Aspect-Oriented Design and Implementation of an AUTOSAR-Like Operating System Kernel

Diploma Thesis in Computer Sciences

by

Wanja Hofer

born 04/29/1983 at Ludwigshafen/Rhein

Department of Computer Sciences 4
Friedrich-Alexander University Erlangen-Nuremberg

Advisors:	Dipl.-Inf. Daniel Lohmann Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat
Start of work:	05/02/2007
End of work:	10/30/2007

Abstract

In the automotive industry, the consolidation of the software features on less but more powerful microcontrollers has led to the advent of a more sophisticated type of embedded operating system, which is targeted by the AUTOSAR OS standard. Since this specification prescribes the newly introduced protection features to be configurable (though on a coarse level), an implementation is to be designed as a product family.

CiAO aims to face that challenge by making use of concepts of aspect-oriented programming (AOP). After identifying the concerns present in an AUTOSAR-like operating system kernel, these concerns are modeled in an aspect-oriented kernel design. Thereby, even highly cross-cutting concerns like kernel synchronization or fault isolation features can be kept encapsulated, configurable, and evolvable.

An evaluation of the approach shows that the CiAO kernel is highly scalable with respect to both memory footprint and performance; a comparison to a commercial implementation of the standard indicates that the aspect-oriented implementation does not induce a significant overhead per se. Additionally, the systems programmer benefits from several advantages that the aspect-oriented design bears: Many requirements stated in the AUTOSAR standard can be formulated in a natural and encapsulated way, facilitating the understanding and maintenance of the kernel design and implementation.

Kurzzusammenfassung

In der Automobilindustrie hat die Konsolidierung der softwarebasierten Funktionen auf weniger aber dafür mächtigere Mikrocontroller zu einer neuen Art von eingebettetem Betriebssystem geführt, welche mit der AUTOSAR-OS-Spezifikation standardisiert wurde. Da dieser Standard vorschreibt, dass die neu eingeführten Schutzmechanismen konfigurierbar angeboten werden müssen (wenn auch nur auf grobem Niveau), muss eine Implementierung desselben als Produktfamilie entworfen werden.

CiAO bewältigt diese Herausforderung durch gezielte Anwendung von Konzepten der aspektorientierten Programmierung (AOP). Nach der Identifikation der Belange eines AUTOSAR-orientierten Betriebssystemkerns werden diese in einem aspektorientierten Kernentwurf umgesetzt. Dadurch können sogar stark querschneidende Belange wie die Kernsynchronisation oder Fehlerisolierungseigenschaften gekapselt werden, was sie einzeln konfigurierbar und einfach weiterentwickelbar macht.

Eine Bewertung des Ansatzes zeigt, dass der CiAO-Kern gut skaliert, sowohl in Bezug auf den Speicherverbrauch, als auch auf die Ausführungsgeschwindigkeit. Ein Vergleich mit einer kommerziellen Implementierung des Standards weist darauf hin, dass die aspektorientierte Implementierung nicht zwangsläufig zu signifikanten Einbußen führt. Zusätzlich kann der Systemprogrammierer von einer Reihe von Vorteilen profitieren, welche der aspektorientierte Entwurf mit sich bringt: Viele Anforderungen des AUTOSAR-Standards können in natürlicher Form und gekapselt formuliert werden, wodurch das Verständnis und die Wartung des Kernentwurfs und dessen Implementierung vereinfacht werden.

Contents

CHAPTER 1

Introduction

Microcontroller Consolidation

Embedded systems in the automotive industry are undergoing a fundamental change at the moment: Different pieces of functionality formerly deployed on several low-end microcontrollers are now joined on few but more powerful platforms for reasons of cost reduction. This step of *microcontroller consolidation* also induces a focus transition of the underlying system software, which now needs to offer additional functionality protecting the different applications from each other. Since automotive manufacturers often purchase the pieces of application software from several competing external vendors, robust protection facilities of the operating system and the hardware are crucial to reduce responsibility and liability conflicts.

AUTOSAR OS

This additional need for *integration* of distinct applications was the driving factor for the evolution of the OSEK operating system standard [OSE05a] to the AUTOSAR OS standard [AUT06b], which additionally features powerful protection facilities, depending on the features of the configured *scalability class*.

Separation of Concerns

Implementations of the AUTOSAR OS standard are far from trivial; especially the newly requested protection mechanisms are highly cross-cutting the system. Hence, a clear *separation of concerns* in the operating system design as well as in its implementation is needed to allow for better maintenance, evolvability, and configurability of an AUTOSAR operating system.

1.1 CiAO

Architectural Properties

CiAO (<u>C</u>iAO <u>i</u>s <u>A</u>spect-<u>O</u>riented) is an embedded research operating system that aims at the configurability of certain *architectural properties* of the system [LS03]. Architectural properties are those that are internal to the system and that are not reflected in the system API; examples include the deployed interrupt synchronization mechanism [LSSSP07], or the kind of protection between system and application components [LSH$^+$07].

Non-Functional Properties

With respect to the application, architectural properties do not affect the functionality of the system but only influence its *non-functional properties* like, for instance, its performance, safety characteristics, or memory consumption.

Hence, many non-functional properties are said to be *emergent* properties, which have no direct representation in the software system code [LSSP05].

AOP

Since architectural properties are inherently cross-cutting, CiAO uses techniques of *aspect-oriented programming* (AOP) to encapsulate distinct configurations of those properties. Furthermore, it systematically deploys AOP design patterns in order to achieve a better separation of concerns (SoC) in general, even with those concerns that are not highly cross-cutting the rest of the system.

Aspect Awareness

Several studies have been conducted attempting to refactor existing operating system code and extract configurable architectural properties into aspect entities [LST^{+}06, CK03, CKFS01]. This proved to be a very tedious task since the architecture of a piece of system software is normally hard-coded and tangled all over the source code, effectively hindering ex-post modularization. Thus, CiAO is designed keeping aspects in mind from the very beginning; its design is *aspect-aware*, in contrast to the traditional AOP claim of obliviousness of the affected source code. An aspect-aware kernel design provides static, explicit join points in order to keep the woven and compiled system code as efficient as possible.

Secondary Goals

Secondary goals of the CiAO project include system *efficiency*, *portability* to several microcontroller platforms, and *variability* and fine *granularity* of configured features. Object-oriented design, for instance, is only applied where its implementation mapping is known to be run-time and memory efficient. Easy portability is reached by designing a hardware access layer, where a common subset can then be implemented for different platforms.

Software Product Line

The configurability of variable features of distinct granularities is enforced by the understanding of CiAO as a *software product line*. The common features are clearly separated from the varying ones denoting the different product line configurations, which themselves correspond to distinct implementation entities (e.g., an object, or an aspect). This way, re-use of components is facilitated and resources are preserved because features that are not needed in an implementation variant can easily be excluded.

