```
    if(*tjp->result()) {
      CYG_INSTRUMENT_MUTEX(LOCKED,tjp->that(),0);
  } }
  advice call("% Cyg_Thread::wake(...)")
        && within("% Cyg_Mutex::unlock(...)") : after() {
    CYG_INSTRUMENT_MUTEX(WAKE,tjp->that(),tjp->target());
  }
  ...
};
```

In the advice bodies, the join-point API is used to retrieve the object instances involved in a particular event. Overall, 13 aspects with 85 code advice definitions, such as above, affect a total of 139 join points (Table 5.1). Again, the number of affected join points is below the number of original macro invocations due to multiple exit points in some functions.

Because of the self-imposed restriction to do only gentle refactorings, Instrumentation could not be refactored completely into aspects. Three out of 162 invocations remained in the code, all of which are related to the scheduler implementation. In these cases, the invocation of a `CYG_INSTRUMENT_xxx` macro was conditional, depending on some internal context such as follows:

```
void Cyg_Scheduler_Implementation::timeslice() {
  if( --timeslice_count <= 0 ) {
    CYG_INSTRUMENT_SCHED(TIMESLICE,0,0);

    // Force a reschedule on each timeslice
    need_reschedule = true;
    timeslice_count = CYGNUM_KERNEL_SCHED_TIMESLICE_TICKS;
  }
}
```

It should be clear, however, that with just slightly more elaborated refactorings (in this case: “extract method” on the inner block) a complete separation of Instrumentation had been possible.

Overall, the refactoring of policy enforcement code into aspects was successful; the scattering of this code over the kernel source base could notably be improved in AspeCos. Figure 5.1 visualizes the distribution of Synchronization and Instrumentation in the AspeCos kernel source base.

5.2.1.2. Refactoring of Configuration Options

For AspeCos, each configuration option has been encapsulated into a single aspect that superimposes the functionality into the base component. Additional member functions and state variables specific for a certain configuration option are applied by introductions, option-specific behavior is applied by code advice. Table 5.2 lists the details. Overall, 98

		(a) eCos	(b) AspeCos		
		#ifdef	#ifdef	join points	
	configuration option	blocks	blocks	code	intro
thread options (8)	CYGVAR_KERNEL_THREADS_NAME	*3*	*1*	*2*	*2*
	CYGVAR_KERNEL_THREADS_LIST	*4*	*0*	*2*	*6*
	CYGVAR_KERNEL_THREADS_STACK_LIMIT	*7*	*0*	*3*	*4*
	CYGVAR_KERNEL_THREADS_STACK_CHECKING	*6*	*1*	*8*	*1*
	CYGVAR_KERNEL_THREADS_STACK_MEASUREMENT	*2*	*0*	*2*	*2*
	CYGVAR_KERNEL_THREADS_DATA	*3*	*0*	*1*	*8*
	CYGVAR_KERNEL_THREADS_DESTRUCTORS	*3*	*0*	*1*	*3*
	CYGVAR_KERNEL_THREADS_DESTRUCTORS_PER_THREAD	*11*	*1*	*2*	*3*
	thread configuration options total	*39*	*3*	*21*	*29*
mutex options (4)	CYGSEM_KERNEL_ SYNC_MUTEX_PROTOCOL	*14*	*4*	*9*	*11*
	CYGSEM_KERNEL_ SYNC_MUTEX_PROTOCOL_INHERIT	*5*	*1*	*4*	*3*
	CYGSEM_KERNEL_ SYNC_MUTEX_PROTOCOL_CEILING	*10*	*1*	*4*	*6*
	CYGSEM_KERNEL_ SYNC_MUTEX_PROTOCOL_DYNAMIC	*5*	*0*	*5*	*6*
	mutex configuration options total	*34*	*6*	*22*	*26*

Table 5.2.: Comparison of configuration option enforcement in eCos and AspeCos.
Listed is the number of `#ifdef` blocks in the C++ (respective AspectC++) parts of the kernel code base (`.cxx`, `.hxx`, `.inl` `.ah` files). **(a)** Results from the original eCos code base (reprinted from Table 3.1). **(b)** Results from the AspeCos code base, together with the number of affected join points.
[Lines of code counted with CCCC [Lit], numbers of join points obtained from the AC++–generated join point repository]

join points are affected: 21 code join points and 29 introductions for *thread configuration options*, and 22 code join points and 26 introductions for *mutex configuration options*. The number of join points is above the number of `#ifdef` blocks in the original (73), as some `#ifdef` blocks embrace the definition of multiple identifiers (state variables, member functions), while in the refactored version each identifier is represented by a separate introduction.[2] Some few `#ifdef` blocks (3 for *thread configuration options*, 6 for *mutex configuration options*), all caused by inter-feature dependencies, were not resolved completely, but simply moved into the corresponding aspect implementation. Again, it would have been possible to completely remove these `#ifdef` blocks with more sophisticated refactorings.

Overall, the refactoring of configuration options into aspects was successful; the scattering of the implementation of mutex and thread configuration options, which leads to a true "`#ifdef` hell" in eCos, could be significantly reduced in AspeCos. Figure 5.2 visualizes this on the example of the distribution of mutex configuration options in the AspeCos kernel sources.

[2] This syntax for introductions (one introduced member per piece of introduction-advice) is deprecated in newer AspectC++ versions, where it has been superseded by the *slice* concept (Appendix section A.3.4).

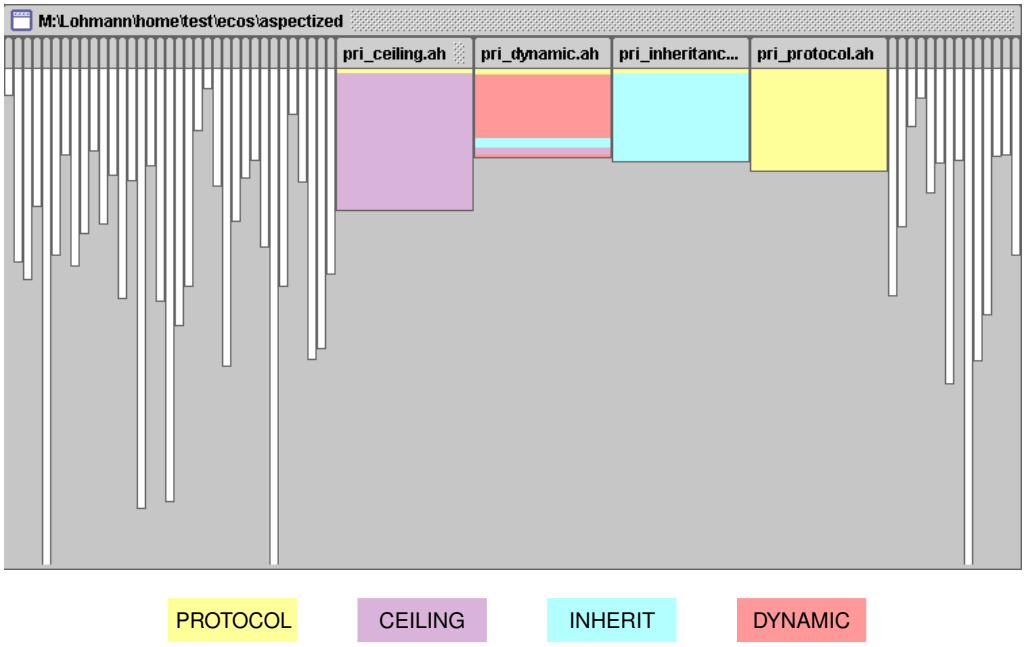


Figure 5.2.: Distribution of mutex configuration options in the AspeCos kernel source base.
Each bar represents a single AspectC++ file (`.hxx`, `.cxx`, `.inl`, `.ah`) from the kernel source base; the colored areas represent lines of code extracted from `#ifdef` blocks for the respective configuration option. Compared to the original eCos (Figure 3.2), the enforcement of the various protocol variants is no longer scattered over the mutex and scheduler implementation files, but was separated out into distinct aspect headers `pri_ceiling.ah`, `pri_dynamic.ah`, `pri_inheritance.ah`, and `pri_protocol.ah`.

5.2.2. The Cost of Large-Scale AOP

In the resulting AspeCos, several kernel policies and various configuration options have been implemented by means of AOP. The performance-critical Synchronization and Instrumentation policies affect around 300 join points; another 100 join points are affected by the refactored configuration options. AspeCos has become a suitable target for the analysis part of objective 3 (Section 5.1) – the analysis of the quantitative effects of a larger-scale utilization of aspects to implement configurability. Furthermore, as AspeCos was derived from eCos by carrying out only gentle refactorings, it has become a suitable target for the evaluation part of objective 3 – the quantitative comparison of an AOP-based with a preprocessor-based implementation.

5.2.2.1. Setup

13 different configurations, each in an AspeCos and an eCos version, were generated and built, including:

- The (Aspect-)eCos basic configuration (*_base*, no additional features), to compare the effects regarding the implementation of Synchronization.

(a) thread

cycles	thread system call
215	cyg_thread_create()
327	cyg_thread_resume()
127	cyg_thread_resume()
274	cyg_thread_yield()
354	cyg_thread_exit()
77	cyg_thread_yield()
91	cyg_thread_resume()
102	cyg_thread_kill()
336	cyg_thread_suspend()
96	cyg_thread_suspend()
53	cyg_thread_suspend()
63	cyg_thread_resume()
115	cyg_thread_resume()
38	cyg_thread_delete()
2268	*total (thread_base)*

(b) mutex

cycles	system call
52	cyg_mutex_init()
47	cyg_mutex_lock()
22	cyg_mutex_unlock()
46	cyg_mutex_try_lock()
49	cyg_mutex_try_lock()
381	cyg_mutex_lock()
429	cyg_mutex_unlock()
17	cyg_mutex_destroy()
1043	*total (mutex_base)*

(c) semaphore

cycles	system call
62	cyg_semaphore_init()
50	cyg_semaphore_post()
723	cyg_semaphore_wait()
416	cyg_semaphore_trywait()
44	cyg_semaphore_trywait()
46	cyg_semaphore_wait
22	cyg_semaphore_post()
19	cyg_semaphore_destroy()
1382	*total (sem_base)*

Table 5.3.: Test applications for the quantitative comparison of AspeCos and eCos.
Listed is the sequence of system calls performed by the *thread*, *mutex*, and *semaphore* test applications. Numbers denote *CPU cycles* taken by the particular system call. CPU cycles differ among several invocations of the same call due to different context-switch operations the kernel has to perform internally. **(a) thread:** three threads activate each other in turns using operations from the kernel thread API. **(b) mutex:** two threads synchronize using kernel mutex objects. **(c) semaphore:** two threads synchronize using kernel semaphore objects.
[eCos original, base configuration, Pentium 3 at 600 MHz. Cycles measured with `rdtsc` and averaged from 10 series of 1000 iterations (σ < 0.1% for all cases). All code compiled with `G++`-3.3.5 using -O3 -mpreferred-stack-boundary=2 -fno-align-functions -fno-align-jumps -fno-align-loops -fno-align-labels -fno-reorder-blocks -fno-prefetch-loop-arrays optimizations.]

- A configuration with additional support for instrumentation (*_instrumentation*), to compare the effects regarding the implementation of Instrumentation.

- Several other configurations, each with one selected extra feature, to compare the effects regarding the refactored *configuration options*.

A set of three multi-threaded test applications that specifically use the affected parts of the eCos kernel was linked against each of the 13 + 13 variants of the eCos library, which results in 58 test application binaries. The test applications themselves are quite simple; their threads just invoke a sequence of system calls and do not perform further calculations. Table 5.3 lists the sequence of system calls performed by each test in conjunction with the number of CPU cycles taken by each system call on a basic eCos system.

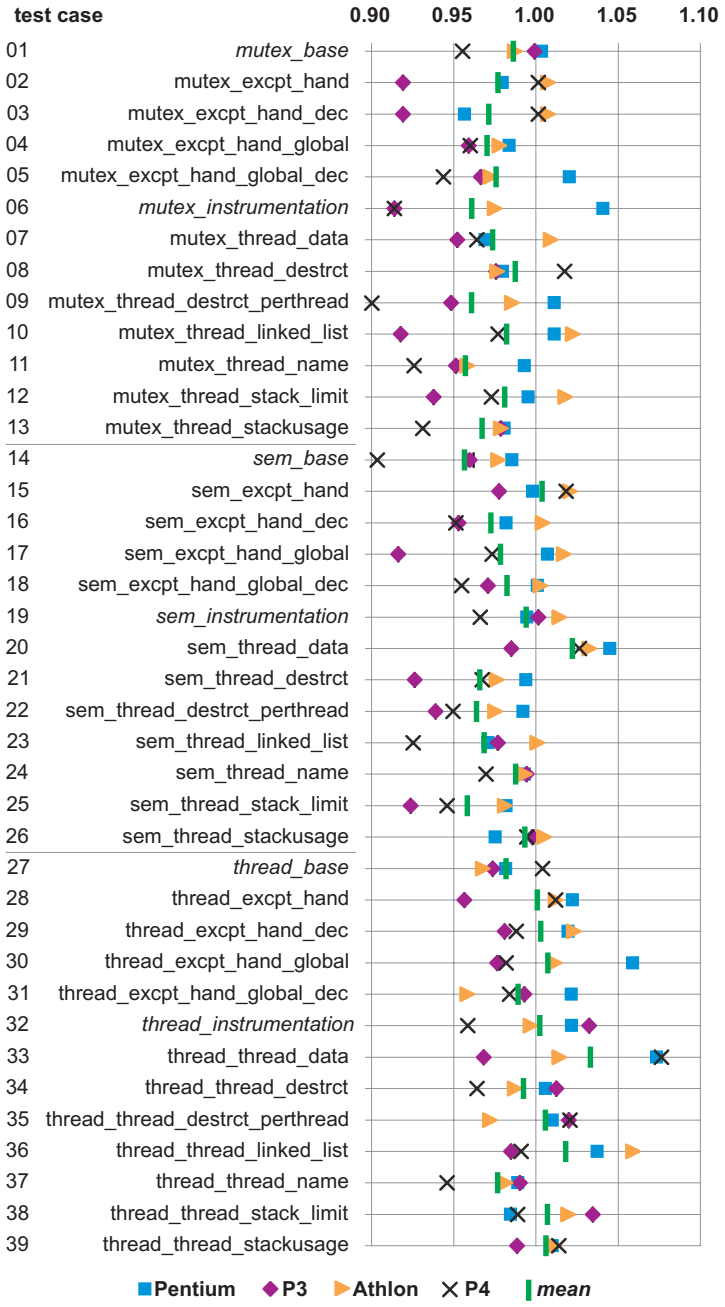


Figure 5.3.: Comparison of the CPU overhead between AspeCos and eCos.
Depicted is the per-test-case AOP runtime cost factor $\left({}^{AspeCos}/_{eCos}\right)$ for 13 different configurations over the three test applications (*mutex*, *sem[aphore]*, *thread*, see Table 5.3) and four different IA32 CPU types. *Mean* denotes the per-test-case average over all CPU types.
[AspeCos woven with AC++1.0PRE1; other details as described in Table 5.3.]

5.2.2.2. Runtime Cost

Runtime cost (*cycles*) was measured in the running test applications and averaged from 10 series of 1000 iterations to reduce cache effects. With a relative standard derivation $\sigma <$ 0.1% for all series, the results can be considered as stable. Besides the P3 CPU, which was the target platform for this study, additional measurements with the same binary images were performed on Pentium, Athlon, and P4 CPUs. The results are depicted in Figure 5.3 with the relative AOP runtime cost factor $(^{AspeCos}/_{eCos})$ for each test case, configuration, and CPU type:[3]

- Over all test cases, the AOP cost factor does vary. The highest variation is in the numbers of the P4 CPU ([0.90, 1.07], σ=0.04); the distribution of all other CPU types is in between (Pentium: [0.96, 1.07], σ=0.03, P3: [0.91, 1.03], σ=0.03, Athlon: [0.96, 1.06], σ=0.02). For an averaged CPU (*mean*), the cost factor is distributed with [0.96, 1.02], σ=0.02.
- Even for each single test case, the AOP cost factor varies noticeably among the different CPU types. The maximum variation is [0.91, 1.04], σ=0.06 (test case 06: *mutex_instrumentation*). The minimum is [0.99, 1.01], σ = 0.01, (test case 39: *thread_thread_stackusage*). In average, the CPU-dependent cost factors are distributed with a standard deviation of σ = 0.03.
- For no single test case, AOP can be considered as clearly beneficial (factor < 0.98) or disadvantageous (factor > 1.02) for all CPU types.
- Over all test cases, a slight tendency towards a beneficial influence of AOP can be observed for the P3 and P4 CPUs, for which the average AOP cost factor is 0.97. For the Pentium and Athlon CPUs, however, no such effect can be found (factor 1.00) – neither so for the averaged CPU (*mean*), for which the factor is 0.99.

We can conclude from these numbers that a general effect of AOP on the runtime cost of AspeCos can not be ascertained. I attribute the observed variation to CPU-internal "performance noise", probably caused by mechanisms such as branch prediction, instruction reordering and memory alignments. The distribution of the per-CPU cost factors, which is stable and reproducible for each single test case, but which differs between different test cases, is a clear indicator for such effects. The amount of cycles consumed by most eCos kernel functions is quite low (Table 5.3), which increases the relative effects of such CPU-internal variability in the measurements. On the other hand, a potential AOP-related overhead should have become evident in the results, as it would affect every CPU type.

To gain additional evidence, I analyzed and compared the generated machine code of AspeCos and eCos in the *_base* configuration (test cases 01, 13, 27). From the 28 analyzed kernel functions that are called directly or indirectly from the test applications

[3] Depicted results were measured with enabled CPU caches. Additional measurements with caching disabled resulted in a much higher standard deviation (probably caused by DRAM-timing and bus-load effects), but basically the same average cost factors.

and that are affected by Synchronization, 19 functions turned out to be actually *identical* in the machine code.[4] Eight functions turned out to be minimally different with respect to instruction ordering and other differences in sub-sequences of up to four instructions. I found major differences between AspeCos and eCos only in the function `Cyg_Scheduler_Implementation::add_thread()`. In AspeCos, more member functions were inlined. I attribute this to the compiler-internal heuristic regarding inlining of functions that are not explicitly marked as `inline`. With the synchronization code moved into aspects, the size of `add_thread()` had probably fallen under a compiler-internal threshold, which caused the extra inlining. This effect seems to be even more prevalent in the other configurations.

Overall, the enforcement of kernel policies and configuration options by AOP instead of conditional compilation does neither lead to better nor to worse performance.

5.2.2.3. Memory Cost

The additional inlining performed by the compiler in AspeCos can as well be observed in the memory overhead. Table 5.4 lists the absolute differences ($\Delta = AspeCos - eCos$) between AspeCos and eCos for all 39 configurations.

In almost all cases, AspeCos induces some *ROM* overhead. Larger code section cause this overhead, which is 0.9 percent in average. The *instrumentation* test cases (06, 19, 32), adding more than 500 extra code bytes (an overhead of three percent) show the maximum difference. Most of this overhead can again be attributed to inlining effects. In eCos, the compiler does not embed the final call to `Cyg_Scheduler::unlock()` if a function is affected by Instrumentation. Instead, a `jmp` statement to a shared copy of `Cyg_Scheduler::unlock()` is generated. In AspeCos, the call is always embedded and no code sharing takes place, regardless of Instrumentation being enabled or not.

Generally, AspeCos also shows some stack overhead: In average, 1.3 percent more stack bytes are used. This is mainly caused by (unnecessary) reservations for call-by-value parameters, as explained in Section 4.3. The generally higher overhead of the *mutex* and *semaphore* test cases (compared to *thread*) can be explained by this effect as well, as the `Cyg_Mutex` and `Cyg_Semaphore` operations lead to higher call depths inside the kernel.

According to the AspectC++ microbenchmark results from the *language level* (Section 4.3), some additional stack usage can also be expected if the advice code accesses context information via the join point API. Surprisingly, test cases affected by Instrumentation (06, 19, 32), which makes extensive use of the join point API, show a very low stack overhead. The reduced call-depth (due to inlining) seems to compensate any AOP-induced overhead in these cases.

No differences can be observed regarding *RAM* utilization. As our aspects are stateless, the linker seems to be able to omit even the aspect instances from the final image.

[4] if ignoring differences regarding symbol addresses and function-wide register allocation.

test case		ROM Δ	%	RAM Δ	%	stack Δ	%
01	*mutex_base*	*101*	*0.6*	*0*	*0.0*	*24*	*2.0*
02	mutex_excpt_hand	*55*	*0.3*	*0*	*0.0*	*24*	*2.0*
03	mutex_excpt_hand_dec	*-35*	*-0.2*	*0*	*0.0*	*24*	*2.0*
04	mutex_excpt_hand_global	*92*	*0.5*	*0*	*0.0*	*24*	*2.0*
05	mutex_excpt_hand_global_dec	*85*	*0.5*	*0*	*0.0*	*24*	*2.0*
06	*mutex_instrumenation*	*543*	*3.0*	*0*	*0.0*	*4*	*0.3*
07	mutex_thread_data	*74*	*0.4*	*0*	*0.0*	*28*	*2.3*
08	mutex_thread_destrct	*74*	*0.4*	*0*	*0.0*	*28*	*2.3*
09	mutex_thread_destrct_perthread	*224*	*1.3*	*0*	*0.0*	*24*	*2.0*
10	mutex_thread_linked_list	*88*	*0.5*	*0*	*0.0*	*8*	*0.6*
11	mutex_thread_name	*146*	*0.9*	*0*	*0.0*	*16*	*1.3*
12	mutex_thread_stack_limit	*311*	*1.8*	*0*	*0.0*	*24*	*2.0*
13	mutex_thread_stackusage	*341*	*2.0*	*0*	*0.0*	*24*	*2.0*
14	*sem_base*	*91*	*0.5*	*0*	*0.0*	*24*	*2.0*
15	sem_excpt_hand	*51*	*0.3*	*0*	*0.0*	*24*	*2.0*
16	sem_excpt_hand_dec	*-54*	*-0.3*	*0*	*0.0*	*24*	*2.0*
17	sem_excpt_hand_global	*88*	*0.5*	*0*	*0.0*	*24*	*2.0*
18	sem_excpt_hand_global_dec	*81*	*0.5*	*0*	*0.0*	*24*	*2.0*
19	*sem_instrumenation*	*588*	*3.1*	*0*	*0.0*	*0*	*0.0*
20	sem_thread_data	*70*	*0.4*	*0*	*0.0*	*28*	*2.3*
21	sem_thread_destrct	*70*	*0.4*	*0*	*0.0*	*28*	*2.3*
22	sem_thread_destrct_perthread	*220*	*1.3*	*0*	*0.0*	*24*	*2.0*
23	sem_thread_linked_list	*84*	*0.5*	*0*	*0.0*	*8*	*0.6*
24	sem_thread_name	*142*	*0.8*	*0*	*0.0*	*16*	*1.3*
25	sem_thread_stack_limit	*307*	*1.8*	*0*	*0.0*	*24*	*2.0*
26	sem_thread_stackusage	*337*	*2.0*	*0*	*0.0*	*24*	*2.0*
27	*thread_base*	*104*	*0.6*	*0*	*0.0*	*12*	*0.7*
28	thread_excpt_hand	*58*	*0.3*	*0*	*0.0*	*12*	*0.7*
29	thread_excpt_hand_dec	*-32*	*-0.2*	*0*	*0.0*	*12*	*0.7*
30	thread_excpt_hand_global	*95*	*0.6*	*0*	*0.0*	*12*	*0.7*
31	thread_excpt_hand_global_dec	*88*	*0.5*	*0*	*0.0*	*12*	*0.7*
32	*thread_instrumenation*	*563*	*3.1*	*0*	*0.0*	*-4*	*-0.2*
33	thread_thread_data	*77*	*0.4*	*0*	*0.0*	*12*	*0.7*
34	thread_thread_destrct	*77*	*0.4*	*0*	*0.0*	*12*	*0.7*
35	thread_thread_destrct_perthread	*227*	*1.3*	*0*	*0.0*	*12*	*0.7*
36	thread_thread_linked_list	*91*	*0.5*	*0*	*0.0*	*-12*	*-0.6*
37	thread_thread_name	*149*	*0.9*	*0*	*0.0*	*0*	*0.0*
38	thread_thread_stack_limit	*333*	*1.9*	*0*	*0.0*	*12*	*0.7*
39	thread_thread_stackusage	*344*	*2.0*	*0*	*0.0*	*12*	*0.7*

Table 5.4.: Comparison of the memory overhead in AspeCos and eCos.
Listed is the absolute [Byte] and relative overhead of AspeCos compared to eCos ($\Delta = AspeCos - eCos$, $\% = {}^{\Delta}/_{eCos} \cdot 100$) with respect to dynamic memory utilization (*stack*, accumulated from all threads) and static memory utilization (*ROM* = code + data + rodata, *RAM* = data + bss).
[Dynamic memory utilization was measured byte-exact in the running test applications. Static memory utilization was retrieved off-line and byte-exact from the linker map files.]

Overall, the enforcement of kernel polices and configuration options by AOP instead of conditional compilation does not lead to a significant (>2%) memory overhead.

5.3. Improving eCos Configurability by AOP

The implementation and enforcement of several central kernel policies that were scattered over the source base in eCos could be separated out into distinct aspects in AspeCos. Thereby, AspeCos also became a good target for experiments regarding objective 4 (Section 5.1) – the evaluation if an implementation with AOP actually improves on the configurability of kernel policies.

5.3.1. Turning Synchronization and Preemption into Optional Features

In eCos, Synchronization is a mandatory policy and threads are always preemptable (Section 3.1.1.2). We have seen, however, that there are good cases for understanding Synchronization and Preemption as optional features (Section 3.1.2). The first experiment therefore was aimed at improving the *granularity* of AspeCos with respect to these concerns. The goal was to add support for the following application cases:

1. DSRs are not used, we only have threads: DSR support already is an optional feature in eCos. However, if DSRs are not used, Synchronization is not necessary either. Hence, we want to be able to select Synchronization as an optional feature as well.
2. Threads are not used, we only have DSRs: This is exactly the opposite of application case 1: Synchronization is not needed either if the system is completely implemented by DSRs and does not use any threads.
3. Preemption of threads is not required, all threads are run-to-completion: For this application case it would be preferable to configure eCos for nonpreemptive scheduling, which would also offer the opportunity to share stack space between threads. Hence, we want thread preemptability represented as an optional Preemption feature.

In eCos, omitting Synchronization is hardly possible. The calls to `Cyg_Scheduler::lock()` and `Cyg_Scheduler::unlock()` are "hard-wired" in the kernel source base. In AspeCos, however, turning Synchronization into an optional feature should be easy: Theoretically, all that is necessary is to omit the `int_sync` aspect from the output generated by ECOSCONFIG.

The closer analysis of the actual implementation revealed that this is by no means that simple. The reason is the way another concern, Preemption, is enforced (not to say: hidden) inside the eCos kernel. Preemption is based on the following requirements:

- Whenever a thread becomes ready (for example, if the current thread releases a mutex for which some other thread has been waiting) this is a potential point of preemption. If the thread that just became ready has a higher priority than the current thread it should immediately get the CPU.

(a) Synchronization

```
void Cyg_Alarm::enable() {
  // Prevent DSR execution
  Cyg_Scheduler::lock();
  if( !enabled ){
    // ensure the alarm time
    // is in our future:
    synchronize();
    enabled = true;
    counter->add_alarm(this);
  }
  // Unlock the scheduler and propagate
  // DSRs. (No thread was set ready, so
  // this is no point of preemption.)
  Cyg_Scheduler::unlock();
}
```

(b) Synchronization and Preemption

```
void Cyg_Mutex::unlock() {
  // Prevent preemption and DSR execution
  Cyg_Scheduler::lock();
  if( !queue.empty() ) {
    Cyg_Thread *thread = queue.dequeue();
    thread->set_wake_reason(
      Cyg_Thread::DONE);
    thread->wake();
  }
  locked      = false;
  owner       = NULL;
  // Unlock the scheduler, propagate DSRs
  // and maybe switch threads
  Cyg_Scheduler::unlock();
}
```

Listing 5.1: Join point ambiguity with respect to Synchronisation and Preemption in the eCos kernel.
Because `Cyg_Scheduler::unlock()` is used to enforce Synchronisation *and* Preemption, the related execution join points are ambiguous. **(a)** The execution of `Cyg_Scheduler::unlock()` represents a join point for Synchronization only. **(b)** The execution of `Cyg_Scheduler::unlock()` represents a join point for Synchronization and Preemption.

- However, in most cases some additional state changes (such as clearing the owner field of the mutex) have to be propagated first to ensure consistency of kernel data. The actual preemption of the running thread must be delayed until these state changes have been carried out and the internal kernel state is guaranteed to be consistent again.

Kernel state consistency has been reached, by definition, whenever `Cyg_Scheduler::unlock()` is invoked.[5] The eCos designers exploited this fact for a "clever optimization": They piggybacked the enforcement of Preemption on the implementation of Synchronization: Internally, `Cyg_Scheduler::unlock()` does not only reactivate DSR execution (Synchronization), but also invokes the scheduler (Preemption).

The result is ambiguity: Apparently, 50 of the 101 invocations of `Cyg_Scheduler::unlock()` that can be found in the eCos kernel sources represent only a point of Synchronization, similar to Listing 5.1.a. The other 51 invocations represent as well a point of Preemption, comparable to Listing 5.1.b. However, this is not distinguishable from the affected join points!

The unaesthetic consequence is that the `int_sync` aspect does not only implement Synchronization, but also parts of Preemption. Even in AspeCos it is not easily possible to leave out just one of them – which would be necessary to support application cases 1 and 3. Application case 2, however, can be supported. By extracting the code into the `int_sync` aspect it became at least possible to leave out Synchronization and Preemption *together*.

Overall, the extraction of Synchronization into an aspect did not lead to much better configurability with respect to this policy.

[5] Otherwise DSRs could run on inconsistent kernel state.

5.3.2. Adding a New Feature: The Kernel Stack Aspect

The second experiment was about improving the variability of eCos by adding a new kernel-stack policy. The idea of this optional policy is to reserve a dedicated part of a thread's stack exclusively for the kernel, which would guarantee that no stack overflow error could occur while executing kernel code. The goal was to implement this new configuration option as an additional aspect in AspeCos without any modifications of other parts of the kernel.

The implementation of the kernel-stack functionality itself is relatively straightforward: The stack pointer is switched to the part reserved for the kernel before a thread enters the kernel. This is the case whenever the application invokes a function from the eCos library. The stack is switched back to the application stack immediately before the kernel is left (when the invoked function terminates):

```
advice stack_switch() : around() {
  HAL_TO_KERNEL_STACK( Cyg_Thread::self() );
  tjp->proceed();
  HAL_FROM_KERNEL_STACK();
};
```

The platform-specific `HAL_TO_KERNEL_STACK` macro performs the actual stack switch. Afterwards, the join point API method `proceed()` calls the original function and thereby copies all parameters from the join point context to the now active kernel stack.[6]

The issue is, again, ambiguous join points. In eCos, most kernel functions are not only invoked from application level, but also used internally, that is, called by other kernel functions. As a consequence, it is not possible to decide statically if some kernel function invocation represents an *application* $\mapsto$ *kernel* transition or not.

However, in this case the ambiguity can be resolved by incorporating extra run-time state. From the AOP point of view this means to use not statically evaluable pointcuts. The ambiguity between *application* $\mapsto$ *kernel* and *kernel* $\mapsto$ *kernel* transitions can be resolved by a pointcut expression similar to the following:[7]

```
pointcut stack_switch() = execution( kernel() )
                          && !cflow( execution( kernel() ) );
```

This definition of the `stack_switch()` pointcut yields all join points where a kernel function is not executed in the context of some other kernel function. However, not statically evaluable pointcut functions, such as the used `cflow()`, have shown nonnegligible cost in the microbenchmarks (Section 4.3.2.4). As the execution of *every* kernel function is

[6] This is platform-dependent. On x86, `proceed()` is inlined and accesses the join-point context via the frame pointer, which is not modified by `HAL_TO_KERNEL_STACK`.

[7] Similar, as `cflow()` is not actually thread-safe. For thread safety, the cflow counter had to be stored in thread-local storage, which is not supported by standard C++. For the actual `kernel_stack` implementation, a "per-thread `cflow()`" was implemented by introducing the counter into the `Cyg_Thread` thread control block.

test application	AspeCos	"alternative AspeCos"
thread	*2.24*	*1.05*
mutex	*1.95*	*1.07*
semaphore	*1.88*	*1.09*

Table 5.5.: CPU overhead of the new Kernel stack feature.
Listed is the relative overhead (factor) of the `kernel_stack` aspect if applied to the three test programs (Table 5.3). In **AspeCos**, not statically evaluable pointcut functions have to be used to resolve the ambiguity between *application* $\mapsto$ *kernel* and *kernel* $\mapsto$ *kernel* transitions at run-time. With an "**alternative AspeCos**" implementation, the same ambiguity was resolved at compile-time, which led to a much lower overhead.

affected by the `kernel_stack` aspect, it is not surprising that this also leads to a noticeable overhead on the global level: Table 5.5 (column "AspeCos") lists the cost factor of the `kernel_stack` aspect for the three *_base* test-cases from Table 5.3. The cost factor is between 1.88 and 2.24 (average 2.02).

To evaluate which part of this cost is induced by the concern implementation itself (the actual stack switch) and which part is induced by the need to use not statically evaluable join points on a larger scale, I compared the results to those of an implementation for an "alternative AspeCos". This "alternative AspeCos" is more aspect-aware by offering a thin "user API layer" on top of the kernel, which is used exclusively by application code to invoke kernel functionality. The user API layer consists of a set of inline wrapper functions and classes and does not lead to additional overhead. However, the extra layer would make it possible to *statically* resolve the ambiguity between *application* $\mapsto$ *kernel* and *kernel* $\mapsto$ *kernel* transitions, so aspects can bind to these events without inducing an overhead.

```
pointcut stack_switch() = call( kernel() )
                        && within( user_api() );
```

As shown in Table 5.5 (column "alternative AspeCos"), the overhead was significantly lower in this case, with a the cost factor between 1.05 and 1.09 (average 1.07). Thus, most of the overhead of the real `kernel_stack` implementation is caused by a weakness in the eCos design by which it became necessary to resolve the join points relevant for the new Kernel stack policy at run-time.

Overall, it was possible to add a new kernel policy to AspeCos without any modification of the kernel – the kernel implementation remained oblivious of the new kernel-stack feature. The overhead induced by the resulting aspect, however, is higher than necessary.

5.4. Discussion of Results

With respect to the objectives defined in Section 5.1, the "eCos" study was quite successful:

1. It could be shown that the enforcement of kernel policies that heavily crosscut the implementation of other concerns can actually be factored out into aspects. The

expressive power of the AspectC++ language turned out to be sufficient for this (objective 1).

2. The same holds for the implementation of fine-grained configuration options. The "`#ifdef`-hell" caused by 12 thread and mutex configuration options could be replaced by a much cleaner separation into aspects. The expressive power of the AspectC++ language turned out to be sufficient for this, too (objective 2).

3. All this does furthermore not lead to a higher overhead at runtime. On the language level, most AspectC++ constructs have shown a small, but nevertheless present overhead. On the larger scale, AOP turned out to be a cost-neutral alternative to conditional compilation. System-software developers can get better separation of concerns for free (objective 3)!

In short: **AOP is qualitatively superior to conditional compilation** for the implementation of configuration options and central kernel policies. Kernel developers can implement both types of concerns with aspects in a much cleaner way. **There are no quantitative disadvantages**. This means that we already can report success with respect to the implementation-level objectives from Section 3.3.2 (page 65) – AOP *does* compare qualitatively *and* quantitatively *very well* to conditional compilation for implementing configurability in software product lines for embedded systems. These results and their validity are further discussed in Section 5.4.1.

On the other hand, the "eCos" study also revealed that the extraction of concerns into distinct aspects does not *per se* improve their configurability:

4. Many join points turned out to be ambiguous. This hindered further configurability by aspects or made further configuration options more expensive than necessary (objective 4).

These issues have to be reflected with respect to the goal of aspect-aware development. This is further discussed in Section 5.4.2.

5.4.1. The Implementation of Configurability by AOP

The results from the "eCos" study are quite promising with respect to the first of problem identifed in the problem analysis of this thesis in Section 3.1.3: State of the art to implement fine-grained configurability is – for the sake of efficiency and flexibility – conditional compilation, which however leads to commingling of concerns and "`#ifdef` hell". AOP combines similar flexibility and efficiency with a much better separation of concerns.

This gives rise to questions regarding the general, not eCos-specific validity of the obtained results. The following two sections discuss this with respect to the qualitative and quantitative dimensions of configurability.

5.4.1.1. Validity of Quantitative Results

The quantitative results were obtained by combining language-level microbenchmarks (Section 4.3) with a refactoring-based case study. The refactoring approach of the "eCos" study makes sure that the *implementation* of the concerns is identical in the AOP and original version. Both versions differ only regarding the *mechanism* used to apply the concern implementations. Thus, the measured results actually depict the net overhead of using aspects instead of tangled code and are not caused by a "smarter implementation". So even though eCos ist just one example for a system-software product line, we can expect the obtained results to have a general significance:

- eCos can be considered as a demanding test subject regarding the cost of aspects in system software. Designed for and widely accepted in the very resource-thrifty domain of embedded systems, eCos has certainly been optimized for runtime and memory efficiency. Most system calls take only a few cycles (Table 5.3), the typical kernel image size is a only few KB. An AOP-induced overhead, if any, should become evident in this system.
- The test applications chosen for cost measurements (Table 5.3) can not be considered as "typical applications". They have been designed specifically for the measurements and spend most of their time executing the affected kernel code. Regarding the cost of aspects, this has to be considered as more demanding than using "real" applications, which only occasionally call kernel functions. Thus, an AOP-induced overhead, if any, should become evident with this setup.
- The refactored concerns represent broad-scaled and cost-critical kernel functionality. With 160 affected code join points (Table 5.1), Synchronization qualifies well for evaluation of the cost AOP may induce for homogeneously crosscutting, performance-critical concerns. Instrumentation qualifies, with 139 affected code join points (Table 5.2) given by 13 aspects with 85 code advice definitions, well for evaluation of the cost AOP may induce for inhomogeneously and context-sensitive crosscutting concerns. Scalability problems, such as a per–join-point overhead, should become evident with this setup.

Overall, we can conclude that the results of the quantitative comparison between AOP on the one side and conditional compilation on the other side are generally valid for our target domain. [8]

5.4.1.2. Validity of Qualitative Results

Naturally, the "eCos" results with respect to the flexibility of AOP are based on only a selection of concerns, namely the central kernel policies Synchronization and Instrumentation

[8] The quantitative results are furthermore backed by another, more application-oriented study for an embedded-system product line. In the "WeatherMon" study I compared AOP, OOP and conditional compilation for the implementation of a weather-station product line; the hardware was based on a small 8-bit microcontroller. The "WeatherMon" study is described and discussed in Appendix B.

and 12 configuration options. However, the refactoring approach ensures that we deal with concerns from "real software"; the selected concerns represent a cross-section of different classes of real-world (and cost-critical) concerns:

- Synchronization is a typical example of a *homogenously crosscutting* concern.
- Instrumentation is a typical example of an *inhomogenously crosscutting* concern.
- The various configuration options are typical examples for *scattered* concerns that are caused by fine-grained configurability.

On the other hand, Instrumentation could not be factored out entirely into aspects. Three out of 162 macro invocations (5%) remained in the code because of the self-imposed restriction to gentle refactorings (Section 5.2.1.1). Even though in this case it would have been possible to factor out the complete concern by a more aggressive refactoring: It should be clear that the text-processing approach of the preprocessor *is* more flexible than AOP – it is not restricted to the boundaries of syntactic entities.

With respect to these boundaries, eCos was furthermore a relatively good-natured target. By its fine-grained C++ implementation, the eCos kernel offered an extensive set of potential join points – ideal conditions for aspect-based refactorings. This might explain why AOP-refactoring studies conducted by other researchers revealed less optimistic results [SWK06, KAB07].

However, our question is not if AOP is the ultimate refactoring tool, but if it is flexible (and efficient) enough to be used instead of conditional compilation for the implementation of configurability when applied from the very beginning. With respect to the different classes of concerns we dealt with in the "eCos" study, we can conclude that this is generally the case.

5.4.2. Consequences for Aspect-Aware Operating-System Developement

Overall the "eCos" study has shown that AOP is beneficial for the *implementation* of configurability. Concerns that already had been designed for configurability (by conditional compilation) could as well be implemented with aspects, resulting in a much cleaner code base. However, AOP did not help much in improving the configurability of concerns that were not designed to be configurable from the very beginning. The attempts to extend the configurability of architectural policies in eCos revealed problems of join-point ambiguity – a simple separation of concerns into aspects does not *per se* improve their configurability.

The lesson from this is: The successful aspect-aware software development does not only imply to design the software *with* aspects, but also design the software *for* aspects. In a sense, both of these interpretations are covered by the meaning of "awareness" (which encompasses, as pointed out in Section 1.3, "knowledge" as well as "perception"):

Design *with* aspects means to use aspects as first-class design elements when decomposing some feature into implementation entities of the solution space. It requires

profound *knowledge* about the cost and expressiveness of AOP language constructs – as well as idioms and rules for how and when to use them.