■ 1.2 OSEK and AUTOSAR

OSEK

OSEK[1] is currently the leading industry standard for operating systems in the automotive area. The specification of the event-triggered OSEK variant (OSEK OS) [OSE05a] defines a clear interface to the operating system, offering control flow abstractions (tasks), synchronization mechanisms (events), and interrupt processing primitives, amongst others. The advent of the OSEK standard led to the portability of OSEK applications between different hardware platforms and OSEK implementations from different vendors.

AUTOSAR Motivation

Due to the need to integrate several applications on one microcontroller, a comprehensive extension to the successful OSEK standard was worked on by the AUTOSAR[2] committee. The resulting *AUTOSAR OS standard* [AUT06b] provides additional support for the isolation of multi-source software compo-

[1]German abbreviation for "Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug", translating into "open systems and the corresponding interfaces for automotive electronics".

[2]Abbreviation for "automotive open system architecture".

nents at run time, and further support to confine fault propagation. These new protection facilities are named *memory protection*, *timing protection*, and *service protection*. The standard also arranges for *scalability classes*, which include defined subsets of the configurable features and protection mechanisms only, therefore only providing for a very small variability, though.

AUTOSAR Architectural Properties

The newly introduced protection features are typical representatives of configurable architectural properties. They do not affect the functionality of the system that is visible in its API, but nevertheless have a big impact on non-functional properties like safety or the resulting performance and therefore on the applicability of the whole system in a specific domain (e.g., in safety-critical parts of automotive electronics).

Suitability for CiAO

Since the configurability of architectural properties is also the main goal of the CiAO operating system, the AUTOSAR OS standard is perfectly suited to look for inspiration and to provide the user interface to CiAO where applicable. This way, the resulting design and implementation can be compared to other (industrial) implementations in an evaluation step. Furthermore, many optional parts of the AUTOSAR specification read like they could be encapsulated in aspect entities in order to reach better separation of (configurable) concerns.

1.3 Goals of This Thesis

CiAO Kernel

The goal of this thesis is to develop the kernel of the CiAO operating system. For the reasons stated in Section 1.2, the kernel is supposed to offer OS abstractions oriented at the AUTOSAR standard where applicable. The AUTOSAR standard is to be seen rather as a guideline than a strict specification, especially when its requirements contradict the higher goals of CiAO (see Section 1.1).

Aspect Orientation

The kernel design, and especially its architectural properties, are to be evaluated with respect to the deployment of aspect-oriented patterns and implemented using AOP techniques where appropriate. Aspect awareness is to be promoted by careful design, enabling the kernel to be properly advised by aspects.

Aspect Traceability

A further facet to be investigated is the development of aspects in the different stages of the kernel development: Which aspects can be identified in the requirements and the specification, how are they mapped to design elements (e.g., objects, aspects, or linker scripts), and which implementation artifacts do they correspond to? This kind of exploration is named *aspect traceability*.

Configurability

Clear separation of concerns (using aspect-oriented design, amongst others) enables further configuration of variable and even cross-cutting features and policies. Where trade-off situations (e.g., run time vs. memory footprint) occur, the alternatives are to be examined and to be made configurable. Especially the error handling mechanisms should be separated in a fine-grained manner in order to be able to control the footprint of the resulting system to match hardware restrictions (e.g., the ROM size). This way, it is possible for a system integrator to monitor the occurrence of errors in the development phase and then only deploy those error handlers on the production systems that are more likely to be needed. Another contact point for the introduction of variability and therefore configurability are those features described in the AUTOSAR OS specification that are (still) ambiguous or not exactly specified and therefore to be interpreted in different ways.

1.4 Related Work

There is some work describing the synthesis of operating system kernels and aspect-oriented programming already published; the environments and aims of those projects are different from those of CiAO, though.

PURE PURE is an operating system family that aims to support even deeply embedded systems [BGP+99]. Its abstractions are designed in minimal extensions, providing for fine-grained configuration possibilities. However, it was originally designed in an object-oriented way, oblivious to AOP. Only after exploring the ability of AOP to modularize the cross-cutting concerns present in a PURE system, aspects were considered [SL04], and only implemented for selected concerns like interrupt synchronization [MSGSP02]. Therefore, PURE was not designed to be aspect-aware from the beginning as is CiAO.

TOSKANA The TOSKANA toolkit by Engel and Freisleben [EF05, EF06] enables the deployment of aspects into an OS kernel. The prototype is demonstrated using the NetBSD kernel; in general, the OS domain targeted for is the PC domain and not embedded systems. Furthermore, the toolkit *instruments* the kernel in order to be able to weave and unweave aspects *dynamically* at run time. The induced run-time and memory overhead is not tolerable in embedded computing; furthermore, the functionality of those systems is only needed to be configured *statically*. Finally, AOP is used as an ex-post mechanism to advise existent kernels; the obliviousness of the target kernel is an explicit design goal of TOSKANA.

Studies by Coady et al. As already noted in Section 1.1, several studies were conducted on how to apply AOP ex-post to existing kernels. Particularly, Coady et al. showed how to modularize pre-fetching in the FreeBSD operating system kernel [CKFS01], and sketched the design of an aspect-oriented page daemon and quota manager [CK03]. However, the aspect weaver proposed for this task, AspectC, does not have a functional implementation; the examples were hand-woven and therefore do not provide a cost evaluation. Furthermore, the re-factoring approach is fundamentally different to the one of aspect awareness in CiAO.

Aspect NesC Walton and Eide researched the applicability of AOP to operating systems deployed on very small sensor nodes, especially TinyOS [WE07]. Since TinyOS is written in the special-purpose, component-based language NesC, their newly developed language *Aspect NesC* targets only this C dialect. Furthermore, the pointcut functions to be provided by NesC differ from the traditional ones proposed by the AOP community and used in CiAO: The goal is to be able to express resource management aspects taking action when overflowing the stack, or when exceeding a task's deadline, for instance.

1.5 Outline of This Thesis

The remainder of this document is organized as follows.

- Chapter 2 provides the reader with the background knowledge introducing the basic terms and concepts necessary to understand the thesis.

- The different concerns present in an AUTOSAR-like OS are introduced and analyzed in Chapter 3.
- Chapter 4 presents the design of those concerns in CiAO and common aspect-oriented design and implementation schemes that were deployed.
- Both the design and the implementation of the CiAO kernel are evaluated in Chapter 5; this comprises both the general footprint of the system and the evaluation of the approach using aspect orientation.
- Chapter 6 provides a summary of the results, combined with an outlook of the ideas to be tackled in future work on CiAO.