Design *for* aspects means to be aware of potential aspects and their interactions when designing and implementing some feature, that is, to be aware of the semantics and quality of the offered join points. It requires a good *perception* of concern interaction and "potentially interesting join points" – as well as experience and methods in how to retrieve them.

Essentially, design *for* aspects requires that we abandon – at least partly – the obliviousness principle (Section 4.1.2.1) of AOP.

5.5. Chapter Summary

The goal of this chapter was to evaluate AOP – in comparison to conditional compilation – with respect to the *implementation level* of configurability in system software for embedded devices. The results from the "eCos" study show that AOP does compare very well here; it leads to a much better separation of concerns without any disadvantages on the cost side. This means that we have solved the first problem addressed by this thesis (called "Problem 1" in Section 3.1.3): We have found a better way to implement configurability!

The results with respect to the second problem (called "Problem 2" in Section 3.1.3) are less convincing, however. Separating the implementation of architectural policies into aspects does not inherently make them more configurable. To achieve configurability of even architectural policies, it seems to be necessary to design the software and its components specifically *for* the application of policy aspects.

The work I have presented so far mostly addresses design *with* aspects and the necessary knowledge about language, expressiveness, and – especially – cost. In the next chapter, we will therefore concentrate on design *for* aspects in system software for embedded devices.

6

Design Level – CiAO Aspects: Aspect-Aware Operating-System Development

In the previous chapter we have seen that many configuration options and architectural kernel policies can as well be *implemented* by aspects instead of conditional compilation. However, the "eCos" study also revealed that a simple extraction of policies into aspects does not automatically improve their configurability. The observed issues were ambiguity of join points, missing join points, and a general lack of influence by aspects on the less obvious facets of eCos components.

In this chapter, we will lift up this knowledge on the *design level* of configurability. The goal is to understand, how system components can be developed *for* aspects to achieve better configurability of both mechanisms and policies. What are useful design principles and idioms for such an *aspect-aware operating-system development*?

On the following pages, I elaborate on these questions by the example of the CiAO family of operating systems. CiAO – the acronym stands for CiAO is Aspect-Oriented – is a highly configurable operating-system product line that has been designed and implemented from scratch with aspects as a first-class development concept.

The chapter is organized as follows: I begin with a brief overview of CiAO – goals, approach, and general structure – in Section 6.1. This is followed by a brief presentation of the CiAO design principles in Section 6.2. These principles are achieved by a set of aspect-aware development idioms, which I first present and discuss by examples from CiAO in Section 6.3, and then elaborate further by three larger case studies in Sections 6.4–6.6 (pages 137–167). The approach of aspect-aware operating-system development is then discussed in Section 6.7 and finally summarized in Section 6.8.

Related Publications

The ideas and results presented in this chapter have partly also been published as:

[LSSP05] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, pages 19–25, Chicago, IL, USA, March 2005. Northeastern University, Boston (NU-CCIS-05-03).

[LSSSP07] Daniel Lohmann, Jochen Streicher, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Interrupt synchronization in the CiAO operating system. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, New York, NY, USA, 2007. ACM Press.

[LSH+07] Daniel Lohmann, Jochen Streicher, Wanja Hofer, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Configurable memory protection by aspects. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS '07)*, pages 1–5, New York, NY, USA, October 2007. ACM Press.

[HLSP08] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Concern impact analysis in configurable system software—the AUTOSAR OS case. In *Proceedings of the 7th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)*, pages 1–6, New York, NY, USA, March 2008. ACM Press.

6.1. A Brief Overview of CiAO

CiAO is a family of research operating systems that has been developed using AOP and software product line concepts from scratch. CiAO targets the domain of embedded systems, such as automotive applications. The *CiAO-AS* family member implements an AUTOSAR-OS–like operating system kernel with configurable protection policies (memory protection, timing protection, service protection) as defined in [AUT06a].

6.1.1. Goals and Approach

The primary goal of CiAO is **architectural configurability**. That is, configurability of even fundamental, *architectural* kernel policies (see Section 3.1.2), like *synchronization* or *protection*. Further engineering goals are **efficiency** with respect to hardware resources, **configurability** in general, and **portability** with respect to hardware platforms.

The approach to achieve these goals in the implementation is **aspect-aware operating-system development**. The basic idea behind aspect-aware operating-system development is the strict decoupling of policies and mechanisms[1] by using aspects as the primary composition technique: Kernel mechanisms are glued together and extended by *binding*, *policy* or *extension* aspects; they support these aspects by ensuring that all relevant internal control-flow transitions are available as potential join points.

6.1.2. General Structure

Figure 6.1 gives an overview of CiAO's architecture. Like most operating systems, CiAO is designed with a *layered architecture*, in which each layer is implemented using the functionality of the layers below (Figure 6.1). The only exceptions from this are the aspects implementing architectural policies, which may take effect across multiple layers.

On the coarse level, we have three layers. From bottom up these are: the *hardware layer* (the hardware programming interface), the *system layer* (the operating system itself), and the *interface layer* (the application programming interface). In the implementation, each layer is represented by a separate top-level namespace (`hw`, `os`, `ciao/as`). Each layer may define additional sublayers or subsystem. The *hardware access layer* (`hw::hal`), for instance, decouples the kernel from the actual hardware platform by providing the fundamental control flow abstraction (`Continuation`) and components for the accessing central devices, such as the processor and interrupt controller (`CPU`) or the memory protection unit (`MPU`). The *system kernel layer* (`os::krn`) contains the actual kernel.

In CiAO, layers do not only serve as conceptual levels of abstraction, but also as a means to provide cross-layer control-flow transitions (especially into and out of `os::krn`) as

[1] *Policies* in the general sense, that is, strategic decisions *how* a certain system behavior (such as preemption or upcalls) is to be achieved using the available mechanisms. (When reading about kernel policies, people tend to think of thread scheduling only.)

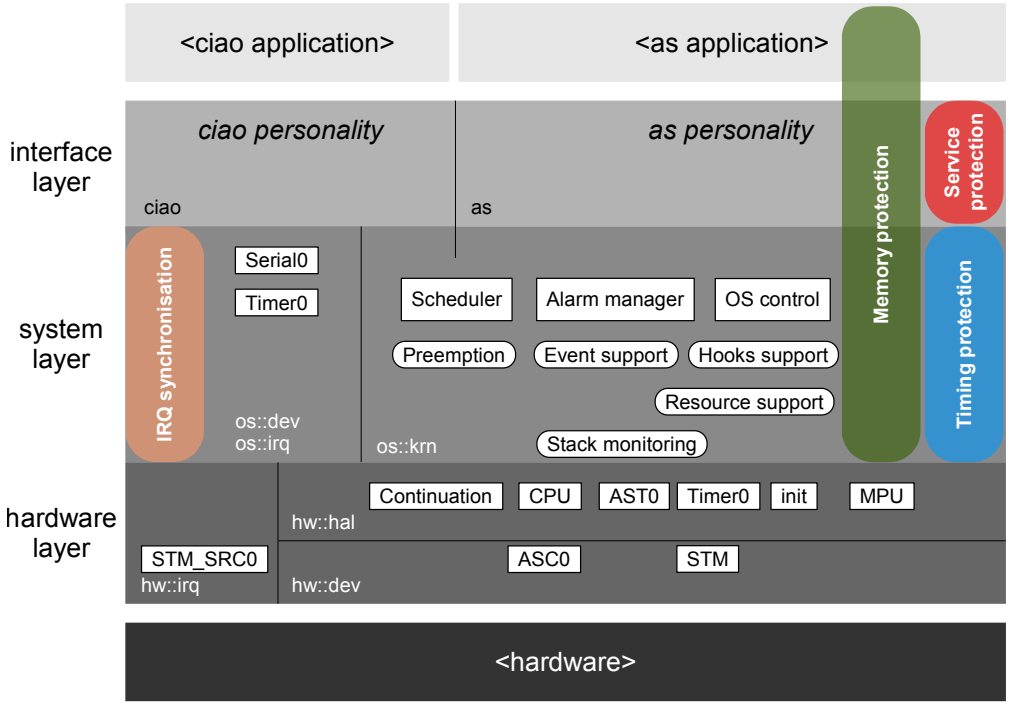


Figure 6.1.: Layered structure of CiAO.
Depicted are the three fundamental layers of the CiAO architecture with a selection of their sublayers, components, abstractions, and aspects (depicted with rounded corners). Each subsystem and sublayer defines a separate namespace. Configurable architectural properties may have an effect across multiple layers.

potential join points. The representation by distinct namespaces makes it easy to grasp such transitions by pointcuts, like the following pointcut yields all join points where a system-layer component accesses the hardware:

```
pointcut OStoHW() = call("% hw::...::%(...)") && within("% os::...::%(...)");
```

Control-flow transitions down the layer hierarchy are established by function calls; aspects can interfere with these transitions by giving advice to a pointcut like `OStoHW`. Transitions up the hierarchy (*upcalls*) are *only* established by binding or policy aspects. Hence, aspects can influence and engage with all control-flow transitions up or down the layers. This also holds for upcalls into the application code (e.g., for signal handlers). We will see examples for this in Sections 6.3, 6.4, and 6.5.

The static structure of CiAO is organized in a similar way: Extensions of a component or abstraction defined in a lower layer are established by means of C++ implementation inheritance and type aliases. Extensions of a component or abstraction defined in a higher layer are established by aspects and extension slices. Again, examples will be presented in Sections 6.3 and 6.4.

6.1.3. Kernel Personalities and Features

On the interface layer (Figure 6.1), CiAO currently provides two different kernel personalities, namely the *native CiAO* personality *(ciao)* and the *CiAO-AS* personality (*as*). Both can be understood as top-level family members of the CiAO operating system family; they represent the outcome of two consecutive development phases with, however, different approaches to aspect-aware operating system development:

1. The **ciao personality** is the outcome of the first development phase. It was developed in a *bottom-up* process – from the hardware up to a minimal kernel. Most components and abstractions of the *hw* layer (and its sublayers, especially *hw::hal*) result from this phase, so does an execution model for device drivers (*os::dev*) and the configurable architectural policy **interrupt synchronization**.
2. The **as personality** is the outcome of the second development phase. It was developed as an extension to the existing system in a *top-down* process – from an external specification (AUTOSAR OS [AUT06b, OSE05]) down to the (afterwards much extended) kernel. Most kernel components and abstractions result from this phase, so do the configurable architectural policies **memory protection**, **timing protection**, and **service protection**.

The focus of the first phase was on the aspect-aware design and implementation of the core control-flow abstractions (continuations, threads, interrupts). In this context I also developed the fundamental principles and idioms for aspect-aware operating-system development. These points are further detailed in Sections 6.2–6.4; the configuration of the interrupt synchronization policy is discussed in Section 6.5.

The goals of the second phase were twofold: Firstly, additional kernel components and abstractions and new configurable architectural policies should be added to the CiAO system. Secondly, the approach of aspect-aware operating system development was to be evaluated with real-world operating-system requirements. CiAO-AS is further discussed in Section 6.6.

6.1.4. Configuration of System Components, Abstractions, and Objects

The CiAO operating system is configured completely statically. There are two configuration sets involved in the specification of the *system components*, *system abstractions*, and *system objects* that make a concrete CiAO variant:

1. The **system configuration** specifies the *features* and the *availability* of **system components** (such as the `Scheduler`, the `EventManager`, or the various hardware devices) and **system abstractions** (such as `Task`, `Event`, `Hook` or `Application`). Both are configured by selecting features from a representation of the feature model in the graphical configuration tool (Figure 6.2 shows a screen shot).

2. The **application configuration** specifies the **system objects**, the application-specific *instantiation* of system abstractions (such as that there are the tasks `myTask1` and `myTask2`, the event `myEvent`, and that all these objects belong to the application (protection domain) `myAppA`). The system objects are configured by an XML-based configuration file. The underlying XML schema mimics the concepts and expressiveness of *OIL* (*OSEK implementation language*), the system-objects configuration language used in OSEK and AUTOSAR OS [OSE04, OSE05, AUT06b].

Both configurations are processed to build the concrete system variant. As upcalls into the application are established by aspects, the kernel has to be woven with application-specific code. However, these aspects are generated from the configuration; they are not to be provided by the application programmer.

6.2. CiAO Design Principles

As pointed out in Section 6.1.1, the basic idea behind *aspect-aware operating system development* is to use aspects to achieve a clear separation between policies and mechanisms in the implementation. The key towards *aspect awareness* is a component structure that makes it possible to influence the composition and shape of components as well as all run-time control flows that run through them by aspects. This leads to the three fundamental principles of aspect-aware operating system development:

The principle of loose coupling. Make sure that aspects can influence and engage with all facets of the integration of system components. This includes *binding*, *instantiation*, *initialization* of components, but also the *interaction* between components, which all should be established (or at least be influenceable) by aspects.

The principle of visible transitions. Make sure that aspects can influence and engage with the control flows that run through the system. All control-flow transitions into, out of, and within the system should be influenceable by aspects. For this they have to be represented on the join-point level as statically evaluable, unambiguous join points.

The principle of minimal extensions. Make sure that aspects can influence and engage with all features provided by the system on a fine granularity. System components and system abstractions should be fine-grained, sparse, and extensible by aspects.

Aspect-awareness, as described by these principles, means that we moderate the AOP ideal of obliviousness. CiAO's system components and abstractions are *not* totally oblivious to aspects – they are supposed to provide explicit support for aspects and even depend on them for their integration.

However, this does not mean that system components and abstractions have to know the *concrete* aspects that (potentially) bind to them. It is the responsibility of the *aspects* to ensure that all components affected and used by them still work correctly. Potentially

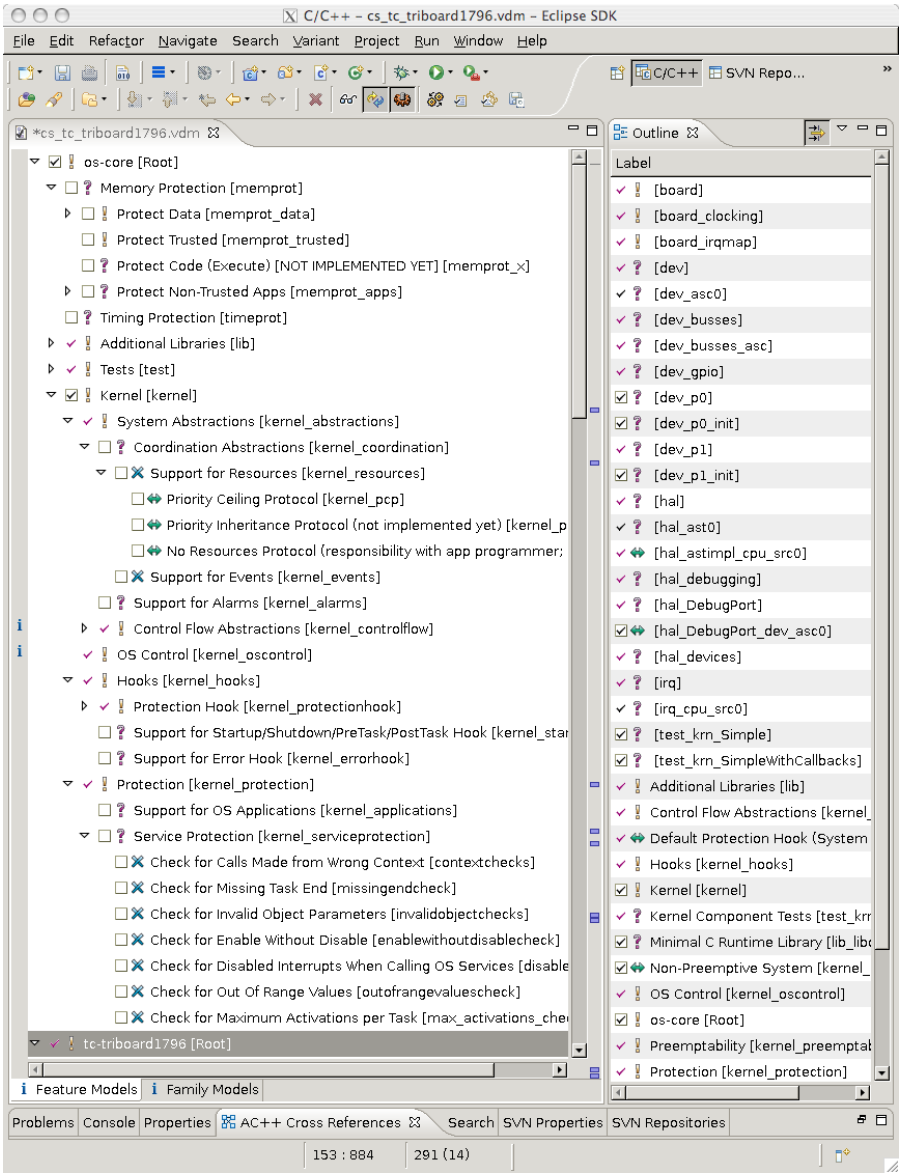


Figure 6.2.: Screen shot of the CiAO system configuration editor.
The system is configured inside the Eclipse IDE using the PURE::VARIANTS variant management plugin [Beu03b]. Depicted is the configuration editor with an extract from the CiAO feature model.

critical are situations in which the application of the aspect leads to changes in the *uses hierarchy* [Par76b]. The developer of some aspect that, for instance, binds some mechanism to a join point of an interrupt control flow has to make sure that the component that provides this mechanism is safe to be used on interrupt level. This often involves the enforcement of additional constraints, which, however, can be applied by another aspect. We will see examples for this in Section 6.3.3 and Section 6.5.

6.3. Aspect-Aware Development Idioms

In the design and implementation of the CiAO system, we can find the recurring application of three development idioms, which we can understand as the operationalization of the CiAO design principles: *advice-based binding*, *explicit join points*, and *extension slices*. In the following, I first present some general details on how aspects and classes are used within CiAO (Section 6.3.1) and then discuss each of these idioms by concrete examples from CiAO (Sections 6.3.3, 6.3.4, and 6.3.5).

6.3.1. Roles and Types of Classes and Aspects

CiAO is developed in AspectC++ using both OOP and AOP language constructs. However, the implementation abstains from all C++ and AspectC++ features that bear a significant overhead. This does not only include advanced run-time features, such as run-time type information (RTTI) and exception handling, but also "OOP basics", such as virtual functions or the support for the construction of global object instances at system start.[2] Instead, the binding of behavior and the initialization of components is implemented via a static mechanism – *advice-based binding*.

In general, CiAO employs static typing as much as possible. System components and system abstractions are represented as distinct classes so they can be distinguished at compile time and yield statically evaluable join points. For instance, each interrupt source is represented by its own class in the *hw::irq* layer instead of being instantiated as a system object from a common interrupt-source abstraction. However, all interrupt classes adhere to a standardized *static* interface that specifies a well-defined set of operations and *explicit join points* so that *policy aspects* can employ generic advice to quantify over all interrupt sources. We will see an example for this in Section 6.5.

System components that maintain run-time state (such as the scheduler), are implemented as singletons and instantiated by the kernel. The singleton instance encapsulates the run-time state (e.g., the ready list); it can only be retrieved via the `...::Inst()` class method. The standardized singleton interface ensures that *policy aspects* can engage with the instantiation of components (e.g., implement a different instantiation scheme by overriding `...::Inst()` with a piece of around-advice).

The classes for system components and system abstraction are sparse and to be "filled" by *extension slices*. The main purpose of classes is to provide a namespace with unambiguous join points for the aspects. We distinguish three general types of aspects:

1. **Extension aspects** add additional features to a system abstraction or component.
2. **Policy aspects** "glue" otherwise unrelated system abstractions or components together to implement some CiAO kernel policy.

[2] Both of which induce quite some overhead, compared to the applied alternative implementation with static typing and aspects. I have compared and analyzed this cost in detail in the "WeatherMon" study. This study can be found in Appendix B, the cost analysis in Appendix section B.4.

3. **Upcall aspects** bind behavior defined by higher layers to events produced in lower layers of the system.

Generally, the effect of *extension aspects* becomes visible in the API of the affected system component or abstraction, whereas *policy aspects* and *upcall aspects* lead to a different behavior. However, the distinction between the three types is not strict; a policy aspect, for instance, may also extend some class if this is part of the policy. We will see examples for all three types of aspects in the following sections.

6.3.2. Diagram Notation

Figure 6.3 describes the diagram notation I use for class-and-aspect diagrams in this thesis. Several attempts have been published to extend UML with a formal notation of AOP elements.[3] However, the existing notations tend to be either too formal, too close to AspectJ, or both. Hence, I developed my own notation that offers a suitable level of detail for the purpose of this thesis. The notation is also related to UML, but intentionally abstains from the official UML extension system (e.g., stereotypes) to bring in the relevant AOP concepts.

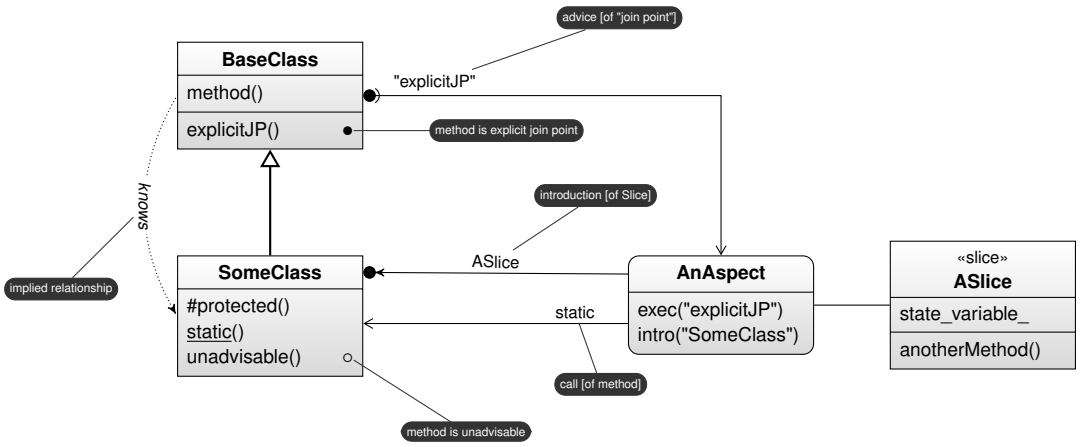


Figure 6.3.: Diagram notation for class-and-aspect diagrams (static structure).
The notation is related to the notation of UML class diagrams, but uses several nonstandard style elements to depict aspects, advice, and introductions.

6.3.3. Loose Coupling by Advice-Based Binding

With **advice-based binding** components and polices *integrate themselves* into the system. This is the most fundamental idiom for the implementation of loose coupling. It exploits

[3] Well-known examples are: *composition patterns* [CW01], *Theme/UML* [BC04], the notation by STEIN and colleagues [SHU02], and *AML* [GB04].

the effect that aspects effectively invert the direction in which control-flow relationships between components are established (see Figure 4.1). Thereby, optional components and policies can easily hook into the system's control flows.

Besides flexibility, advice-based binding also has the advantage that it can be bound at compile time if the affected join points are statically evaluable. Where appropriate, the advice code can even be inlined directly at the join point occurrence to avoid the overhead of extra function calls. This is more efficient than the common approaches for indirect binding of components, which bind at link time (external functions) or run time (virtual functions and function pointers).

6.3.3.1. Component Self-Integration

The canonical example for self-integration by advice-based binding is component initialization: Every CiAO component has an accompanying `_Init` aspect that gives advice to the system initialization handler `hal::init()` to invoke the component's `init()` method at system startup time (see Figure 6.4). Thereby, the startup code does not have to know which components are present in the actual CiAO configuration – nevertheless this flexibility does not come at a price, as all initialization code gets bound and inlined at compile time.

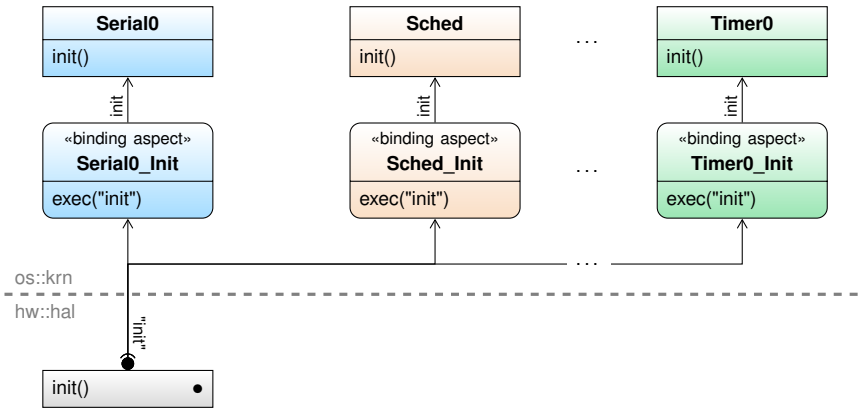


Figure 6.4.: Self-integration of components by advice-based binding.
Depicted is the CiAO component initialization scheme. Every CiAO component integrates itself into the global system initialization handler `hal::init()` by an accompanying `_Init` aspect.

Figure 6.4 also demonstrates another advantage of advice-based binding, namely the **transparent support of** $1 : n$ **relationships**. Without any further preparations, multiple clients can bind to the same join point by several aspects giving advice for it. The result is sequential activation of the respective advice implementations at run time.[4]

[4]Another obvious use case for this facility is the transparent chaining of interrupt handlers on platforms where interrupt lines are shared between multiple devices.

However, with respect to loose coupling and aspect awareness, the most important benefit of advice-based binding is that we can *further* influence it by additional aspects. Consider, for instance, an (optional) aspect `Serial0Ext` that extends the serial driver from Figure 6.4 by a task of its own (e.g., for some background protocol handling). This aspect effectively inserts a new functional dependency between the serial driver and the scheduler; the serial driver now *uses* the scheduler. The consequence for the implementation is that the scheduler has now to be initialized *before* the serial driver. This new constraint can easily be realized with **order-advice**. Additional to the extension of the class `Serial0`, the aspect `Serial0Ext` can specify a relative invocation order for the *foreign* aspects `Sched_Init` and `Serial0_Init` at the join point `execution( "void hw::hal::init()" )` as follows:[5]

```
aspect Serial0Ext {
  ...
  advice execution( "void hw::hal::init()" ): order(
    "Sched_Init", "Serial0_Init" );
};
```

Essentially, the aspect thereby re-establishes the *uses* hierachy of the system.[6]

6.3.3.2. Policy Self-Integration

Another common use case for advice-based binding in CiAO is the self-integration of policies. Self-integration of policies is crucial for the aspired decoupling of policies and mechanisms. Most policy implementations induce new interactions between (otherwise unrelated) components. This may, again, lead to new functional dependencies that we also have to deal with.

Figure 6.5 demonstrates self-integration of policies by the example of two variants of the CiAO preemption policy. Generally, system components report the need for rescheduling (and, thus, potential preemption of the running task) by calling `Sched::setNeedReschedule()`. The actual activation of the scheduler is, however, delayed:

The aspect `Sched_LeaveBinding` in Figure 6.5.a implements a simple delayed activation policy for a cooperative system; with this policy, preemption is only possible at the return from some system service.

The aspect `Sched_ASTBinding` in Figure 6.5.b implements a more sophisticated delayed activation policy for an interruptive system; with this policy, preemption can also take place after interrupt termination. Technically, this is realized by binding the scheduler activation (`Sched::reschedule()`) to the function `AST0::ast()`, which is the handler of

[5] At weave time, the aspect weaver analyzes all specified partial orders to derive a valid total order of advice invocation for every join point. If this is not possible, a weave-time error is thrown. A detailed explanation of the syntax of *order-advice* can be found in Appendix section A.1.3.3.

[6] The effect that changes in the functional hierarchy have also to be caught up in the initialization order of system components was a serious problem for the aspect-based configuration of architectural polices in PURE [SL04]. In PURE, system components are initialized by C++ global instance construction, for which the order cannot be influenced on the language level.

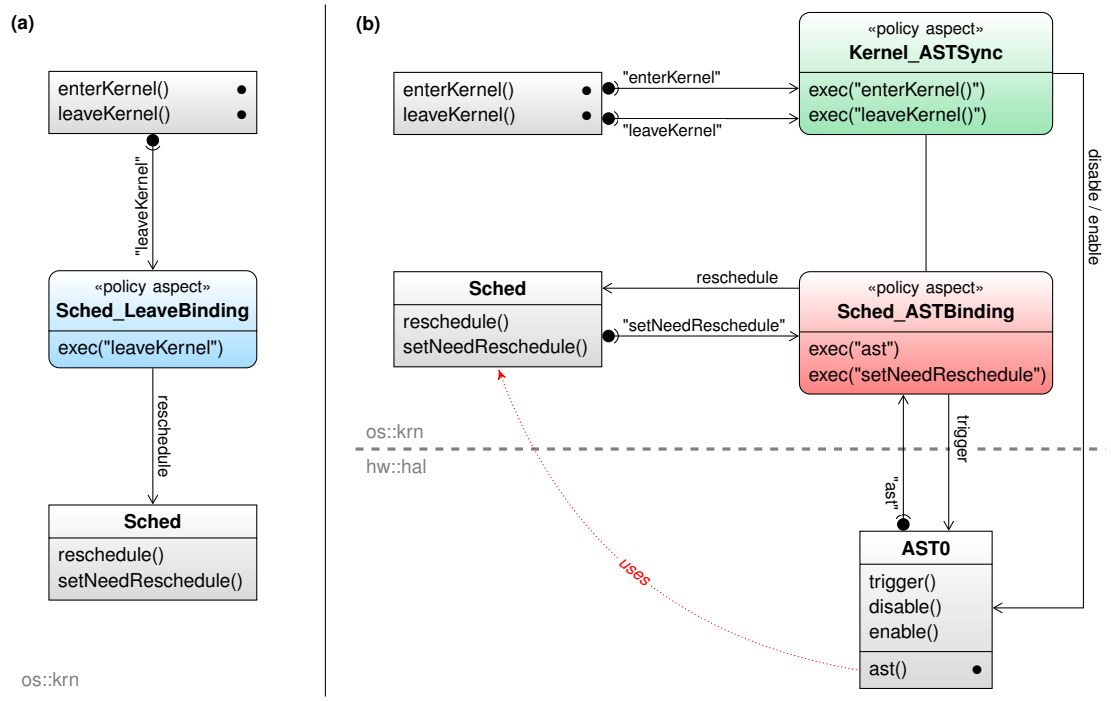


Figure 6.5.: Self-integration of policies by advice-based binding.
Depicted are two alternatives for the delayed preemption policy in CiAO. **(a)** The aspect `Sched_LeaveBinding` binds to `leaveKernel()` to activate the scheduler when some task leaves the kernel. **(b)** The aspect `Sched_LeaveBinding` binds to the handler of an *asynchronous system trap* (*AST*) to activate the scheduler when all (potentially nested) interrupt handlers have terminated. This has the consequence that the scheduler now runs on AST level, which leads to a new functional dependency between the (otherwise unrelated) system components `AST0` and `Sched`. The aspect `Kernel_ASTSync` re-establishes a correct *uses* hierarchy by synchronizing AST propagation with other control flows in the kernel.

an *asynchronous system trap* (*AST*). Additionally, the triggering of the AST is bound to `setNeedReschedule()`. The fact that the scheduler is now activated from `AST0::ast()` leads to a new functional dependency, which has the consequence that the kernel now has to be synchronized on AST level. We can, however, easily enforce this constraint with additional pieces of advice that are given by the `Kernel_ASTSync` aspect.

6.3.4. Visible Transitions by Explicit Join Points

Much of the flexibility of advice-based binding comes from the fact that CiAO's components provide – by a fine-grained separation of concerns and the syntactic richness of the C++ language (we have discussed this Section 4.1.2.1) – a rich **implicit join-point interface** to which aspects can bind. Every method call or execution adds to the set of potential join points. This fine-grained separation of concerns is an important factor for aspect-awareness.

However, in many cases the implicit join-point interface is not ample enough. This has conceptual as well as technical reasons:

1. Implicit join points are inherently implementation dependent. Their amount – but especially their semantics – may be inconsistent between different implementations of the same concept. This is absolutely acceptable for component-specific extension aspects, as these aspects have to know the component they extend anyway. It is, however, not satisfying for system aspects that implement more general policies and for aspects that bind application code to kernel events, such as the binding of user-level signal handlers (AUTOSAR OS hook functions).
2. Some semantically important control-flow transitions are not visible on join-point level because they do not occur on the boundary of function calls or executions. In other cases, their place of occurrence is configuration-dependent, or there are multiple places of occurrence. For example, *application* $\mapsto$ *kernel* transitions might occur if a kernel function is called, when a trap handler is activated, or during task switching to another task. However, in CiAO this is a matter of configuration.
3. Several semantically important control-flow transitions are not available as join points because of technical reasons. This is often the case with low-level system abstractions, such as interrupt handlers or the implementation of the context switch mechanism. If the implementation uses assembly language, the join points are not visible to the aspect weaver. In other cases, the implementation depends on assumptions about the compiler's code generation that are not covered by the semantics of ISO C++ (e.g., that a function will always be inlined, or that a parameter is always passed in a certain register). Join points in those fragile parts of the code are visible, but should not be advised as the nontrivial transformations by the aspect weaver (compare Section 4.3.1) might easily break it. In the diagram notation, an open dot ($\circ$) is used to mark such unadvisable methods.

For these reasons, many CiAO components and layers provide furthermore a well-defined **explicit join-point interface** that defines one or several *explicit join points*. An **explicit join point** is a named join point in the kernel control flow that bears a precisely defined semantics and can safely be advised. Technically, explicit join points are implemented as empty methods – provided for the sole purpose that aspects can bind to them. The join-point provider invokes these methods at run time, either directly or indirectly by component-specific adapter aspects. In the diagram notation, a filled dot ($\bullet$) marks a method that represents an explicit join point.

We have already seen several examples: The methods `hal::init()` and `AST0::ast()`, for instance, are actually explicit join points. Both have an empty implementation, but represent the occurrence of a well-defined, semantically important run-time event. They are explicitly triggered by their providing components (the startup code; and the hardware-based or software-based implementation of the AST facility, both of which are platform dependent).

Conceptually, explicit join-point interfaces can be compared to hooks or interceptor

	type	representing function or method	description
os::krn	U	`interalErrorHook()`	Explicit join points for the support and binding of OSEK OS and AUTOSAR OS user-level hook functions, as specified in [OSE05, p. 39] and [AUT06b, p. 46]. Triggered in case of an *error*, a *protection* violation, before (*pre*) and after (*post*) at high-level task switch, and at operating-system startup and shutdown time.
	U	`internalProtectionHook(StatusType error)`	
	U	`internalPreTaskHook()`	
	U	`internalPostTaskHook()`	
	U	`internalStartupHook()`	
	U	`internalShutdownHook()`	
	T	`enterKernel()`	Triggered when a control flow enters (respectively leaves) the kernel domain.
	T	`leaveKernel()`	
		...	
hw::hal	U	`ThreadFunc()`	Entry point of a new thread (continuation).
	T	`before_CPURelease(Continuation*& to)`	Triggered immediately before the running continuation is deactivated or terminated; `to` is going to become the next running continuation.
	T	`before_LastCPURelease(Continuation*& to)`	
	T	`after_CPUReceive()`	Triggered immediately after the (new) running continuation got reactivated or started.
	T	`after_FirstCPUReceive()`	
	U	`AST<#>::ast()`	Entry point of the respective AST.
	U	`init()`	Triggered during system startup after memory busses and stack have been initialized.
		...	
hw::irq	U	`<IRQ_NAME>::handler()`	Entry point of the respective interrupt handler. (Interrupts are still disabled.)
		...	

Table 6.1.: Explicit join points in CiAO.
Listed is a selection of *upcall* (U) and *transition* (T) join points offered by the different layers (respectively components in these layers) of CiAO. The actual set of available explicit join points is configuration dependent.

interfaces in other component models. An advantage of explicit join points is, however, their low overhead. In most cases (that is, when they do not have to be triggered from parts written in assembly language) they can be implemented as empty inline methods, which get optimized away by the compiler if no aspect binds to them. Another advantage is the inherent support for $1 : n$ relationships, as explained in the previous section on the example of `hal::init()`.

Table 6.1 lists a selection of the explicit join points provided by CiAO. We distinguish between **upcall join points** and **transition join points**. This differentiation is not strict (depending on the client, an upcall can also represent a transition and vice versa), however, underlines the primary purpose of the respective explicit join point.

6.3.4.1. Explicit Upcall Join Points

For the sake of configurability, the processing of most system-internal events is postponed to a higher layer than the layer on which they occur. By representing these events as explicit join points, higher layers (up to the application itself) can subscribe to them with advice-based binding.

The methods `hal::init()`and `AST0::ast()` are examples for (internally used) upcall join points. Other examples include interrupt handlers (`<IRQ_NAME>::handler()`), signal handlers (AUTOSAR OS hook functions), the `TCBUser::ThreadFunc()` start function of a new coroutine (will be further detailed in Section 6.4), or the method `Sched::idle()` that is called from the scheduler idle loop. Besides the direct binding (and potential inlining) of the event-processing code, the kernel can also exploit these join points to implement configuration-dependent upcall policies. An upcall-policy aspect may, for instance, filter events, or translate them into virtual function invocations, thread activations, or send message operations.

6.3.4.2. Explicit Transition Join Points

Control flow transitions inside the kernel, such as the transition from application level to kernel level, from thread level to interrupt level, or the context switch from one thread to another one, are important events for the implementation of many policies. Many of these events, however, have multiple sources ($m : n$ relationships); or they occur in fragile, low-level parts of the implementation. By representing them as explicit join points, providers and publishers of transition events can be decoupled.

An already mentioned example for transition join points are the *application* $\mapsto$ *kernel* and *kernel* $\mapsto$ *application* transitions, which are represented in CiAO by the explicit join points `krn::enterKernel()` and `krn::leaveKernel()`. Another example are the transition join points provided by the CiAO dispatcher (the class `Continuation`), which are further detailed in Section 6.4.

6.3.5. Minimal Extensions by Extension Slices

The classes that represent CiAO's system components and system abstractions are generally sparse: they are either completely empty or implement only the minimal base of some feature. Optional features are implemented as **extension slices** and introduced by extension aspects into these classes.

The use of *extension slices* is the most relevant idiom for the implementation of minimal extensions: The implementation of optional features does usually not affect a single component or abstraction, but crosscuts with the implementation of several other concerns – often even across multiple layers. This is, for instance, always the case if the extension is also to become visible in the API provided by the interface layer.

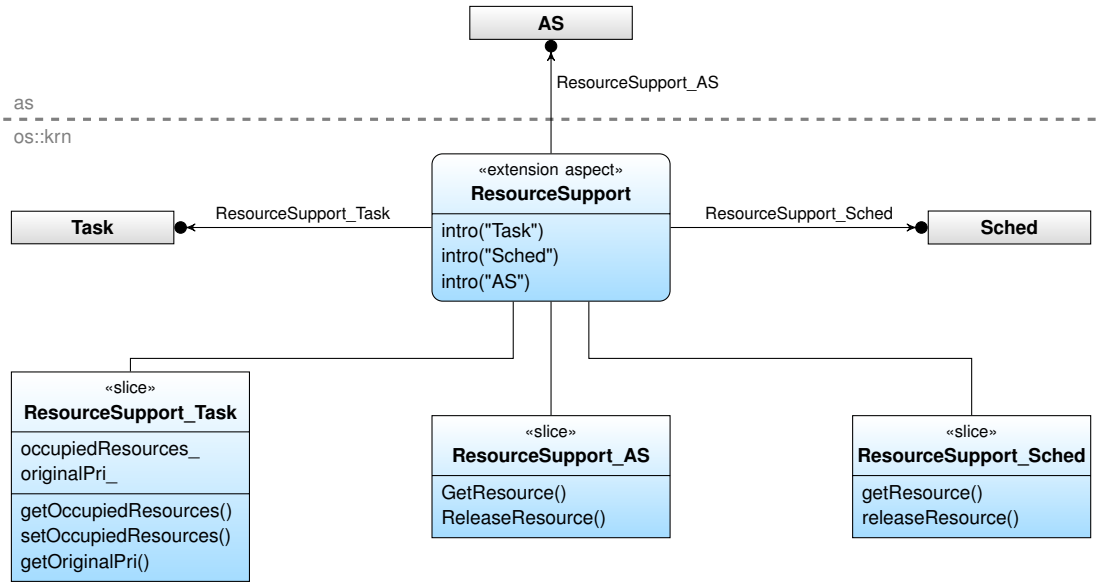


Figure 6.6.: Integration of an optional feature by extension slices.
The aspect `ResourceSupport` adds support for AUTOSAR-OS resources to the CiAO kernel. This requires the introduction of respective *extension slices* into the `Task` system abstraction and the `Sched` system component from the *os::krn* layer, as well as into CIAO-AS API, which is contained in the class `AS` on the interface layer.

Figure 6.6 demonstrates this by the example of the `ResourceSupport` extension aspect that adds support for AUTOSAR-OS resources to the CiAO-AS kernel. The actual implementation is introduced as methods and state variables into the `os::krn` classes `Task` and `Sched`. However, to be accessable from applications, the CIAO-AS API on the interface layer has to be adapted as well – which requires the introduction of the respective methods into the class `AS`.