CHAPTER 2

Background

The following chapter provides the basic background knowledge needed to understand this thesis; this includes an overview of the OSEK and AUTOSAR standards (see Section 2.1) and an introduction to the concepts of aspect-oriented programming (see Section 2.2). Moreover, references are given for further reading where necessary.

2.1 Overview of OSEK/AUTOSAR

This section gives a short introduction to the terminology used in the OSEK and AUTOSAR OS specifications. It briefly describes basic concepts, offered services, and the configuration process of a standard system.

2.1.1 From OSEK to AUTOSAR

OSEK Specifications

The OSEK committee was founded in 1993 by German automotive manufacturers and suppliers in order to develop a standardized interface to the system software of the control units in an automobile. This was mainly because it was recognized that the integration of the heterogeneous software products by different vendors costs a significant amount of time and money. The resulting specifications include a standard for an event-triggered operating system [OSE05a] (OSEK OS), a time-triggered operating system [OSE01] (OSEKtime OS), network management [OSE04c] (OSEK NM), and the communication between the microcontrollers in a car [OSE04b] (OSEK COM). Only the OSEK OS standard is relevant for this thesis, though. Furthermore, helper specifications for the configuration language of an OSEK system [OSE04a] (OSEK implementation language, OIL), and for an interface to the debugger [OSE05b, OSE05c] (OSEK run-time interface, ORTI) are provided. Some parts of OSEK, including OSEK OS, are now also an ISO standard (ISO 17356).

AUTOSAR Specifications

The need for further isolation between different applications running on a single microcontroller system has led to the formation of a successor committee: AUTOSAR. Now being a real international organization including automotive manufacturers and suppliers from Asia and the U.S., it took the comprehensive input as a motivation to develop improved versions of the OSEK standards as well as new ones. These new specifications include documents on the software engineering process that is to be applied [AUT06a] and standards for new abstractions offered to the applications, like the AUTOSAR run-time environment [AUT06c] (RTE).

AUTOSAR OS The specification relevant to this thesis is the core operating system standard: AUTOSAR OS [AUT06b]. It is completely based on the OSEK OS standard and therefore backwards compatible; only minor OSEK elements are excluded in some scalability classes of AUTOSAR OS. The main extension of AUTOSAR comprises the protection facilities to provide for isolation between distinct applications, where protection is understood in two different dimensions: spatial (i.e., memory protection), and temporal (i.e., timing protection). OSEK had already introduced an *extended mode*, which checks for additional error classes compared to the *standard mode* that is to be used for production systems; yet this mode did not go far enough for the new demands. That is why AUTOSAR OS features a more comprehensive facility named *service protection*. The rest of the document merely uses the term “AUTOSAR OS” or just “AUTOSAR”, even when parts of it also apply to the OSEK OS standard.

2.1.2 AUTOSAR OS Abstractions

Tasks There are basically two control flow abstractions defined in AUTOSAR: *tasks* and *interrupt service routines* (ISRs). Tasks have defined start functions that are executed whenever they are set running by the OS scheduler. Among the configurable features of a task are its priority and its preemptability, which are respected by the scheduler decisions.

Interrupt Service Routines ISRs themselves are separated into two categories: Category 2 ISRs are scheduled and synchronized by the OS, whereas the OS is *not* aware of ISRs of category 1, which are also specific to the hardware platform and the OS implementation. If synchronization of category 1 ISRs is needed, the user has to care about that by disabling the corresponding interrupt source or all interrupts manually. Due to the unawareness of the OS, ISRs of category 1 bear less overhead, though, and might therefore be better suited for reactions where a low latency is needed.

Resources The AUTOSAR OS object for synchronization of tasks is a *resource*. Whenever a task acquires a specific resource, the OS guarantees that no other task can acquire it; mutual exclusion is reached. The standard prescribes the implementation of an *OSEK priority ceiling protocol* to fulfil that guarantee, but other solutions are possible (e.g., a priority inheritance protocol).

Events Signalization between tasks is performed using *events*. Tasks can wait for events and be blocked by the scheduler until another task or ISR sets the event it is waiting for.

Counters and Alarms *Counters* can be implemented in software as well as in hardware. *Alarms* are the abstractions triggering a configured behavior (e.g., the activation of a task, or the setting of an event) when a counter reaches a specific value.

Hooks The standard also proposes the use of callback routines from the OS executing user-specific code on distinct occasions. Proposed points include the occurrence of an error (`ErrorHook`) or a protection violation (`ProtectionHook`), the start-up and shut-down of the system (`StartupHook` and `ShutdownHook`), and before and after the execution of a task (`PreTaskHook` and `PostTaskHook`, respectively).

Schedule Tables

A new AUTOSAR OS abstraction combining alarms with task activations are *schedule tables.* Similar to an alarm, a schedule table is bound to a hardware or software counter, but consists of multiple expiry points upon which the activation of a task is triggered. The tables can be set to bear one-shot or periodic behavior.

■ 2.1.3 AUTOSAR OS System Services

In the AUTOSAR interface, several system service groups can be distinguished to classify the 42 defined system services.

OS Control Services

The user is able to start-up the OS in different application modes (`StartOS ()`) depending on short run-time routines at boot time (e.g., the polling of a port level). This mode can be queried at run-time using `GetActiveApplicationMode ()` to write mode-dependent code. The operating system can be shut down in a controlled manner using `ShutdownOS ()`.

Task Services

All of the task management functions refer to the scheduler. They include services to put another task into the ready state (`ActivateTask ()`), to terminate the running task (`TerminateTask ()`), to do both at once (`ChainTask ()`), to query the running task's ID (`GetTaskID ()`), to query a task's state (`GetTaskState ()`), and to force a scheduling decision manually (`Schedule ()`; only makes sense when called from within a non-preemptable task).

Interrupt Services

Interrupt recognition can globally be disabled and enabled by using `DisableAllInterrupts ()` and `EnableAllInterrupts ()`. If nested calls are possible, the use of `SuspendAllInterrupts ()` and `ResumeAllInterrupts ()` is suggested, which save and restore the current recognition status, respectively. `SuspendOSInterrupts ()` and `ResumeOSInterrupts ()` deactivate and reactivate the recognition of ISRs of category 2 only. The ID of a running category 2 ISR can be queried with `GetISRID ()`, and the source triggering a specific ISR can be activated and deactivated through the use of `EnableInterruptSource ()` and `DisableInterruptSource ()`, respectively.

Resource Services

The services offered to control the use of resources only include one to acquire a resource (`GetResource ()`), and one to release a resource (`ReleaseResource ()`).

Event Services

The comprehensive services for event management include the notification of an event to a task (`SetEvent ()`), the clearing of a task's event mask (`ClearEvent ()`), and the querying of the current event mask of a task (`GetEvent ()`). Additionally, a potentially blocking service to wait for the occurrence of an event is provided (`WaitEvent ()`).