With extension slices, collateral adaptations of several kernel components, abstractions, and the API can be separated and grouped into a single logical module – the extension aspect. Basically all optional system services provided by the CiAO-AS kernel are implemented this way. This ensures that services and abstractions that have not been configured for the kernel are not reflected in the API either; hence, many configuration errors can be detected early at compile time.

We can understand extension slices as the static counterpart of advice-based binding (see Section 6.3.3); the latter is used for loose coupling and self-integration with respect to the system's control flows, whereas the former fulfills the same task for the static system structure, that is, abstractions and components. Therefore, extension slices and advice-based binding are often used together. We will see examples for this in the "Continuation" study in Section 6.4.

6.3.6. Summary

The CiAO design principles of *loose coupling, visible transitions*, and *minimal extensions,* are to a high degree implemented by three development idioms: *advice-based binding*, *explicit join points*, and *extension slices.* Technically, these idioms exploit the mechanisms AOP provides for obliviousness – code advice and introductions; they here facilitate the self-integration (and thereby decoupling) of mechanisms and policies in the implementation. CiAO components and abstractions are, however, not completely oblivious to aspects – they even depend on them for their integration, all “glueing” is done by advice.

The application of an extension or policy aspect often leads to side effects in the functional hierarchy. However, if the CiAO design principles have been applied consequently, it is always possible to enforce the additional constraints that result from new functional dependencies by additional aspects. Essentially, aspects are used in CiAO to implement different functional hierarchies from a common set of implementation assets.

The following sections further elaborate on the application of the discussed principles and idioms by three larger case studies from CiAO:

“Continuation”. In the “Continuation” study I exemplify the aspect-aware decomposition and implementation of a **configurable system abstraction** – from features down to code. The purpose of this study is to demonstrate the need and the benefits of explicit join points in even the fundamental low-level system abstractions of the operating system to achieve exensibility by visible transitions and loose coupling (Section 6.4).

“Interrupt synchronisation”. The “Interrupt synchronisation” study demonstrates the aspect-aware design (and partly also the implementation) of a **configurable architectural policy**. The purpose of this study is to show how architectural configurability can be achieved by means of aspect-aware development (Section 6.5).

“CiAO AS”. The “CiAO-AS” study finally presents results from the aspect-aware development of a complete **configurable operating system** – from requirements and specification down to evaluation results. The purpose of this study is to show that the approach does scale up and is applicable for the development of a complete kernel (Section 6.6).

6.4. Case Study “Continuation”

In this section I present a detailed example for the aspect-aware design and implementation of a system abstraction. The Continuation concept is CiAO’s system abstraction for the instantiation of preemptable control flows. It is provided by the *hw::hal* layer and serves as the base for other subsystems, especially the kernel, to implement higher-level thread or task abstractions.

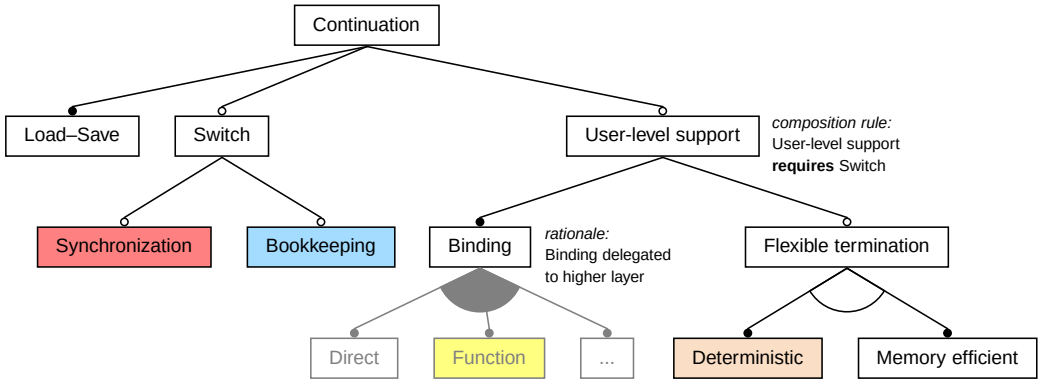


Figure 6.7.: Feature diagram of CiAO's control flow abstraction.
The Continuation concept provides mechanisms to load, save, and initialize a control flow context (Load–Save); extension stages on top of this are mechanisms for dispatching (Switch) and the support for user-level control flows (User-level support). Context switches can be atomic (Synchroniziation) and tracked (Bookkeeping); user-level control flows can be allowed to terminate while executing subfunctions (Flexible termination). (The coloring is for the purpose of feature traceability with Figure 6.8 and Figure 6.9.)

6.4.1. Continuation Features

Figure 6.7 depicts the variability of the Continuation concept as a feature diagram. A CiAO continuation provides at least mechanisms to load, save, and initialize a control-flow context (Load–Save). Optional extension stages are the mechanisms to dispatch from one continuation to another (Switch) and the interface for the kernel to implement user-level control flows (User-level support). The extension stages can be further specialized by additional features to keep track of the running continuation (Bookkeeping), ensure atomicity of context switches (Synchronization), and permit user-level control flows to terminate from subfunctions (Flexible termination).

6.4.2. Continuation Design

Figure 6.8 depicts the functional hierarchy (dependency graph) for the Continuation concept under the assumption of an aspect-aware design. The three (logical) basic functions (Load–Save, Switch, User-level support) build upon one another. The functions that implement the remaining features, however, do not depend directly on one of the basic functions. Instead, they depend on *transitions* that result from *others* using the basic functions. The Bookkeeping function, for instance, does not directly depend on any kind of context switch functionality, but on being informed about context switches. We can understand this as a dependency on some (implicit or explicit) event interface. In Figure 6.8, these event interfaces are made explicit as special functions (depicted in italics).

The Continuation abstraction is provided by the *hw::hal* layer as a set of classes and aspects around the class `Continuation`. Figure 6.9 shows the resulting class-and-aspect diagram. The three extension stages Load–Save, Switch, and User-level support have been implemented as classes `ContinuationBase`, `Contination`, and `TCBUser`, respectively.

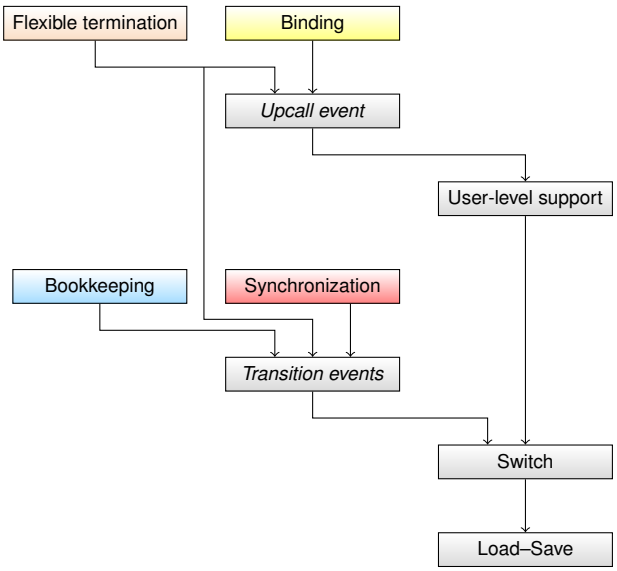


Figure 6.8.: Functional hierarchy of the features provided by CiAO’s control flow abstraction.
The nodes represent logical functions, that is, implementations of the (equally named/colored) features from Figure 6.7; the edges represent functional dependencies between the logical functions.

In theory it would also have been possible to implement Switch and User-level support as extension aspects of `ContinuationBase`; the class-based design was chosen to support control-flow instances of different extension stages to coexist in the system (e.g., to execute interrupt handlers as `Continuation` instances while user-level threads are instances of `TCBUser`).

Besides the relevant mechanisms, the classes `Continuation` and `TCBUser` each provide certain events by an explicit join-point interface (methods marked with •) for potential aspects to bind to. Thereby, all other functions could be implemented as loosely-coupled policy, extension, or binding aspects that use advice-based binding and extension slicing to integrate themselves into the respective abstraction.

The explicit join-point interface is of particular importance here. As pointed out in Section 6.3.4, implicit join points in the implementation of low-level mechanisms can be fragile or completely invisible and, hence, have to be considered as unadvisable (methods marked with ○). Even if they are visible, their semantics can bear subtle differences across platforms and implementations. The four explicit transition join points provided by the class `Continuation`, on the other hand, make all relevant transitions in the life cycle of a control flow (*start*, *deactivation*, *reactivation*, and *termination,* see also Table 6.1) visible on join-point level with well-defined semantics. In a similar manner the class `TCBUser` provides with `ThreadFunc()` an upcall join point with a well-defined semantics for the binding of further kernel policies or abstractions.

In the following, I illustrate these constraints, the join point interfaces, and how they are used by extension, policy or binding aspects by an actual implementation of the Continuation abstraction.

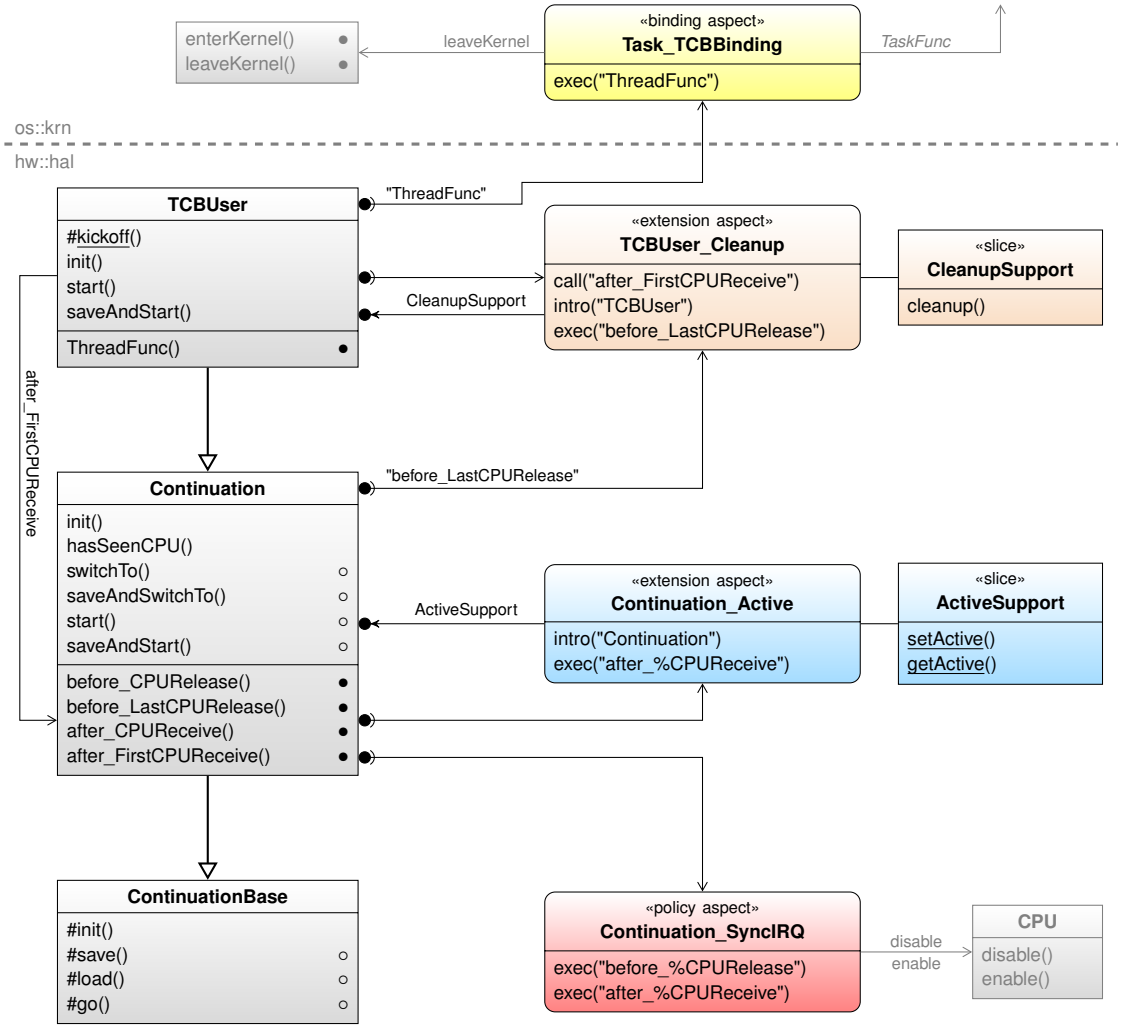


Figure 6.9.: Classes and aspects that implement CiAO's control flow abstraction.
Depicted is the static structure of classes and aspects that implement the features from Figure 6.7 according to the functional hierarchy from Figure 6.8. Central element is the class `Continuation`, which provides an interface of four explicit transition join points; the class `TCBUser` extends `Continuation` by an additional upcall join point for the kernel. All other features are modeled as policy, binding, or extension aspects that bind to these join points.

6.4.3. Implementation for TriCore

On the TriCore platform with G++ as the compiler, all context switch functionality could be implemented in C++ (with utitliziation of a few assembler intrinsics).

6.4.3.1. The Fundament: Continuation, ContinuationBase, and TCBUser

ContinuationBase. Listing 6.1 shows the TriCore implementation of the class `ContinuationBase`. The purpose of this class is to encapsulate the elementary state of a

```
1  class ContinuationBase {
2   protected:
3    _tc::PCXI_t_nonv  pcxi_;      // previous context pointer (caller's context)
4    void*             addr_;      // return address (caller)
5
6    void init() {
7      pcxi_.reg = 0;
8    }
9    void CIAO_INLINE save() {
10     addr_     = _getRA();       // save return address (caller)
11     pcxi_.reg = _mfcr( $pcxi ); // save PCXI register (caller's context)
12     _dsync();                   // sync data pipeline
13   }
14   void CIAO_INLINE load() {
15     _mtcr( $pcxi, pcxi_.reg );  // restore PCXI register (caller's context)
16     _setRA( addr_ );            // restore return address (caller)
17     _isync();                   // sync instruction pipeline
18   }
19   void CIAO_INLINE go( void* tos, StartFunc starter ) {
20     _mtcr( $pcxi, 0 );          // new control flow has no caller
21     _setSP( tos );              // set the stack pointer
22     _isync();
23     hw::JUMP1( starter, this ); // to not waste CSAs, directly jump to start address
24   }
25 };
```

Listing 6.1: TriCore implementation of the class `ContinuationBase`

continuation and to provide the elementary operations to initialize, save, load, and begin a continuation context (`init()`, `save()`, `load()`, `go()`). As the TriCore CPU automatically saves and restores all nonvolatile registers around function calls, the state to be managed in the continuation object itself is relatively small: Only the register that contains the return address and the register that points to the implicitly saved caller context have to be dealt with in the `save()`, `load()`, and `go()` operations.[7]

However, even though these operations are implemented in C++ they must not be advised – they are fragile. As `save()` and `load()` effectively store and restore the *caller's* context (the actual switch after a `load()` operation takes place when the *surrounding* function returns), they *must* be inlined into their invoker, which itself *must not* be inlined. The `go()` operation must be inlined, too – otherwise the call context that was implicitly created for the call to `go()` would never be freed.

The explicit control over inlining versus noninlining depends on compiler-specific language extensions.[8] As pointed out in Section 6.3.4, the nontrivial transformations of the aspect

[7] A peculiarity of the TriCore platform is that call frames are not managed on the stack, but in linked lists of dedicated *context save areas* (*CSAs*) that are implicitly created and destroyed by the `call` and `ret` instructions. A CSA is a memory block of 128 bytes that represents a function frame including all nonvolatile registers and the pointer to the previous context. The *PCXI* (*previous context information*) CPU register always points to the most recent CSA, which is the head of the CSA list of the currently active thread. Consult [Inf05, pp. 5-1ff] for further details.

[8] On `G++` `CIAO_INLINE` and `CIAO_NOINLINE` expand to `__attribute__((always_inline))` and `__attribute__((noinline))`, respectively.

weaver might easily break such code. However, even if it were safely possible to give advice to these implicit join points we should refrain from doing so – they might be unavailable or bear subtle semantical differences in other implementations of the load–save mechanisms.

Continuation. The class `Continuation` implementation-inherits from `ContinuationBase` and uses the elementary operations to provide the four higher-level context-switch operations that are available to clients:

```
class Continuation : public ContinuationBase {
  void ... start( void* tos, StartFunc starter, Continuation* to );
  void ... switchto( Continuation* to );
  void ... saveAndStart( void* tos, StartFunc starter, Continuation* to );
  void ... saveAndSwitchto( Continuation* to );
  ...
```

These operations have to be considered as nonadvisable, too. They are also fragile with respect to inlining and may bear subtle semantical differences in other implementations. However, all control-flow transitions that result from using these operations are made visible on join-point level by an explicit join-point interface:

```
  ...
  void before_CPURelease( Continuation*& to ) {}
  void before_LastCPURelease( Continuation*& to ) {}
  void after_CPUReceive() {}
  void after_FirstCPUReceive() {}
};
```

The explicit join points are triggered by the context switch operations. The `start()` and `switchto()` operations, for instance, are used to start, respectively reactivate, another continuation `to` without saving their own context. Both operations never return; the calling continuation terminates. Immediately before termination they trigger `before_LastCPURelease()` to signal this transition to interested aspects. In the implementation of `start()` this looks as follows:

```
void CIAO_INLINE start( void* tos, StartFunc starter, Continuation* to ) {
  before_LastCPURelease( to ); // we are going to leave forever
  to->go( tos, starter );      // start 'to'
}                              // <-- we never come here
```

Thereby, an aspect that gives advice to `before_LastCPURelease()` is activated whenever the current continuation control flow is about to terminate for `to` to receive the CPU. Note that `to` is passed as a *reference* parameter to `before_[Last]CPURelease()` – an aspect could not only inspect, but even influence the ongoing transition by choosing another continuation to receive the CPU.

The `saveAnd...()` context switch operations work similarly, but save the calling's continuation context first. They return when the calling continuation gets reactivated. Hence, they trigger `before_CPURelease()` and `after_CPUReceive()`, which signal these transitions. In the implementation of `saveAndSwitchto()` this looks as follows:

```
void Continuation::saveAndSwitchto( Continuation* to ){
  int_saveAndSwitchto( to );  // call (!) internal implementation
                              // <-- point of reactivation
  after_CPUReceive();         // hello, again
}
void CIAO_NOINLINE Continuation::int_saveAndSwitchto( Continuation* to ) {
  before_CPURelease( to );    // we are going to leave for 'to'...
  save();                     // but not forever
  to->load();                 // load 'to'
}                             // <-- point of 'to' reactivation
```

The explicit join point `after_FirstCPUReceive()` signals that a continuation has been started (activated for the very first time). It has to be triggered by the start function that is passed as parameter `starter` to the `start()` or `saveAndStart()` operations. Furthermore, `starter` has to adhere to a specific platform/configuration-dependent signature. These two constraints are part of the contract between the class `Continuation` and its clients. A third constraint imposed by `Continuation` is that the control flow must not terminate (invoke `start()` or `switchTo()`) from a function call level below `starter`.

TCBUser. Whereas the above constraints are perfectly acceptable for system-internal control flows, they might be inappropriate for continuations that start in user-level code. In general, the decision *how* the user-level code has to be shaped, is bound, and gets activated should be understood as a policy decision of the kernel – and not be prescribed by the *hw::hal* layer.

The class `TCBUser` implements a continuation interface for the kernel that deals with these issues. It is intended as the base for user-level control flows and provides an explicit upcall join point (`ThreadFunc()`) to which the kernel can bind its own policy for the binding and activation of user code without having to deal with the first two constraints imposed by `Continuation`. (The third constraint is tackled by the `TCBUser_Cleanup` extension aspect, which will be discussed further below.):

```
class TCBUser : public Continuation {
  ...
  // the internal Continuation start function
  static void cfHAL_STARTFUNC_ATTRIBUTES kickoff( TCBUser* me ) {
    me->after_FirstCPUReceive();   // we are alive
    me->ThreadFunc();              // and this is what we do
    _debug();                      // <-- we should never come here!
  }
 public:
  void inline TCBUser::ThreadFunc() { /* upcall JP for the kernel*/ }
};
```

Technically, `TCBUser` uses an internal start function (`kickoff()`) that fulfills the `Continuation` contract and then triggers the explict upcall join point. As `ThreadFunc()` can be inlined by the compiler, this indirection does not cause an overhead.

6.4.3.2. The Aspects: Utilizing the Explicit Join-Point Interfaces

The remaining optional or alternative features have been implemented as aspects; they integrate themselves into the continuation classes with advice-based binding and extension slices (see Figure 6.9).

Continuation_IRQSync. `Continuation_IRQSync` is a platform-independent policy aspect that implements the optional Synchronization feature. It ensures atomicity of context switches by disabling interrupts in `before_[Last]CPURelease()` and reenabling them in `after_[First]CPUReceive()`. The implementation of this aspect is straightforward:

```
aspect Continuation_IRQSync {
  ...
  advice execution( "void ...::before_%CPURelease( ... )" ) && ... : before() {
    hw::hal::CPU::disable();
  }
  advice execution( "void ...::after_%CPUReceive( ... )" ) && ... : after() {
    hw::hal::CPU::enable();
  }
};
```

Continuation_Active. `Continuation_Active` is a platform-independent extension aspect that implements the optional Bookkeeping feature. It upgrades the class `Continuation` to a full-blown dispatcher that "knows" the currently running continuation. For this purpose, an extensions slice is used to introduce the static `active_` pointer along with corresponding accessor functions into class `Continuation`; advice-based binding is employed to update the pointer after a context switch transition:

```
aspect Continuation_Active {
  pointcut pcClass() = "hw::hal::Continuation";
  advice pcClass() : slice class ActiveSupport {
    static ActiveSupport* active_;
   public:
    static void setActive(ActiveSupport* to) {active_ = to;}
    static ActiveSupport* getActive() {return active_;}
  };
  advice execution( "void ...::after_%CPUReceive( ... )" )
        && within( pcClass() )  : before() {
    JoinPoint::That::setActive( tjp->that() );
  }
};
```

TCBUser_Cleanup. `TCBUser_Cleanup` is an extension aspect that implements the Flexible termination feature. For the sake of potential stack sharing, continuation control flows are generally constrained to terminate only from the call-depth level of their start function – otherwise remaining call contexts may not be freed. On platforms that implicitly share call contexts between all control flows (as on the TriCore) this constraint is compulsory.

However, as a restriction for the user level it might be inappropriate.[9] With the Flexible termination feature, it becomes permissible for user-level control flows to terminate even while executing some subfunction.

```
1  aspect TCBUser_Cleanup {
2    pointcut pcClass() = "hw::hal::TCBUser";
3    advice pcClass() : slice class CleanupSupport {
4      _tc::PCXI_t_nonv fcxi_;                          // tail pointer (for dummy context)
5     public:
6      void CIAO_INLINE cleanup() {
7        _tc::PCXI_t_nonv head = _mfcr( $pcxi );        // get head of CSA list
8        _mtcr( $pcxi, 0 );                             // "forget" it
9        _tc::free_cx_list( head, fcxi_ );              // add CSAs to the CPU's free list
10     }
11   };
12   advice call( "% ...::after_FirstCPUReceive(...)" )  // when a *TCBUser* starts...
13         && within ( pcClass()  ) : after() {
14     _svlcx();                                        // let the CPU create (and link) a CSA
15     tjp->target()->fcxi_ = _mfcr( $pcxi );           // remember its adress in fcxi_
16   }
17   advice execution( "% ...::before_LastCPURelease(...)" )
18         && within ( base( pcClass() ) ) : after() {
19     if( _mfcr( $pcxi) != 0 ) {                       // if there are still CSAs left
20       ((hw::hal::TCBUser*)(tjp->target()))->cleanup(); // we are a TCBUser and need to cleanup
21     }
22   }
23 };
```

Listing 6.2: TriCore implementation of the aspect TCBUser_Cleanup

Listing 6.2 shows the source code of `TCBUser_Cleanup`, which implements the Deterministic alternative of Flexible termination for the TriCore platform. When a continuation terminates, it adds all remaining call contexts to the free list of the CPU (lines 17–22). In order to not have to collect them by walking down the whole list (which would take an indeterministic amount of time), the aspect creates an extra dummy context that serves as a tail pointer (`fcxi_`) at the beginning of a `TCBUser` continuation. By using call-advice instead of execution-advice, the creation of this extra context only affects `TCBUser` continuations (lines 12–16).

Task_TCBBinding. The last aspect under discussion is not provided by the *hw::hal* layer as part of the continuation concept; it is an example for the implementation of the alternative Binding feature by the kernel. `Task_TCBBinding` is the binding aspect that is employed by the CiAO-AS kernel implementation to bind the user-level task implementations to continuations via an AUTOSAR-OS–compatible `TaskFunc` function pointer. As the switch to user level is itself a semantically important transition (for which the kernel employs separate explicit join points, see Table 6.1), the aspect furthermore signals this transition before activating the user code:

[9] This depends on the kernel personality and, hence, should be configurable. AUTOSAR OS, for instance, constrains the allowed termination points of user-level task functions in a similar manner; however, other operating systems do not impose such a restriction.

```
aspect Task_TCBBinding {
  ...
  advice execution( "% hw::hal::TCBUser::ThreadFunc(...)" ) : around() {
    os::krn::leaveKernel();            // we are going up
    hw::JUMP ( tjp->that()->start_ );  // execute user-level code
  }
};
```

6.4.4. Summary

The CiAO control flow abstraction is a good example for the aspect-aware design and implementation of a low-level system abstraction by applying the idioms discussed in Section 6.3. Visible transitions by explicit join points and advice-based binding facilitate the loose coupling of the core abstraction, its optional extensions, and the related policies. The result is a perfect one-to-one mapping from features to implementation components. New policies or extensions – either platform-independent or for some particular hardware platform – are easy to add. A good example for the latter would be an extension with respect to the amount of context information. If, for instance, a CPU architecture provides dedicated floating-point registers, saving and restoring these registers could be left to an extension aspect.

Especially the four explicit join points provided by class `Continuation`, which make all relevant transitions from the life cycle of a control flow visible on join-point level, turned out as particularly useful. Besides the aspects depicted in Figure 6.9, we can find customer aspects that make use of them in many other parts of the kernel. This includes policy aspects from the implementation of the architectural policies memory protection and timing protection, but also extension aspects that implement optional kernel features, such as the support for AUTOSAR-OS hook functions.

6.5. Case Study “Interrupt Synchronization”

In the domain of event-triggered systems, *interrupt requests* (*IRQs*) are the common approach to signal events from peripheral devices (such as the expiry of a timer or a level change on a digital I/O line) to the CPU. The CPU deals with the event by the (immediate or delayed) execution of a corresponding *interrupt service routine* (*ISR*). To ensure consistency of system state that is accessed by ordinary control flows as well as by ISRs, the operating system has to apply measures for *interrupt synchronization*. As pointed out in Section 3.1.2.2, interrupt synchronization is an architectural policy – the chosen strategy is transparent to the application, but can have a notable influence on non-functional properties, such as *latency* and *performance*. For these reasons, interrupt synchronization is implemented in CiAO as a configurable policy.

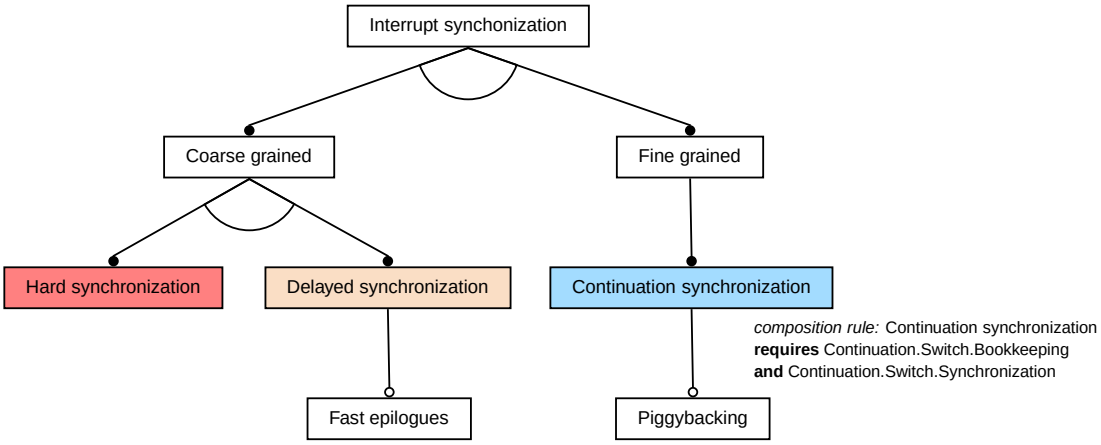


Figure 6.10.: Feature diagram of the configurable architectural policy *interrupt synchronization*.
CiAO supports three different strategies for interrupt synchronization: Hard synchronization, Delayed synchronization, and Continuation synchronization. Continuation synchronization requires, however, certain Continuation features (see Figure 6.7) to be present. (The coloring is for the purpose of feature traceability with Figure 6.11.)

6.5.1. CiAO Interrupt Synchronization Models

As in eCos (compare Section 3.1.1.2), interrupt handling in CiAO device drivers is explicitly divided into two parts: The first part, called *prologue*, is intended for time-critical actions and restricted with respect to the resources it may access, typically only hardware registers. Before termination, the prologue may request the (potentially delayed) execution of a second part. The second part, called *epilogue* in CiAO, is allowed to access other system components, such as the scheduler. The general idea is to execute the time-critical part immediately on interrupt level and the synchronized second part with a lower priority or at a later time when the required resources are available.

The feature diagram in Figure 6.10 depicts the offered variability for the architectural property interrupt synchronization. CiAO currently provides two different models for coarse-grained interrupt synchronization (Hard synchronization and Delayed synchronization) and one model for fine-grained interrupt synchronization (Continuation synchronization). They are all based on well-known techniques that are also used in other operating systems.

6.5.1.1. Hard Synchronization

In this configuration, the two parts are actually combined into one. When an interrupt occurs, prologue and epilogue are just executed consecutively on the interrupt level (interrupts remain disabled). Before accessing shared resources, threads have to enter interrupt level as well, that is, they have to disable interrupts.

The advantage of this model is its simplicity and low overhead. It is employed in many proprietary operating systems (compare Section 2.1.4) but also in sensor-network operating systems like TinyOS [Ber]. However, if interrupts are disabled too long or the

interrupt handler has to perform a time-consuming task, latency rises and IRQ signals might be lost.

6.5.1.2. Delayed Synchronization

The prologue is executed with low latency at interrupt level. Epilogues are executed on their own epilogue level; they are queued until the kernel propagates them for execution, which is the case after all nested prologues have terminated and before the scheduler is activated. Epilogues thereby have priority over threads, but are interruptible by prologues if new IRQ signals come in. Threads inside the kernel can temporarily disable the propagation of epilogues to access shared resources. In this case, epilogue propagation is delayed until the thread finishes its access. Interrupts only have to be disabled if a thread or epilogue operates on prologue-accessible state.

If low latencies for critical handler code are crucial, this is our model of choice, as prologue deferment is rare and short. Many operating systems employ means for such a delayed execution of the major handler code: The term *epilogues* as used in CiAO stems from PEACE [SP94]; in eCos, epilogues are called *delayed service routines* (*DSRs*) (see Section 3.1.1.2), in Linux *Tasklets* [RC01, Lov05], and Windows calls them *deferred procedure calls* (*DPCs*) [SR00].

The optional Fast epilogues feature represents an optimization. When this feature is applied, epilogues are – if possible – executed directly without queueing them first.[10]

6.5.1.3. Continuation Synchronization

In this configuration, the role of the prologue is the same as above. If an epilogue is requested, interrupts are reenabled and a new *continuation* (basic thread abstraction in CiAO, see Section 6.4) is started to execute the epilogue code in its own control-flow context. Epilogues synchronize with other continuation objects (epilogues or threads) via mutex objects using a priority inheritance protocol: The execution of the epilogue continuation can block on such mutex if a shared resource is currently in use by some thread or lower-priority epilogue; in this case, the owning control flow is reactivated until it frees the mutex. Hence, interrupts (respectively epilogues) and threads share (logically) a common priority space.[11]

The major advantage of this model is that it thereby becomes permissible for interrupt control flows to block. This facilitates on-demand and fine-grained locking of kernel components. The implementation in CiAO was inspired by the interrupt-as-coroutines

[10] This is possible, when (a) there are currently no nested prologues and (b) epilogue propagation has not been temporarily disabled.

[11] To avoid a scheduling overhead this is restricted in the current implementation. Epilogues just keep (and hand down) the current interrupt level, so they have priority over all threads and can only be interrupted and preempted by IRQs with a higher priority.

approach from Solaris [KE95]; however, the earliest references to this idea can be found in Moose [SP86] and Mach [ABG+86]. Several other systems, such as FreeBSD [MKM04] and L4 [Lie95], also execute interrupt handlers as threads.

The optional Piggybacking feature represents an optimization. When this feature is applied, interrupts “borrow” – if possible – the interrupted continuation control flow for the execution of their epilogue instead of starting their own continuation for this purpose.[12]

6.5.2. Design

In the following, I sketch the functional layers and their relevant components of interrupt synchronization in a bottom-up manner. The achieved separation of concerns through aspect-aware design will then be further detailed in Section 6.5.2.2; the implementation in Section 6.5.3.

6.5.2.1. Functional Layers

Figure 6.11 shows the structure of interrupt synchronization in CiAO as a layered model:

(1) Interrupt handling starts in the *hw*::*irq* layer (the interface to the underlying hardware), which contains one separate system component (a C++ class) for each (platform-specific) interrupt source. Each interrupt class provides a static `handler()` method as an explicit upcall join point (see Table 6.1). This join point is triggered when the corresponding interrupt occurs.

(2a) Binding aspects from the *os::dev* layer establish the link from hardware interrupt sources to corresponding system layer components (drivers). As a driver may service more than one IRQ, prologue and epilogue are contained in *virtual IRQ* (*VIRQ*) components inside the driver. VIRQs are the operating system’s software abstraction for hardware interrupt sources. Each VIRQ class provides an empty `handler()` method as an explicit transition join point for the execution policy.

(4a) The `Executor` policy aspect from the *policy* layer binds `VIRQ::handler()` to the proper activation of the actual interrupt handler implementation in `VIRQ::prologue()` and `VIRQ::epilogue()`.

(2) The *os* layer and its sublayers (*os::dev*, *os::krn*) contain the functional parts of the operating system, which are independent of the interrupt synchronization policy. Device drivers, but also other system components (such as the scheduler) are placed in this layer. Device drivers implement the interrupt service code as VIRQs (that is, define the behavior of `prologue()` and `epilogue()`), but have neither information nor any influence on the actual circumstances of their execution. Depending on the chosen synchronization model, a VIRQ may also act as a continuation or a delayed execution object. Every system component used by interrupts (either directly or

[12] This is possible when the interrupted continuation has not aquired a mutex.

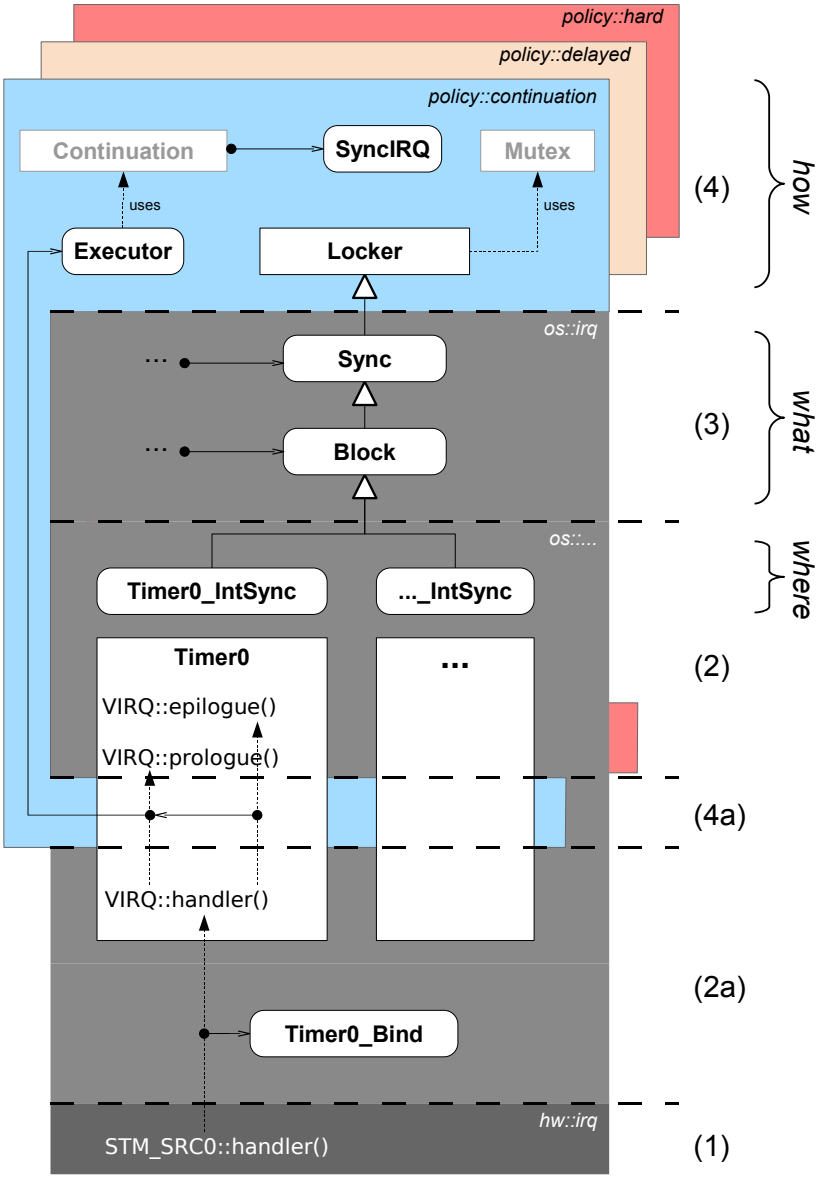


Figure 6.11.: Design model of the architectural policy interrupt synchronization in CiAO.
Depicted is, on the example of a `Timer0` device driver and the Continuation synchronization strategy, how interrupt processing engages with the policy implementation (`Executor`, `Locker`) of the chosen strategy. (The coloring indicates the corresponding feature from Figure 6.10.)

indirectly) is subject to interrupt synchronization and provides an accompanying `..._IntSync` aspect that describes its synchronization requirements.

(3) The *os::irq* layer is responsible for enforcing the synchronization constraints. The aspect `Block` enforces disabling of interrupts when methods are called that operate

on prologue-accessible state (will be explained in following next section). It may be deactivated if we want to combine prologues and epilogues, which means that they are actually synchronized with the same mechanism. This is always the case with the Hard synchronization strategy. The `Sync` aspect enforces protection of all methods that run on epilogue level.

(4) Finally, a *policy* layer implements the chosen model of interrupt synchronization.[13] A policy layer contains at least a `Locker` and an `Executor`. Whereas the `Locker` encapsulates the implementation of the locking mechanism, the `Executor` applies the policy-dependent mechanisms for the activation of prologues and epilogues. This responsibility includes the necessary transformations of VIRQs in a way that they are able to act as a continuation or a delayed execution object.

In the shown *policy::continuation* strategy, the `Executor` activates epilogues as new continuations using the `Continuation` abstraction as dispatcher. VIRQs have to be equipped with their own `Continuation` context for this purpose; locking is implemented by mutex objects. As the context switch mechanisms are now also activated from interrupt level (the `Executor` aspect introduces a new functional dependency between interrupts and the class `Continuation`), they have to be synchronized on interrupt level. For this purpose the `Continuation_SyncIRQ` aspect we discussed in Section 6.4 is employed.

6.5.2.2. Separation of Policies and Mechanisms

With respect to aspect-aware operating system development, the most interesting point of the sketched design is how the separation of policies and mechanisms is achieved. We can roughly divide the architectural policy of interrupt handling in two concerns:

Synchronization. The *synchronization* concern deals with the question which mechanism is used for the coordination of interrupt and thread control flows (deferring of interrupts, deferring of epilogues, mutex) and where it has to be applied.

Execution. The *execution* concern deals with the question which mechanism is used for the activation of interrupt control flows (direct, delayed, as separate continuation) and where it has to be applied.

Both concerns are not independent of each other – if we choose a blocking synchronization mechanism (such as mutex) we also need a preemptable execution mechanism (such as continuation). Hence, they together constitute an implementation of the interrupt synchronization policy.

To be able to develop system components (such as device drivers) in a way that they are transparent with respect to such a policy, a further separation is necessary. It can be expressed as simple questions of *how*, *where*, and *what*:

[13] The *policy* layers are not real layers in CiAO; they serve only illustrative purposes here to group the mechanisms that are used for a particular policy.

How. The answer to the question *how* we want to synchronize is the *policy-dependent* part of interrupt synchronization and execution. It defines which mechanisms are to be used for synchronization of components (`Locker`) together with a suitable execution model for prologues and epilogues (`Executor`).

Where. The answer to the question *where* synchronization measures have to be applied is the *component-dependent* part of interrupt synchronization. We need a representation of the synchronization requirements of each operating system component. As this knowledge depends on the component implementation, it has to be provided by the component developer. In the current implementation, synchronization requirements are encoded on a per-method level in the accompanying ...`IntSync` aspects, which provide named pointcuts that specify the members of each synchronization class.[14] We distinguish between three synchronization classes.

1. Most methods belong to the class *synchronized*, which means that they get synchronized on epilogue level.
2. Methods that access prologue-accessible state belong to the class *blocked* instead, which means that they are synchronized on prologue level.
3. Methods that only perform atomic or interrupt-transparent operations do not need to be subject of any synchronization measures and belong to the class *transparent*.

In principle, the *where* of prologue/epilogue activation is component-dependent, too. However, this knowledge is already encoded in an aspect-aware manner by the common structure and join-point interface of the VIRQ classes. Thereby it is possible to quantify over all points of prologue/epilogue activation with a single pointcut expression. Hence, in the actual implementation the *where* of the execution concern is not component-dependent and can directly be encoded as a pointcut in the `Executor` aspect.

What. Finally, we have to ensure that the chosen synchronization mechanism actually gets applied appropriately at the correct positions in the control flow. This part is independent of both the policy and the component. It is accomplished by pieces of advice given by the aspects `Sync` and `Block` from the *os::irq* layer. For the execution mechanism it is taken care of by pieces of generic advice given by the `Executor` aspect.

The concerns and their corresponding aspects are briefly summarized in Table 6.2.

[14] The specification of the synchronization requirements on method granularity and by manual maintenance of named pointcuts is not optimal. Ideally, we would be able to tag methods directly with their synchronization class – instead of listing them in a named pointcut. Even better would be the possiblity to apply tags on a finer granularity, such as on the level of code blocks. Currently, AspectC++ does not provide language means for this. However, if a concept of tagged attributes becomes available in AspectC++, it will be easy to catch this up in the discussed design as no policy-related parts will be affected.

aspect	#	concern
Binder	per IRQ–VIRQ mapping	upcall binding, hardware decoupling
VIRQ	per VIRQ per component	execution domain specification
Executor	per policy	execution mechanism and enforcement
Locker	per policy	synchronization mechanism
Sync	1	synchronization enforcement
Block	1	synchronization enforcement
IntSync	per component	synchronization domain specification

Table 6.2.: Concerns of the architectural policy *interrupt handling* and corresponding aspects in CiAO

6.5.3. Implementation

In the following, I provide a closer look at some selected parts of the implementation. A typical characteristic of architectural policies is that their implementation homogeniously crosscuts with the implementation of a (potentially unknown) number of kernel components. In the actual implementation, this is tackled with generic advice.

6.5.3.1. Component Implementation

Listing 6.3.a shows excerpts from the driver implementation for the `Timer0` timer device: The `windupPeriodical()` method arms the timer device to request an interrupt after the specified `time`. However, for the timer to do this periodically, the `tick()` method has to be invoked from the interrupt handler to re-arm the timer. This is considered time-critical, so it is done in the prologue. Therefore, the timer hardware registers and the period belong to *prologue-accessible state*. Consequently, `windupPeriodical()` and `tick()` belong to the synchronization class *blocked*. The callback functions, which are registered by `addEvent()` and eventually get triggered by `processEvents()`, are held in a queue. This queue belongs to the *epilogue-accessible s*tate; hence `addEvent()` and `processEvents()` are members of the synchronization class *synchronized*. The `value()` method simply reads the timer value; this happens atomically on this hardware architecture and does not need to be synchronized, so `value()` belongs to the synchronization class *transparent*. The accompanying `Timer0_IntSync` aspect in Listing 6.3.b encodes these decisions by specifying a named pointcut for each class. The named pointcuts `pcSynchronized()`, `pcBlocked()`, and `pcTransparent()` are actually definitions of pure virtual pointcuts from the aspects `Block` and `Sync` (Section 6.5.3.3), from which (indirectly) every `..._IntSync` aspect inherits.[15]

[15] An aspect may give advice to an yet undefined (pure virtual) pointcut, which turns the aspect into an *abstract aspect*; the pure virtual pointcut eventually has to be defined by some derived aspects. A more detailed example for *aspect inheritance* and (pure) virtual pointcuts in AspectC++ can be found in Appendix section A.2.

(a) Timer0.h

```
1  class Timer0 ... {
2    ... // state
3   public:
4    void windupPeriodical( long time );
5    long value() const;
6    void addEvent( const EventCB* cb );
7   private:
8    void tick();
9    void processEvents();
10   class VIRQ ... {
11     void handler();
12     void prologue() {
13       tick();
14     }
15     void epilogue() {
16       process_events();
17     }
18   };
19   ...
20 };
```

(b) Timer0_IntSync.ah

```
1  aspect Timer0_IntSync : public IntSync {
2    pointcut pcClass = "os::dev::Timer0";
3
4    pointcut virtual pcSynchronized() =
5      within( pcClass() ) && (
6         "% addEvent(...)"
7      || "% processEvents()"
8      );
9
10   pointcut virtual pcBlocked() =
11     within( pcClass() ) && (
12        "% windupPeriodical(...)"
13     || "% tick()"
14     );
15
16   pointcut virtual pcTransparent() =
17     within( pcClass() ) && (
18        "% value()"
19     );
20 };
```

Listing 6.3: Example for a CiAO device driver with corresponding ..._IntSync aspect.
(a) Class `Timer0` with inner class `Timer0::VIRQ` implements the driver for an interrupt-driven timer device. **(b)** The accompanying aspect `Timer0_IntSync` encodes the synchronization requirements of `Timer0` methods by assigning them to one of the pointcuts `pcSynchronized()`, `pcBlocked()`, or `pcTransparent()`.

6.5.3.2. Policy Implementation

The `Locker` is simply a type alias to a class which provides methods to be called in order to protect critical method calls. In the case of Hard synchronization that class looks like this:

```
struct Hard {
  static void enter() {
    hw::hal::CPU::disable();
  }
  static void leave() {
    hw::hal::CPU::enable();
  }
};
```

For the other models, more sophisticated actions are to be performed by these two methods. With Continuation synchronization, for instance, `Locker` is aliased to the kernel mutex class `IMutex`, which has to deal with priority inheritance and potential dispatching in `enter()` and `leave()`.

With respect to configurability, the more interesting part of model implementation is the `Executor` aspect, which again looks relatively simple in the case of Hard synchronization:

```
aspect Executor_Hard {
  advice execution("% os::...::VIRQ%::handler(...)") : after() {
    if (JoinPoint::That::prologue())  // execute prologue
```

```
      JoinPoint::That::epilogue();    // execute epilogue
  }
};
```

The `Executor` aspect is quantified over all VIRQ classes; it uses *generic advice* (Section 4.2) to bind `prologue()` and `epilogue()` via their static type (given by `JoinPoint::That`), so they can be inlined.

If we want to run the Delayed synchronization policy, the aspect has to enqueue the VIRQ for later execution of the epilogue. For this, it is necessary to transform VIRQ classes into queueable objects by introducing a `Queueable` base class:

```
aspect Executor_Delayed  {
  advice "os::...::VIRQ%" : slice class Gate : public Queueable {};
  advice execution("...::VIRQ%::handler(...)") : after() {
    if ( JoinPoint::That::prologue() ) {          // execute prologue
        Guard::relay( JoinPoint::That::Inst() );  // enqueue for later execution
  } }
};
```

To be queueable, an actual instance is needed for each VIRQ class, even though the VIRQ classes contain just static elements. To provide such instance the introduced slice `Gate` also transforms VIRQs into singletons.

The realization of Continuation synchronization requires one separate continuation context per VIRQ to which the `Executor` may switch for epilogue activation:

```
aspect Executor_Continuation  {
  advice "...::VIRQ%" : slice class {     // introduce its own Continuation
    static Continuation ctx;              // into every VIRQ
    static char *stack[cfIRQ_EPISTACK];
    static void cfHAL_STARTFUNC_ATTRIBUTES entry( Continuation* me) {
      me->after_FirstCPUReceive();
      epilogue();                         // run epilogue
  } };

  advice execution("...::VIRQ%::handler(...)") : after() {
    ...
    typedef JoinPoint::That VIRQ;
    if ( VIRQ::prologue() ) {                   // execute prologue
      Continuation::getActive()->saveAndStart(  // save current context and
        &VIRQ::stack[cfIRQ_EPISTACK],           //  start epilogue in its own
        VIRQ::entry, &VIRQ::ctx);               //  continuation
  } ... }
};
```

No object instance for the VIRQ is needed in this case as all members can be static.

6.5.3.3. Enforcement of Synchronization

To accomplish the task of combining *what*, *how* and *where*, the aspects of this layer use pure virtual pointcuts to which they give pieces of advice that contain the synchronization code. These pointcuts are later defined by the component-specific ..._IntSync aspects. The pcExclude() pointcut protects *synchronized* but magic code (for example the epilogue itself) from being affected by the piece of advice:

```
aspect Sync : Locker {

  pointcut virtual pcSynchronized() = 0;
  pointcut pcToSync() = call(pcSynchronized()
                        && !pcExclude())
                        && !within(pcSynchronized());

  advice pcToSync() : around() {
    enter();
    tjp->proceed();
    leave();
  }
};
```

For fine-grained locking as used by the Continuation synchronization strategy, every component has to be synchronized independently. This is achieved by the fact that a separate instantiation of the whole synchronization hierarchy is performed for each (component-specific) ..._IntSync aspect, resulting in one IMutex per component. In this case the Sync aspect instruments all calls into "foreign" synchronization domains to obtain and release the respective Mutex instance around the call.

With coarse-grained locking as used by the Hard synchronization and Delayed synchronization strategies, all components share a single synchronization domain. This is realized by combining the component-specific definitions of the virtual pointcuts pcSynchronized(), pcBlocked(), and pcTransparent().

6.5.4. Interrupt Latency Comparison

Table 6.3 compares the relative prologue and epilogue activation overhead of the three implementations. The numbers represent the latency in the optimal case: no other control flow is in the kernel that blocks or delays the execution of prologues and epilogues. It is therefore not surprising that the implementation of Hard synchrionization performs best as this model involves the lowest ground overhead. With Delayed synchronization, the prologue activation time is identical, however the potentially delayed execution of the epilogue causes some overhead. As expected, the overhead is highest in the implementation of Continuation synchronization. For the later context switch out of interrupt state, the TriCore CPU requires some additional processing before entering the prologue, which causes the higher latency for its activation. The context switch to activate the epilogue itself comes

	t_{prologue}	t_{epilogue}	t_{iret}
hard	*8*	*8*	*16*
delayed	*8*	*40*	*60*
continuation	*16*	*60*	*108*
ProOSEK category 1 ISR	*12*	–	*20*
ProOSEK category 2 ISR	–	*12*	*20*

Table 6.3.: Latencies for non-delayed interrupts in CiAO and ProOSEK.
Depicted numbers are the elapsed time [*cycles*] from the begin of the hardware interrupt handler to the first *prologue* instruction, *epilogue* instruction, and until interrupt termination (*iret*)
[Measurements were performed on a TC1796b running at 50MHz clock speed. Code was compiled with AC++1.0PRE3 and TRICORE-GCC-3.3 using -O3 optimizations and executed from internal no-wait-state RAM. Measurements were performed with a hardware trace analyzer (Lauterbach). All results were measured (and turned out to be stable) over 10 iterations.]

at a price, too, even though 60 cycles can still be considered as a fairly small overhead for the gained flexibility of fine-grained locking.

Even though the latency of Continuation synchronization is highest in Table 6.3 (the ideal case without any delays), it can quickly pay off: If the length of epilogue locks caused by other interrupt handlers or kernel components exceeds the amount of 20 cycles, we already could have a break-even to Delayed synchronization. This, however, always depends on particularities of the concrete application (event frequency, deadlines, processor utilization, ...) and, thus, should be configurable.

The last two rows show the interrupt latency of category 1 and category 2 ISRs in ProOSEK.[16] In OSEK OS, ISRs cannot be split into two parts; instead they run either, as a prologue, outside of the kernel (category 1 ISR) or, similar to an epilogue, synchronized with the kernel (category 2 ISR) [OSE05, p. 25].

6.5.5. Summary

Interrupt synchronization is a good example for the decoupling of policies and mechanisms of even architectural policies in CiAO. Whereas the enforcement of a usual kernel policy (such as the preemption strategy presented in Figure 6.5) affects only a small selection of well-known components, it is characteristic for an architectural policy that its implementation crosscuts with the implementation of a (potentially unknown) number of kernel components. Hence, it has to be quantifiable – which requires some preparations on the side of the affected components. CiAO's kernel components are *aware* of interrupt synchronization – they adhere to a common driver model for VIRQs and explicitly specify their synchronization requirements. However, they do not have to know the concrete strategy. Thanks to generic advice and static typing, this architectural transparency can be implemented in way that leads to quite efficient yet flexible and concern-separated code.

[16] ProOSEK by Elektrobit Automotive GmbH is a commercial OSEK OS implementation that is widely used in the German automotive industry.

6.6. Case Study "CiAO-AS"

The purpose of the "CiAO-AS" study was to evaluate the identifed principles and idioms of aspect-aware operating-system development on a larger scale, that is, by construction of a complete kernel. For a variety of reasons, the AUTOSAR-OS standard is a particularly interesting test subject for this kind of evaluation (see also Section 3.1.2.3):

- AUTOSAR OS is not a concrete system but a standard, described by a set of requirements and a detailed specification of the system services (API) and abstractions. This makes it possible to evaluate the approach with a top-down implementation of real-world requirements.
- The suggested protection facilities (*memory protection*, *timing protection*, and *service protection*) make AUTOSAR OS a convincing case for architectural configurability.
- AUTOSAR is a "hot topic" in the embedded systems domain.

CiAO-AS is an AUTOSAR-OS–like operating system based on the CiAO kernel. It is AUTOSAR-OS–*like* in the sense that it implements almost all functional requirements and features specified by AUTOSAR OS [AUT06a, AUT06b] (tasks, ISRs, events, resources, alarms, hooks, ...) but does not claim to adhere entirely to the specification.[17]

In the following sections, I present and discuss some results from the "CIAO-AS" study. As we already have discussed several examples for the aspect-aware design and implementation of CiAO's system abstractions on a relatively high level of detail, I shall concentrate in the following more on the achieved general results. This includes the global analysis of AUTOSAR-OS concerns and their interdependencies, the implementation of these concerns by aspect-aware operating-system development in CiAO, and the memory and execution-time footprint of the resulting kernel.

6.6.1. Analysis Results – From Requirements to Concerns

As described in Section 3.1.2.3, the AUTOSAR-OS standard proposes a set of *conformance-and-scalability classes* for the purpose of system tailoring. These classes are, however, relatively coarse-grained and do not clearly separate between conceptually distinct concerns. As CiAO aims at a much better granularity (see Section 6.1.1), every AUTOSAR-OS concern is represented as an individual feature in CiAO-AS.

Table 6.4 presents – in a condensed form – the results of the analysis of the AUTOSAR-OS concerns. It lists the identified concerns of AUTOSAR OS (column headings) and how we can expect them to interact with the named entities of the implementation (row headings), that is, the 44 system services (e.g., `ActivateTask()`) and the relevant system object types (e.g., `TaskType`) specified in [OSE05, AUT06b]. Additionally examined in Table 6.4

[17] The differences are mostly related to language and tools. For instance, CiAO-AS applications have to be written in C++ and woven with AC++.

are some *internal concerns* (Preemption, Kernel synchronization) and *internal transitions* (events) that are not mentioned explicitly in the AUTOSAR-OS specification, but that are nevertheless of high relevance. These concerns were examined by experience and deduction.

Table 6.4 thereby provides an overview on how we can expect AUTOSAR-OS concerns to crosscut with each other in the structural space (types, services) and behavioral space (control flows events) of the implementation. We can see, for instance, that the design of the API is canonical – each system service and object type is motivated (introduced) by exactly one concern. The state to be maintained in the listed object types, however, is influenced by several concerns, so is the behavior that is associated with the execution of a system service. In fact, there is not a single AUTOSAR-OS service that is influenced by only one concern!

Some concerns are “highly crosscutting”, in the sense that their implementation is expected to touch a high number of system services or object types. The Hooks facility, for instance, includes support for several application-specific signal handlers, among them the `ErrorHook` that is to be invoked in case of an error.[18] In the implementation, this touches every system service that may return with an error code (`StatusType`).

As expected, some of the architectural protection classify as “highly crosscutting”, too. This is particularly true for the various Service protection constraints to be checked for – the enforcement of these constraints is naturally associated with the execution of system services. Memory protection, on the other hand, is mostly associated with system-internal events.[19]

The identified system-internal events are of particular importance with respect to an aspect-aware development as they reflect relevant transitions that are *not* implicitly provided by a system service. Instead, I had to deal with these transitions explicitly in the design and implementation, for example, model them as explicit join-points. We have already seen several examples for this in Section 6.3.4.

Overall, the concerns defined by the AUTOSAR-OS specification documents [AUT06b, OSE05] bear a surprisingly high amount of crosscutting in the specified services and abstractions. A concrete implementation should profit significantly from the aspect-aware development approach.

[18] In Table 6.4, the Hooks concern subsumes all six hooks specified by AUTOSAR OS (`ErrorHook`, `ProtectionHook`, `StartupHook`, `ShutdownHook`, `PreTaskHook`, `PostTaskHook`); in the CiAO-AS implementation the support for each hook can be selected invididually (see also Figure 6.2 on page 127).

[19] The architectural policy Memory protection is not further elaborated in this thesis. Please consult [LSH+07] for details regarding the design and implementation of CiAO’s Memory protection facilities.

Table 6.4. (both pages): Influence of configurable features on system services, system types, and internal events in AUTOSAR OS.

The table displays the results a concern impact analysis on the base of the AUTOSAR-OS requirements [AUT06a] and specification [AUT06b, OSE05]. Depicted is the influence of the identified AUTOSAR-OS concerns (columns) on specified and deduced implementation entities (rows).

The implementation entities in the rows are (from top to bottom): *system services* (functions), *system objects* (types), and, slightly separated, *internal events* that were deduced during the analysis (not part of the AUTOSAR-OS specification). The concerns in the columns are (from left to right): *system abstractions* (functional concerns), *callbacks* (upcalls into the application), *protection facilities* (architectural concerns), and, slightly separated, *internal* concerns deduced during the analysis (not part of the AUTOSAR-OS specification).

Kind of influence:
⊕ = extension of the API by a service or type
⊛ = extension of an existing type
◑ = modification after service or event
◐ = modification before
● = modification before *and* after

	System abstractions (functional)						
	OS control	Tasks	ISRs category 1	ISRs category 2	Resources	Events	Alarms
`GetActiveApplicationMode()`	⊕						
`StartOS()`	⊕						
`ShutdownOS()`	⊕						
`ActivateTask()`		⊕					
`TerminateTask()`		⊕					
`ChainTask()`		⊕					
`Schedule()`		⊕					
`GetTaskID()`		⊕					
`GetTaskState()`		⊕					
`EnableAllInterrupts()`			⊕				
`DisableAllInterrupts()`			⊕				
`ResumeAllInterrupts()`			⊕				
`SuspendAllInterrupts()`			⊕				
`ResumeOSInterrupts()`				⊕			
`SuspendOSInterrupts()`				⊕			
`GetISRID()`				⊕			
`DisableInterruptSource()`				⊕			
`EnableInterruptSource()`				⊕			
`GetResource()`					⊕		
`ReleaseResource()`					⊕		
`SetEvent()`						⊕	
`ClearEvent()`						⊕	
`GetEvent()`						⊕	
`WaitEvent()`						⊕	
`IncrementCounter()`							⊕
`GetAlarmBase()`							⊕
`GetAlarm()`							⊕
`SetRelAlarm()`							⊕
`SetAbsAlarm()`							⊕
`CancelAlarm()`							⊕
`StartScheduleTableRel()`							⊕
`StartScheduleTableAbs()`							⊕
`StopScheduleTable()`							⊕
`NextScheduleTable()`							⊕
`SetScheduleTableAsync()`							⊕
`SyncScheduleTable()`							⊕
`GetScheduleTableStatus()`							⊕
`GetApplicationID()`							
`TerminateApplication()`							
`CallTrustedFunction()`							
`CheckObjectAccess()`							
`CheckObjectOwnership()`							
`CheckISRMemoryAccess()`							
`CheckTaskMemoryAccess()`							
`AppModeType`	⊕	⊛					⊛
`TaskType`		⊕			⊛	⊛	
ISR category 2				⊕			
`ResourceType`					⊕		
`AlarmType/ScheduleTableType`		⊛				⊛	⊕
`ApplicationType`							
alarm expiry		◑				◑	
category 2 ISR execution							
system startup		◑					◑
system shutdown							
protection violation							
task switch							
application switch							
uncontrolled task end							
user ↦ kernel transition							
kernel ↦ user transition							

Callbacks		Protection facilities (architectural)											Internal		
Alarm callbacks	Hooks	OS applications	Memory protection	Stack monitoring	Timing protection	Invalid parameters	Out of range	Wrong context	Missing task end	Enable w/o disable	Interrupts disabled	Nontrusted shutdown	Foreign OS objects	Preemption	Kernel sync	
	◑							◐			◐					`GetActiveApplicationMode()`
						◐		◐			◐		◐			`StartOS()`
	◑							◐			◐	◐				`ShutdownOS()`
	◑					◐		◐			◐		◐	◑		`ActivateTask()`
	◑							◐			◐			◑		`TerminateTask()`
	◑					◐		◐			◐		◐	◑		`ChainTask()`
	◑							◐			◐			◑		`Schedule()`
	◑							◐			◐					`GetTaskID()`
	◑					◐		◐			◐		◐			`GetTaskState()`
					◐			◐		◐						`EnableAllInterrupts()`
					◑			◐								`DisableAllInterrupts()`
					◐			◐		◐						`ResumeAllInterrupts()`
					◑			◐								`SuspendAllInterrupts()`
					◐			◐		◐						`ResumeOSInterrupts()`
					◑			◐								`SuspendOSInterrupts()`
								◐			◐					`GetISRID()`
	◑					◐		◐			◐		◐			`DisableInterruptSource()`
	◑					◐		◐			◐		◐			`EnableInterruptSource()`
	◑				◑	◐		◐			◐		◐			`GetResource()`
	◑				◐	◐		◐			◐		◐	◑		`ReleaseResource()`
	◑					◐		◐			◐		◐	◑		`SetEvent()`
	◑							◐			◐					`ClearEvent()`
	◑					◐		◐			◐		◐			`GetEvent()`
	◑							◐			◐			◑		`WaitEvent()`
	◑					◐		◐			◐		◐	◑		`IncrementCounter()`
	◑					◐		◐			◐		◐			`GetAlarmBase()`
	◑					◐		◐			◐		◐			`GetAlarm()`
	◑					◐	◐	◐			◐		◐			`SetRelAlarm()`
	◑					◐	◐	◐			◐		◐			`SetAbsAlarm()`
	◑					◐		◐			◐		◐			`CancelAlarm()`
	◑					◐	◐	◐			◐		◐			`StartScheduleTableRel()`
	◑					◐	◐	◐			◐		◐			`StartScheduleTableAbs()`
	◑					◐		◐			◐		◐			`StopScheduleTable()`
	◑					◐		◐			◐		◐			`NextScheduleTable()`
	◑					◐		◐			◐		◐			`SetScheduleTableAsync()`
	◑					◐		◐			◐		◐			`SyncScheduleTable()`
	◑					◐		◐			◐		◐			`GetScheduleTableStatus()`
		⊕						◐			◐					`GetApplicationID()`
	◑	⊕						◐			◐					`TerminateApplication()`
	◑	⊕						◐			◐					`CallTrustedFunction()`
		⊕				◐		◐			◐		◐			`CheckObjectAccess()`
		⊕						◐			◐					`CheckObjectOwnership()`
			⊕			◐		◐			◐		◐			`CheckISRMemoryAccess()`
			⊕			◐		◐			◐		◐			`CheckTaskMemoryAccess()`
																`AppModeType`
		⊛	⊛	⊛	⊛								⊛	⊛		`TaskType`
		⊛			⊛								⊛			ISR category 2
		⊛											⊛			`ResourceType`
⊛		⊛					⊛						⊛			`AlarmType/ScheduleTableType`
		⊕	⊛													`ApplicationType`
◑														◑		alarm expiry
			●		●									◑		category 2 ISR execution
	◑															system startup
	◐															system shutdown
	◑															protection violation
	●		◑	◐	●											task switch
			◑										◑			application switch
									◑							uncontrolled task end
			◑												◐	user ↦ kernel transition
			◑												◑	kernel ↦ user transition

6.6.2. Development Results – From Concerns to Classes and Aspects

In its full configuration, the CiAO-AS kernel source bears three basic system components that are singletons by definition and represented as classes:

1. The *scheduler* (`Scheduler`) takes cares about the dispatching of tasks and the scheduling strategy.
2. The *alarm manager* (`AlarmManager`) takes cares about the management of alarms and the underlying (hardware / software) counters.
3. The *OS control facility* (`OSControl`) provides services for the controlled startup and shutdown of the system and the management of OSEK-OS / AUTOSAR-OS application modes.

Also represented as classes are the system abstractions (the types that represent instantiable system objects, such as `TaskType`, `ResourceType`, and so on) and the namespace of the API (`AS`).

However, as described in Section 6.3.1, these classes are sparse or even empty. If at all, they implement only a minimal base of their respective concern. All further features and variants are brought into the system by aspects. The class `Scheduler`, for instance, provides only the minimal base of the scheduling facility, which is nonpreemptive scheduling with single task activations. Support for more sophisticated preemption or activation modes is provided by additional *policy aspects* and *extension aspects*.

Table 6.5 displays an excerpt of the list of AUTOSAR-OS concerns that have been implemented as aspects in CiAO-AS.

The first three columns list for each concern the number of *extension*, *policy*, and *upcall aspects* that implement the concern. For all concerns, the implementation could be realized as a single aspect. (The *further* separation into an extension aspect and a policy aspect in two cases (Resource support and Protection hook) is owed to the goal of strict decoupling of mechanisms and policies suggested by the CiAO approach, see Section 6.1.1.)

The majority of concerns from Table 6.5 contributes to the set of *policy aspects* (12 aspects), which by extend is followed by the set of *extension aspects* (9 aspects). The number of *upcall aspects* $(3+n+m)$ differs from these in so far as it does not only depend on the system configuration, but also on the application configuation: Each specified ISR in the application is bound with the respective interrupt source in the kernel or HAL by its own upcall aspect. These aspects are, however, not to be provided by the application developer; they are generated automatically from the application configuration.

Another interesting point is the realization of synergies by *quantification*. If for some concern the number of pieces of advice is lower than the number of affected join points (displayed in the last two columns of Table 6.5) we have actually profited from the AOP concept of quantification. For 14 out of the 22 concerns listed in Table 6.5 this is the case.

The net amount of this profit, however, depends on the type of the concern and aspect. *Extension aspects* typically crosscut *inhomogeniously* with the implementation of other

concern	extension	policy	upcall	advice	join points	extension of | advice-based binding to
ISR cat. 1 support	1		m	$2+m$	$2+m$	API, OS control | m ISR bindings
ISR cat. 2 support	1		n	$5+n$	$5+n$	API, OS control, scheduler | n ISR bindings
ISR abortion support	1			2	$1+m+n$	scheduler | $m+n$ ISR functions
Resource support	1	1		3	5	scheduler, API, task | PCP policy implementation
Resource tracking		1		3	4	task, ISR | monitoring of Get/ReleaseResource
Event support	1			5	5	scheduler, API, task, alarm | trigger action JP
Alarm support	1			1	1	API
OS application support	1			2	3	scheduler, task, ISR
Full preemption		1		2	6	| 3 points of rescheduling
Mixed preemption		1		3	7	task | 3 points of rescheduling for task / ISR
Multiple activation support	1			3	3	task | binding to scheduler
Stack monitoring		1		2	3	task | CPU-release JPs
Context check		1		1	s	| s service calls
Disabled interrupts check		1		1	30	| all services except interrupt services
Enable w/o disable check		1		3	3	| enable services
Missing task end check		1		1	t	| t task functions
Out of range check		1		1	4	| alarm set and schedule table start services
Invalid object check		1		1	25	| services with an OS object parameter
Error hook			1	2	30	scheduler | 29 services
Protection hook	1	1		2	2	API | default policy implementation
Startup / shutdown hook			1	2	2	| explicit hooks
Pre-task / post-task hook			1	2	2	| explicit hooks

Table 6.5.: CIAO-AS kernel concern implemented as aspects with number of affected join points.
Listed are kernel concerns that are implemented as *extension*, *policy*, or *upcall aspects* (not including aspects for memory protection, timing protection, and *hw::hal*-bindings), together with the related pieces of *advice* (not including order-advice), the affected *join points*, and a short explanation for the purpose of each join point (separated by “|” into *introductions of extension slices* | *advice-based binding*).

concerns, which does not leave much potential for synergies by quantification. *Policy aspects* on the other hand – especially those for architectural policies – tend to crosscut *homogeniously* with the implementation of other concerns; here quantification creates significant synergies. Note, however, that in many cases the necessary homogenity of the affected join points could only be achived by using generic advice.[20]

Overall, the identified AUTOSAR-OS concerns (cf. Section 6.6.1) could be well separated into distinct implementation units by applying the principles and idioms of aspect-aware operating system development.

[20] You may remember the `ServiceProtection_InvalidObjectCheck` aspect that served as an example for generic advice in Section 4.2.3. It actually implements the Invalid object check feature from Table 6.5.

6.6.3. Evaluation Results – From Configurations To Cost

In the following, I present some results from the performance and memory footprint evaluation of the CiAO-AS implementation that demonstrate the achieved granularity.

6.6.3.1. Scalability of Memory Requirements

Table 6.6 displays how the memory requirements of the CiAO-AS kernel scale up with the amount of selected features. The *base system*, which stands for the Task management and OS control features, comprises about three KiB of code and 80 bytes of RAM. This seems to be surprisingly high, however, is distorted by the not yet optimized startup code and library code. The resulting image contains several debug facilities, including a serial driver and a `printf()` implementation. Hence, this figure should only be understood as the base size for the relative cost of the other features.

Each `Task` object, for instance, takes 20 bytes of *data* for the kernel task context (priority, state, function, stack, interrupted flag) and 16 bytes (*bss*) for the underlying `TCBUser` continuation structure (saved PCXI register, return address, start function, task ID). Aspects from the implementation of other features, however, may extend the size of the kernel task context. Event support, for instance, crosscuts with Task management in the implementation of the `Task` structure, which it extends by 8 bytes to accommodate the current and waited-for event masks.

The cost of several features scales up with the amount of affected join points, which in turn depends on the presence of other features. For example, the overhead of selecting a distinct preemption policy than the implicit nonpreemptive depends on the number of additional points of rescheduling (the execution of the services `ActivateTask()`, `ReleaseResource()`, and `SetEvent()`), which have to be affected by the preemption policy aspect. Hence, the total overhead is determined by the actual presence of these system services in the configuration. If, for instance, Resource support is not part of the current configuration, the piece of advice given by the preemption policy aspect to `ReleaseResource()` has just no effect and, thus, does not induce a cost. This effect underlines again the flexibility of loose coupling by advice-based binding.

Overall, we can ascertain CiAO a good granularity. The memory requirements scale up well with the set of selected features.

6.6.3.2. Execution-Time Comparison

Table 6.7 displays the execution times of several tasking-related test scenarios (explained in Listing 6.4) on CiAO and ProOSEK. For each of the microbenchmarks, both systems were configured to support the smallest possible set of features.

The differences between CiAO and ProOSEK are formidable. CiAO is noticeable faster in all test scenarions, with up to 2.6 times better execution times. This is due to several reasons:

feature	with feature or instance	text	data	bss
base system (OS control and tasks)		*2890*	*24*	*56*
	with Alarm support	*+ 32*	*0*	*0*
	per task	*+ task function size + 0*	*+ 20*	*+ stack size + 16*
	per application mode	*0*	*+ 4*	*0*
	per alarm	*0*	*+ 8*	*0*
ISR cat. 1 support		*0*	*0*	*0*
	per ISR	*+ ISR function size + 0*	*0*	*0*
	per disable–enable	*+ 4*	*0*	*0*
Resource support		*+ 128*	*0*	*0*
	per resource	*0*	*+ 4*	*0*
	per task	*0*	*+ 8*	*0*
Event support		*+ 280*	*0*	*0*
	with Alarm support	*+ 54*	*0*	*0*
	per task	*0*	*+ 8*	*0*
	per alarm	*0*	*+ 12*	*0*
Alarm support		*+ 568*	*0*	*+ 24*
	per alarm	*0*	*+ 16*	*0*
	per application mode	*0*	*+ 4*	*0*
Alarm callback support		*+ 24*	*0*	*0*
	per alarm	*0*	*+ 8*	*0*
Full preemption		*0*	*0*	*0*
	per join point	*+ 12*	*0*	*0*
Mixed preemption		*0*	*0*	*0*
	per join point	*+ 44*	*0*	*0*
	per task	*0*	*+ 4*	*0*
Stack monitoring		*0*	*0*	*0*
	per join point	*+ 44*	*0*	*0*
	per task	*0*	*+ 4*	*0*
Context check		*0*	*0*	*0*
	per void join point	*0*	*0*	*0*
	per StatusType join point	*+ 8*	*0*	*0*
Disabled interrupts check		*0*	*0*	*0*
	per join point	*+ 64*	*0*	*0*
Enable without disable check		*+ 14*	*0*	*0*
Missing task end check		*0*	*0*	*0*
	per task	*+ 58*	*0*	*0*
Out of range values check		*0*	*0*	*0*
	per join point	*+ 152*	*0*	*0*
	per alarm	*0*	*+ 8*	*0*
Invalid objects check		*0*	*0*	*0*
	per join point	*+ 36*	*0*	*0*
ErrorHook		*0*	*0*	*+ 4*
	per join point	*+ 54*	*0*	*0*
StartUpHook or ShutDownHook		*0*	*0*	*0*
PreTaskHook or PostTaskHook		*0*	*0*	*0*

Table 6.6.: Scalability of CiAO’s memory footprint.

Listed are the static memory demands [Byte] of the CiAO base system (first line) and the relative increment for a selection of optional features.

[Number of bytes retrieved from linker map files. All variants were woven and compiled with AC++-1.0PRE3 and TRICORE-G++-3.4.3 using -O3 -fno-rtti -funit-at-a-time -ffunction-sections -Xlinker --gc-sections optimization flags.]

test scenario	CiAO	ProOSEK
(a) voluntary task switch	*160*	*218*
(b) forced task switch	*108*	*280*
(c) preemptive task switch	*192*	*274*
(d) system startup	*194*	*399*

Table 6.7.: Performance measurement results from CiAO and ProOSEK.
Listed numbers represent execution times [*cycles*] taken by CiAO and ProOSEK when executing the test scenarios from from Listing 6.4 (execution time between `FROM` and `TO` in the corresponding test case). For each figure the operating systems were configured with the minimal set of features supporting the respective test case.
[Measurements were performed on a TC1796b running at 50MHz clock speed while executing from internal no-wait-state RAM. Measurements were performed with a hardware trace analyzer (Lauterbach). All results were measured (and turned out to be stable) over at least 10 iterations.]

- First and foremost, CiAO provides a much better configurability (and thereby granularity), than ProOSEK. As the test scenarios utilize only a relatively small set of AUTOSAR-OS features, and both systems where configured to support the *smallest possible* amount of features, this has a significant effect on the resulting execution times. The smallest possible configurations of ProOSEK still contained a lot of ballast. The scheduler is synchronized with ISRs, for instance; however, there are no ISRs in the application scenario possibly interrupting the kernel.[21]
- Another reason is that ProOSEK's internal implementation of continuations is less efficient than CiAO's (see Section 6.4.3). ProOSEK additionally saves the volatile registers when preempting a task. Furthermore, the dispatching code does not distinguish internally between `start()` of a new control flow and `switch()` to a preempted control flow. Hence, whenever a new task gets activated, a context has to be created for this task that resembles the context of a preempted task. On the TriCore platform this is expensive, as it requires manual fiddling with the list of free CSAs, for which – even worse – interrupts have to be disabled.

Overall, we can ascertain CiAO not only a good granularity, but also a quite competitive performance.

6.6.4. Summary

The results from the "CiAO-AS" study show that the approach of aspect-aware operating-system development is both feasible and benefitial for the implementation of real-world operating systems. The concerns, services, and abstractions defined by the AUTOSAR-OS standard bear a noticeable amount of internal crosscutting. Nevertheless, by applying the principles and idioms of aspect-aware operating-system development, they could be implemented in a well-separated and fine-grained manner in CiAO-AS.

[21] Apparently, ProOSEK bears the same unnecessary coupling of *preemption* and *synchronization* we already have observed in eCos (see Section 5.3).

(a) voluntary switch, nonpreemptive system

```
TASK( Task0 ) { // low priority
  ActivateTask( Task1 );
  asm volatile( "FROM:" );
  Schedule();
  ChainTask( Task0 );
}

TASK( Task1 ) { // high priority
  asm volatile( "TO:" );
  TerminateTask();
}
```

(b) forced switch, nonpreemptive system

```
TASK( Task0 ) {
  ActivateTask( Task1 );
  asm volatile( "FROM:" );
  TerminateTask();
}

TASK( Task1 ) {
  asm volatile( "TO:" );
  ActivateTask( Task0 );
  TerminateTask();
}
```

(c) preemptive switch, fully preemptive system

```
TASK( Task0 ) { // low priority
  asm volatile( "FROM:" );
  ActivateTask( Task1 );
  ChainTask( Task0 );
}
TASK( Task1 ) { // high priority
  asm volatile( "TO:" );
  TerminateTask();
}
```

(d) system startup

```
int main() {
  asm volatile( "FROM:" );
  StartOS( OSDEFAULTAPPLICATIONMODE );
}

TASK( Task0 ) {
  asm volatile( "TO:" );
  TerminateTask();
}
```

Listing 6.4: Performance measurement test scenarios.
Taken into account are all instructions executed from the `FROM` label to the `TO` label. **(a)** Voluntary task switch in a nonpreemptive system: `Task1` gets the CPU when running `Task2` invokes `Schedule()`. **(b)** Forced task switch in a nonpreemptive system: activated `Task1` gets the CPU when running `Task0` terminates. **(c)** Preemptive switch in a fully preemptive system: `Task1` gets the CPU the moment it is activated by `Task0`. **(d)** System startup: `StartOS()` initializes the operating system and starts the scheduler, which dispatches `Task0`.

6.7. Discussion of Results

With respect to the goals described in Section 6.1.1, the approach of aspect-aware operating system development was quite successful. CiAO reaches the primary research goal of **architectural configurability**. The system combines a good separation of concerns in the implementation with excellent granularity and configurability, and – thereby – a quite competitive efficiency regarding hardware resources.

This means that we can report success with respect to the design-level objectives from Section 3.3.3 (page 65) – AOP *does* lead to a better configurability of policies in an operating-system kernel; **even architectural policies can be implemented as configurable features**.

Overall, we can conclude that AOP is a well-suited tool for the implementation of configurability in system-software product lines for resource-constrained devices. It particularly

solves the problems that motivated this thesis (see Section 3.1.3 on page 63). These results, however, were achieved by a somewhat different, "pragmatic" understanding and application of AOP. In the following sections, I discuss some "aspects" of my approach.

6.7.1. Obliviousness Versus Awareness

An important difference between aspect-aware operating-system development and the "classical" understanding and application of AOP is how I dealt with the inherent tension between obliviousness and awareness.[22]

In their publications [FF00, FECA05], FILMAN and FRIEDMAN describe the obliviousness *ideal* of AOP. Ideally, obliviousness can be a *bidirectional* relationship between components and aspects: The programmers of the base system and the aspect developers can work completely independent of each other. ("Just program like you always do, and we'll be able to add the aspects later.")

However, total obliviousness is unrealistic for nontrivial aspect–component interactions. Hence, in actual applications of AOP obliviousness is usually understood *unidirectional*: The components of the base system are kept oblivious of aspects – at the price that the aspects have to be perfectly aware of the components they affect. This often involves knowledge about certain implementation details, which in turn leads to fragile pointcuts if the component *developers* are kept oblivious of the aspects, too. Furthermore, this approach hits its limits when the base code just does not offer the required join points. The "eCos" study from Chapter 5 is a good example.

Aware of these problems, the AOP community tends to suggest "better pointcut languages" with "semantic pointcuts" as the remedy to go for. However, I doubt that, for instance, the semantic ambiguity between points of *preemption* and points of *synchronization* in eCos (see Section 5.3.1 on page 113) could really be resolved by a better pointcut language.

Aspect-aware operating system development moderates these issues by pragmatically considering *obliviousness* and *awareness* as two ends of a continuum: The more oblivious a component should be of the aspects that potentially engage with it, the more aware the aspects have to be of the component – and vice versa. Much of the flexibility and configurability of CiAO stems from the freedom to decide for each relationship about the placement on this continuum. This is reflected in the three design principles of *loose coupling*, *visible transitions*, and *minimal extensions*.