Alarm Services

OS functions for the management of alarms comprise services to query the characteristics of an alarm at run time (`GetAlarmBase ()`), and the number of ticks until it expires (`GetAlarm ()`). Alarms can be set at run time to expire at an absolute counter value (`SetAbsAlarm ()`) or one relative to the current value (`SetRelAlarm ()`). An alarm can also be aborted, using `CancelAlarm ()`. If the underlying counter is a software counter, it is advanced by `IncrementCounter ()`.

Schedule Table Services

Pre-defined schedule tables can be started using `StartScheduleTableRel ()` or `StartScheduleTableAbs ()`, at a relative or absolute tick value of the underlying counter, respectively. A running table can be stopped with `StopScheduleTable ()`. `NextScheduleTable ()` starts another schedule table after the length or period of a currently still running one. The state of a schedule table can be queried using `GetScheduleTableStatus ()`. Concerning synchronization, `SetScheduleTableAsync ()` sets a schedule table to be asynchronous manually, while `SyncScheduleTable ()` synchronizes a schedule table with a global time value.

OS Application Services

The running AUTOSAR OS application can be queried by `GetApplicationID ()`, and all its corresponding behavior can be terminated with `TerminateApplication ()`. The access rights to and the owner of distinct OS objects can be queried at run time using `CheckObjectAccess ()` and `CheckObjectOwnership ()`, respectively. The access rights to specific memory regions, however, can be found out with `CheckISRMemoryAccess ()` or `CheckTaskMemoryAccess ()`, depending on the type of control flow abstraction to be examined. A function exported by another (trusted) OS application has to be configured at compile time to be then be called at run time using `CallTrustedFunction ()`.

2.1.4 Configurability of AUTOSAR

Static Configuration

AUTOSAR OS is a statically configured operating system, that is, the supplied configuration is not alterable at run time. There are two parts that are meant to be configured: the settings of the applications, and the system settings. Both of them are configured in an OSEK implementation language file (OIL file) that is to be provided for every system; the latter settings are encapsulated in an artificial `OS` configuration object.

Application Configuration

Information about the applications running on an AUTOSAR system must be provided at compile time. This information comprises the number and properties of the needed OS objects. For instance, tasks need to be initialized with a priority and a maximum number of recorded activations, and ISRs need to be bound to an interrupt source. Other examples include the action taken on the expiry of each available alarm, or the binding of every OS object to an OS application.

System Configuration

There are also system-wide properties and settings that are to be configured before compiling the system. Those features are configurable because they bear an inherent safety–costs trade-off that is to be decided upon separately for each target system. Global properties defined in AUTOSAR are the support for the different sorts of hooks, or whether stack monitoring should be activated, for example.

Scalability Classes

Furthermore, the *scalability class* needs to be specified, implicating the protection facilities that are to be enabled. The protection types defined by AUTOSAR are the *memory protection*, *timing protection*, and *service protection*. The AUTOSAR standard does not allow for more fine-grained configuration of each of these protection facilities, neither does it allow arbitrary on/off configurations of these three types. For a detailed analysis of the configurable features, see Chapter 3.

2.2 Introduction to Aspect-Oriented Programming

Aspect-oriented programming (AOP) is a term firstly coined by Kiczales et al. [KLM+97] as the result of a research project at Xerox PARC. Section 2.2.1

introduces the concepts and terminology of AOP, whereas Section 2.2.2 presents the basics of AspectC++, the AOP language used to implement CiAO.

■ 2.2.1 AOP Concepts

Expressivity of Languages

A programming language or programming paradigm is often judged by its ability to express the thoughts of the programmer in an appropriate way. This was duly noted by Edsger W. Dijkstra, for instance [Dij72]:

> [...] 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!

Aspects

AOP postulates to provide programming languages with an additional expressivity through the introduction of the *aspect* concept, which allows to program the concerns of a system in a more natural and explicit form. The final executable code is produced by *weaving* the aspect code programs.

Cross-Cutting Concerns

A fundamental feature of the AOP decomposition technique is that it provides special means to be able to separate *cross-cutting concerns*. Two concerns are said to be cross-cutting if they affect each other at distinct static and dynamic places named *join points*. Therefore, with traditional decomposition mechanisms, the design and implementation of these concerns is either *scattered*, that is, spread out rather than localized, or *tangled*, that is, intertwined rather than separated, or both. AOP allows for the separation of even these cross-cutting conerns.

Advice Types

Aspects encapsulate several pieces of *advice*, which can be given in different forms. *Introduction advice* introduces members (i.e., methods or variables) to advised classes. Additionally, join points in the control flow of a piece of software can be advised to execute additional code *before*, *after*, or *around* (i.e., instead) the advised join points.

Pointcuts

The join points that an advice affects are declared in a descriptive language, the *pointcut expression language*. This language often features wildcard expressions to match several static join points, and *pointcut functions* to further filter the set of target join points both statically at compile time and dynamically at run time. These functions often include *calls* of functions, *execution* of functions, and the context of calls (named *within*), but additional pointcut functions may be offered.

Example

A simple example is an arbitrary (possibly complex and complicated) concern encapsulated in a class, and the concern of tracing its execution (e.g., for debugging or profiling purposes). These concerns cross-cut each other, since they are inherently dependent on each other (see also Figure 2.1 on the following page). A traditional implementation of these two concerns would result in scattering and tangling (see Figure 2.2 on the next page). In contrast, a formulation of the concerns in two aspects would result in a clear separation; the aspects would then be treated by an aspect weaver to produce the final code (see Figure 2.3 on the following page).

■ 2.2.2 AOP with AspectC++

AspectC++ Basics

An implementation of an AOP language extension and aspect weaver for C++

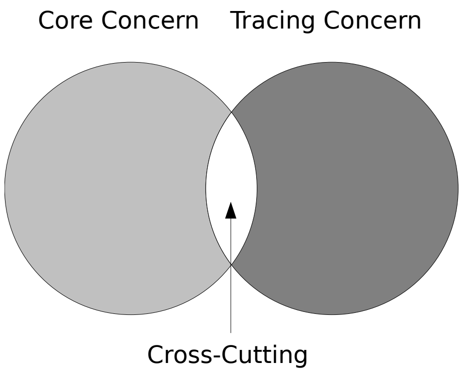

Figure 2.1: Two cross-cutting concerns.

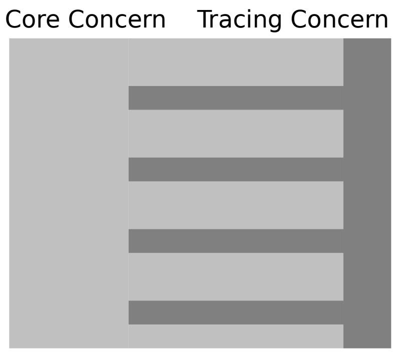

Figure 2.2: A both scattered and tangled implementation of two cross-cutting concerns.

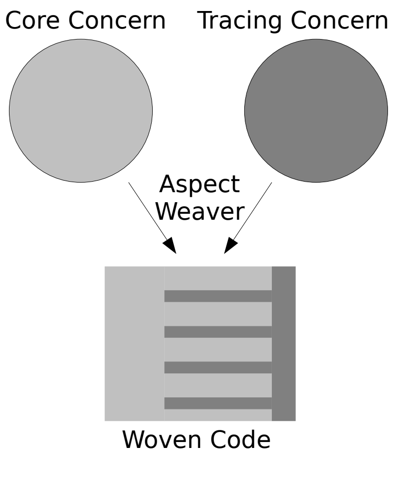

Figure 2.3: Two clearly separated concerns, implemented by aspects. The weaver combines the aspect code at the designated join points.

```
aspect Tracing {
    pointcut target () = "% ComplicatedClass::% (...)";

    advice execution (target ()) : around () {
        cout << "Entering " << tjp->signature () << endl;
        tjp->proceed ();
        cout << "Leaving " << tjp->signature () << endl;
    }
};
```

Figure 2.4: An example tracing aspect implemented in AspectC++.

is *AspectC++* [SL07][1]. It is based on a source-to-source transformation approach, that is, the AspectC++ sources are woven into standard C++ code, which can then be compiled with a standard-compliant C++ compiler to get an executable program.

Syntax Elements

Figure 2.4 shows the example discussed above, implemented in AspectC++. The `aspect` keyword begins an aspect construct, which is otherwise similar to class. Aspects can include `pointcut` definitions denoting a set of join points in the static or dynamic control flow of a C++ program. An example pointcut function is the `execution` function used in the example, which filters the points in the control flow when the target join points are executed. Furthermore, the type of the execution advice is an `around` advice, meaning that it basically substitutes the affected join points. The advice body, denoting the substituted behavior, prints out a line on standard output, then proceeds with the functionality of the join point, and then closes with another output line. It makes use of the *join point API*, which is accessible in AspectC++ through the implicit object `tjp` (this join point). The functionality triggered by the calls of the `tjp` methods is adapted by the weaver to reflect the affected join point; that is, both `signature ()` and `proceed ()` are join-point-dependent.

More Examples

More sophisticated examples of AspectC++ fragments are located in Chapter 4, where the design of CiAO and specific parts of its implementation are presented.

[1] http://www.aspectc.org

CHAPTER 3

Concern Impact Analysis

This chapter presents an analysis of the AUTOSAR OS standard and the underlying OSEK specification with respect to separable system concerns. The proposed conformance classes and scalability classes provide a basic indication but are too coarse-grained and little variable, and, first and foremost, do not clearly *separate* distinct concerns. Hence, the resulting concern list transcends the standards.

The concerns are analyzed with respect to their potential impact on the abstract OS services and OS-managed state as specified in the AUTOSAR standard; these encapsulated functionalities are *aspect candidates* for the following design step (see Chapter 4). Most of the state that is introduced by the distinct concerns is assigned to an OS object type since it needs to be instantiable dependent on the number of OS objects of the specific type available in the configuration (e.g., *four* tasks and *two* alarms).

Additionally, the impact of the analyzed concerns on system-internal points is examined in order to be able to designate *pointcut candidates* of special importance to the operating system.

Only the perceivable influence on the semantics and functionality of the operating system is documented; nevertheless, most of the examined concerns bear an indirect impact on non-functional concerns like performance or safety.

Section 3.1 gives an overview of the identified features in the AUTOSAR OS, whereas Sections 3.2 to 3.5 provide a classification and an analysis of those. Section 3.6 summarizes the results of the impact analysis, while Section 3.7 provides a further discussion of those.

■ 3.1 Feature Overview

In order to get an overview of the concerns addressed in this chapter, a diagram of the features extracted from the concerns is presented, visualizing the configurability on the level of the problem space defined in the AUTOSAR specification. A feature is defined as a distinguishable characteristic of a concept that is relevant to some stakeholder of the concept [CE00], in this case

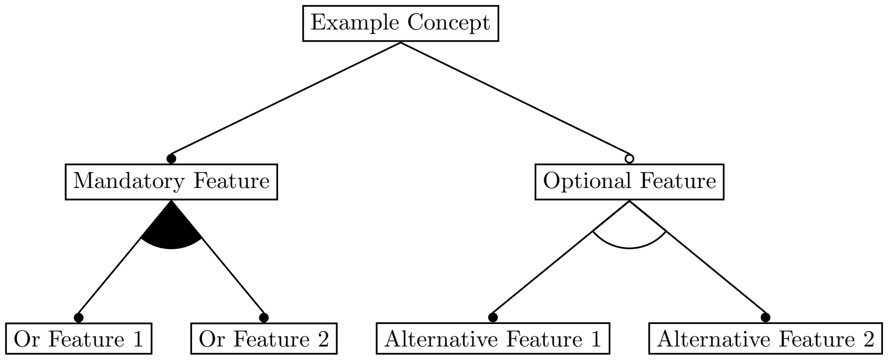

Figure 3.1: An example feature diagram.

the deployer of an AUTOSAR system. Most features can be derived directly from the identified concerns.

Feature Diagram Semantics

Since there are many different versions of feature diagrams in circulation, the semantics used here (as defined by Czarnecki and Eisenecker in [CE00]) are explained shortly. A feature connected with a filled circle is *mandatory* if the parent feature is selected, while one with an empty circle is *optional*. Arcs represent feature *groups*; a regular arc is depicted if only one of the group's features is to be selected (*alternative* features), while a filled arc allows for several features from the group to be selected (*or* features). A simple diagram containing all types of features is shown in Figure 3.1.

AUTOSAR OS Features

The main diagram representing an AUTOSAR OS is depicted in Figure 3.2 on the next page, while two of its subfeatures (system abstractions and service protection) are outsourced to Figure 3.3 on the facing page, and Figure 3.4 on page 18, respectively. The concerns implemented by the features are presented in detail in the following sections.

Feature Starter Set

Since the CiAO implementation of AUTOSAR OS is a product line, most of the identified concerns can be mapped to configurable features later-on. A potential *feature starter set* of the most basic CiAO operating system variant is the set of features mapped from the OS control concern (see also Section 3.2.1) and one of the control flow abstraction concerns (i.e., ISR 1 management, ISR 2 management, or task management).

■ 3.2 System Abstraction Concerns

The most basic features of an operating system are the *abstractions* it offers. The distinct abstractions and their impact on the API and behavior of the OS are analyzed in this section.