6.7.2. AOP Critique – Revisited

This brings us back to the common AOP critique we have discussed in Section 2.3.8 on page 43. In his OOPSLA essay, STEIMANN concludes on the tension between obliviousness and awareness that "the problems of AOP cannot be fixed without giving up its distinguishing characteristics" [Ste06]. However, he implies that obliviousness (or awareness)

[22] After all, *aware* and *oblivious* are antonyms [McK05].

is an all-or-nothing property – which is not the case with aspect-aware operating-system development. Even though CiAO components are not completely oblivious to aspects, – the AOP mechanism behind obliviousness (which is inversion of control flow specifications by advice, compare Section 4.1.2.1 on page 70) turned out as particularly useful. The same holds for the mechanism behind quantification. Even though considered by some authors as “rarely applicable” [KAB07] – *only* quantification (that is, the implicit per–join-point instantiation of advice, compare Section 4.1.2.2 on page 72) made it possible to implement architectural polices as configurable features in CiAO.

6.8. Chapter Summary

The goal of this chapter was to evaluate AOP with respect to the *design level* of configurability in system software for embedded devices. This led to the design principles and development idioms of *aspect-aware operating-system development*.

The results from the CiAO project show that aspect-aware operating-system development is a sound and applicable approach.

7

Summary, Conclusions, and Further Ideas

In this thesis, I evaluated the application of AOP as a first-class mechanism for the implementation of configurability in operating-system product lines for resource-constrained embedded systems. The overall goal was to show that a well-directed, broad-scale application of AOP significantly improves on the state of the art to implement configurability in embedded operating systems without disadvantages on the hardware cost side.

7.1. Summary and Conclusions

The idea to evaluate AOP for the implementation of configurability in operating-system product lines for resource-constrained embedded systems was motivated by two problems:

Problem 1: Conditional compilation – the current state of the art for the implementation of fine-grained configurability – leads to commingling of concerns instead of their separation. While feasible for small projects, this technique does not scale, as the increasingly commingled preprocessor statements severely impact the readability and maintainability of the code – a situation commonly described by the pictorial term "`#ifdef` hell".

Problem 2: The configurable operating systems available today are yet not configurable enough – they do not provide configurability of architectural policies. Even though these policies are transparent to the application, they influence important nonfunctional properties and, thus, should be understood as configurable features.

Both problems are essentially problems of crosscutting, so aspect-oriented programming was a promising candidate for a remedy. It was, however, completely open if AOP can compete – qualitatively, but especially quantitatively – with the existing approaches, and if it facilitates to implement architectural policies as configurable features.

My research approach to answer these questions was to analyze and evaluate the suitability of AOP in a *bottom-up* manner on three levels of increasing abstraction: *language level*, *implementation level*, and *design level*.

Language-level conclusions: AOP does not induce an inherent overhead that makes it *per se* unacceptable for the domain of efficient system software. Key to success, however, is (1) to bind advice to join points at compile time whenever possible, that is, to avoid not-statically evaluable pointcuts, and (2) to provide means for overhead-free, static advice polymorphism – such as generic advice.

Implementation-level conclusions: AOP does not induce an overhead if applied to real system software with aspects that quantify over hundreds of join points. With the "eCos" study I could show that AOP compares very well to conditional compilation for the implementation of configurability; it leads to a much better separation of concerns without any disadvantages on side of the hardware cost. **Hence, AOP is a suitable remedy for Problem 1**.

However, at the same time we had to conclude that separating the implementation of architectural policies into aspects does not inherently make them more configurable.

Design-level conclusions: Instead, to achieve configurability of architectural policies, it is necessary to design the software and its components specifically *for* the application of policy aspects from the very beginning. The design rules of *aspect-aware operating-system development* facilitate such a consequent decoupling of policies and mechanisms in the implementation of an operating-system kernel. Thereby, it

becomes possible to implement even fundamental architectural policies as configurable features. I could show this on the example of the AUTOSAR OS standard and the CiAO operating-system product line, which combines a very competetive granularity and variability with the configurability of architectural policies. **Hence, AOP is a suitable remedy for Problem 2**.

7.2. Contributions

This thesis advances the state of the art in several directions. The following list describes the most relevant contributions:

- The new **generic advice** concept, which facilitates advice polymorphy in statically typed aspect languages, such as AspectC++. Only thereby it became possible to exploit quantification for the implementation of architectural policies without compromising on efficiency.
- The detailed **analysis and reduction of the overhead** induced by AspectC++ language constructs, which also led to a set of **best practices** on how to apply AOP in resource-critical environments. Developers of embedded systems can now be sure that the separation of concerns by AOP does not induce an extra overhead.
- The proof that AOP can lead to a **significantly better separation of concerns** in the implementation of configurable system software. Code scattering and tangling in real-world and state-of-the-art system software, such as the eCos operating system, can significantly be reduced without any compromise on the hardware cost side.
- The method of **aspect-aware operating system development**, which puts the ideas of obliviousness, quantification, and awareness in a new balance and has been applied in the development of the CiAO operating system.
- The proof that by aspect-aware operating system development it becomes possible to implement even fundamental **architectural policies as configurable features**.
- The **CiAO family of operating systems**, which is the first system family that has been developed with AOP concepts from scratch and offers a competitive implementation of the AUTOSAR OS standard.

7.3. Further Ideas

The Ideal Aspect Language for System Development. The implementation of CiAO and AspeCos with AspectC++ was successful. Still we learned that applying AOP to the low-level concerns of system sofware involves some very particular requirements that are not addressed in an optimal way so far. Generally desirable for this domain would be language extensions that provide more control over the generated code. It

should be possible, for instance, to assign compiler-specific attributes that control inlining and calling conventions to advice code as well. Language means to prevent advising of fragile parts would be helpful; so would be a better notion of explicit join points, such as the possibility to "tag" syntactic entities, code blocks, or even single statements with an annotation mechanism.

Optimization of Nonfunctional Properties. In Section 3.1.2, I discussed the relation between nonfunctional and architectural properties. Because of their emergent nature, nonfunctional properties can only be influenced indirectly. The motivation of a configurable architecture stems from the observation that architectural policies have a high influence on certain nonfunctional properties without impairing the functional specification of the system. Hence, they are ideal variation points for optimizing a system with respect to certain nonfunctional properties, such as latency, performance, or memory footprint.

With its configurable architecture, CiAO now provides excellent means for such optimizations. Still open is, however, *how* we can find the best – or at least a reasonably good – configuration for a particular application and its characteristic workload. A brute-force approach (that is, to manually or automatically generate and evaluate all possible variants that fulfill the functional specification) will quickly hit its limits, as the number of variants to check grows exponentially with the number of architectural features. Hence, we need measures for an upfront assessment of promising and unpromising feature selections to reduce the number of configurations to test.

A

Appendix: AspectC++

AspectC++ [SGSP02, SLU05a, SL07] is a general-purpose AOP extension for the C++ language [Str97] as specified by ISO/IEC 14882:2003 (C++98) [Ins03]. The AspectC++ language and all related tools (especially the static weaver AC++) are available under an open-source license from the AspectC++ project homepage [AC+].

AspectC++ was originally developed by OLAF SPINCZYK as part of his PhD thesis [Spi02] on top of the general-purpose C++ transformation framework PUMA, developed by OLAF SPINCZYK and MATTHIAS URBAN. GEORG BLASCHKE and RAINER SAND have provided additional tool support.[1] Since I joined the AspectC++ team in 2003 I have been involved in questions of language design (such as the support for *generic advice* discussed in Section 4.2), overhead evaluation, and optimization.

On the following pages, I provide additional information about the AspectC++ language and tools. This includes:

- a general language overview in section A.1,
- some advanced application examples in section A.2,
- the AspectC++ language quick reference in section A.3, and
- a comparison of the around-advice code generation by AC++0.9 and AC++1.0PRE1 in section A.4.

[1] namely the ag++ front-end for the GNU g++ compiler and the *AspectC++ Development Tools for Eclipse* (*ACDT*)

Related Publications

The ideas and results presented in this chapter have partly also been published as:

[LBS04] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 55–74. Springer-Verlag, October 2004.

[SL07] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.

[SLU05b] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP extension for C++. *Software Developers Journal*, (5):68–76, May 2005.

A.1. Language Overview

AspectC++ is a superset of the C++ language; every valid C++ program is also a valid AspectC++ program. As in AspectJ, the most relevant terms are *advice* and *pointcut*. Advice types include *code-advice* (before, after, around), *introductions* (known as *inter-type declarations* in AspectJ) and *aspect order* definitions. The building blocks of pointcut expressions are *match expressions* (to some degree comparable to *generalized type names* (*GTNs*) and *signature patterns* in AspectJ), which can be combined using *pointcut functions* and *algebraic operations*. Pointcuts can be *named* and thereby reused in a different context. While named pointcuts can be defined anywhere in the program, only *aspects* can give advice. The aspect

```
aspect TraceService {
  pointcut Methods() = "% Service::%(...)";
  advice call( Methods() ) : before() {
    cout << "Service function invocation" << endl;
}};
```

gives *before-advice* to all calls to functions defined by the pointcut `Methods()`, which is in turn defined as all functions of the class or namespace `Service`, like `void Service::foo()` or `int Service::bar(char*)`. The special `%` and `...` symbols are wildcards, comparable to `*` and `..` in AspectJ.[2] More about the match expression language can be found in section A.1.3.2 and section A.3.5. A list of the pointcut functions and algebraic operations currently supported by AspectC++ can be found in section A.3.6, respectively.

A.1.1. Design Rationale

AspectC++ has been developed with the following goals in mind:

AOP in C++ should be easy. We want practitioners to use AspectC++ in their daily work. The aspect language has to be general-purpose, applicable to existing projects and needs to be integrated well into the C++ language and tool chain.

AspectC++ should be strong where C++ is strong. Even though general-purpose, AspectC++ should specifically be applicable in the C/C++-dominated domains of "very big" and "very small" systems. Hence, it must not lead to a significant overhead at run-time.

These goals of AspectC++, in conjunction with the properties of the C++ language itself, led to some fundamental design decisions:

- *"AspectJ-style" syntax and semantics*, as the AspectJ approach of AOP is used and approved.

[2] AspectJ supports with `+` a third wildcard for *subtype matching*. In AspectC++ this is implemented by the `derived()` pointcut function.

- *Comply with the C++ philosophy*, as this is crucial for acceptance. As different as C++ is from Java, as different AspectC++ has to be from, e.g., AspectJ.
- *Source-to-source weaving*, as it is the only practicable way to integrate AspectC++ with the high number of existing tools and platforms. The AspectC++ weaver AC++ transforms AspectC++ code into C++ code.
- *Avoid using expensive C++ language features* in the generated code, such as exception handling or RTTI, as this would lead to a general run-time overhead.
- *Apply strict pay-as-you-use semantics* with respect to used AOP features. The generated code must not depend on additional run-time libraries, as this would lead to a general run-time overhead.
- *Careful, minimal extension of the C++ grammar*, as the grammar of C++ already is a very fragile building, which should not be shaken more than absolutely necessary.

A.1.2. AspectC++ Grammar Extensions

The AspectC++ extensions to the C++ grammar can be found in section A.3.1. Probably the most important design decision for keeping the set of grammar extensions small and simple was to use *quoted match expressions*. By quoting match expressions, pointcuts can be parsed with the ordinary C++ expression syntax. The real evaluation of the pointcuts itself can be postponed to a separate parser. This clear separation also helps the user to distinguish on the syntax level between ordinary code expressions and match expressions, which are quite different concepts. Additionally, it keeps the match-expression language extensible.

A.1.3. Join-Point Model

AspectC++ uses a unified join-point model to handle all types of advice in the same way. This is different from AspectJ, which distinguishes between *pointcuts* and *advice* on the one hand, and *GTNs* and *introductions* on the other. As shown in the `TraceService` example above, in AspectC++ even match expressions are pointcuts and can be named. While such a coherent language design is a good thing anyway, this is particularly useful in combination with aspect inheritance and (pure) virtual pointcuts. In AspectC++, even the pointcuts used for introductions and base-class introductions can be (pure) virtual and, thus, be defined or overridden in derived aspects. This will be demonstrated in the "Reusable Observer" example in section A.2.

A.1.3.1. Join-Point Types

Regarding the implementation, the unified model requires join points to be *typed* in AspectC++. The basic join-point types are *Name* (**N**) and *Code* (**C**). A name join point

represents a named entity from the C++ program, like a class, a function or a namespace. It typically results from a match expression. A code join point represents a node in the program execution graph, like the call to or execution of a function. Code join points result from applying pointcut functions to name pointcuts. The basic types are additionally separated into more specialized subtypes like *Class* ($\mathbf{N}_C$) or *Function* ($\mathbf{N}_F$), *Execution* ($\mathbf{C}_E$), or *Construction* ($\mathbf{C}_{Cons}$). The aspect weaver uses the type information to ensure that, for example, code-advice (before, after, around) is only given to code join points. section A.3.2 lists all join-point types. The subtypes are, however, mostly irrelevant for the user as most pointcut functions accept the general types as a compound; an $\mathbf{N}_C$ join point, for instance, is implicitly treated as a set of $\mathbf{N}_F$ join points of all member functions.

A.1.3.2. Match Expression Language

C++ has a rather complex type system, consisting of *fundamental types* (`int`, `short`, ...) and *class types* (`class`, `struct`, `union`), *derived types* (pointer, array, reference, function) and optional *cv qualifiers* (`const`, `volatile`). Besides *ordinary functions* and *member functions*, C++ also supports overloading a predefined set of *operator functions*. Classes can define *type-conversion functions* to provide implicit casts to other types. And finally, classes or functions can be parameterizable with *template arguments*.

The AspectC++ match expression language covers all these elements, because in C++ they are an integral part of a type's or function's name or signature. Hence, they should be usable as match criteria for yielding name join points. The match expression language is defined by its own grammar consisting of more than 25 rules [US06, pp. 33ff]. I shall not describe these rules here, instead refer to the match expressions examples in section A.3.5, which should be sufficient to understand the match expressions used in this thesis.

A.1.3.3. Order-Advice

Besides code-advice (before, after, around) and introductions (see section A.3.4), AspectC++ supports with *order-advice* a third type of advice. Order-advice is used to define a partial order of aspect precedence *per pointcut*. This makes it possible to have a different precedence for different join points. For example, the order declarations

```
advice "Service" : order("Locking", "Tracing");
advice "Client" : order("Tracing", !"Tracing");
```

define that in the context of the class or namespace `Service` all advice given by the aspect `Locking` should be applied first, followed by advice from any aspect but `Locking` and `Tracing`. Advice given by `Tracing` should have the lowest precedence. However, for `Client` the order is different. Advice given by `Tracing` should be applied first. As order declarations are themselves pieces of advice, they benefit from the unified join-point model. They can be given to virtual pointcuts and can be declared in the context of any aspect. Hence, it is possible to separate the precedence rules from the aspects they affect.

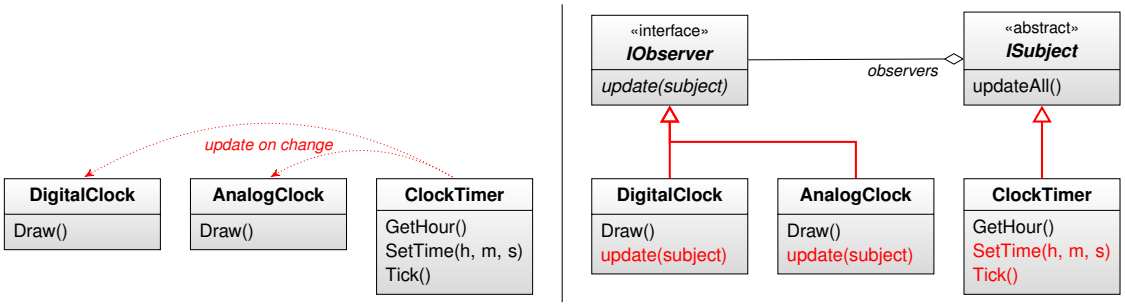


Figure A.1.: Tangled code in the application of the observer protocol (scenario).
The existing stand-alone classes `DigitalClock` and `AnalogClock` should become observers of the existing stand-alone class `ClockTimer`. However, applying the observer pattern requires significant modifications in these classes. To turn `DigitalClock` and `AnalogClock` into observers of `ClockTimer`, they have to derive from an additional base class, an `update()` method has to be added to `AnalogClock` and `DigitalClock`, and each state-changing method in `ClockTimer` has to be modified to trigger the update. The implementation of the observer protocol crosscuts with the implementation of its participants.

The AC++ weaver collects all partial order declarations for a join point and derives a valid total order. In case of a contradiction, a weave-time error is reported.

A.1.3.4. Intentionally Missing Features

AspectC++ intentionally does not implement the `get()` and `set()` pointcut functions, known from AspectJ to give advice for field access. Even if desirable, they are not implementable in a language that supports free pointers. Field access through pointers is quite common in C/C++ and implies a danger of "surprising" side effects for such advice.

A.2. Examples

The following examples for real-world aspects demonstrate some of the more advanced AspectC++ language features.

A.2.1. Observer Pattern with AspectC++

Design patterns, such as the patterns published by the GANG OF FOUR (GOF) [GHJV95], are generally considered as excellently reusable pieces of software design. Their *implementation*, however, typically crosscuts with the implementation of the participating classes, which leads to tangled, less reusable code. Figure A.1 demonstrates this problem by an example scenario for the observer pattern.

By means of AOP most design patterns can be implemented in a reusable, noninvasive way. This was first presented by HANNEMANN and KICZALES with AspectJ and a subset

```
1  aspect ObserverPattern {
2  public:
3    // Interfaces for each role
4    struct ISubject {};
5    struct IObserver { virtual void update (ISubject *) = 0; };
6
7    // To be defined / overridden by the concrete derived aspect
8    pointcut virtual observers() = 0;
9    pointcut virtual subjects() = 0;
10   pointcut virtual subjectChange() = execution( "% ...::%(...)"
11     && !"% ...::%(...) const" ) && within( subjects() );
12
13   // Introduce role interfaces into actual subject/observer classes;
14   advice observers() : slice class : public ObserverPattern::IObserver;
15   advice subjects()  : slice class : public ObserverPattern::ISubject;
16         // Update observers when the subject changes
17   advice subjectChange() : after () {
18     ISubject* sub = tjp->that();
19     updateObservers( sub );
20   }
21   // Data structures and helper functions to manage subjects and observers
22   void updateObservers( ISubject* sub ) {...}
23   void addObserver( ISubject* sub, IObserver* ob ) {...}
24   ...
25 };
```

Listing A.1: A reusable implementation of the observer pattern in AspectC++.
The abstract aspect `ObserverPattern` defines interfaces `IObserver` and `ISubject` to be introduced as base classes into all classes specified by the pure virtual pointcuts `observers()` and `subjects()`. Observers are notified in case of a change by giving after-advice to the virtual pointcut `subjectChange()`, which defaults to all state-changing methods.

of the GoF patterns [HK02]. Listing A.1 implements a reusable observer pattern with AspectC++.

The aspect `ObserverPattern` defines interfaces `ISubject` and `IObserver` (lines 4f), which are inserted via base-class introductions into all classes that take part in the observer protocol (lines 14f). These roles are represented by the pointcuts `subjects()` and `observers()`, which are declared as *pure virtual*. Thus, `ObserverPattern` is an *abstract aspect* and the definition of `subjects()` and `observers()` is delegated to derived concrete aspects.

A third pointcut, `subjectChange()`, describes all methods that potentially change the state of a subject and thereby should lead to a notification of the registered observers (lines 11f). This pointcut is a good example for the expressive power of the AspectC++ join-point description language; it is defined as:

```
execution(               // execution of
  "%...::%(...)"         //   any function in any scope
  &&                     //   intersected with
  !"%...::%(...) const"  //   any nonconst function
) &&                     // intersected with
within(subjects())       // anything within a class from subjects()
```

```
1 #include "ObserverPattern.ah"
2 #include "ClockTimer.h"
3
4 aspect ClockObserver : public ObserverPattern {
5
6   // define the pointcuts
7   pointcut subjects()   = "ClockTimer";
8   pointcut observers()  = "DigitalClock" ||"AnalogClock";
9
10  // insert the concrete update() implementation
11  advice observers() : slice class {
12      void update( ObserverPattern::ISubject* sub ){
13        Draw();
14      }
15 };};
```

Listing A.2: Application example for the reusable observer aspect.
The concrete aspect `ClockObserver` implements the application scenario from Figure A.1 without the necessity to touch the existing classes `AnalogClock`, `DigitalClock`, and `ClockTimer`. The pattern is instantiated by inheriting from the reusable `ObserverPattern` aspect described in Listing A.1 and overwriting the pure virtual pointcuts `subjects()` and `observers()`. Additionally, the instantiation-specific implementation of `IObserver::update()` is introduced into `DigitalClock` and `AnalogClock`.

which evaluates to the *execution* of all *nonconst methods* that are defined *within* a class from `subjects()`. This is a reasonable default. However, `subjectChange()` is declared as *virtual*; it can be overridden in a derived aspect – if, for instance, not all state-changing methods should trigger a notification. Finally, the notification of observers is implemented by giving after-execution-advice to `subjectChange()` (lines 17ff).

The `ClockObserver` aspect in Listing A.2 implements the scenario from Figure A.1 as an example for a concrete aspect derived from `ObserverPattern`. To apply the pattern, the developer only would have to define the inherited, pure virtual pointcuts `subjects()` and `observers()` (lines 7f) and to write the introduction that inserts the implementation of `update()` into the observer classes (lines 11ff).[3]

A.2.2. Caching with AspectC++

The following example demonstrates how the possibility to use generative programming techniques for aspect implementations can lead to a very high level of abstraction. The example is a simple caching policy for function invocations. The implementation idea is straightforward: before executing the function body, a cache is searched for the passed

[3] This implementation of "reusable observer" is even more generic than the AspectJ implementation suggested by HANNEMANN in [HK02], where the *derived* aspects have to perform the base class introductions for the role interfaces. Purpose and name of these interfaces are, however, implementation details of the protocol and should be hidden. Moreover, the derived aspect has to define the `subjectChange()` pointcut in any case. In the AspectC++ solution this is not necessary, as it is possible to take advantage from the C++ notion of nonmodifying (`const`) methods in match expressions and thereby find all potentially state-changing methods automatically.

```
1  class Calc {
2  public:
3
4    Vec Expensive( const Vec& a,
5                   const Vec& b) const;
6
7    int AlsoExpensive( double a,
8                       double b) const;
9
10   int VeryExpensive( int a, int b,
11                      int c) const;
12 };
```

```
13 struct Vec {
14   Vec( double _x = 0, double _y = 0 )
15     : x( _x ), y( _y ) {}
16   Vec(const Vec& src){operator =(src);}
17   Vec& operator =( const Vec& src ){
18     x = src.x; y = src.y; return *this;
19   }
20   bool operator==(const Vec& w) const{
21     return w.x == x && w.y == y;
22   }
23   double x, y;
24 };
```

Listing A.3: Target code for caching (Scenario).
Class `Calc` provides computationally intensive, yet side-effect–free (`const`) functions `Expensive()`, `AlsoExpensive()`, and `VeryExpensive()`. In the function signatures, simple (`int`, `double`) as well as complex (`Vec`) data types are used.

argument values. If an entry is found, the corresponding result is returned immediately, otherwise the function body is executed and the cache is updated.

Caching is a typical example for a crosscutting policy. However, to be generic, an aspect must be able to *compare*, *copy*, and *allocate* arbitrary sequences of function arguments. All this is type-dependent; hence, for C++ the implementing code has to be generated at compile-time.

A.2.2.1. Application Scenario

Consider an application that uses the class `Calc` (Listing A.3). The goal is to improve the overall application execution speed. With the help of a tracing aspect we figured out that the application spends most time in the computationally intensive functions `Calc::Expensive()`, `Calc::AlsoExpensive()`, and `Calc::VeryExpensive()`. We detected that in our application these functions are often called several times in order with exactly the same arguments. Therefore, we want to improve the execution speed by caching.

A.2.2.2. The Generative Caching Aspect

Listing A.4 describes the implementation of a reusable caching aspect for this purpose. The implementation is based on common idioms for template meta-programming. It uses services from *Loki*, a well-known library for template meta-programming and policy-based design [Ale01]: The `Loki::Tuple` template generates (at compile time) a tuple from a list of types, passed as a `Loki::Typelist`.[4] The resulting tuple is a class that contains one data member for each element in the type list. The rough implementation idea is to pass a list of parameter types to the `Cache` template and to store the cache data in a `Loki::Tuple`

[4] consult [Küm] or [Ale01] for a detailed documentation of these classes.

```
1  #include "loki/TypeTraits.h"
2  #include "loki/Typelist.h"
3  #include "loki/HierarchyGenerators.h"
4
5  template< class T > struct Traits {
6    typedef typename Loki::TypeTraits<
7      typename Loki::TypeTraits< T
8        >::ReferredType >::NonConstType
9      BaseType;
10 };
11
12 namespace AC {
13   // Generate Loki::Typelist from args
14   template< class TJP, int J >
15   struct JP2TL {
16     typedef Loki::Typelist< typename
17       Traits< typename TJP::Arg<
18         TJP::ARGS-J >::Type >::BaseType,
19       typename JP2TL< TJP, J-1 >::Result
20     > Result;
21   };
22   template< class TJP >
23   struct JP2TL< TJP, 1 > {
24     typedef Loki::Typelist< typename
25       Traits< typename TJP::Arg<
26         TJP::ARGS-1 >::Type >::BaseType,
27       Loki::NullType
28     > Result;
29   };
30 }
31
32 aspect Caching {
33   template<class TJP> struct Cache
34     : public Loki::Tuple< typename
35     AC::JP2TL< TJP, TJP::ARGS >::Result >
36   {
37     // Comp. TJP args with a Loki tuple
38     template<class C,int I>struct Comp_N{
39       static bool proc(TJP* tjp,
40         const C& cc)
41       { return *tjp->arg< I >() ==
42           Loki::Field< I >( cc ) &&
43           Comp_N<C,I-1>::proc(tjp,cc);
44       }
45     };
46     template<class C>struct Comp_N<C,0>{
47       static bool proc(TJP* tjp,
48         const C& cc)
49       { return *tjp->arg<0>()
50           == Loki::Field<0>(cc);
51       }
52     };
53
54     // Copies TJP args into a Loki tuple
55     template<class C,int I>struct Copy_N{
56       static void proc(TJP* tjp, C& cc) {
57         Loki::Field< I >(cc) =
58           *tjp->arg< I >();
59         Copy_N< C, I-1 >::proc(tjp, cc);
60       }
61     };
62     template<class C>struct Copy_N<C,0>{
63       static void proc(TJP* tjp,C& cc) {
64         Loki::Field< 0 >( cc ) =
65         *tjp->arg< 0 >();
66       }
67     };
68
69     bool  valid;
70     typename TJP::Result res;
71     Cache() : valid( false ) {}
72
73     bool Lookup( TJP* tjp ) {
74       return valid && Comp_N< Cache,
75         TJP::ARGS-1 >::proc(tjp, *this);
76     }
77     void Update( TJP* tjp ) {
78       valid = true;
79       res   = *tjp->result();
80       Copy_N< Cache, TJP::ARGS-1 >
81         ::proc(tjp, *this);
82     }
83   };
84
85   advice execution("% Calc::%(...)")
86   : around() {
87     static Cache< JoinPoint > cc;
88
89     if( cc.Lookup( tjp ) )
90       *tjp->result() = cc.res;
91     else {
92       tjp->proceed();
93       cc.Update( tjp );
94     }
95   }
96 };
```

Listing A.4: Generative caching aspect.

The aspect `Caching` implements a generic, one-elemental cache by giving around-advice to the execution of methods (lines 85–95). The actual parameter values (encoded in `tjp`) are looked up and returned from the the cache if found (lines 89f); otherwise the original method is invoked (`tjp->proceed()`) and the cache is updated. The inner template class `Cache`, instantiated with `JoinPoint`, implements the actual caching strategy. To store the argument values, it derives from the `Loki::Tuple` template, instantiated with the signature of the affected function (lines 33ff), which first has to be converted by the `AC::JP2TL` template meta-program (lines 14–29) into a `Loki::Typelist`. Additional template meta-programs generate the code to compare the affected function's argument values with the values stored in the cache (meta-program `Comp_N`, lines 38–52) and to copy them into the cache (meta-program `Copy_N`, lines 55–67). These meta-programs are instantiated to implement the `Cache` methods `Lookup()` (lines 74f) and `Update()` (lines 80f), respectively.

(a)	VC++ 2003 AC++0.9	tangled cycles	caching aspect cycles	Δ
	cache hit	*39*	*54*	*15*
	cache miss	*36*	*67*	*31*

(b)	VC++ 2003 AC++1.0PRE1	tangled cycles	caching aspect cycles	Δ
	cache hit	*39*	*40*	*1*
	cache miss	*36*	*47*	*11*

Table A.1.: Performance overhead of the generative caching aspect.
Overhead in clock *cycles* in case of a *cache hit* and a *cache miss* when a simple one-elemental caching strategy is applied to the method `Vec Calc::Expensive(const Vec&, const Vec&)` from Listing A.3. The column *tangled* shows the overhead in case of a manual implementation of caching in the body of the method; the column *caching aspect* shows the overhead if the generative caching aspect from Listing A.4 is used instead; Δ denotes the difference to *tangled*. **(a)** Results with AC++0.9. **(b)** Results with AC++1.0PRE1 (new around-advice implementation).
[Measurements were performed on an Intel PIII E ("Coppermine") machine running at 650 MHz under Windows XP SP1. Cycles were measured with `rdtsc` for 1000 iterations and averaged over 25 series ($\sigma < 0.1$). Used VisualC++ optimization flags: /O2 /Ot]

instead of distinct data members; thereby adapt the cache data automatically to the actual parameter list of the affected function. The code for the methods `Cache::Lookup()` and `Cache::Update()` is partly generated at compile time, as the number of arguments (and therefore the number of necessary comparisons and assignments) is unknown until the point of advice instantiation.

A.2.2.3. Performance Evaluation

The aspect `Caching` is indeed a generic and broadly reusable implementation of a caching strategy. It can be applied noninvasively to functions with 1 to *n* arguments; each argument being of any type that is comparable and assignable. Type safety is achieved and code redundancy is completely avoided. The source code complexity, on the other hand, is notably high. The encapsulation as an aspect may also result in a performance overhead, as the aspect weaver has to create special code to provide the infrastructure for the woven-in advice.

Table A.1 compares the performance overhead of the generative caching aspect with a manual "in-place" implementation with AC++0.9 and AC++1.0PRE1 as aspect weavers. For the in-place cache, *cache hit* represents the cost of a successful call to `Lookup()`, *cache miss* represents the cost of an unsuccessful call to `Lookup()`, followed by a call to `Update()`.[5] For the caching aspect, these numbers include the additional overhead introduced by AspectC++. This overhead, explicitly displayed in column Δ, can be understood as the price of applying caching by an aspect. The significantly better numbers with AC++1.0PRE1 underline the benefits of the new around-advice implementation, which is discussed in section A.4.

[5] The effect that the overhead of a cache miss is even *lower* than for a cache hit can be explained by the four necessary (and relatively expensive) floating point comparisons to ensure a cache hit, while a cache miss can ideally be detected after the first comparison. The skipped comparisons outweigh the cost of `Update()`.

The additional overhead of the aspect implementations is partly induced by the fact that AspectC++ needs to create an array of references to all function parameters to provide a unified access to them.

A.2.2.4. Excursus: A Generic Caching Aspect in AspectJ

To understand if and how a similar genericity is reachable in a language that does not support static meta-programming and follows a more run-time–based philosophy, I also analyzed what a generic caching aspect would look like in AspectJ.

The AspectJ implementation (see Listing A.5) uses reflection to store the argument and result values in the cache by dynamically creating instances of the corresponding types and invoking their copy constructors. However, whereas in C++ the copy constructor is part of the canonical class interface (and its absence is a strong indicator for a noncopyable and therefore noncacheable class), in Java for many classes copy constructors are "just not implemented". This limits the applicability of the aspect, and, even worse, it is not possible to detect the missing copy constructor before run-time.

Another strong issue of a reflection-based solution is performance: On a 1GHz Athlon machine the additional cost of a cache-hit are 0.44 μs; a cache miss costs, because of the expensive reflection-based copying, about 26.47 μs. These numbers are orders of magnitude higher than the corresponding 0.001 μs and 0.015 μs (1 respectively 11 cycles on a PIII-700) of the C++ solution.[6] In short: A solution based on run-time reflection instead of compile-time genericity induces a higher overhead, reduced applicability, and may even lead to unexpected errors at run-time.

[6] Java measurements were performed on a 1GHz Athlon 4 ("Thunderbird") running Sun's Java 1.4.2_03 on a Linux 2.6.3 kernel. Because of the different setups, the Java and C++ numbers are not directly comparable. We can however assume that the 700MHz PIII machine used for the C++ measurements does not perform better than the 1GHz Athlon machine used for the Java measurements.

```
1  public class Copy {
2    static public Object copy(Object _src) {
3      Object copy = null;
4      Class tmpClass = _src.getClass();
5      Field[] tmpFields = tmpClass.getFields();
6      boolean isPrimitive = false;
7      for (int j = 0; j < tmpFields.length
8      && !isPrimitive; j++) {
9        isPrimitive |=
10         tmpFields[j].getName() == "TYPE";
11     }
12     // if argument is primitive just copy ref
13     if (isPrimitive) {
14       copy = _src;
15     } else { try {
16         // try to invoke a copy constructor
17         Class[] initTypes = { tmpClass };
18         Constructor tmpConst =
19           tmpClass.getConstructor(initTypes);
20         Object[] initArgs = { _src };
21         copy = tmpConst.newInstance(initArgs);
22       } catch (Exception e) {
23         ...
24     } }
25     return copy;
26 } }
27
28 public class Cache {
29   boolean valid;
30   public Object result;
31   public Object[] arguments;
32
33   public Cache() {
34     valid = false;
35   }
36   public void update(Object[] _args,
37   Object _result) {
38     this.args = _args;
39     this.result = _result;
40     this.valid = true;
41   }
42   public boolean lookup(Object[] _args) {
43     // check if cache contains valid data
44     if (valid) {
45       boolean inCache = true;
46       For (int i = 0; i < this.args.length
47       && inCache; i++) {
48         inCache =
49           this.args[i].equals(_args[i]);
50       }
51       return inCache;
52     }
53     else return false;
54 } }
55
56 public aspect Caching {
57   /* A map containing all caches
58      for all cached functions */
59   Cache cc = new Cache();
60
61   public pointcut cachedFunctions() :
62     within(Calc) && execution(* * (..));
63
64   Object around() : cachedFunctions() {
65     /*get arguments */
66     Object[] args = thisJoinPoint.getArgs();
67     /* lookup arguments in cache */
68     if (cc.lookup(args)) {
69       return cc.result;
70     } else {
71       // get result by executing the original
72       Object result = proceed();
73       // copy result
74       Object resCopy = Copy.copy(result);
75       // copy arguments
76       Object[] argsCopy =
77         new Object[args.length];
78       for (int i = 0; i < argsCopy.length; i++) {
79         argsCopy[i] = Copy.copy(args[i]);
80       }
81       // perform Cache update
82       cc.update(argsCopy, resCopy);
83       return result;
84 } } }
```

Listing A.5: A generic caching aspect in AspectJ.

The AspectJ implementation of the generic caching aspect is structurally similar to the AspectC++ implementation from Listing A.4. The main difference is that the actions to *compare*, *copy*, and *allocate* an unknown sequence of function arguments have to be implemented with run-time mechanisms. The class `Cache` provides the services to `lookup()` and `update()` a cache entry. The comparison of values in `lookup()` can be implemented via the `Object.equals()` service. The copying of values, however, requires the use of Java run-time reflection and is therefore delegated to a separate class. In `update()`, only the references to these copies are stored. The class `Copy` implements the value-copier for Java object instances (given as `Object`) under the assumption that the actual type implements a copy-constructor.

A.3. AspectC++ Language Quick Reference

The following is a (slightly improved) reprint of the AspectC++ language quick reference [SL06], which in turn is a condensed version of the AspectC++ language reference [US06].

A.3.1. Syntax Extensions

The AspectC++ syntax is an extension to the C++ syntax as defined in the ISO/IEC 14882:2003 standard [Ins03].

class-key:
aspect

declaration / member-declaration:
pointcut-declaration
advice-declaration
slice-declaration

pointcut-declaration:
pointcut *declaration*

pointcut-expression:
constant-expression

advice-declaration:
advice *pointcut-expression* **:** *order-declaration*
advice *pointcut-expression* **:** *declaration*

order-declaration:
order (*pointcut-expression-list* **)**

pointcut-expression-list:
pointcut-expression
pointcut-expression, *pointcut-expression-list*

slice-declaration:
slice *declaration*

A.3.2. Join-Point Types

Code

C, C_C, C_E, C_{Cons}, C_{Des} :
any, *Call*, *Execution*, *Construction*, *Destruction*

Name

N, N_N, N_C, N_F, N_T :
any, *Namespace*, *Class*, *Function*, *Type*

A.3.3. Aspects

aspect *A* **{ ... };**
defines the aspect *A*

aspect *A* **:** *public B* **{ ... };**
A inherits from class or aspect *B*

A.3.4. Advice Declarations

advice *pointcut* : **before**(...) {...}
the advice code is executed before the join points in the pointcut

advice *pointcut* : **after**(...) {...}
the advice code is executed after the join points in the pointcut

advice *pointcut* : **around**(...) {...}
the advice code is executed in place of the join points in the pointcut

advice *pointcut* : **slice class** : **public** *Base*;
introduces a new base class *Base* into the target classes matched by *pointcut*

advice *pointcut* : **slice class** : **public** *Base* {...};
introduces a new base class *Base* and new members

advice *pointcut* : **slice** *ASlice* **;**
introduces the slice *ASlice* into the target classes matched by *pointcut*; the slice (a class fragment) has to be defined separately in any class or namespace scope

advice *pointcut* : **order**(*high*, *...low*)**;**
high and *low* are pointcuts, which describe sets of aspects; aspects on the left side of the argument list always have a higher

precedence than aspects on the right hand side at the join points, where the order declaration is applied

A.3.5. Match Expressions

Type Matching

`"int"`
matches the C++ built-in scalar type `int`

`"% *"`
matches any pointer type

Namespace and Class Matching

`"Chain"`
matches the class, struct, or union *Chain*

`"Memory%"`
matches any class, struct, or union whose name starts with "Memory"

Function Matching

`"void reset()"`
matches the function *reset* having no parameters and returning `void`

`"% printf(...)"`
matches the function *printf* having any number of parameters and returning any type

`"% ...::%(...)"`
matches any function, operator function, or type conversion function (in any class or namespace)

`"% ...::Service::%(...) const"`
matches any const member function of the class `Service` defined in any scope

`"% ...::operator %(...)"`
matches any type conversion function

Template Matching

`"std::set<...>"`
matches all template instances of the class *std::set*

`"std::set<int>"`
matches only the template instance *std::set<int>*

`"% ...::%<...>::%(...)"`
matches any member function from any template class in any scope

A.3.6. Predefined Pointcut Functions

Functions

call(*pointcut*) $\mathbf{N}\rightarrow\mathbf{C}_C$
provides all join points where a named entity in the *pointcut* is called

execution(*pointcut*) $\mathbf{N}\rightarrow\mathbf{C}_E$
provides all join points referring to the implementation of a named entity in the *pointcut*

construction(*pointcut*) $\mathbf{N}\rightarrow\mathbf{C}_{Cons}$
all join points where an instance of the given class(es) is constructed

destruction(*pointcut*) $\mathbf{N}\rightarrow\mathbf{C}_{Des}$
all join points where an instance of the given class(es) is destructed

pointcut may contain function names or class names; a class name is equivalent to the names of all functions defined within its scope combined with the || operator (see below)

Control Flow

cflow(*pointcut*) $\mathbf{C}\rightarrow\mathbf{C}$
captures join points occurring in the dynamic execution context of join points in the *pointcut*; the argument pointcut is forbidden to contain context variables or join points with runtime conditions (currently cflow, that, or target)

Types

base(*pointcut*) $\mathbf{N}\rightarrow\mathbf{N}_{C,F}$
returns all base classes resp. redefined functions of classes in the *pointcut*

derived(*pointcut*) $\mathbf{N}\rightarrow\mathbf{N}_{C,F}$
returns all classes in the *pointcut* and all classes derived from them resp. all redefined functions of derived classes

Context

that(*type pattern*) **N→C**
returns all join points where the current C++ `this` pointer refers to an object which is an instance of a type that is compatible to the type described by the *type pattern*

target(*type pattern*) **N→C**
returns all join points where the target object of a call is an instance of a type that is compatible to the type described by the *type pattern*

result(*type pattern*) **N→C**
returns all join points where the result object of a call/execution is an instance of a type described by the *type pattern*

args(*type pattern*, ...) **(N,...)→C**
a list of *type patterns* is used to provide all join points with matching argument signatures

Instead of the *type pattern* it is possible here to pass the name of a **context variable** to which the context information is bound; in this case the type of the variable is used for the type matching

Scope

within(*pointcut*) **N→C**
filters all join points that are within the functions or classes in the *pointcut*

Algebraic Operators

pointcut **&&** *pointcut* **(N,N)→N, (C,C)→C**
intersection of the join points in the *pointcuts*

pointcut **||** *pointcut* **(N,N)→N, (C,C)→C**
union of the join points in the *pointcuts*

! *pointcut* **N→N, C→C**
exclusion of the join points in the *pointcut*

A.3.7. Join-Point API

The join-point API is provided within every advice code body by the built-in object ***tjp*** of class ***JoinPoint***.

Compile-Time Types and Constants

That [type]
object type (object initiating a call)

Target [type]
target object type (target object of a call)

Result [type]
result type of the affected function

Arg<i>::Type, Arg<i>::ReferredType [type]
type of the i^{th} argument of the affected function (with $0 \leq i < ARGS$)

ARGS [const]
number of arguments

JPID [const]
unique numeric identifier for this join point

JPTYPE [const]
numeric identifier describing the type of this join point (*AC::CALL*, *AC::EXECUTION*, *AC::CONSTRUCTION, AC::DESTRUCTION*)

Run-Time Functions and State

static const char ***signature**()
gives a textual description of the join point (function name, class name, ...)

static const int **args**()
returns the number of arguments

That ***that**()
returns a pointer to the object initiating a call or 0 if it is a static method or a global function

Target ***target**()
returns a pointer to the object that is the target of a call or 0 if it is a static method or a global function

Result ***result**()
returns a typed pointer to the result value or 0 if the function has no result value

Arg<i>::ReferredType ***arg**()
returns a typed pointer to the argument value with compile-time index *number*

void ***arg**(*int number*)
returns a pointer to the memory position holding the argument value with index *number*

void **proceed**()
: executes the original code in an around-advice

AC::Action &**action**()
: returns the runtime action object containing the execution environment to execute (*trigger()*) the original code encapsulated by an around-advice

Run-Time Type Information

static AC::Type **type**()

static AC::Type **resulttype**()

static AC::Type **argtype**(*int* i)
: return a C++ ABI V3 conforming string representation of the signature / result type / argument type of the affected function

A.4. Around-Advice Implementation in AC++-0.9 and AC++-1.0PRE1

The evaluation results from the generative caching aspect (Table A.1) and the AspectC++ benchmarks in Section 4.3.2.1 revealed a significant improvement of AC++-1.0PRE1 compared to AC++-0.9 with respect to the cost of around-advice. This effect is caused by the better code generation of AC++-1.0PRE1.

Consider the example program below, in which some around-advice is given for all calls to a function `foo( int, int )`:

```
void foo( int a, int b) {}
int main() {
  foo( 47, 11 );
}

aspect SimpleAround {
  advice call("% foo(...)") : around() {
    // before proceed()
    tjp->proceed();
  // after proceed()
  }
};
```

A.4.1. Code Generation with AC++-0.9

After weaving the example program with AC++-0.9, the generated code looks (slightly edited) as follows:

```
#ifndef __ac_h_
#define __ac_h_
namespace AC {
  typedef const char* Type;
  enum JPType { CALL = 0x0004, EXECUTION = 0x0008,
                CONSTRUCTION = 0x0010, DESTRUCTION = 0x0020 };
  struct Action {
    void **_args;
    void *_result;
    void *_target;
    void *_that;
    void (*_wrapper)(Action &);
    void *_fptr;
    inline void trigger () { _wrapper (*this); }
  };
  template <class Aspect, int Index>
  struct CFlow {
    static int &instance () {
      static int counter = 0;
      return counter;
    }
    CFlow () { instance ()++; }
    ~CFlow () { instance ()--; }
    static bool active () { return instance () > 0; }
  };
}
```

```
#endif // __ac_h_

#ifndef __forward_declarations_for_SimpleAround__
#define __forward_declarations_for_SimpleAround__
class SimpleAround;
namespace AC {
  template <class JoinPoint>
  inline void invoke_SimpleAround_SimpleAround_a0_around (JoinPoint *tjp);
}
#endif

void foo( int a, int b) {}

struct TJP__ZN4mainEv_0 {
  typedef void Result;
  typedef void That;
  typedef void Target;
  static const int JPID = 0;
  static const AC::JPType JPTYPE = (AC::JPType)4;
  enum { ARGS = 2 };
  template <int I, int DUMMY = 0> struct Arg {
    typedef void Type;
    typedef void ReferredType;
  };
  template <int DUMMY> struct Arg<0, DUMMY> {
    typedef int Type;
    typedef int ReferredType;
  };
  template <int DUMMY> struct Arg<1, DUMMY> {
    typedef int Type;
    typedef int ReferredType;
  };

  AC::Action *_action;
  inline void proceed () { _action->trigger (); }
  AC::Action &action() {return *_action;}
};

static void __action_call__ZN4mainEv_0_0 (AC::Action &action) {
  ::foo(*((TJP__ZN4mainEv_0::Arg<0>::ReferredType*) action._args[0]),
        *((TJP__ZN4mainEv_0::Arg<1>::ReferredType*) action._args[1]));
}
inline void __call__ZN4mainEv_0_0 (int arg0, int arg1){
  void *args__ZN4mainEv_0[] = { (void*)&arg0, (void*)&arg1 };
  AC::Action tjp_action__ZN4mainEv_0 = { args__ZN4mainEv_0, 0, 0, 0,
                                         __action_call__ZN4mainEv_0_0, 0 };
  TJP__ZN4mainEv_0 tjp__ZN4mainEv_0 = {&tjp_action__ZN4mainEv_0};

  AC::invoke_SimpleAround_SimpleAround_a0_around<TJP__ZN4mainEv_0> (
    &tjp__ZN4mainEv_0);
}

int main() {
  __call__ZN4mainEv_0_0 ( 47, 11 );
}

class SimpleAround {
public:
  static SimpleAround *aspectof () {
```

```
    static SimpleAround __instance;                                                    86
    return &__instance;                                                                87
  }                                                                                    88
  static SimpleAround *aspectOf () {                                                   89
    return aspectof ();                                                                90
  }                                                                                    91
  template<class JoinPoint> void __a0_around (JoinPoint *tjp) {                        92
    typedef typename JoinPoint::That __JP_That;                                        93
    typedef typename JoinPoint::Target __JP_Target;                                    94
    typedef typename JoinPoint::Result __JP_Result;                                    95
                                                                                       96
    // before proceed()                                                                97
    tjp->proceed();                                                                    98
    // after proceed()                                                                 99
  }                                                                                    100
};                                                                                     101
template <class JoinPoint>                                                             102
inline void AC::invoke_SimpleAround_SimpleAround_a0_around (JoinPoint *tjp) {          103
  ::SimpleAround::aspectof()->__a0_around (tjp);                                       104
}                                                                                      105
```

The original call from `main()` to `foo()` has been replaced by a call to the wrapper function `__call__ZN4mainEv_0_0()` (line 80). In this function (lines 69–77) an `Action` object is allocated on the stack and initialized. This object encapsulates the complete context of the original call, including a pointer to a wrapper function `__action_call__ZN4mainEv_0_0()` that finally performs the call to `foo()` (line 65). The `Action` object is passed as part of the join point context to the `proceed()` method (line 61), which uses it to invoke the original call via the pointer to the wrapper function.

A.4.2. Code Generation with AC++-1.0PRE1

Compare this to the (slightly edited) output generated by AC++-1.0PRE1, in which the implementation of `proceed()` (lines 75ff) is no longer based on the presence of an `Action` object, but calls `foo()` directly:

```
#ifndef __ac_h_                                                                        1
#define __ac_h_                                                                        2
namespace AC {                                                                         3
  typedef const char* Type;                                                            4
  enum JPType { CALL = 0x0004, EXECUTION = 0x0008,                                     5
                CONSTRUCTION = 0x0010, DESTRUCTION = 0x0020 };                         6
  struct Action {                                                                      7
    void **_args; void *_result; void *_target; void *_that; void *_fptr;              8
    void (*_wrapper)(Action &);                                                        9
    inline void trigger () { _wrapper (*this); }                                       10
  };                                                                                   11
  struct AnyResultBuffer {};                                                           12
  template <typename T> struct ResultBuffer : public AnyResultBuffer {                 13
    char _data[sizeof (T)];                                                            14
    ~ResultBuffer () { ((T*)_data)->T::~T(); }                                         15
    operator T& () const { return *(T*)_data; }                                        16
  };                                                                                   17
  template <class Aspect, int Index>                                                   18
  struct CFlow {                                                                       19
```

```
    static int &instance () {
      static int counter = 0;
      return counter;
    }
    CFlow () { instance ()++; }
    ~CFlow () { instance ()--; }
    static bool active () { return instance () > 0; }
  };
}
inline void * operator new (unsigned int, AC::AnyResultBuffer *p) { return p; }
inline void operator delete (void *, AC::AnyResultBuffer *) { } // for VC++
#endif // __ac_h_

#ifndef __ac_fwd_SimpleAround__
#define __ac_fwd_SimpleAround__
class SimpleAround;
namespace AC {
  template <class JoinPoint>
  inline void invoke_SimpleAround_SimpleAround_a0_around (JoinPoint *tjp);
}
#endif

void foo( int a, int b) {}

struct TJP__ZN4mainEv_0_0 {
  typedef void Result;
  typedef void That;
  typedef void Target;
  static const int JPID = 0;
  static const AC::JPType JPTYPE = (AC::JPType)4;
  struct Res {
    typedef void Type;
    typedef void ReferredType;
  };
  enum { ARGS = 2 };
  template <int I, int DUMMY = 0> struct Arg {
    typedef void Type;
    typedef void ReferredType;
  };
  template <int DUMMY> struct Arg<0, DUMMY> {
    typedef int Type;
    typedef int ReferredType;
  };
  template <int DUMMY> struct Arg<1, DUMMY> {
    typedef int Type;
    typedef int ReferredType;
  };

  void **_args;

  inline void *arg (int n) {return _args[n];}
  template <int I> typename Arg<I>::ReferredType *arg () {
    return (typename Arg<I>::ReferredType*)arg (I);
  }

  void proceed () {
    ::foo(*((TJP__ZN4mainEv_0_0::Arg<0>::ReferredType*)_args[0]),
          *((TJP__ZN4mainEv_0_0::Arg<1>::ReferredType*)_args[1]));
  }
```

```
};                                                                                    79
                                                                                      80
inline void __call__ZN4mainEv_0_0 (int arg0, int arg1){                               81
  void *args__ZN4mainEv_0[] = { (void*)&arg0, (void*)&arg1 };                         82
  TJP__ZN4mainEv_0_0 tjp;                                                             83
  tjp._args = args__ZN4mainEv_0;                                                      84
  AC::invoke_SimpleAround_SimpleAround_a0_around<TJP__ZN4mainEv_0_0> (&tjp);          85
 }                                                                                    86
                                                                                      87
int main() {                                                                          88
  __call__ZN4mainEv_0_0 ( 47, 11 );                                                   89
}                                                                                     90
                                                                                      91
class SimpleAround {                                                                  92
public:                                                                               93
  static SimpleAround *aspectof () {                                                  94
    static SimpleAround __instance;                                                   95
    return &__instance;                                                               96
  }                                                                                   97
  static SimpleAround *aspectOf () {                                                  98
    return aspectof ();                                                               99
  }                                                                                   100
  template<class JoinPoint> void __a0_around (JoinPoint *tjp) {                       101
    typedef typename JoinPoint::That __JP_That;                                       102
    typedef typename JoinPoint::Target __JP_Target;                                   103
    typedef typename JoinPoint::Result __JP_Result;                                   104
                                                                                      105
    // before proceed()                                                               106
    tjp->proceed();                                                                   107
    // after proceed()                                                                108
  }                                                                                   109
};                                                                                    110
namespace AC {                                                                        111
  template <class JoinPoint>                                                          112
  inline void invoke_SimpleAround_SimpleAround_a0_around (JoinPoint *tjp) {           113
    ::SimpleAround::aspectof()->__a0_around (tjp);                                    114
  }                                                                                   115
}                                                                                     116
```

Hence, no `Action` object has to be allocated and initialized in the call-advice wrapper `__call__ZN4mainEv_0_0()` (lines 81–86) and the invocation of the original call to `foo()` does no longer happen indirectly via a function pointer. This code is much better optimizable by current C++ back-end compilers.

B

Appendix: Case Study “WeatherMon”

The goal of the “WeatherMon” study presented in the following was to evaluate AOP in comparison to OOP for the development of software product lines for small, resource-thrifty “deeply embedded” systems. For this purpose, a real-world example of a small embedded system was designed and implemented from scratch as a configurable software product line. The real-world example is an *embedded weather-station product line* that is based on an AVR ATmega 8-bit microcontroller and can be equipped with various sensors and actuators to measure and process weather data. Figure B.2 shows a picture of the actual hardware.

Besides the AOP-based design and implementation, I also implemented an OOP-based version of the same product line. The aim of this version is to provide a qualitative measure for the analysis of the idioms for configurability used in the AOP-based version. A third version was implemented with C and the C preprocessor; it provides the quantitative measure for the cost of configurability by AOP respective OOP.

Related Publications

The ideas and results presented in this chapter have partly also been published as:

[LSSP06] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Lean and efficient system software product lines: Where aspects beat objects. In Awais Rashid and Mehmet Aksit, editors, *Transactions on AOSD II*, number 4242 in Lecture Notes in Computer Science, pages 227–255. Springer-Verlag, 2006.

[LS06] Daniel Lohmann and Olaf Spinczyk. Developing embedded software product lines with AspectC++. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 740–742, Portland, Oregon, USA, 2006. ACM Press.

B.1. WeatherMon Overview

A weather station basically consists of one or more *sensors* to gain environmental information and one or more *actuators* to process the gathered weather data. Figure B.1 shows the the weather-station feature model. Note that the lists of available sensors and actuators are expected to grow in future versions, for instance, by additional sensors for wind direction and humidity. Not visible in the feature model, but nevertheless part of the product-line definition, is that we have to distinguish between two kinds of actuators:

Generic actuators are able to process or aggregate information of any set of sensors. Display and XMLProto are examples for generic actuators. The XML representation of weather information, for instance, can easily and automatically be extended for additional sensors.

Nongeneric actuators process or aggregate sensor data of some specific sensors only. SNGProto is an example for a specific actuator. It implements a legacy binary data representation for compatibility with existing PC applications. This format encodes information of certain sensors only.

A weather-station variant is configured by selecting features from a representation of the feature model in the PURE::VARIANTS configuration tool [Beu03b]. The configuration and variant generation process with PURE::VARIANTS is comparable to the process with ECOSCONFIG, which I described in Section 3.1.1.1. However, a nice and unique feature

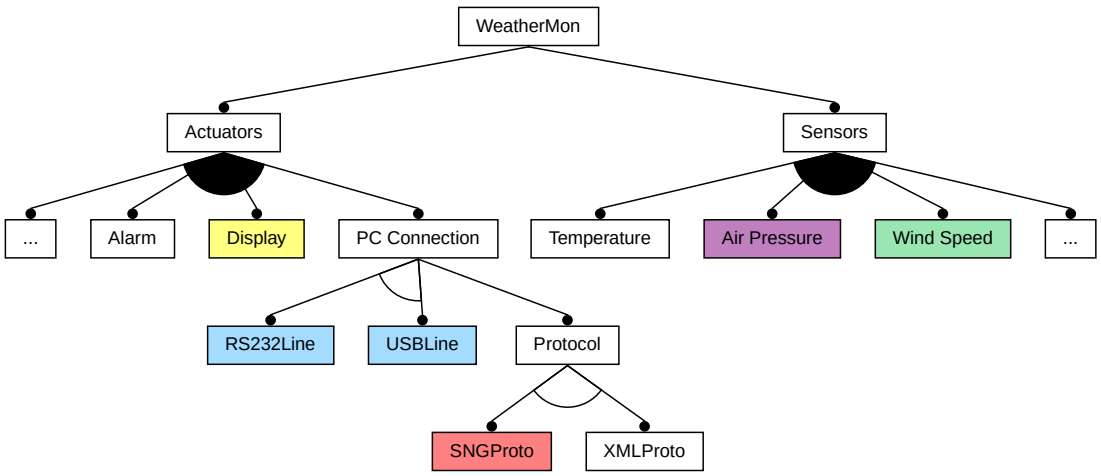


Figure B.1.: Feature diagram of the embedded weather-station product line.
The concept WeatherMon consists of at least one sensor and one actuator. Sensors gain information about the environment, such as Temperature, Air Pressure, or Wind Speed. Actuators process sensor data: Weather information can be printed on an (LCD) Display, monitored to raise an Alarm if values exceed some threshold, and passed to a PC over a PCConnection, which can be either an RS232Line or a USBLine, using either an XML-based data representation (XMLProto) or a proprietary binary format (SNGProto). (The coloring is for the purpose of feature traceability with Figure B.3 and Figure B.4)

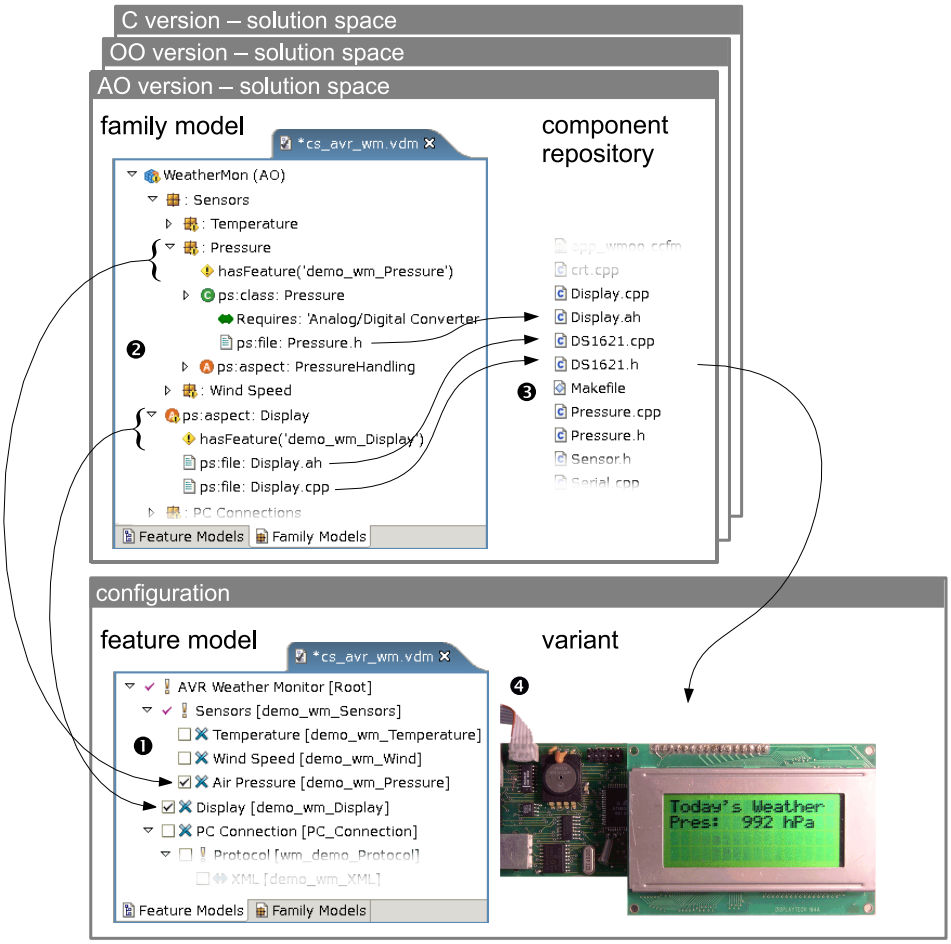


Figure B.2.: WeatherMon configuration process with PURE::VARIANTS.
The user selects all desired features from a representation of the *feature model* of the configuration editor (❶). Each solution-space version consists of a *family model* and a *component repository*. The family model maps features to *logical implementation components* (❷), which in turn are mapped to *physical implementation files* (❸). The thereby determined set of implementation files for a concrete configuration is copied into the target directory, and finally compiled and linked into the actual weather-station *variant* (❹).

of PURE::VARIANTS is that for the same *problem space* (defined by the feature model) *alternative versions* of an implementing *solution space* can be specified. This was used to manage the different versions of the WeatherMon product line (C-based, AOP-based, and OO-based implementation), as depicted in Figure B.2.

The AOP-based and OOP-based versions of the weather-station product line provide *configurability by separation of concerns*: Each feature is implemented as a distinct set of implementation artifacts; technically, the configuration decisions are enforced by including or omitting implementation artifacts from the output of PURE::VARIANTS.

The C-based implementation, in contrast, was solely optimized for low resource requirements. Configuration is enforced by means of conditional compilation; basically the actual code looks like the code we found in eCos (Listing 3.1). As the only purpose of the C-based implementation is to provide the lower bounds of the resource consumption that can be reached in the different product-line configurations, I shall not discuss it in further detail.

B.2. Designing for Configurability with AOP and OOP

In the following I present a very brief description of the OOP-based and AOP-based implementations (which I will abbreviate as *AO version* respective *OO version* from now on). The details about both implementations are discussed together (to set them in contrast to each other) in section B.3.

B.2.1. Requirements

Besides the functional features that have to be implemented, the additional requirements on the implementations can be summarized as *resource thriftiness* and *separation of concerns*. This means in particular:

Granularity. Components should be fine-grained. Each implementation element should be either mandatory (such as the application main loop) or dedicated to a single feature only.

Economy. The use of expensive language features should be avoided as far as possible. For instance, a method should only be declared as `virtual` if polymorphic behavior of this particular method is required.

Pluggability. Changing the set of selected sensors or actuators should not require modifications in any other part of the implementation. This basically means that sensor and actuator implementations should be able to "integrate themselves" into the system.

Extensibility. The same should hold for new sensor or actuator types, which may be available in a future version of the product line.

B.2.2. The OO Version

Figure B.3 shows the class model of the OO version. Central elements are the `Weather` and `Sink` classes. `Weather` aggregates all sensors, which are registered at run time by calling `Weather::registerSensor()`. The `Sink` class aggregates all actuators, respectively. Internally both, sensors and actuators, are managed by chaining them into light-weight single-linked lists (`ChainBase`).

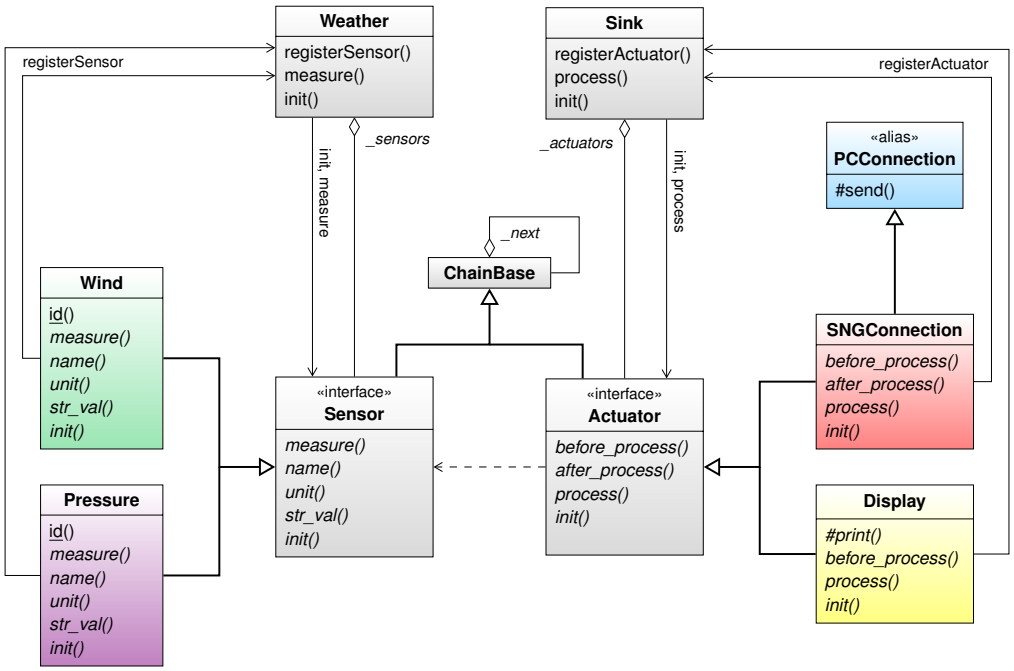


Figure B.3.: Static structure of the OO version of the WeatherMon product line.
The class diagram shows an excerpt with two sensors and two actuators. Virtual functions are depicted in *italics*, static functions are underlined.

Principle of Operation

1. Weather information is acquired by calling `Weather::measure()`, which in turn invokes the `Sensor::measure()` method on each registered sensor to update the sensor data.

2. Weather information is processed by the `Sink::process()` method. `Sink::process()` calls `Actuator::before_process()` for each registered actuator to initialize the processing.

3. Sensor information is passed to the actuators by calling `Actuator::process()` for each registered sensor. Actuators retrieve the actual sensor name, unit, and measured data (as character strings) by calling the respective `Sensor` methods.

4. At last, data processing is finalized by `Sink::process()` invoking `Actuator::after_process()` on each actuator.

5. The whole process is repeated by the application main loop every second.

Further details of the design will be discussed in section B.3.

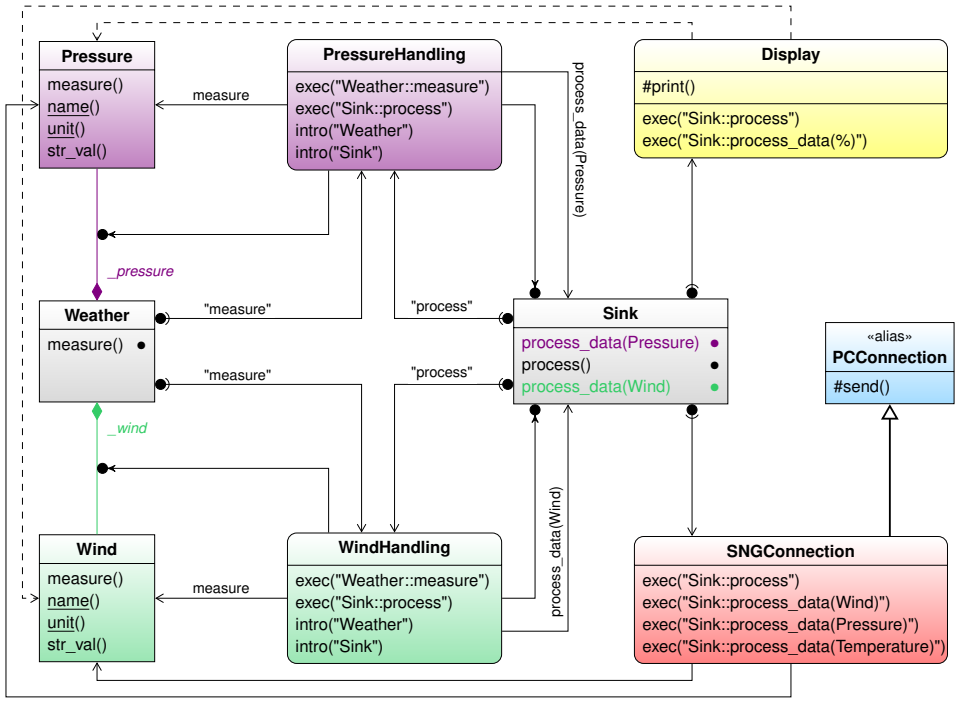


Figure B.4.: Static structure of the AOP version of the WeatherMon product line.
The class and aspect diagram shows an excerpt with two sensors and two actuators. Introduced elements in classes `Sink` and `Weather` are depicted in the color of the introducing aspect.

B.2.3. The AO Version

The class and aspect model of the AO version is shown in Figure B.4. Central elements are, again, the classes `Weather` and `Sink`. Each sensor class is accompanied by a *handling aspect*, which performs the actual integration into `Weather` and `Sink`. A handling aspect performs two introductions: It aggregates the sensor as an instance variable into class `Weather` and a sensor-specific (empty) `process_data()` method into class `Sink`. Additionally, it gives execution-advice to `Weather::measure()` and `Sink::process()`.

Actuators are implemented as aspects, which give execution-advice to `Sink::process()` (for initialization and finalization) and the sensor-introduced `Sink::process_data()` methods (for the actual data processing).

Principle of Operation

1. Weather information is acquired by calling `Weather::measure()`, which is advised by the handling aspects of each sensor to call the sensor's `measure()` method.
2. Weather information is processed by the `Sink::process()` method. `Sink::process()` is before-advised by any actuator that needs to initialize before processing.
3. `Sink::process()`is advised for each sensor to call the introduced sensor-specific

`process_data()` method.

4. The sensor-specific `process_data()` method is advised by each actuator that processes data of this sensor.
5. At last, data processing is finalized by `Sink::process()` being after-advised for any actuator that needs to finalize its processing.
6. The whole process is repeated by the application main loop every second.

In the next section, I discuss further details of the design and also compare it to the OOP solution.

B.3. AOP and OOP Idioms for Configurability

To achieve configurability, both versions use approach-specific idioms and patterns in the design and implementation. In the following I analyze the AOP idioms that have to be used to reach separation of concerns – and thereby, configurability – by comparing them to the OOP idioms used for the same purpose.

To reach separation of concerns, three major issues had to be solved in the design.

B.3.1. Issue 1: Working with Configuration-Dependent Sensor and Actuator Sets

Both implementations have in common that the `Weather` and `Sink` classes are used to abstract from the configured sets of sensors and actuators. These two abstractions are unavoidable, because the `main()` function, which runs the endless measurement and processing loop, should be configuration-independent and, thus, robust with respect to extensions. However, the implementation of the $1:n$ relationship of `Weather` or `Sink` and the concrete sensors or actuators is very different:

Interface dependency between Weather, Sink and sensors, actuators

The `Weather` and `Sink` classes need to be able to invoke the measuring and processing of data independent of the actual sensor or actuator *types*. Otherwise, pluggability and extensibility would be broken:

OO version. In the OO version, this is solved by *common interfaces* and *late binding*. Sensors have to inherit from the abstract class `Sensor`, actuators from the abstract class `Actuator`, respectively. `Weather` or `Sink` invoke sensors or actuators via these interfaces, thus depend on them. `Sensor::measure()`, `Actuator::before_process()`, `Actuator::after_process()`, and `Actuator::process()` have to be declared as virtual functions:

```
struct Sensor : public ChainBase {
  virtual void measure() = 0;
  ...
};
...
class Weather {
public:
  ...
  void measure () {
    for( Sensor* s = ... )
      s->measure();       // virtual function call!
  }
};
```

AO version. In the AO version, the interface dependency is exactly opposite; we have an inverted dependency from sensors and actuators to `Weather` and `Sink`, which do not have to know about sensors and actuators. Instead, sensors and actuators integrate themselves by *advice-based binding*:

```
class Weather {
public:
  void measure () {} // empty implementation
};
...
aspect PressureHandling {
public:
  advice "Weather" : slice struct{ Pressure _pressure; };
  advice execution ("void Weather::measure()") : before () {
    // nonvirtual inlineable function call
    theWeather._pressure.measure ();
  }
  ...
};
```

Potential cost drivers. In the OO version, four methods in two classes have to be declared and called as virtual functions. In the AO version we do not expect any extra cost as the sensor or actuator code can be inlined.

Quantifying over sets of sensors and actuators

`Weather` and `Sink` need to be independent of the actual *number* of sensors and actuators configured into the system. There is a $1:n$ relationship between `Weather` or `Sink` and sensors or actuators:

OO version. In the OO version, this is solved by a simple *publisher and subscriber* mechanism. Sensors are chained into a linked list of "publishers", which is frequently iterated by `Weather::measure()` to update the data of each sensor:

```
class Weather {
  static  Sensor* _sensors;
public:
  Sensor* firstSensor() const {
    return _sensors;
  }
  void measure () {
    for( Sensor* s = firstSensor(); s != 0;
         s = static_cast< Sensor* >( s->getNext() ) )
      s->measure();
  }
  ...
};
```

Actuators are similarly chained into a list of “subscribers” which are invoked when new sensor data is available. The `Sink` class acts as a “mediator” between sensors and actuators. For the sake of memory efficiency, actuators do not subscribe for single sensors but are implicitly subscribed for the complete list and ignore unwanted sensor data at run time.

AO version. In the AO version, the interface dependency is again inverted. `Weather` and `Sink` do not have to know about actuators and sensors. The $1:n$ relationships are realized by more than one aspect giving advice for the same `Weather::measure()` and `Sink::process()` execution join points. Thereby, invocations of actuators and sensors are implicitly chained at compile time.

Potential cost drivers. In the OO version, `Weather` and `Sink` as well as sensors and actuators need to carry an extra pointer for the chaining. Some additional code is required to iterate over the chains of sensors and actuators.[1]

Registration of sensors and actuators

This problem is closely related to the previous one. Sensors and actuators need to be able to register themselves for the publisher or subscriber chain:

OO version. In the OO version, registration is done at run time by calling `Weather::registerSensor()` or `Sink::registerActuator()`. Self-registration requires some C++ trickery: Each sensor or actuator is instantiated as a global object. The registration is performed by the constructor, which is “automatically” triggered during system startup.

AO version. In the AO version, no extra efforts are required for self-registration. The chaining of actuators and sensors is implicitly performed at compile time by advice-code weaving. Basically, it is the presence of some aspect in the source tree that triggers the registration.

[1] The `firstSensor()` and `getNext()` operations are, however, inlined as they just perform a pointer look-up.

Potential cost drivers. In the OO version, the use of global object instances causes some overhead, as the compiler has to generate extra initialization code and has to put a reference to this code in a special linker section. A system startup function has to be provided that iterates over this section and invokes the constructors of all global instances.

B.3.2. Issue 2: Implementation of Generic Actuators

Generic actuators process sensor data from any sensor type. For example, the Display actuator should print the measured values of any configured sensor, regardless of the current configuration or future extensions with new sensor types. However, this leads to an interface dependency between actuators and sensors, because (at least) the value has to be obtained in a generic way:

OO version. In the OO version, this is again solved by *interfaces* and *late binding*. To enable actuators to retrieve sensor information from any sensor, the `Sensor` interface has to be extended by three additional virtual functions: `Sensor::name()`, `Sensor::unit()`, and `Sensor::str_val()`:

```
class Display : public Actuator {
public:
  UInt8 _line;

  // print a line on the display, increment _line
  void print(const char *name, const char *val_str, const char *unit );
  ...
  virtual void before_process() {
    _line = 1;
  }

  // called by Sink::process for each sensor
  virtual void process( Sensor* s ) {
    char val[ 5 ];
    s->str_val( val );
    print( s->name(), val, s->unit() );
  }
};
```

AO version. In the AO version, this is solved with *explicit join points*, and *quantification* of *generic advice* (Section 4.2). For each sensor `MySensor`, the corresponding `MySensorHandling` aspect introduces an empty sensor-specific `process_data(const MySensor&)` method into the class `Sink` and gives advice to `Sink::process()` to invoke the introduced method:

```
aspect PressureHandling {
public:
```

```
  advice "Weather" : slice struct{ Pressure _pressure; };
  ...
  // introduce an empty process_data function for the pressure
  advice "Sink" :
    slice struct{ void process_data( const Pressure & ) {} };
  // call Sink::process_data for the pressure
  advice execution("void Sink::process()") : after () {
    theSink.process_data( theWeather._pressure );
  }
};
```

The introduced `process_data(const MySensor&)` method represents an *explicit join point* provided by the particular sensor. To bind to these join points, an actuator gives generic advice that quantifies over all (overloaded) `Sink::process_data()` functions and uses the AspectC++ join-point API to retrieve a typed reference to the sensor instance in the advice body:

```
aspect Display {
  ...
  // display each element of the weather data
  advice execution("void Sink::process_data(%)") : before () {
    typedef JoinPoint::template Arg<0>::ReferredType Data;
    char val[5];
    tjp->arg<0>()->str_val( val );
    print( Data::name(), val, Data::unit() );
  }
};
```

AspectC++ instantiates advice bodies per join point. Therefore, the calls to the actual sensor's `str_val()`, `name()`, and `unit()` methods can be bound at compile-time. As an additional optimization, `name()` and `unit()` are implemented as static (class) functions.

Potential cost drivers. In the OO version, three additional virtual functions are required in the sensor classes, as well as additional virtual function calls in the actuator classes. In the AO version, the join-point API has to be used for a uniform and type-safe access to the sensor instance, which induces some overhead (Section 4.3.2.3). Furthermore, the advice body of a generic actuator is instantiated once per sensor, which may lead to code bloating effects (Section 4.4.2).

B.3.3. Issue 3: Implementation of Nongeneric Actuators

Nongeneric actuators process data of some sensors only. The legacy SNG protocol, for instance, encodes weather data in a record of wind speed, temperature, and air pressure. It exposes the actual data using sensor-specific interfaces. The record may be sparse, meaning that a specific sensor feature, such as Temperature, may or may not be present in the actual system configuration.

OO version. In the OO version, a nongeneric actuator filters the sensors it is interested in by *run-time type checks* in the `process()` method. Each passed sensor is tested against the handled sensor types. If the run-time type matches a handled sensor, a downcast is performed to get access to the sensor-specific interface:

```
class Pressure : public Sensor {
  static const char *_id;
public:
  ...
  static const char * id () { return _id; }
  const char *name () const { return _id; }
};
...
class SNGConnection : public Actuator, protected PCConnection {
  UInt8 _p, _w, _t1, _t2;     // weather record
public:
  virtual void before_process() { ... /* init record */ };
  virtual void after_process() { ... /* transmit record */ };
  ...
  // collect wind, pressure, temperature data
  virtual void process( Sensor* s ) {
  const char *name = s->name();
    if (name == Wind::id()) // pointer comparison
      _w = ((Wind*)s)->_w;
    else if (name == Pressure::id())
      _p = ((Pressure*)s)->_p - 850;
    else if (name == Temperature::id()) {
      _t1 = (UInt8)((Temperature*)s)->_t1;
      _t2 = ((Temperature*)s)->_t2;
  } }
};
```

The idiom commonly used for run-time type checks in C++ is the `dynamic_cast` operator, which is part of the C++ *run-time type interface* (*RTTI)*. RTTI is quite expensive, as it requires additional run-time support and leads to some extra overhead in *every* class that contains virtual functions. To avoid this overhead, the implementation uses a "home-grown" dynamic type-check mechanism: The test is performed by comparing the string address returned by the (late-bound) `Sensor::name()` method with the address of the name string stored in the concrete class, which is also returned by the static `Sensor::id()` method.[2] The expensive C++ RTTI mechanism has been disabled.

Normally, dynamic type checks are considered harmful because of a lack of extensibility and the accumulated cost of type checks, which sometimes outweigh the cost

[2] This basically reduces the overhead of a run-time type test to a virtual function call and a pointer comparison. As the storage for the name string has to be provided anyway, this mechanism also induces no extra overhead in the sensor classes.

of a single virtual function call. However, in our case a *nongeneric* actuator shall be implemented. Therefore, extensibility is not an issue and the overhead of our type check implementation is acceptable. At the same time, alternative designs such as *visitor* [GHJV95] fail, because a visitor interface would have to list a `visitSensor()` method for every sensor type `Sensor`. However, the set of sensors is configurable and should nowhere be hard-wired.

AO version. In the AO version, this is again based on the *explicit join points* provided by sensors. The only difference to a generic actuator is that a nongeneric actuator gives advice for specific `Sink::process_data()` overloads only instead of quantifying over all potential sensors with generic advice:

```
aspect SNGConnection : protected PCConnection {
  UInt8 _p, _w, _t1, _t2;     // weather record
  ...
  // let this aspect take a higher precedence than <Sensor>Handling
  advice process () : order ("SNGConnection", "%Handling");
  advice execution("void Sink::process(const Weather&)")
      : before () { ... /* init record */ }
  advice execution("void Sink::process(const Weather&)")
      : after () { ... /* transmit record */ }

 // collect wind, pressure, temperature data by giving specific advice
  advice execution("void Sink::process_data(...)") && args (wind)
      : before (const Wind &wind) {
    _w = wind._w;
  }
  advice execution("void Sink::process_data(...)") && args (pressure)
      : before (const Pressure &pressure) {
    _p = pressure._p - 850;
  }
  advice execution("void Sink::process_data(...)") && args (temp)
      : before (const Temperature &temp) {
    _t1 = (UInt8)temp._t1;
    _t2 = temp._t2;
  }
};
```

Potential cost drivers. Run-time type checks in the OO version induce nevertheless some overhead. In the AO version, some overhead is induced by the `args()` pointcut function (Section 4.3.2.4), which is used here to get the actual sensor instance.

B.3.4. Design Summary

The OO version as well as the AO version of the embedded weather-station product achieve a good separation of concerns and, thereby, configurability. In particular, the requirements described in section B.2.1 are met by both versions:

Granularity is achieved by the OO version as well as the AO version. Each implementation component is either mandatory (such as the `Weather` and `Sink` classes), or dedicated to a single feature only.

Economy is achieved as far as possible. In the OO version, only methods that have to be available via a generic interface are declared as virtual. RTTI is not used, as the required run-time type checks can be implemented with less overhead. In the AO version, join-point–specific context information is used only sparingly.

Pluggability is achieved as well. In both versions, no component has to be adapted if the set of selected sensors/actuators is changed. Sensors and actuators basically integrate themselves if their implementation component is present in the configured source tree. The OO version uses global instance construction for this purpose. In the AO version, the integration is performed by advice.

Extensibility is also achieved. In the OO version, new sensor or actuator types just need to implement the common `Sensor` or `Actuator` interface. In the AO version, new sensor or actuator types just need to provide some aspect that performs the integration into `Weather` or `Sink`.

Overall, AOP and OOP turned out to be equipollent with respect to a "design for configurability" in the "WeatherMon" study. We identified, however, more potential cost drivers in the OO design. Especially virtual functions were unavoidable in many places to realize loose coupling and genericity of components. In the next section, I analyze how this affects scalability and memory demands of the product line.