■ 3.2.1 OS Control

The AUTOSAR OS system designer is allowed to write his own start-up sequence determining the desired application mode, which must be handed over to the OS by calling `StartOS ()` [OSE05a, p. 24]. The application mode is

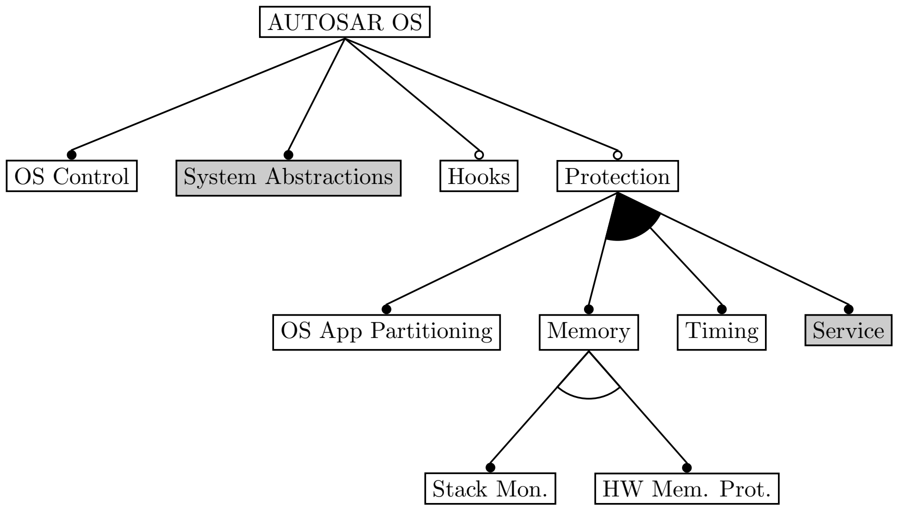

Figure 3.2: A feature diagram of the AUTOSAR OS standard. The system abstraction and service protection concerns are detailed in Figure 3.3 and Figure 3.4 on the next page, respectively.

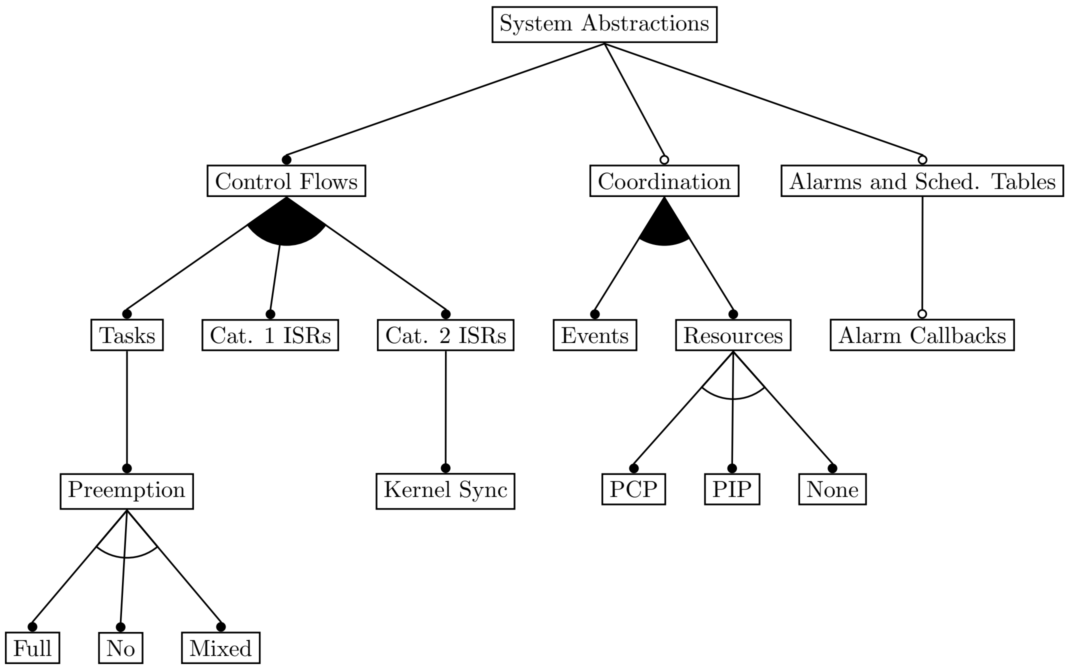

Figure 3.3: A feature diagram of the AUTOSAR OS system abstraction concerns.

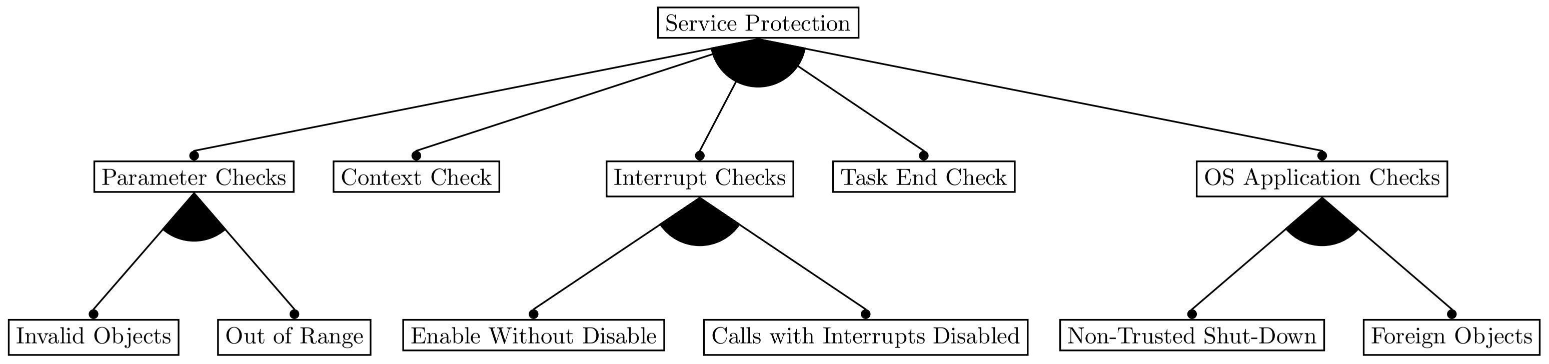

Figure 3.4: A feature diagram of the AUTOSAR OS service protection concerns.

queryable at run time (by the system service `GetActiveApplicationMode ()`) to be able to write mode-dependent code.

The system can be shut down by user code in a controlled manner using the service `ShutdownOS ()` [OSE05a, p. 43].

Impact on Services. The OS needs to provide the implementations of the three AUTOSAR OS control services: `StartOS ()`, `GetActiveApplication-Mode ()`, and `ShutdownOS ()`.

Impact on OS-Managed State. An identifier for the distinction of application modes is to be supplied. The active application mode is to be held in the OS-managed state.