B.4. The Cost of Configurability in Deeply Embedded Systems

In the following I evaluate the scalability of the embedded weather-station product line and analyze the particular cost of separation of concerns with AOP or OOP for such small and deeply embedded systems.

B.4.1. Setup

Six different configurations of the weather station were generated as an AO, OO, and C version. For each variant and version I measured:

- The static memory demands, which are determined by the amount of generated machine code (*text*), static initialized data (*data*), and static noninitialized data (*bss*).
- The dynamic memory demands, which are determined by the maximum stack space used by the running application (*stack*).[3]

[3] The weather station software uses no heap, which otherwise would also contribute to dynamic memory demands.

- The run time of a complete measure-and-process–cycle.

On the actual hardware, *text* occupies flash memory space. *Data* occupies flash memory *and* RAM space, as it is writable at run time and therefore has to be copied from flash into RAM during system startup. *Bss* and *stack* occupy RAM space only.

Static memory demands (*text*, *data*, *bss*) could easily be retrieved directly from the linker map file. Dynamic memory demands (*stack*) and run time had to be measured in the running targets:

Measuring Stack Utilization

The common technique for run-time stack monitoring is to initialize the entire stack space with some specific *magic pattern* during system startup. The maximum amount of stack used can then be measured at any time by searching (from the bottom of the stack) for the first byte where the pattern has been overwritten. This technique was used to implement stack measurement as an additional *sensor type*. Understanding stack measurement as just another sensor had some nice advantages: Because of the achieved pluggability (in the AO and OO versions) it was very easy to apply stack measurement to any WeatherMon configuration. Of course, some extra care had to be taken to ensure that the maximum stack utilization is not caused by the stack measurement sensor itself. For this reason, the stack measurement implementation uses only global variables, which do not occupy stack space. By analyzing the generated machine code I furthermore made sure that the stack utilization of the stack sensor methods is minimal among all sensors. As all sensor methods are invoked from the same call depth level and at least one “real” sensor besides stack measurement is used in a weather station configuration, it can thereby be guaranteed that stack measurement itself does not tamper with the maximum stack utilization. In the actual targets, the thereby acquired maximum stack utilization remained stable after a short startup time and could be read from one of the attached generic actuators.

Measuring Runtime

Runtime measuring was not implemented as another sensor type, as it would have been too difficult to distinguish the run time taken by the sensor processing itself from the run time of the target to measure. Instead, I used a less invasive approach that could be implemented with just two additional assembler statements: In the application main loop, a digital I/O port of the AVR microcontroller is set to *high* before the call to `Weather::measure()` and reset to *low* after the return of `Sink::process()`. The result is a rectangular signal on this port, which was recorded and analyzed with a storage oscilloscope.[4] A *high* period in the signal represents the run time taken by a complete `Weather::measure()` plus `Sink::process()` cycle. After a short startup time, period and phase of the signal remained stable and the length of the *high* phase could be measured.

[4] Tektronix TDS 2012, 100MHz resolution

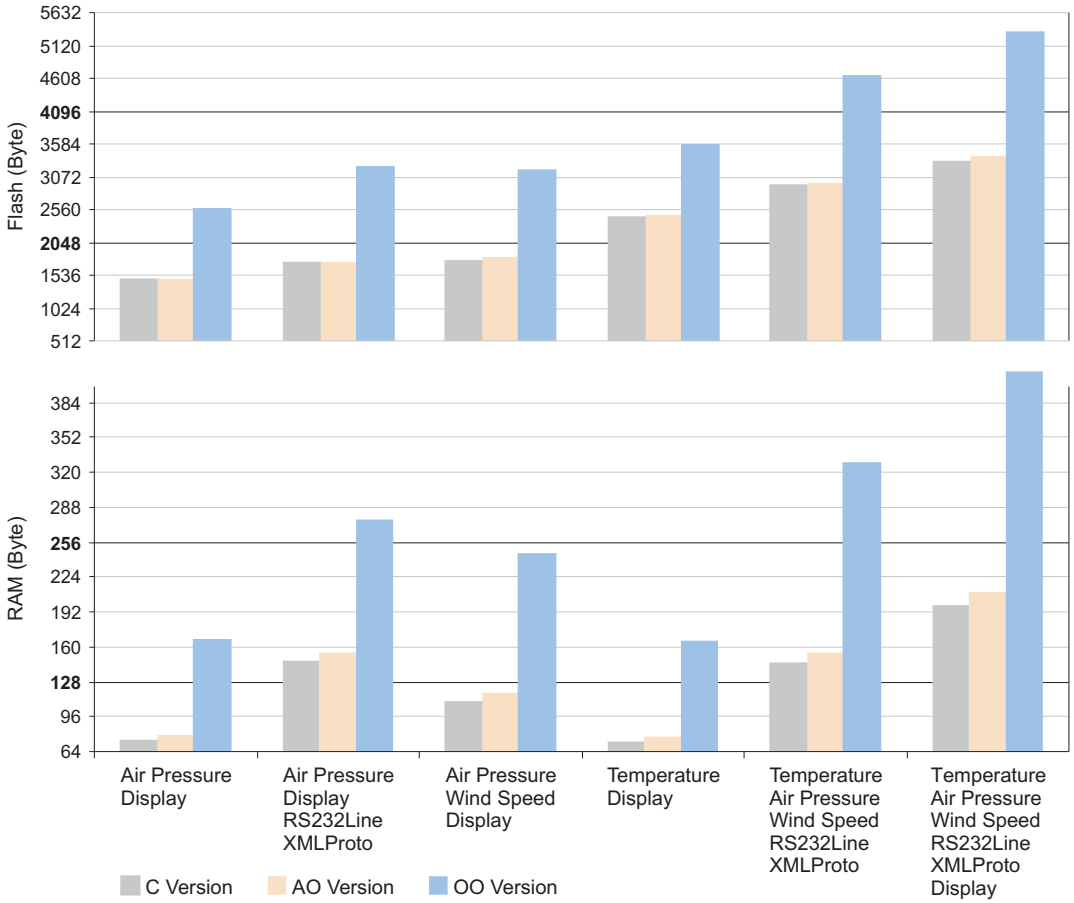


Figure B.5.: Footprint and cost scalability for different configurations.
Depicted are the various configurations of the WeatherMon product line and the resulting *RAM* and *Flash* footprint for the *C*, *AO*, and *OO* versions of the solution space. The emphasized numbers mark common RAM or Flash hardware limits (compare Table 2.1). If memory consumption exceeds one of these thresholds, a switch to a bigger and more expensive microcontroller is necessary. Note that the *OO* version generally requires a more expensive microcontroller.

B.4.2. Overall Scalability

As the graphs in Figure B.5 show, the resulting RAM and Flash demands of the weather-station software scale quite well with the amount of selected features. The “Barometer” configuration, consisting of just the features Air Pressure and Display, induces significantly smaller memory demands than the “Deluxe-PC” configuration, which bundles three sensors (Temperature, Air Pressure, Wind Speed) and two actuators (Display, PC Connection using XMLProto over RS232Line). The memory requirements of the other examined configurations are in between. The noticeably high amount of Flash memory required by the “Thermometer” configuration (Temperature, Display) can be explained by the fact that this sensor is connected via the I^2C bus to the microcontroller, which requires additional

variant	version	text	data	bss	stack	= flash	= RAM	time (ms)
Air Pressure, Display	**C**	*1392*	*30*	*7*	*34*	*1422*	*71*	*1.21*
	AO	*1430*	*30*	*10*	*38*	*1460*	*78*	*1.21*
	OO	*2460*	*100*	*22*	*44*	*2560*	*166*	*1.29*
Air Pressure, Display,	**C**	*1578*	*104*	*7*	*34*	*1682*	*145*	*60.40*
RS232Line, XMLProto	**AO**	*1622*	*104*	*12*	*38*	*1726*	*154*	*59.20*
	OO	*3008*	*206*	*26*	*44*	*3214*	*276*	*60.80*
Air Pressure, Wind Speed,	**C**	*1686*	*38*	*14*	*55*	*1724*	*107*	*2.96*
Display	**AO**	*1748*	*38*	*18*	*61*	*1786*	*117*	*2.96*
	OO	*3020*	*146*	*33*	*65*	*3166*	*244*	*3.08*
Temperature, Display	**C**	*2378*	*28*	*8*	*34*	*2406*	*70*	*1.74*
	AO	*2416*	*28*	*11*	*38*	*2444*	*77*	*1.73*
	OO	*3464*	*98*	*23*	*44*	*3562*	*165*	*1.82*
Temperature, Wind Speed,	**C**	*2804*	*90*	*17*	*35*	*2894*	*142*	*76.40*
Air Pressure, RS232Line,	**AO**	*2858*	*90*	*23*	*41*	*2948*	*154*	*76.40*
XMLProto	**OO**	*4388*	*248*	*39*	*41*	*4636*	*328*	*76.40*
Temperature, Wind Speed,	**C**	*3148*	*122*	*17*	*57*	*3270*	*196*	*79.60*
Air Pressure, RS232Line,	**AO**	*3262*	*122*	*24*	*63*	*3384*	*209*	*77.60*
XMLProto, Display	**OO**	*5008*	*300*	*44*	*67*	*5308*	*411*	*80.00*

Table B.1.: Memory usage and run time of the AO and OO versions for different configurations.
Measured were the *text*, *data*, *bss*, and *stack* memory demands (Byte) and the *time* of a complete `measure()` and `process()` cycle. *Flash* = *text* + *data*, *RAM* = *data* + *bss* + *stack*
[All variants were woven and compiled with AC++-1.0PRE3 and AVR-G++-3.4.1 using -Wall -fno-rtti -Os -fno-exceptions -fomit-frame-pointer -ffunction-sections optimization flags.]

driver code that does not have to be included for other sensors.

Overall, all three versions meet the goal of scalability, which is an indicator for achieved granularity. In every case, however, the OO version requires significantly more memory space than its AO counterpart, which comes very close to the C version. Depending on the configuration, the required amount of RAM is up to 138 percent higher in the OO version, while the AO version takes only an extra of 10 percent at maximum (up to 13 byte) – both compared to the C-based version that does not provide configurability by separation of concerns. The amount of Flash memory is up to 91 percent higher in the OO version, but only 4 percent at maximum in the AO version. The net difference between using AOP and OOP has to be considered as even higher, as all versions of each configuration are linked with the same (configuration-, but not version-dependent) set of device drivers whose memory requirements are included in these numbers.[5]

[5]The driver library is actually an early prototype of the CiAO OS for the AVR platform. Parts of the driver code are implemented as inline functions, so it is not possible to differentiate between driver-induced and application-induced code here.

B.4.3. Memory Cost

Table B.1 breaks down the required overall amount of RAM and flash memory into their origins. The OO variants induce especially higher static memory demands. The *text* sections of the OO variants are up to 78 percent, the *data* sections up to 284 percent bigger than in the AO variants. The following cost drivers can be mainly accounted for this:

Virtual functions are the main source of additional code and data, as they induce overhead on both the caller and the callee side. On the caller side, each call to a virtual function requires, compared to advice code inlining, at least 16 additional byte in the machine code. On the callee side, virtual function tables have to be provided (4 byte + 2 byte per entry, *data*), object instances need to carry a pointer to the virtual function table (2 byte per instance), and constructors to initialize the vtable pointer of an instance have to be generated (at least 24 byte per class, *text*). In the "Barometer" configuration (Air Pressure and Display), for instance, 52 byte of the *data* section are occupied solely by virtual function tables.[6]
Regarding code size, the situation may become even worse in larger projects: Due to late binding, the bodies of virtual functions are never removed from the final image by means of function-level linking. Thus, "dead" function code that is never called at run time becomes nevertheless part of the *text* section.

Dynamic data structures are another source of additional overhead. The chaining of actuators and sensors induces 8 additional data byte in the "Barometer" configuration, plus some extra code to access and iterate over the lists.

Static instance construction causes some more "hidden" code to be generated. For each translation unit that defines one or more object instances with static linkage, the compiler has to generate a specific initialization-and-destruction function (88 bytes, *text*). Pointers to these functions are additionally stored in the *data* section.

Regarding dynamic memory usage (*stack*), the differences between the AO and OO version are less significant. The OO variants only need a few byte (up to 16%) more stack space than the related AO variants. This seems surprising – given that virtual function calls can not be inlined, therefore lead to a higher call depth and, thus, higher stack utilization. Part of this effect can be explained by the fact that the AOP version requires some additional stack space as well (2-4 byte), namely by context-binding pointcut functions and the join-point API (Section 4.3.2.3). The main reason is, however, that the maximum virtual function call depth is with 2 levels quite low. As the AVR architecture provides 32 general-purpose registers, which are also used for passing function parameters, it can furthermore be considered as quite "stack-friendly". On other CPU architectures (such as Intel x86), the differences between AO-based and OO-based solutions would be more significant.

[6] The AVR RISC core uses a Harvard architecture, thus vtables can not be placed in the *text* section.

B.4.4. Run-Time Cost

Table B.1 also lists the measured run times. In all configurations that support a serial connection the measurement and processing cycle time is mainly dominated by the transmission time over the serial interface. Here the cost is almost the same. In all other configurations the performance of the AO version and the C implementation are equal. The run-time overhead of the OO version is between 4 and 6.6 percent. It can be explained with the extra cost of virtual function calls.

B.5. Summary

With respect to the evaluation of AOP as a means to implement configurability in software product lines for resource-thrifty embedded devices the “WeatherMon” study was successful:

- It could be shown that AOP provides excellent means to design and implement for configurability in software product lines. Thanks to the match mechanism and the advice concept, it becomes very easy to design loosely coupled components that automatically integrate themselves into the system. This facilitates fine granularity, pluggability, and extensability.
- To a certain degree, similar benefits are also achievable with OOP and design patterns. The OO version, however, induces dramatically higher memory requirements, which has a direct impact on the hardware cost: In all cases, the “luxury” of a clear separation of concerns by OOP makes it necessary to apply a more expensive microcontroller than with the scattered C implementation. For the AO version, in contrast, there is no single case in which the clear separation of concerns does induce higher hardware cost than with the C version. Separation of concerns by AOP clearly is affordable – even in the domain of deeply embedded systems.

Considering that expressiveness deficiencies of OOP have been the motivating factor for the development of AOP (Section 2.3), it might be somewhat surprising that AOP revealed only quantitative but no hard qualitative benefits over OOP in the “WeatherMon” study. The reason is that all the WeatherMon features represent fine-grained configuration options that affect only a few join points – there is only little need for quantification in these cases.

Bibliography

[ABG+86] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proceedings of the 1986 USENIX Technical Conference*, pages 93–112. USENIX Association, June 1986.
(Cited on page 149.)

[ABV92] Mehmet Aksit, Lodewijk Bergmans, and Sinan Vural. An object-oriented language-database integration model: The composition-filters approach. In Ole Lehrmann Madsen, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '92)*, volume 615 of *Lecture Notes in Computer Science*, pages 372–395. Springer-Verlag, 1992.
(Cited on pages 41 and 79.)

[AC+] AspectC++ homepage. `http://www.aspectc.org/`.
(Cited on pages 38 and 177.)

[ACH+05] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Optimising AspectJ. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 117–128, New York, NY, USA, June 2005. ACM Press.
(Cited on page 97.)

[Ada] Bram Adams. Aspicere homepage. `http://users.ugent.be/~badams/aspicere`.
(Cited on page 78.)

[Ada06] Bram Adams. AOP on the C-side. In *AOSD Workshop on Linking Aspect Technology and Evolution (AOSD-LATE '06)*, March 2006.
(Cited on page 77.)

[Ada08] Bram Adams. *Co-Evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems*. PhD thesis, Universiteit Gent, Faculteit Ingenierswetenschappen, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium, 2008.
(Cited on pages 42, 77, and 78.)

[AGMO06] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Overview of CaesarJ. In *Transactions on AOSD I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, 2006.
(Cited on page 77.)

[Ale00] Andrei Alexandrescu. Traits: The else-if-then of types. *C++ Report*, April 2000.
(Cited on pages 79 and 87.)

[Ale01] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
(Cited on pages 85 and 185.)

[Ale02] Andrei Alexandrescu. Aspect-Oriented Programming in C++. In *Tutorial held on the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, November 2002.
(Cited on page 96.)

[ÅLS+03] Rickard A. Åberg, Julia L. Lawall, Mario Südholt, Gilles Muller, and Anne-Françoise Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE '03)*, pages 196–204, Montreal, Canada, March 2003. IEEE Computer Society Press.
(Cited on pages 5, 42, 43, 78, and 251.)

[Atk01] Colin Atkinson. *Component-Based Product Line Engineering with UML*. Addison-Wesley, 2001.
(Cited on page 27.)

[aut] AUTOSAR homepage. `http://www.autosar.org/`, visited 2009-03-26.
(Cited on page 62.)

[AUT06a] AUTOSAR. Requirements on operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.
(Cited on pages 45, 62, 123, 158, and 160.)

[AUT06b] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.
(Cited on pages 3, 17, 45, 62, 125, 126, 134, 158, 159, 160, and 250.)

[BC01] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.
(Cited on page 58.)

[BC04] Elisa Baniassad and Siobhán Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 158–167, Washington, DC, USA, 2004. IEEE Computer Society Press.
(Cited on page 129.)

[Ber] UC Berkeley. TinyOS homepage. `http://www.tinyos.net`.
(Cited on pages 17, 22, and 147.)

[Ber94] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, 1994.
(Cited on page 41.)

[Beu03] Danilo Beuche. *Composition and Construction of Embedded Software Families*. Dissertation, Otto-von-Guericke Universität Magdeburg, 2003.
(Cited on page 27.)

[Beu06] Danilo Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2006. `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`, visited 2009-03-26.
(Cited on pages 127 and 201.)

[BF] Avi Bryant and Robert Feldt. AspectR — simple aspect-oriented programming in Ruby. `http://aspectr.sourceforge.net/`.
(Cited on page 42.)

[BFK+99] Joachim Bayer, Oliver Flege, Peter Knauber, Roland Laqua, Dirk Muthig, Klaus Schmid, Tanya Widen, and Jean-Marc DeBaud. PuLSE: A methodology to develop software product lines. In *Proceedings of the 1999 Symposium on Software Reusability (SSR '99)*, pages 122–131, New York, NY, USA, 1999. ACM Press.
(Cited on page 27.)

[BGP+99a] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. On the development of object-oriented operating systems for deeply embedded systems - the PURE project. In *Object-Oriented Technology: ECOOP '99 Workshop Reader*, number 1743 in Lecture Notes in Computer Science, pages 27–31, Lisbon, Portugal, June 1999. Springer-Verlag.
(Cited on page 20.)

[BGP+99b] Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.
(Cited on pages 5, 17, 20, and 251.)

[BKPS04] Günter Böckle, Peter Knauber, Klaus Pohl, and Klaus Schmid. *Software-Produktlinien: Methoden, Einführung und Praxis*. dpunkt.verlag GmbH, Heidelberg, 2004.
(Cited on page 29.)

[Bro06] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*, pages 33–42, New York, NY, USA, 2006. ACM Press.
(Cited on page 11.)

[BvDDT07] Magiel Bruntink, Arie van Deursen, Maja D'Hondt, and Tom Tourwé. Simple crosscutting concerns are not so simple: Analysing variability in large-scale idioms-based implementations. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD '07)*, pages 199–211. ACM Press, 2007.
(Cited on page 71.)

[CDE01] Krysztof Czarnecki, Lutz Dominick, and Ulrich W. Eisenecker. Aspektorientierte Programmierung in C++, Teil 1–3. *iX, Magazin für professionelle Informationstechnik*, 8–10, 2001.
(Cited on page 96.)

[CE99] Krzysztof Czarnecki and Ulrich W. Eisenecker. Synthesizing objects. In R. Guerraoui, editor, *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, number 1628 in Lecture Notes in Computer Science, pages 18–42, Lisbon, Portugal, 1999. Springer-Verlag.
(Cited on pages 28 and 85.)

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000.
(Cited on pages 25, 27, 29, 30, 59, 84, and 85.)

[Chi95] Shigeru Chiba. Metaobject protocol for C++. In *Proceedings of the 10th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '95)*, pages 285–299, October 1995.
(Cited on page 82.)

[CIMR93] Roy Campbell, Nayeem Islam, Peter Madany, and David Raila. Designing and implementing Choices: An object-oriented system in C++. *Communications of the ACM*, 36(9), 1993.
(Cited on page 21.)

[CK03] Yvonne Coady and Gregor Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 50–59, Boston, MA, USA, March 2003. ACM Press.
(Cited on pages 5, 42, 78, and 251.)

[CKFS01] Yvonne Coady, Gregor Kiczales, Michael Feeley, and Greg Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 3rd Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE '01)*, 2001.
(Cited on pages 42 and 78.)

[CL03] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report TR 03-01, Department of Computer Science, Iowa State University, Ames, Iowa 50011-1040,

USA, 2003. http://archives.cs.iastate.edu/documents/disk0/00/00/02/96/00000296-00/paper.pdf.
(Cited on pages 43 and 44.)

[CN04] Shigeru Chiba and Kiyoshi Nakagawa. Josh: An open AspectJ-like language. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, 2004.
(Cited on page 77.)

[Coo03] Jim Cooling. *Software Engineering for Real-Time Systems*. Addison-Wesley, 2003.
(Cited on page 11.)

[CSS04] Constantinos Constantinides, Therapon Skotiniotis, and Maximilian Störzer. AOP considered harmful. In *1st European Interactive Workshop on Aspects in Software (EIWAS '04)*, 2004.
(Cited on pages 43 and 44.)

[CW01] Siobhán Clarke and Robert J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pages 5–14, Washington, DC, USA, 2001. IEEE Computer Society Press.
(Cited on page 129.)

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technische Universität Ilmenau, Ilmenau, Germany, 1998.
(Cited on pages 26 and 28.)

[Del05] Delta Software Technology GmbH. Angie – an introduction, June 2005.
(Cited on page 79.)

[Deu89] L Peter Deutsch. Design reuse and frameworks in the Smalltalk-80 programming system. In *Software Reusability, volume II*, pages 55–71. ACM Press, 1989.
(Cited on page 21.)

[DGH+04] Bruno Dufour, Christopher Goard, Laurie Hendren, Clark Verbrugge, Oege de Moor, and Ganesh Sittampalam. Measuring the dynamic behaviour of AspectJ programs. In *Proceedings of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*, pages 150–169, New York, NY, USA, 2004. ACM Press.
(Cited on pages 91 and 97.)

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki — a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.
(Cited on page 23.)

[DH96] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*, October 1996.
(Cited on page 21.)

[Dig04] Christopher Diggins. Aspect-oriented programming & C++. *Dr. Dobb's Journal of Software Tools*, 408(8), August 2004.
(Cited on page 96.)

[Dij68] Edsger Wybe Dijkstra. The structure of the THE-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.
(Cited on pages 20 and 26.)

[Dij72] Edsger Wybe Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.
(Cited on page 67.)

[DJ04] Maja D'Hondt and Viviane Jonckers. Hybrid aspects for weaving object-oriented functionality and rule-based knowledge. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 132–140, 2004.
(Cited on page 97.)

[DPM02] G. Denys, F. Piessens, and F. Matthijs. A survey of customizability in operating systems research. *ACM Computing Surveys*, 34(4):450–468, December 2002.
(Cited on page 18.)

[Dur] Pascall Durr. WeaveC project homepage. `http://weavec.sourceforge.net`.
(Cited on page 78.)

[EA06] Marc Eaddy and Alfred Vaino Aho. Statement annotations for fine-grained advising. In Walter Cazzola, Shigeru Chiba, Yvonne Coady, and Gunter Saake, editors, *Proceedings of the ECOOP'06 Workshop on Reflection, AOP, and Meta-Data for Software Evolution (RAM-SE '06)*, pages 89–99, Magdeburg, Germany, July 2006. Fakultät für Informatik, Universität Magdeburg.
(Cited on page 43.)

[EBB05] Magnus Eriksson, Jürgen Börstler, and Kjell Borg. The PLUSS approach — domain modeling with features, use cases and use case realizations. In J. Henk Obbink and Klaus Pohl, editors, *Proceedings of the 9th Software Product Line Conference (SPLC '05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 33–44. Springer-Verlag, 2005.
(Cited on page 27.)

[EBN02] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002.
(Cited on page 23.)

[eCo] eCos homepage. `http://ecos.sourceware.org/`.
(Cited on pages 3, 4, 23, 49, 101, and 249.)

[EF05] M. Engel and B. Freisleben. Supporting autonomic computing functionality via dynamic operating system kernel aspects. In Peri Tarr, editor, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 51–62, Chicago, Illinois, March 2005. ACM Press.
(Cited on page 42.)

[EF06] Michael Engel and Bernd Freisleben. TOSKANA: a toolkit for operating system kernel aspects. In Awais Rashid and Mehmet Aksit, editors, *Transactions on AOSD II*, number 4242 in Lecture Notes in Computer Science, pages 182–226. Springer-Verlag, 2006.
(Cited on page 42.)

[EFB01] Tzilla Elrad, Robert E. Filman, and Atef Bader. Aspect-oriented programming. *Communications of the ACM*, pages 29–32, October 2001.
(Cited on page 36.)

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, ACM SIGOPS Operating Systems Review, pages 251–266. ACM Press, 1995.
(Cited on pages 18 and 19.)

[Fav97] Jean-Marie Favre. A rigorous approach to support the maintenance of large portable software. In *1st Euromicro Working Conference on Software Maintenance and Reengineering (CSMR '97)*, page 44. IEEE Computer Society Press, 1997.
(Cited on page 23.)

[FBB+97] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A substrate for kernel and language research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, ACM SIGOPS Operating Systems Review, pages 38–51. ACM Press, October 1997.
(Cited on page 22.)

[FCF+06] Fernando Castor Filho, Nelio Cacho, Eduardo Figueiredo, Raquel Maranhão, Alessandro Garcia, and Cecília Mary F. Rubira. Exceptions and aspects: The devil is in the details. In *Proceedings of ACM SIGSOFT '06 / FSE-14*, pages 152–162, New York, NY, USA, 2006. ACM Press.
(Cited on page 43.)

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.
(Cited on pages 44 and 168.)

[FF00] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced SoC (OOPSLA '00)*, October 2000.
(Cited on pages 44 and 168.)

[FGCW05] Marc Fiuczynski, Robert Grimm, Yvonne Coady, and David Walker. patch(1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS '05)*. USENIX Association, 2005.
(Cited on pages 5, 42, 78, and 251.)

[Fil01] Robert E. Filman. What is aspect-oriented programming, revisited. Technical Report 01.14, Research Institute for Advanced Computer Science, Mountain View, CA, USA, May 2001. `http://www.riacs.edu/research/technical_reports/TR_pdf/TR_01.14.pdf`.
(Cited on page 36.)

[Fre87] P. Freeman. A conceptual analysis of the Draco approach to constructing software systems. *IEEE Transactions on Software Engineering*, 13(7):830–844, 1987.
(Cited on page 26.)

[FRT] FreeRTOS homepage. `http://freertos.org/`.
(Cited on page 23.)

[FSH+01] L. Fernando Friedrich, John Stankovic, Marty Humphrey, Michael Marley, and John Haskins. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, 21(3):54–68, 2001.
(Cited on pages 16 and 18.)

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia Lawall, and Gilles Muller. THINK: A software framework for component-based operating system kernels. In *Proceedings of the 2002 USENIX Technical Conference*, pages 73–86. USENIX Association, June 2002.
(Cited on page 22.)

[FSP99] A. Fröhlich and W. Schröder-Preikschat. High performance application-oriented operating systems – the EPOS aproach. In *Proceedings of the 11th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD '99)*, pages 3–9, Natal, Brazil, September 1999. IEEE Computer Society Press.
(Cited on page 17.)

[GB03] Kris Gybels and Johan Brichau. Arranging language features for pattern-based crosscuts. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 60–69, Boston, MA, USA, March 2003. ACM Press.
(Cited on page 97.)

[GB04] Iris Groher and Thomas Baumgarth. Aspect-orientation from design to code. In *Proceedings of the 2004 AOSD Early Aspects Workshop (AOSD-EA '04)*, March 2004.
(Cited on page 129.)

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
(Cited on pages 21, 182, and 212.)

[GJ] Michael Gong and Hans-Arno Jacobsen. AspeCt-oriented C (ACC) homepage. `http://www.aspectc.net`.
(Cited on pages 42 and 78.)

[GJ08] Weigang Gong and Hans-Arno Jacobsen. *AspeCt-oriented C Language Specification (Version 0.8)*. Middleware Systems Research Group, Department of Computer Science, University of Toronto, January 2008. `http://www.aspectc.net/acc_manual.pdf`.
(Cited on page 78.)

[GLC05] David Gay, Phil Levis, and David Culler. Software design patterns for TinyOS. In *Proceedings of the 2005 Workshop on Languages, Compilers and Tools for Embedded Systems (LCTES '05)*, pages 40–49. ACM Press, 2005.
(Cited on pages 17 and 22.)

[GLv+03] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, pages 1–11, San Diego, CA, USA, 2003. ACM Press.
(Cited on page 22.)

[Gom04] Hassan Gomaa. *Designing Software Product Lines with UML. From Use Cases to Pattern-Based Software Architectures*. Addison-Wesley, 2004.
(Cited on page 27.)

[Gom05] Hassan Gomaa. Architecture-centric evolution in software product lines. In *Proceedings of the ECOOP '05 Workshop on Architecture-Centric Evolution (ECOOP-ACE '05)*, Glasgow, UK, July 2005.
(Cited on page 27.)

[Gri] William G. Grisswold. AspectBrowser for Eclipe. `http://www.cse.ucsd.edu/~wgg/Software/AB/`.
(Cited on pages 53 and 55.)

[GS04] Jack Greenfield and Keith Short. *Software Factories*. John Wiley & Sons, Inc., 2004.
(Cited on page 29.)

[Han97] M. Hansen. How to reduce code blat from STL containers. *C++ Report*, pages 34–41, Jan 1997.
(Cited on page 95.)

[HE07] Kevin Hoffman and Patrick Eugster. Bridging Java and AspectJ through explicit join points. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ '07)*, pages 63–72, New York, NY, USA, 2007. ACM Press.
(Cited on page 43.)

[HFC76] Arie Nicolaas Habermann, Lawrence Flon, and Lee W. Cooprider. Modularization and hierarchy in a family of operating systems. *Communications of the ACM*, 19(5):266–272, 1976.
(Cited on pages 15, 18, 20, and 26.)

[HG04] Bruno Harbulot and John R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 122–131. ACM Press, March 2004.
(Cited on page 43.)

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in AspectJ. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 26–35, New York, NY, USA, March 2004. ACM Press.
(Cited on page 71.)

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Sebastian Schönberg, and Jean Wolter. The performance of μ-kernel-based systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, New York, NY, USA, October 5–8 1997. ACM Press.
(Cited on page 61.)

[Hir03] Robert Hirschfeld. Aspect-oriented programming with AspectS. In *NetObjectDays (NODe '02)*, volume 2591 of *Lecture Notes in Computer Science*. Springer-Verlag, October 2003.
(Cited on page 42.)

[HK02] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '02)*, pages 161–173, New York, NY, USA, 2002. ACM Press.
(Cited on pages 183 and 184.)

[HLSP08] Wanja Hofer, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Concern impact analysis in configurable system software—the AUTOSAR OS case. In *Proceedings of the 7th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)*, pages 1–6, New York, NY, USA, March 2008. ACM Press.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming—a critique of pure objects. In *Proceedings of the 8th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, September 1993.
(Cited on pages 41 and 79.)

[HU03] Stefan Hanenberg and Rainer Unland. Parametric introductions. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 80–89, Boston, MA, USA, March 2003. ACM Press.
(Cited on page 97.)

[Inf05] Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. *TriCore 1 User's Manual (V1.3.5), Volume 1: Core Architecture*, February 2005.
(Cited on page 141.)

[Ins03] The British Standards Institute. *The C++ Standard (Incorporating Technical Corrigendum No. 1)*. John Wiley & Sons, Inc., second edition, 2003. Printed version of the ISO/IEC 14882:2003 standard.
(Cited on pages 79, 177, and 190.)

[Joh97] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, 1997.
(Cited on page 21.)

[JSS08] Rasmus Johansen, Peter Sestoft, and Stephan Spangenberg. Zero-overhead composable aspects for .NET. In Antonio Cisternino Egon Börger, editor, *Proceedings of the Advances in Software Engineering (Lipari Summer School '07)*, volume 5316 of *Lecture Notes in Computer Science*, pages 185–215. Springer-Verlag, 2008.
(Cited on page 97.)

[JZ01] Stan Jarzabek and Hongyu Zhang. XML-based method and tool for handling variant requirements in domain model. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering (RE '01)*, pages 116–123, Toronto, Canada, August 2001. IEEE Computer Society Press.
(Cited on page 79.)

[KAB07] Christian Kästner, Sven Apel, and Don Batory. A case study implementing features using AspectJ. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, pages 223–232. IEEE Computer Society Press, 2007.
(Cited on pages 43, 44, 119, and 169.)

[KAK08] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in software product lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, pages 311–320, New York, NY, USA, 2008. ACM Press.
(Cited on page 43.)

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Novak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1990.
(Cited on pages 26 and 28.)

[KE95] Steve Kleiman and Joe Eykholt. Interrupts as threads. *ACM SIGOPS Operating Systems Review*, 29(2):21–26, April 1995.
(Cited on page 149.)

[KHH+01a] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. Getting started with AspectJ. *Communications of the ACM*, pages 59–65, October 2001.
(Cited on pages 37, 41, 42, and 77.)

[KHH+01b] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In J. Lindskov Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, June 2001.
(Cited on pages 37, 41, and 77.)

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.
(Cited on pages 4, 32, 37, 41, and 250.)

[KR06] Günter Kniesel and Tobias Rho. A definition, overview and taxonomy of generic aspect languages. *L'Objet, Special Issue on Aspect-Oriented Software Development*, 11(2–3):9–39, September 2006.
(Cited on pages 77 and 97.)

[KRH04] Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable pattern implementations need generic aspects. In *Proceedings of the ECOOP '04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution (ECOOP-RAM-SE '04)*, Oslo, Norway, June 2004.
(Cited on pages 77 and 97.)

[Küm] Peter Kümmel. Loki: A C++ library of designs, containing flexible flexible implementations of common design patterns and idioms. `http://loki-lib.sourceforge.net`.
(Cited on page 185.)

[LBS04] Daniel Lohmann, Georg Blaschke, and Olaf Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, volume

3286 of *Lecture Notes in Computer Science*, pages 55–74. Springer-Verlag, October 2004.
(Cited on page 83.)

[LE93] A.M. Lister and R.D. Eager. *Fundamentals of Operating Systems*. Macmillian, 5th edition, 1993.
(Cited on pages 20, 58, and 60.)

[LGS04] Daniel Lohmann, Wasif Gilani, and Olaf Spinczyk. On adapable aspect-oriented operating systems. In O. Spinczyk, A. Gal, and M. Schoettner, editors, *Proceedings of the 2004 ECOOP Workshop on Programming Languages and Operating Systems (ECOOP-PLOS '04)*, pages 47–52, June 2004.
(Cited on page 38.)

[Lie95] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, ACM SIGOPS Operating Systems Review. ACM Press, December 1995.
(Cited on page 149.)

[Lie96] Karl J. Lieberherr. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996.
(Cited on pages 41 and 79.)

[Lit] Tim Littlefair. CCCC - C and C++ Code Counter homepage. `http://cccc.sourceforge.net/`.
(Cited on pages 54, 103, and 106.)

[LLW03] Karl J. Lieberherr, David H. Lorenz, and Pengcheng Wu. A case for statically executable advice: checking the law of demeter with AspectJ. In Mehmet Akşit, editor, *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 40–49, Boston, MA, USA, March 2003. ACM Press.
(Cited on pages 41 and 77.)

[LMU05] Julia L. Lawall, Gilles Muller, and Richard Urunuela. Tarantula: Killing driver bugs before they hatch. In *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, pages 13–18, Chicago, IL, March 2005.
(Cited on page 78.)

[LN79] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, April 1979.
(Cited on page 61.)

[Lop97] Cristina Isabel Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, Boston, MA, USA, 1997.
(Cited on page 41.)

[Lov05] Robert Love. *Linux Kernel Development*. Novell Press, 2nd edition, 2005.
(Cited on page 148.)

[LS03] Daniel Lohmann and Olaf Spinczyk. Architecture-neutral operating system components. *23rd ACM Symposium on Operating Systems Principles (SOSP '03)*, October 2003. WiP presentation.

[LS05a] Ralf Lämmel and Kris De Schutter. What does aspect-oriented programming mean to Cobol? In Peri Tarr, editor, *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 99–110, Chicago, Illinois, March 2005. ACM Press.
(Cited on page 42.)

[LS05b] Daniel Lohmann and Olaf Spinczyk. On typesafe aspect implementations in C++. In F. Geschwind, U. Assmann, and O. Nierstrasz, editors, *Proceedings of Software Composition 2005 (SC '05)*, volume 3628 of *Lecture Notes in Computer Science*, pages 135–149, Edinburgh, UK, April 2005. Springer-Verlag.

[LS06] Daniel Lohmann and Olaf Spinczyk. Developing embedded software product lines with AspectC++. In Peri L. Tarr and William R. Cook, editors, *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 740–742, Portland, Oregon, USA, 2006. ACM Press.

[LSH+07] Daniel Lohmann, Jochen Streicher, Wanja Hofer, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Configurable memory protection by aspects. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS '07)*, pages 1–5, New York, NY, USA, October 2007. ACM Press.
(Cited on page 159.)

[LSPS05] Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS '05)*, pages 413–420, Sedona, AZ, USA, February 2005.

[LSSP05] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, pages 19–25, Chicago, IL, USA, March 2005. Northeastern University, Boston (NU-CCIS-05-03).

[LSSP06] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Lean and efficient system software product lines: Where aspects beat objects. In Awais Rashid and Mehmet Aksit, editors, *Transactions on AOSD II*, number

4242 in Lecture Notes in Computer Science, pages 227–255. Springer-Verlag, 2006.

[LSSSP07] Daniel Lohmann, Jochen Streicher, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Interrupt synchronization in the CiAO operating system. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, New York, NY, USA, 2007. ACM Press.

[LST+06] Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM Press.

[Mar06] Peter Marvedel. *Embedded System Design*. Springer-Verlag, Heidelberg, Germany, 2006.
(Cited on page 11.)

[Mas02] Anthony Massa. *Embedded Software Development with eCos*. New Riders, 2002.
(Cited on pages 49 and 101.)

[McC04] Steve McConnell. *Code Complete*. Microsoft Press, second edition, 2004.
(Cited on page 67.)

[McK05] Eric McKean, editor. *The New Oxford American Dictionary*. Oxford University Press Inc, USA, second edition, 2005.
(Cited on pages 6, 168, and 252.)

[MKD04] Hideihiko Mashuhara, Gregor Kiczales, and Chris Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proceedings of Compiler Construction (CC '04)*, volume 2622 of *Lecture Notes in Computer Science*, pages 46–60. Springer-Verlag, 2004.
(Cited on page 71.)

[MKM04] George V. Neville-Neil Marshall Kirk McKusick. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2004.
(Cited on page 149.)

[MSGSP02] Daniel Mahrenholz, Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. An aspect-oriented implementation of interrupt synchronization in the PURE operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems (ECOOP-OOOS '02)*, pages 49–54, Malaga, Spain, June 2002.
(Cited on pages 5, 42, and 251.)

[Mye96] Nathan Myers. *A new and useful template technique: "traits"*, pages 451–457. C++ gems. SIGS Publications, Inc., New York, NY, USA, 1996.
(Cited on pages 79 and 87.)

[NC01] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
(Cited on pages 24 and 26.)

[NvGvdP08] István Nagy, Remco van Gngelen, and Dark van der Ploeg. An overview of Mirjam and WeaveC: an industrial-strength aspect-oriented language and weaver for C. In Wouter Joosen, editor, *7th International Conference on Aspect-Oriented Software Development (AOSD '08)–Industry Track Proceedings*, pages 68–76, April 2008.
(Cited on pages 42 and 78.)

[OSE] OSEK/VDX group homepage. `http://www.osek-vdx.org/`.
(Cited on page 61.)

[OSE04] OSEK/VDX Group. OSEK implementation language specification 2.5. Technical report, OSEK/VDX Group, 2004. `http://portal.osek-vdx.org/files/pdf/specs/oil25.pdf`.
(Cited on page 126.)

[OSE05] OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group, February 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2009-03-26.
(Cited on pages 17, 30, 61, 62, 125, 126, 134, 157, 158, 159, and 160.)

[OT00] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer Academic Publishers, 2000.
(Cited on pages 37, 41, and 77.)

[OT01] Harold Ossher and Peri Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, pages 43–50, October 2001.
(Cited on pages 41 and 77.)

[oU] University of Utah. OSKit homepage. `http://www.cs.utah.edu/flux/oskit`.
(Cited on page 22.)

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, pages 1053–1058, December 1972.
(Cited on pages 26 and 44.)

[Par76a] David Lorge Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.
(Cited on pages 20 and 26.)

[Par76b] David Lorge Parnas. Some hypothesis about the "uses" hierarchy for operating systems. Technical report, TH Darmstadt, Fachbereich Informatik, 1976.
(Cited on pages 20, 26, and 127.)

[Par79] David Lorge Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.
(Cited on pages 15 and 58.)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.
(Cited on page 27.)

[PLM06] Yoann Padioleau, Julia L. Lawall, and Gilles Muller. Understanding collateral evolution in Linux device drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006 (EuroSys '06)*, pages 59–71, Leuven, Belgium, April 2006. Also available as INRIA Research Report RR-5769.
(Cited on page 78.)

[PLMH08] Yoann Padioleau, Julia L. Lawall, Gilles Muller, and René Rydhof Hansen. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (EuroSys '08)*, Glasgow, Scotland, March 2008.
(Cited on page 78.)

[Pro] ProOSEK homepage. `http://www.proosek.de/`.
(Cited on page 23.)

[RC01] Alexandro Rubini and Jonathan Corbet. *Linux Device Drivers*. O'Reilly, 2001.
(Cited on page 148.)

[RFS+00] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *4th Symposium on Operating System Design and Implementation (OSDI '00)*, pages 347–360, San Diego, CA, USA, October 2000. USENIX Association.
(Cited on page 22.)

[RPCB04] O. Rohlik, A. Pasetti, V. Cechticky, and I. Birrer. Implementing adaptability in embedded software through aspect oriented programming. In *Proceedings of Mechatronics & Robotics (MechRob '05)*, Aachen, Germany, September 2004. IEEE Computer Society Press.
(Cited on page 79.)

[Sak98] Ken Sakamura. *µItron 3.0: An Open and Portable Real-Time Operating System for Embedded Systems : Concept and Specification*. IEEE Computer Society Press, 1998.
(Cited on pages 17 and 30.)

[SBB02] Bjorn De Sutter, Bruno De Bus, and Koen De Bosschere. Sifting out the mud: low level C++ code reuse. *SIGPLAN Not.*, 37(11):275–291, 2002.
(Cited on page 95.)

[SC92] Henry Spencer and Gehoff Collyer. #ifdef considered harmful, or portability experience with C News. In *Proceedings of the 1992 USENIX Technical Conference*. USENIX Association, June 1992.
(Cited on page 23.)

[Sch86] Wolfgang Schröder. *Eine Familie von UNIX-ähnlichen Betriebssystemen – Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes beim strukturierten Betriebssystementwurf*. Dissertation, Technische Universität Berlin, December 1986.
(Cited on page 149.)

[SGG05] Abraham Silberschatz, Greg Gagne, and Peter Bear Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., seventh edition, 2005.
(Cited on pages 18 and 58.)

[SGS+05] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 166–175, New York, NY, USA, 2005. ACM Press.
(Cited on pages 43 and 44.)

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific '02)*, pages 53–60, Sydney, Australia, February 2002.
(Cited on pages 38 and 177.)

[SHU02] Dominik Stein, Stefan Hanenberg, and Rainer Unland. A UML-based aspect-oriented design notation for AspectJ. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD '02)*, pages 106–112, New York, NY, USA, 2002. ACM Press.
(Cited on page 129.)

[Sim95] Mark A. Simos. Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. In *Proceedings of the 1995 Symposium on Software Reusability (SSR '95)*, pages 196–205, New York, NY, USA, 1995. ACM Press.
(Cited on pages 25, 26, and 27.)

[SKW+06] Dilma Da Silva, Orran Krieger, Robert W. Wisniewski, Amos Waterland, David Tam, and Andrew Baumann. K42: an infrastructure for operating system research. *ACM SIGOPS Operating Systems Review*, 40(2):34–42, 2006.
(Cited on page 21.)

[SL04] Olaf Spinczyk and Daniel Lohmann. Using AOP to develop architecture-neutral operating system components. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 188–192, New York, NY, USA, September 2004. ACM Press.
(Cited on pages 5, 42, 131, and 251.)

[SL06] Olaf Spinczyk and Daniel Lohmann. *AspectC++ Quick Reference (Version 1.11)*. pure::systems GmbH, Agnetenstr. 14, 39106 Magdeburg, Germany, March 2006. `http://www.aspectc.org/fileadmin/documentation/ac-quickref.pdf`.
(Cited on page 190.)

[SL07] Olaf Spinczyk and Daniel Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, 2007.
(Cited on pages 38 and 177.)

[SLU05a] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. Advances in AOP with AspectC++. In Hamido Fujita and Mohamed Mejri, editors, *New Trends in Software Methodologies, Tools and Techniques (SoMeT '05)*, number 129 in Frontiers in Artificial Intelligence and Applications, pages 33–53, Tokyo, Japan, September 2005. IOS Press.
(Cited on pages 38 and 177.)

[SLU05b] Olaf Spinczyk, Daniel Lohmann, and Matthias Urban. AspectC++: an AOP extension for C++. *Software Developers Journal*, (5):68–76, May 2005.

[SP94] Wolfgang Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall PTR, 1994.
(Cited on page 148.)

[SP02] W. Schult and A. Polze. Aspect-oriented programming with C# and .NET. In *Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '02)*, pages 241–248, 2002.
(Cited on page 42.)

[Spi02] Olaf Spinczyk. *Aspektorientierung und Programmfamilien im Betriebssystembau*. Dissertation, Otto-von-Guericke Universität Magdeburg, Institut für Verteilte Systeme, December 2002.
(Cited on pages 5, 42, 177, and 251.)

[SPLG+06] Wolfgang Schröder-Preikschat, Daniel Lohmann, Wasif Gilani, Fabian Scheler, and Olaf Spinczyk. Static and dynamic weaving in system software with AspectC++. In Yvonne Coady, Jeff Gray, and Raymond Klefstad, editors, *Proceedings of the 39th Hawaii International Conference on System Sciences (HICSS '06) - Track 9*. IEEE Computer Society Press, 2006.
(Cited on page 38.)

[SR00] David A. Solomon and Mark Russinovich. *Inside Microsoft Windows 2000*. Microsoft Press, third edition, 2000.
(Cited on page 148.)

[SSPSS98] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Design rationale of the PURE object-oriented embedded operating system. In *Proceedings of the International IFIP TC10 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, pages 231–240, Paderborn, Germany, October 1998.
(Cited on page 20.)

[SSSPS07] Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the linux kernel a software product line? In Frank van der Linden and Björn Lundell, editors, *Proceedings of the International Workshop on Open Source Software and Product Lines (SPLC-OSSPL 2007)*, Kyoto, Japan, 2007.
(Cited on page 30.)

[SSW02] Mario Schüpany, Christa Schwanninger, and Egon Wuchner. Aspect-oriented programming for .NET. In *Proceedings of the 1st AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '02)*, March 2002.
(Cited on page 42.)

[Sta08] William Stallings. *Operating Systems. Internals and Design Principles*. Prentice Hall PTR, sixth edition, 2008.
(Cited on page 58.)

[Ste05] Friedrich Steimann. Domain models are aspect free. In *Proceedings of the 8th International Conference on Model-Driven Engineering, Languages, and Systems (MoDELS '05)*, volume 3713 of *Lecture Notes in Computer Science*, pages 171–185. Springer-Verlag, 2005.
(Cited on page 44.)

[Ste06] Friedrich Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*, pages 481–497, New York, NY, USA, 2006. ACM Press.
(Cited on pages 43, 44, and 168.)

[Str97] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.
(Cited on page 177.)

[SWK06] Jamal Siadat, Robert J. Walker, and Cameron Kiddle. Optimization aspects in network simulation. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD '06)*, pages 122–133. ACM Press, March 2006.
(Cited on pages 43 and 119.)

[Tan06] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall PTR, fifth edition, 2006.
(Cited on page 15.)

[Tan07] Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, third edition, 2007.
(Cited on page 58.)

[Ten00] David Tennenhouse. Proactive computing. *Communications of the ACM*, pages 43–45, May 2000.
(Cited on pages 3, 11, 14, and 15.)

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*, pages 107 – 119. IEEE Computer Society Press, May 1999.
(Cited on pages 37 and 41.)

[Tou05] Jean-Charles Tournier. A survey of configurable operating systems. Technical Report TR-CS-2005-43, University of New Mexico, Albuquerque, NM, USA, November 2005. `http://www.cs.unm.edu/~treport/tr/05-12/Tournier.pdf`.
(Cited on page 18.)

[Tur02] Jim Turley. The two percent solution. *embedded.com*, December 2002. `http://www.embedded.com/story/OEG20021217S0039`, visited 2009-03-26.
(Cited on pages 11 and 12.)

[Tur05] Jim Turley. Survey says: Software tools more important than chips. *embedded.com*, November 2005. `http://www.embedded.com/showArticle.jhtml?articleID=160700620`.
(Cited on page 12.)

[Tur06] Jim Turley. Operating systems on the rise. *embedded.com*, June 2006. `http://www.embedded.com/columns/showArticle.jhtml?articleID=187203732`.
(Cited on pages 17 and 49.)

[US06] Matthias Urban and Olaf Spinczyk. *AspectC++ Language Reference (Version 1.6)*. pure::systems GmbH, Agnetenstr. 14, 39106 Magdeburg, Germany, March 2006. `http://www.aspectc.org/fileadmin/documentation/ac-languageref.pdf`, visited 2009-03-26.
(Cited on pages 181 and 190.)

[Vel95] Todd Veldhuizen. Template metaprograms. *C++ Report*, May 1995.
(Cited on page 84.)

[Wal04] Collin Walls. The Perfect RTOS. Keynote at embedded world '04, Nuremberg, Germany, 2004.
(Cited on pages 17 and 18.)

[Wit96] James Withey. Investment analysis of software assets for product lines. Technical report, Carnegie Mellon University, Software Engineering Institute, Pittsburgh, PA, November 1996.
(Cited on page 24.)

[WL99] David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.
(Cited on page 26.)

[XWe] The XWeaver project homepage. `http://www.xweaver.org`.
(Cited on page 79.)

[YKCI06] Yoshisato Yanagisawa, Kenichi Kourai, Shigeru Chiba, and Rei Ishikawa. A dynamic aspect-oriented system for OS kernels. In *Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 69–78, New York, NY, USA, October 2006. ACM Press.
(Cited on page 42.)

[Yu] Yijun Yu. Implementation of aspectPHP. `http://www.cs.toronto.edu/~yijun/aspectPHP/`.
(Cited on page 42.)

[ZHS04] David Zook, Shan Shan Huang, and Yannis Smaragdakis. Generating AspectJ programs with Meta-AspectJ. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 1–18. Springer-Verlag, October 2004.
(Cited on page 97.)

Aspect Awareness in the Development of Configurable System Software

—

Aspektgewahrheit bei der Entwicklung konfigurierbarer Systemsoftware

Der Technischen Fakultät der
Universität Erlangen-Nürnberg

zur Erlangung des Grades

DOKTOR-INGENIEUR

vorgelegt von

Daniel Lohmann

Erlangen — 2008

Kurzfassung

Mehr als 98 Prozent der weltweiten, jährlichen Mikroprozessorproduktion wird in eingebetteten Systemen verbaut – überwiegend in massenproduzierten Gegenständen des täglichen Bedarfs, wie beispielsweise Autos, Haushaltsgeräte, oder Spielzeug. Eingebettete Systeme sind daher einem enormen Hardware-Kostendruck ausgesetzt. Systemsoftware für eingebettete Systeme muss nicht nur unterschiedlichste Anforderungen und Plattformen erfüllen, sie muss vor allem auch ausgesprochen genügsam sein im Bezug auf die erforderlichen Hardware-Ressourcen. Um gegen proprietäre Systeme bestehen zu können (und damit Wiederverwendung zu ermöglichen), ist es erforderlich, dass Systemsoftware-Produktlinien für eingebettete Systeme hochgradig konfigurierbar und maßschneiderbar sind. Technisch muss diese Flexibilität in einer Weise umgesetzt werden, die dem Anspruch der Ressourcengenügsamkeit gerecht wird.

Stand der Kunst für die ressourcenschonende Implementierung feingranularer Konfigurierbarkeit in Systemsoftware ist die bedingte Übersetzung unter Zuhilfenahme des C-Präprozessors. Dieser Ansatz führt jedoch zu schlecht wartbarem Code – er skaliert nicht. Gleichzeitig wachsen die Konfigurierbarkeitsanforderungen an Systemsoftware. Der neue AUTOSAR-OS Industriestandard verlangt beispielsweise, dass auch grundlegende architekturelle Strategien des Systems, wie die zu verwendenden Schutz- und Isolationmaßnahmen, konfigurierbar ausgelegt werden.

Diese Arbeit untersucht die Eignung der aspektorientierten Programmierung (AOP) als grundlegenden Mechanismus für die Implementierung von Konfigurierbarkeit in ressourcenbeschränkten Systemen. Es wird gezeigt, dass die gezielte und pragmatische Verwendung von AOP zu einer deutlichen Verbesserung bei der Trennung der Belange in konfigurierbarer Systemsoftware führt, ohne dass dieses Nachteile im Bezug auf die Genügsamkeit hat. Der vorgestellte Ansatz der *aspektgewahren Betriebssystementwicklung* ermöglicht es darüber hinaus, selbst grundlegende Architektureigenschaften als konfigurierbare Merkmale aufzufassen und zu implementieren.

Der Ansatz wird am Beispiel von aktuellen Betriebssystemen aus dem Umfeld der eingebetteten Systeme evaluert. Die entstandene CiAO-Betriebssystemfamilie verbindet eine konkurrenzfähige Implementierung des AUTOSAR-OS-Standards mit einer hochgradig konfigurierbaren Architektur. CiAO ist das erste Betriebssystem, das von Beginn an mit Aspekttechniken entworfen und entwickelt wurde.

Inhaltsverzeichnis

Einleitung

Motivation

Etwa 98 Prozent der hergestellten Mikroprozessoren (alleine im Jahr 2000 mehr als acht Milliarden Einheiten) finden Verwendung in einem eingebetteten System. Eingebettete Systeme sind Spezialzweckrechner, oft zu finden in Gegenständen des täglichen Bedarfs (wie zum Beispiel Autos, Hausgeräte oder Spielzeug), wo sie spezielle Berechungs- und Steuerungsaufgaben übernehmen. Als elementarer Bestandteil von Massenprodukten sind eingebettete Systeme einem enormen Kostendruck ausgesetzt. Wenige Cent bei den Herstellungskosten können entscheidend sein für Erfolg oder Misserfolg auf dem Markt. Dies hat Einschränkungen bei den verfügbaren Hardwareressourcen (wie CPU-Leistung und Speicherkapazität) zur Folge; 8-Bit-Prozessoren mit wenigen KiB Speicher sind eine übliche Konfiguration.

Die Beschränkungen der Hardware haben Konsequenzen für den Entwurf und die Entwicklung von Betriebssystemen und anderer Systemsoftware. Systemsoftware für eingebettete Systeme muss auf einfache Weise für die jeweilige konkrete Anwendung maßzuschneidern sein. Eine Möglichkeit ist dabei der Entwurf der Systemsoftware als Softwareproduktlinie, welche zusammen mit einem Konfigurationswerkzeug ausgeliefert wird. Damit kann der Anwendungsentwickler aus einer Menge von feingranularen Merkmalen auswählen und sich eine maßgeschneiderte Variante für den jeweiligen Zweck generieren lassen.

Bei der Implementierung der Systemsoftware muss diese Flexibilität in einer Weise durchgesetzt werden, die den Hardwareeinschränkungen gerecht wird. Stand der Kunst ist dabei, feingranulare Konfigurationsoptionen mittels bedingter Übersetzung durch den C-Präprozessor durchzusetzen. Die Implementierung eines konfigurierbaren Merkmals (zum Beispiel eine Synchronisationsstrategie) wird dabei mit `#ifdef` – `#endif` Blöcken in die Implementierung anderer Merkmale eingebettet. Die resultierende Verteilung des Codes bei einer solchen Implementierung führt jedoch zu schlechter Lesbarkeit und Wartbarkeit; werden gar mehrere Konfigurationsoptionen so implementiert, ist das Ergebnis eine "#ifdef Hölle".[7] Listing 1.1 zeigt ein reales Beispiel aus dem Betriebssystem eCos [eCo]. Obwohl wir es in diesem Beispiel nur vier Konfigurationsoptionen gibt, ist der Code bereits kaum noch zu verstehen. Die Implementierung von Konfigurierbarkeit durch bedingte Übersetzung skaliert nicht.

[7] Der Begriff "`#ifdef` Hölle" ist Hacker-Jargon. Seine erste dokumentierte Verwendung findet sich in einem Usenet-Beitrag von BRIAN HOOK vom 5. November 1993 in `comp.os.opengl`.

```
Cyg_Mutex::Cyg_Mutex() {
  CYG_REPORT_FUNCTION();
  locked      = false;
  owner       = NULL;
#if defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT) && \
  defined(CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DYNAMIC)
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_INHERIT
  protocol    = INHERIT;
#endif
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_CEILING
  protocol    = CEILING;
  ceiling     = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
#endif
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_NONE
  protocol    = NONE;
#endif
#else // not (DYNAMIC and DEFAULT defined)
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_CEILING
#ifdef CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY
  // if there is a default priority ceiling defined, use that to initialize
  // the ceiling.
  ceiling = CYGSEM_KERNEL_SYNCH_MUTEX_PRIORITY_INVERSION_PROTOCOL_DEFAULT_PRIORITY;
#else
  // Otherwise set it to zero.
  ceiling = 0;
#endif
#endif
#endif // DYNAMIC and DEFAULT defined
  CYG_REPORT_RETURN();
}
```

Listing 1.1: Konstruktor der Mutex-Klasse aus eCos.
Die gewählte Strategie zur Verhinderung von Prioritätsumkehreffekten wird in der Klasse `Cyg_Mutex` mit Hilfe des Präprozessors durchgesetzt. Das Ergebnis ist eine "`#ifdef` Hölle".

Ungeachtet dieser Problematik steigen die Anforderungen an die Konfigurierbarkeit von eingebetteten Betriebssystemen weiter. Ein gutes Beispiel ist der neue AUTOSAR OS Betriebssystemstandard [AUT06b], der Konfigurierbarkeit auch im Hinblick auf die verwendeten Strategien zur zeitlichen und räumlichen Isolation verlangt. Die dafür erforderlichen Varianten mit einer einzigen Kernimplementierung abzudecken, ist eine große Herausforderung. Entscheidungen über derart fundamentale Systemstrategien (wie die Entscheidung ob und wie Adressraumgrenzen durchgesetzt werden sollen) werden üblicherweise in einer frühen Phase der Betriebssystementwicklung getroffen. Sie führen zu den grundlegenden, "architekturellen" Entwurfsentscheidungen, die viele weitere Bausteine der Systemimplementierung beeinflussen.

Benötigt wird ein besserer Ansatz für die Implementierung von Konfigurierbarkeit in Systemsoftware-Produktlinien.

Die aspektorientierte Programmierung (AOP) [KLM+97] ist eine vielversprechende Möglichkeit. AOP bietet zusätzliche Sprachmittel für eine feingranulare Trennung der Belange, was einen Ausweg aus der "`#ifdef` Hölle" sein könnte. Eine besondere Stärke

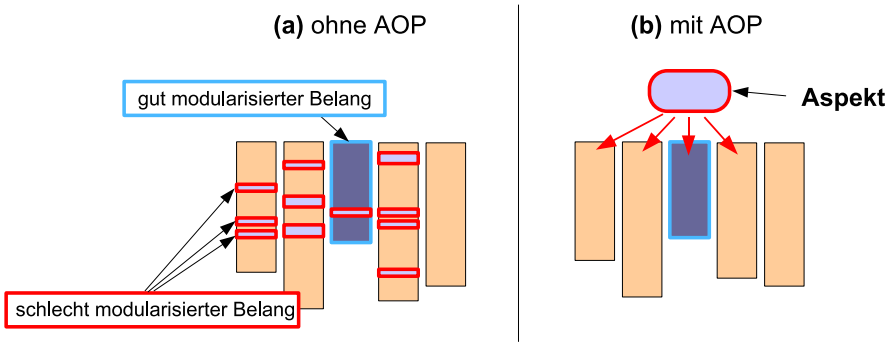


Abbildung 1.1.: Implementierung eines "querschneidenden Belangs" mit und ohne AOP.
Die Säulen stehen für Quelltextartefakte (z. B. Klassen); ein Belang gilt as gut modularisiert, wenn er eigenständig in dedizierten Quelltextartefakten gekapselt wurde. **(a)** Ohne AOP ist die Implementierung "querschneidender Belange" verteilt über die Implementierung der anderen Belange. **(b)** Mit AOP kann die Implementierung "querschneidender Belange" in eigene Quelltextartefakte, genannt *Aspekte*, separiert werden.

von AOP ist dabei die Separierung von "querschneidenden Belange": das sind Belange, die sich ansonsten über die Implementierung anderer Belange verteilen (Abbildung 1.1). Mit der Hilfe von AOP könnte es deshalb möglich sein, auch grundlegende Betriebssystemstrategien als konfigurierbare Merkmale bereit zu stellen.

Zweck der Arbeit

Diese Arbeit untersucht die Zweckmäßigkeit von AOP als grundlegenden Mechanismus zur Implementierung von Konfigurierbarkeit in Betriebssystem-Produktlinen für ressourcenbeschränkte eingebettete Systeme. Das Ziel ist dabei zu zeigen, dass die gezielte Verwendung von AOP zu einer deutlichen Verbesserung bei der Implementierung von Konfigurierbarkeit in Betriebssystemen führt, ohne dass dafür Nachteile bei den Hardwarekosten zu befürchten sind.

Ein positiver Einfluss von AOP auf die Wartbarkeit und Erweiterbarkeit von Betriebssystemcode konnte bereits anhand von Beispielen mit FreeBSD [CK03] und Linux gezeigt werden [ÅLS+03, FGCW05]. Eine vorangegangene Dissertation [Spi02] und einige Workshop-Papiere [MSGSP02, SL04] aus dem Umfeld der eigenen Arbeitsgruppe konnten außerdem Verbesserungen durch AOP bei der Konfigurierbarkeit der PURE Betriebssystemfamilie [BGP+99b] aufzeigen.

Die bisherigen Ergebnisse sind vielversprechend. Nichtsdestotrotz konnte damit die Frage, ob die Verwendung von AOP für die Implementierung von Konfigurierbarkeit in Systemsoftware *in der Breite* Vorteile unter dem Strich erbringt, noch nicht vollständig beantwortet werden:

1. Die vorangegangenen Arbeiten konzentrierten sich auf *qualitative Resultate* bei der Verwendung von AOP, d. h. auf den damit erzielbaren Einfluss auf Wartbarkeit

und Konfigurierbarkeit. Was bislang fehlt, gerade auch angesichts der angedachten Verwendung in ressourcenbeschränkten Systemen, ist eine umfassende Analyse der *quantitativen Auswirkungen* von AOP.

Fragen: Welche AOP-Sprachmerkmale führen zu Mehrkosten im Bezug auf Laufzeit und Speicherbedarf? Ist es möglich, diese Kosten zu reduzieren? Ist die Ausdrucksstärke einer mächtigen AOP-Sprache ein bezahlbarer Luxus bei der Softwareentwicklung für ressourcenbeschränkte eingebettete Systeme?

2. In den vorangegangenen Arbeiten wurde AOP überwiegend als "letzter Strohalm" zur Implementierung von Konfigurierbarkeit verwendet, die auf anderem Wege nicht erreichbar schien. Immer noch unbeantwortet ist die Frage, wie sich AOP als Implementierungstechnik für Konfigurierbarkeit *generell* im Vergleich zu den bekannten Ansätzen schlägt.

   Fragen: Ist mit AOP eine mindestens gleichwertige oder sogar bessere Effizient und Flexibilität erreichbar? Was sind gute Vorgehensweisen für die Implementierung von Konfigurierbarkeit mit Aspekten?

3. In den vorangegangenen Arbeiten wurde AOP ausschließlich im Nachhinein zur Extraktion von Belangen aus bestehenden Systemen verwendet. Weiterhin unbeantwortet ist die Frage, wie sich die Verwendung von AOP auswirkt, wenn man ein Betriebssystem von Anfang an unter der Berücksichtigung von Aspekttechniken entwirft und entwickelt.

   Fragen: Was sind gute Entwurfsregeln für den aspektorientierten Betriebssystembau? Welche Vorteile lassen sich erreichen? Wird es durch die Verwendung von Aspekttechniken möglich, selbst grundlegende, architekturelle Betriebssystemstrategien als konfigurierbare Merkmale bereit zu stellen?

Diese Arbeit baut auf den vorhandenen Arbeiten auf, bietet Antworten auf die genannten Fragen und erweitert den Stand der Kunst durch *Aspektgewahrheit bei der Entwicklung konfigurierbarer Systemsoftware.*

Titel und Ziele

Diese Arbeit hat den Titel "Aspektgewahrheit bei der Entwicklung konfigurierbarer Systemsoftware". *Gewahrheit* (englisch *awareness*) ist die Substantivierung des Adjektives *gewahr* (*aware*), was soviel bedeutet wie *"having knowledge or perception of a situation or fact"* [McK05]. Im *Merriam-Webster Online Dictionary*[8] finden wir außerdem: *"*AWARE *[...] mean[s] having knowledge of something [...]* AWARE *implies viligance in observing or alertness in drawing inferences from what one experiences*."

[8] http://www.merriam-webster.com/dictionary/aware

Demnach hat *Gewahrheit* zwei wesentliche Konnotationen: (hartes, faktenbasiertes) *Wissen* (*knowledge*) und (eher weiches, erfahrungsbasiertes) *Bewusstsein* (*perception*). Ziel dieser Arbeit ist es, *Wissen* und *Bewusstsein* bezüglich der Besonderheiten von AOP für die Implementierung von Konfigurierbarkeit in ressourcenbeschränkten Systemen zu erweitern. Dieses erfordert Arbeiten auf drei verschiedenen Ebenen, der *Sprachebene*, der *Implementierungsebene* und der *Entwurfsebene*:

Sprachebene: Auf der *Sprachebene* soll gezeigt werden, dass durch einen zielgerichteten Entwurf der Aspektsprache, hohe Ausdrucksstärke in Verbindung mit Kostenneutralität erreichbar ist. Dazu soll im Detail untersucht werden, welche AOP-Sprachmerkmale welche Kosten verursachen und Möglichkeiten aufgezeigt werden, diese zu verbessern.

Implementierungsebene: Aufbauend auf diesen Ergebnissen soll auf der *Implementierungsebene* gezeigt werden, dass AOP qualitativ *und* quantitativ den Vergleich mit dem Stand der Kunst besteht. Dabei soll im Detail untersucht werden, welche Faktoren eine Konfigurierung durch Aspekte begünstigen oder erschweren.

Entwurfsebene: Aufbauend auf diesen Ergebnissen soll schließlich auf der *Entwurfsebene* gezeigt werden, dass durch eine konsequente Verwendung von AOP sogar grundlegende Architektureigenschaften als konfigurierbare Merkmale implementiert werden können. Dabei sollen Vorgehensweisen und Regeln für die *aspektgewahre* Entwicklung von Betriebssystemen aufgestellt werden.

Aufbau der Arbeit

Es folgt ein kurzer Überblick über den weiteren Aufbau der Arbeit:

Kapitel 2: *Hintergrund, Kontext und Stand der Kunst* (Seiten 9–45)
Drei größere Themenbereiche sind relevant für diese Arbeit: *Systemsoftware für eingebettete Systeme, Software-Produktlinien* und *Aspektorientierte Programmierung*. Das zweite Kapitel führt den Leser in diese Themenbereiche ein und diskutiert den aktuellen Stand der Kunst.

Kapitel 3: *Problemanalyse und Forschungsansatz* (Seiten 47–66)
Stand der Kunst für die kostenneutrale Durchsetzung feingranularer Konfigurierbarkeit im Code ist die Verwendung des Präprozessors. Dieser Ansatz skaliert jedoch nicht – während gleichzeitig die Anforderungen an Betriebssystem-Produktlinien steigen. Im dritten Kapitel werden die Probleme im Detail analysiert (anhand von Beispielen aus eCos und AUTOSAR OS), die Forschungsvermutung formuliert (dass die Situation durch AOP erheblich verbessert werden kann) und der Forschungsansatz für eine diesbezügliche Evaluation von AOP vorgestellt.

Kapitel 4: *Sprachebene: Evaluierung und Verbesserung von AspectC++* (Seiten 67–98)
Die Evaluation von AOP beginnt auf der *Sprachebene*. Im vierten Kapitel werden

die Sprachmittel von AOP im generellen und von AspectC++ im Speziellen einer detaillierten Untersuchung unterworfen, um deren Kosten und Nutzen für die Implementierung von Konfigurierbarkeit zu verstehen. Im Rahmen dieser Arbeit sind dabei außerdem umfangreiche Verbesserungen an AspectC++ entstanden, die ebenfalls kurz vorgestellt werden.

Kapitel 5: *Implementierungsebene: Konfigurierbarkeit durch AOP in der Praxis* (Seiten 99–120)
Fokus des fünften Kapitels ist die *Implementierungsebene*. In diesem Kapitel wird AOP in die Praxis gebracht, indem einige der Belange aus eCos, die bislang mit Hilfe des Präprozessors konfiguriert wurden, in Aspekte extrahiert werden. Das Ziel ist zu verstehen, ob und unter welchen Bedingungen die vermuteten Vorteile von AOP tatsächlich zum Tragen kommen, wenn AOP in der Breite bei der Implementierung von Konfigurierbarkeit in Systemsoftware für eingebettete Systeme eingesetzt wird.

Kapitel 6: *Entwurfsebene: Aspektgewahre Betriebssystementwicklung* (Seiten 121–169)
Im sechsten Kapitel geht es dann um die *Entwurfsebene*. Der Ansatz der *aspektgewahren* Betriebssystementwicklung wird anhand der CiAO-Betriebssystemfamilie demonstriert. CiAO, die Abkürzung steht für CiAO ist Aspekt-Orientiert, ist eine hochkonfigurierbare Betriebssystem-Produktline, die im Rahmen dieser Arbeit entstanden ist. Dabei kamen von Beginn an Aspekttechniken zum Einsatz, wodurch CiAO im Ergebnis sowohl die Konfigurierbarkeit grundlegender architektureller Eigenschaften als auch außerordentliche Granularität und Variabilität bietet.

Kapitel 7: *Zusammenfassung und Ausblick* (Seiten 171–175)
Im siebten Kapitel wird die Arbeit schließlich zusammengefasst und es werden Ideen für weiterführende Arbeiten angesprochen.

An die sieben Hauptkapitel schließen sich noch zwei Anhänge an, die als relevant erachtetes begleitendes Material beinhalten:

Anhang A: *AspectC++* (Seiten 177–198)
Der erste Anhang enthält weiterführende Informationen zu AspectC++ und den zugehörigen Werkzeugen. Das Kapitel umfasst eine kurze Spracheinführung, Anwendungsbeispiele und eine detaillierte Analyse der verbesserten Codegenerierung, die im Rahmen dieser Arbeit entstanden ist.

Anhang B: *Fallstudie "Wetterstation"* (Seiten 199–218)
Im zweiten Anhang wird eine weitere Fallstudie präsentiert. In der "Wetterstation"-Fallstudie werden AOP und OOP bezüglich ihrer Eignung für die Entwicklung von Software-Produktlinien für kleinste, tief eingebettete Systeme qualitativ und quantitativ verglichen.

Typographische Konventionen

Die Einführung eines neuen Begriffs sowie die Akzentuierung eines wichtigen Ergebnisses wird durch **Fettdruck** aufgezeigt. In einigen Fällen wird ein Begriff verwendet, jedoch erst im folgenden Absatz formal eingeführt. In diesen Fällen wird der Begriff zunächst *kursiv* gesetzt. Kursive Schrift ist darüber hinaus das generell verwendete Stilmittel für *Hervorhebungen*. Programmiersprachliche Bezeichner werden in der Schriftart `Schreibmaschine` gesetzt; Bezeichner, die sich auf konzeptionelle Merkmale beziehen, hingegen in serifenloser Schrift. Die Namen von Personen, Werkzeugen und Kommandozeilenprogramme, wie z.B. Übersetzer und Binder, werden als KAPITÄLCHEN gesetzt.

Zusammenfassung und Ausblick

In dieser Arbeit wurde die Eignung von AOP als grundlegender Mechanismus für die Implementierung von Konfigurierbarkeit in Betriebssystemproduktlinien für die Domäne der eingebetteten Systeme untersucht. Das Ziel war zu zeigen, dass eine gezielte und breite Anwendung von AOP den Stand der Kunst bei der Implementierung von Konfigurierbarkeit erheblich verbessern kann, ohne dabei Nachteile im Hinblick auf Hardwarekosten in Kauf nehmen zu müssen.

Zusammenfassung

Zwei grundsätzliche Problemfelder motivierten die Untersuchung von AOP für die Implementierung von Konfigurierbarkeit in Betriebssystem-Produktlinien für eingebettete Systeme:

Problem 1: Die bedingte Übersetzung, der aktuelle Stand der Kunst für die Implementierung feingranularer Konfigurierbarkeit, führt zur Vermischung anstelle der angestrebten Trennung der Belange. Das Verfahren skaliert nicht, führt über kurz oder lang in die "`#ifdef` Hölle" und ist bereits an seine Grenzen gestoßen.

Problem 2: Die heute verfügbaren konfigurierbaren Betriebssysteme sind trotz allem noch nicht konfigurierbar genug. Sie bieten keine Konfigurierung architektureller Strategien. Zwar sind diese Strategien für die Anwendung transparent, sie beeinflussen jedoch wichtige nichtfunktionale Belange. Konsequenterweise sollten sie als konfigurierbare Merkmale ausgelegt sein.

Beide Problemstellungen sind im Grunde durch Querschnittsbelange verursacht, weshalb die aspektorientierte Programmierung als vielversprechender Ansatz im Hinblick auf eine Besserung angesehen wurde. Es war jedoch vollkommen unklar, ob AOP im Hinblick auf Ausdrucksstärke (qualitativ) und Ressourceneffizienz (quantitativ) gegen die bisherigen Ansätze bestehen kann. Ferner war offen, ob es auf diese Weise tatsächlich möglich wird, architekturelle Strategien als konfigurierbare Merkmale zu implementieren.

Der gewählte Forschungsansatz zur Untersuchung dieser Fragestellung basierte auf der Analyse und Evaluation der Eignung von AOP auf drei unterschiedlichen Abstraktionsebenen: *Sprachebene*, *Implementierungsebene* und *Entwurfsebene*.

Ergebnisse der Sprachebene: AOP verursacht keine inheränten Zusatzkosten, die eine Verwendung in der Domäne der ressourcebeschränkten eingebetteten Systeme *per se* in Frage stellen würden. Unbedingt erforderlich sind jedoch (1) die Bindung von Advice und Verknüpfungspunkten zur Übersetzungszeit und (2) Sprachkonzepte für kostenneutrale, statische Advice-Polymorphie.

Ergebnisse der Implementierungsebene: AOP verursacht ebenfalls keine Zusatzkosten bei Verwendung in echter Systemsoftware, selbst wenn hunderte von Verknüpfungspunkten betroffen sind. Anhand der "eCos" Studie konnte gezeigt werden, dass AOP im Hinblick auf Kosten und Ausdrucksstärke den Vergleich mit bedingter Übersetzung besteht und gleichzeitig zu einer deutlich besseren Trennung der Belange im Code führt. **Insgesamt konnte gezeigt werden, dass der Einsatz von AOP für die Lösung von Problem 1 zweckmäßig ist.**

Auf der anderen Seite musste jedoch auch festgestellt werden, dass die einfache Separation der Implementierung von Architektureigenschaften in Aspekte die Konfigurierbarkeit nicht signifikant verbessert.

Ergebnisse der Entwurfsebene: Für die Konfigurierbarkeit auch architektureller Strategien ist es hingegen erforderlich, dass die Systemsoftware von vorneherein im Hinblick auf die Trennung von Strategien und Mechanismen entworfen und implementiert wird. Die Entwurfsregeln für den *aspektgewahren Betriebssystembau* ermöglichen diese durchgängige Trennung von Strategien und Mechanismen bei der Implementierung eines Betriebssystemkerns. Damit lassen sich auch grundlegende Architektureigenschaften konfigurierbar gestalten. Dieses konnte am Beispiel des AUTOSAR-OS Standards und der CiAO Betriebssystem-Produktlinie gezeigt werden. CiAO vereint ausgesprochen konkurrenzfähige Laufzeit- und Speicherkosten mit der Möglichkeit, auch architekturelle Eigenschaften zu konfigurieren. **Damit konnte gezeigt werden, dass der Einsatz von AOP für die Lösung von Problem 2 zweckmäßig ist.**

Wissenschaftlicher Beitrag

Diese Arbeit erweitert den Stand der Kunst in mehreren Bereichen. Die folgende Auflistung beschreibt noch einmal die wesentlichsten Beiträge:

- Das Konzept des **generic advice**, welches Advice-Polymorphie in überwiegend statisch getypten Sprachen wie AspectC++ erst ermöglicht. Nur durch *generic advice* war es letztlich möglich, Advice-Quantifizierung bei der Implementierung von architekturellen Strategien zu verwenden.
- Die umfassende **Analyse und teilweise Verminderung der Kosten** der AspectC++-Sprachmerkmale und die daraus resultierenden **Handlungsempfehlungen** für den Einsatz von AOP in ressourcenbeschränkten eingebetteten Systemen. Entwickler derartiger Systeme können sich damit sicher sein, dass die Trennung der Belange durch AOP keine zusätzlichen Kosten verursacht.

- Der Beweis, dass mit AOP eine **deutlich bessere Trennung der Belange** in der Implementierung konfigurierbarer Systemsoftware erreichbar ist. Die Code-Qualität aktueller Systemsoftware, wie dem eCos Betriebssystem, kann deutlich verbessert werden ohne dass dieses zu zusätzlichen Kosten auf der Hardwareseite führt.
- Die Methode der **aspektgewahren Betriebssystementwicklung**, welche die AOP-Grundideen von *Obliviousness, Quantification,* und *Awareness* neu austariert und erfolgreich bei der Entwicklung der CiAO-Betriebssystemfamilie zum Einsatz kam.
- Der Beweis, dass es mit aspektgewahrer Betriebssystementwicklung möglich wird, selbst **grundlegende architekturelle Eigenschaften konfigurierbar zu gestalten**.
- Die **CiAO-Betriebssystemfamile**, welche das erste Betriebssystem überhaupt ist, dass von vorneherein mit AOP-Konzepten entworfen und entwickelt wurde. CiAO kann dabei mit einer konkurrenzfähigen Implementierung des AUTOSAR-OS Standards aufwarten.

Weiterführende Ideen

Die ideale Aspektsprache für Systemsoftwareentwicklung. Obgleich die Implementierung von CiAO und AspeCos in AspectC++ erfolgreich war, zeigte sich dabei auch, dass die Verwendung von AOP bei der Implementierung gerade auch hardwarenaher Systembelange Herausforderungen und Anforderungen mit sich bringt, die bislang nicht optimal durch die Sprache unterstützt werden. Grundsätzlich erstrebenswert für die Anwendung in dieser Domäne wären Spracherweiterung, die eine deutlich bessere Kontrolle über den generierten Code ermöglichen. Es sollte beispielsweise möglich sein, übersetzerspezifische Attribute für die Kontrolle von Aufrufkonventionen, Funktionseinbettung und ähnlichem auch auf Advice anzuwenden. Ebenfalls wünschenswert wären Sprachmittel, um die Applizierung von Advice an fragilen Stellen explizit einschränken zu können. Eine weitere wichtige Verbesserung wäre Sprachunterstützung für explizite Verknüpfungspunkte, beispielsweise durch die Möglichkeit Programmelemente, Code-Blöcke oder sogar einzelne Anweisungen durch einen Annotations-Mechanismus zu markieren.

Die Optimierung nichtfunktionaler Systemeigenschaften. In der Problemanalyse wurde kurz das Verhältnis zwischen nichtfunktionalen und architekturellen Eigenschaften angesprochen (siehe Abschnitt 3.1.2). Da nichtfunktionale Eigenschaften überwiegend emergent sind, können sie nur indirekt beeinflusst werden. Die Motivation für konfigurierbare Architektureigenschaften geht letztlich auf die Erfahrung zurück, dass gerade diese Eigenschaften einen hohen Einfluss auf das nichtfunktionale Verhalten des Gesamtsystems haben. Demnach sind konfigurierbare Architektureigenschaften ideale Stellschrauben für die Optimierung eines Systems bezüglich bestimmter nichtfunktionaler Eigenschaften, wie zum Beispiel Latenz, Performance, oder Speicherbedarf.

Mit der konfigurierbaren Architektur bietet CiAO nun gute Vorraussetzungen für derartige Optimierungen. Offen ist jedoch weiterhin, wie man die *beste* (oder wenigstens eine

hinreichend gute) Konfiguration für eine bestimmte Anwendung und die für sie charakteristische Auslastung findet. Der Ansatz alle möglichen Varianten, welche die funktionale Spezifikation erfüllen, manuell oder automatisch zu generieren und durchzutesten, dürfte schnell an Grenzen stoßen – die Anzahl der möglichen Varianten wächst exponentiell mit der Anzahl konfigurierbarer architektureller Eigenschaften. Benötigt werden also Heuristiken und Werkzeuge für eine sinnvolle Vorabbewertung der möglichen Merkmalsselektionen, um so die Anzahl der zu untersuchenden Varianten auf ein praktisch handhabbares Maß zu beschränken.