■ 3.2.2 Task Management

Tasks are the basic control flow abstractions of AUTOSAR OS; they are organized by the OS scheduler [OSE05a, p. 16]. Tasks can activate other tasks, terminate themselves, chain other tasks after their termination, activate the scheduler manually, query their own ID number, and query the state of other tasks.

Impact on Services. The task management services are to be introduced to the OS API: `ActivateTask ()`, `TerminateTask ()`, `ChainTask ()`, `Sched-ule ()`, `GetTaskID ()`, and `GetTaskState ()`.

Impact on Internal Behavior. Upon start-up of the system, the chosen application mode determines which tasks are auto-started.

When alarms are managed by the OS, their configuration is to be checked upon expiry and the configured task is to be activated if applicable.

Impact on OS-Managed State. The concern of task management needs the corresponding task OS object type. Statically configured information that needs to be made available to the OS includes the task's priority, its stack, and its entry function. Dynamically modified information pertaining to a task comprises its state (e.g., running, ready, or suspended).

Moreover, configuration information about the tasks to be auto-started with an application mode is to be introduced, and alarms and schedule tables are given the configuration possibility to activate a task upon expiration.

■ 3.2.3 ISR Category 1 Management

ISRs of category 1 are implementation- and platform-specific, not allowed to use any OS service (except the ones to disable and enable other interrupts), and the OS is unaware of them [OSE05a, p. 25]. OS-managed tasks and ISRs of category 2 can synchronize with them by disabling their recognition and re-enabling it at the appropriate places.

Impact on Services. The concern to manage the recognition of ISRs of category 1 comprises four services: `EnableAllInterrupts ()`, `DisableAll-Interrupts ()`, `ResumeAllInterrupts ()`, and `SuspendAllInterrupts ()`.

Impact on OS-Managed State. None, since category 1 ISRs are not managed by the OS.

3.2.4 ISR Category 2 Management

The functions for processing interrupts that are managed by the OS are the ISRs of category 2 [OSE05a, p. 25]. Each category 2 ISR is assigned an interrupt at system generation time. Tasks can delay the execution of ISRs of category 2 through a well-defined API, where they are called *OS interrupts*, in contrast to *all interrupts*, which also comprise the ISRs of category 1 not managed by the OS.

There are also services to query the running ISR ID number, and to enable or disable the interrupt source triggering a defined ISR.

Impact on Services. ISR management functions that are to be offered by the AUTOSAR OS comprise `ResumeOSInterrupts ()`, `SuspendOSInter-rupts ()`, `GetISRID ()`, `DisableInterruptSource ()`, and `EnableInter-ruptSource ()`.

Impact on OS-Managed State. An OS object type encapsulating the static configuration of an ISR category 2 needs to be provided, which includes an abstraction of the bound interrupt source.

3.2.5 Resources

Resources are the AUTOSAR means to co-ordinate potentially concurrent access to shared resources [OSE05a, p. 29].

In order to address the problem of possible deadlocks and uncontrolled priority inversion, a protocol can be deployed. When acquiring a resource, the *OSEK priority ceiling protocol* (hereby denoted just as PCP) immediately raises the priority of a task to the maximum priority of all tasks possibly accessing the resource. This way, a resource can never be occupied when trying to acquire it. The *priority inheritance protocol* (PIP) only raises the priority if a task with a higher priority tries to acquire it—and then only to the priority of *that* task. The third alternative is merely not to address the problem and exclude it through the configuration of the application. AUTOSAR specifies that PCP is to be used [OSE05a, p. 31], although that might not be the best solution for all cases.

Impact on Services. There are two resource-related services that are introduced with resource support: `GetResource ()` and `ReleaseResource ()`.

Impact on OS-Managed State. Resources are OS objects of their own; hence, an identification needs to be introduced with the corresponding concern.

Depending on the chosen protocol, the resource and task OS object types are to be extended by the ceiling priority, the original priority (both constant), the occupied resources, or the occupying tasks (both dynamic). Furthermore, if a PIP is deployed, or no protocol is used, tasks can be blocked. Hence, a blocked state needs to be introduced in the dynamic part of the task OS structure.

3.2.6 Events

Signalisation between tasks is provided via the AUTOSAR OS mechanism of events [OSE05a, p. 27].

Impact on Services. If events are to be supported by the operating system, all of the event services are introduced to the API: `SetEvent ()`, `ClearEvent ()`, `GetEvent ()`, and `WaitEvent ()`.

Impact on Internal Behavior. Upon alarm expiry, if the corresponding alarm or schedule table was configured to set an event, this event is to be set for the configured task.

Impact on OS-Managed State. Since events are always set in combination with a task, each task needs to hold additional state about the events currently set and waited for. Furthermore, a task's state can be changed to *waiting* for an event.

Configured event support adds the possibility to the statically configured part of the alarm type and the related schedule table type to set an event upon alarm expiry, comprising the affected event and task (see also Section 3.2.7).

Note that an event does not constitute an OS object itself since it can not exist independently and conceptually only comprises an identifier that can be represented as a bit in a mask of events. The OSEK specification accommodates that fact by defining an `EventMaskType`, but no event type of its own.

3.2.7 Alarms and Schedule Tables

In AUTOSAR, alarms are the second stage of a two-stage system to process recurring events, the first one being counters [OSE05a, p. 36]. The alarm management functionality can be encapsulated very concisely.

Furthermore, the AUTOSAR OS specification introduces schedule tables, which encapsulate a statically defined set of alarms [AUT06b, p. 22]. Since their functionality goes hand in hand with regular alarms, they are grouped together.

Impact on Services. Firstly, software counters need an advancement interface called `IncrementCounter ()`. Secondly, the support for alarm management introduces the corresponding alarm services to the usable API: `GetAlarmBase ()`, `GetAlarm ()`, `SetAbsAlarm ()`, `SetRelAlarm ()`, and `CancelAlarm ()`. Furthermore, it provides the user with the schedule tables API, which consists of `StartScheduleTableRel ()`, `StartScheduleTableAbs ()`, `StopScheduleTable ()`, `NextScheduleTable ()`, `SetScheduleTableAsync ()`, `SyncScheduleTable ()`, and `GetScheduleTableStatus ()`.

Impact on Internal Behavior. When the system is started, the alarms and schedule tables configured to be auto-started need to be considered.

Impact on OS-Managed State. The alarm concern also introduces the alarm OS object type and the schedule table type (containing the static configuration information of the expiry points). The static part of the alarm state is its ticks per base unit. The dynamic part comprises the information about its current expiry point and cycle time, and if it is currently armed. Furthermore, the alarms and schedule tables to be auto-started need to be associated to the available application mode OS objects.

■ 3.3 Concerns Internal to the OS Kernel

Concerns that are not directly reflected in the system API perceivable by the user, but that nevertheless possess functionality essential to the kernel (in contrast to the fault isolation concerns listed in Section 3.4) are denoted *kernel-internal concerns*; they are not always explicitly listed in the analyzed specifications.

■ 3.3.1 Control Flow Management: Preemption Policy

An AUTOSAR system can be configured to be non-preemptive, fully-preemptive, or mixed-preemptive [OSE05a, pp. 22f.]. Mixed-preemptive means that the preemptability is configured per task, not system-wide.

Impact on Services. For a non-preemptive system, OSEK defines four explicit points of rescheduling: `TerminateTask ()`, `ChainTask ()`, `Schedule ()`, and `WaitEvent ()` (if events are supported, see Section 3.2.6). All of these are points when a task is not longer ready to run, plus when an explicit rescheduling is requested.

A fully-preemptive system additionally makes a scheduling decision after calls to `ActivateTask ()`, `SetEvent ()`, `ReleaseResource ()`, and `IncrementCounter ()`.

Mixed-preemptive systems potentially have the same scheduling points as fully-preemptive systems, but furthermore have to check whether the currently running task is preemptable before making a decision.

All of these points of rescheduling are only applicable if called from within a task. If the services are called from within an ISR, the rescheduling is performed only upon return from that ISR.

Impact on Internal Behavior. Fully- and mixed-preemptive systems have further points of rescheduling when a task is activated due to an alarm expiry (if alarm support is configured), or when an event is set to a task due to an alarm expiry (if alarm support and event support are configured).

Furthermore, upon return from an ISR, rescheduling is performed in fully- or mixed-preemptive systems if needed.

Mixed-preemptive systems additionally have to check the preemptability flag of the currently executing task at these rescheduling points to determine if rescheduling is actually performed.

Impact on OS-Managed State. If the system is configured to be mixed-preemptive, each task object is to be designated preemptable or non-preemptable at configuration time. Therefore, an additional property has to be introduced to the task OS object type in the static part.

3.3.2 Kernel Synchronization

In order to avoid corruption of certain kernel-internal data structures, critical operations on these structures need to be protected from interruption. This way, the kernel is kept *in sync* with the control flow abstractions making use of it (i.e., tasks and ISRs of category 2).

Since category 2 ISRs are both allowed to make use of a specified set of AUTOSAR OS system services (in contrast to the ISRs of category 1) *and* are triggered and executed asynchronously, they potentially interrupt the scheduler manipulating its data structures and invoke it themselves, leaving it in a possibly *inconsistent* state. This state is to be avoided by deploying an appropriate *kernel synchronization* policy.

Nevertheless, kernel synchronization is only needed *if* there are ISRs of category 2. Hence, the kernel synchronization concern is dependent on the concern introducing this type of ISRs (see Section 3.2.4).

Impact on Services. None.

Impact on Internal Behavior. Depending on the synchronization policy that is deployed and its granularity, there are different points in the kernel that are affected by the synchronization concern. These points include the entering and leaving of the whole kernel (if the kernel constitutes a single, coarse synchronization domain), and the entering and leaving of critical portions of the kernel that are susceptible to corruption due to asynchronous interruptions (if more fine-grained synchronization is applied). In an AUTOSAR system, the latter points only include the invocation of scheduler code, since the device drivers, which are normally also candidates for bearing critical sections, are not part of the core OS kernel.

Impact on OS-Managed State. None.

3.4 Fault Isolation Concerns

The big innovation of AUTOSAR over OSEK are its newly introduced facilities to improve early fault isolation. This comprises concepts for spatial isolation between software components (OS applications, memory protection, and stack monitoring) and temporal isolation (timing protection).

The third fault isolation facility consists of several plausibility checks subsumed under the term *service protection*. Since the applications interact with the OS through the defined system services, care has to be taken that the OS is not corrupted this way [AUT06b, p. 36].

All of these concerns are *super-functional* in the sense that they do not directly influence the functionality of the operating system that is visible by the applications. Nevertheless, they encapsulate an indirect functionality, which is in this case perceivable in potential fault situations.

■ 3.4.1 Support for OS Applications

OS applications are a new AUTOSAR concept of OS objects groups in order to reach better separation of functional units [AUT06b, p. 29]. The inter-application access is restricted; explicit access rights have to be granted at configuration time if desired (see also Section 3.4.12).

Note that the concept of OS applications is not classified as a system abstraction since, in contrast to the other system abstractions, it does not provide further functionality to the user. It is merely a means to partition the existing system abstractions in order to provide an additional concept that can be tackled by other fault isolation concerns.

Impact on Services. The AUTOSAR service to query the running OS application, `GetApplicationID ()`, needs to be introduced and implemented. Additionally, the OS application equivalent to `TerminateTask ()` that terminates the running OS application, `TerminateApplication ()`, is to be provided.

Furthermore, the services to check for access privileges need to be implemented: `CheckObjectAccess ()` and `CheckObjectOwnership ()`.

Finally, the interface for the invocation of so-called shared trusted functions is to be provided through the implementation of `CallTrustedFunction ()`.

Impact on OS-Managed State. The identification type for an OS application is to be introduced. All other OS objects are to be assigned an owning OS application in their statically configured parts.

■ 3.4.2 Memory Protection

The optional memory protection feature (see [AUT06b, p. 31]) is one that is very comprehensive and allows for a very flexible and fine-grained configuration. It is the scope of a related diploma thesis by Jochen Streicher [Str07]. For a detailed analysis of the impact of different memory protection scenarios on the AUTOSAR system services, consider his thesis. For an overview of the implementation of memory protection in CiAO, see [LSH+07].

AUTOSAR only defines memory protection to be enabled or not. A further option would be to protect single tasks and their stacks against each other on a *configurable* basis since this most fine-grained protection granularity also induces the most costs in terms of memory usage and performance loss.

Impact on System Services. Memory protection domains can be queried at run time using the AUTOSAR services `CheckISRMemoryAccess ()` and `CheckTaskMemoryAccess ()`. These services are to be implemented to yield the access characteristics of the given ISR or task, respectively.

Impact on Internal Behavior. In general, a memory protection switch is necessary at those points in time when the newly executed control flow belongs to a different application than the previously executing one. This comprises both tasks and ISRs; therefore, the set of application switches are a subset of the dispatch points of tasks and ISRs.

The memory protection domain needs to be switched at task switch time if the old task and new task belong to different applications. It *always* needs to be switched if protection is applied between single tasks and their stacks.

Since ISRs of category 2 also belong to an OS application, the protection domain might also need to be switched when an ISR is triggered and be switched back when it ends.

Furthermore, since the kernel memory is not directly accessible by the applications, the transitions from kernel to application context and vice versa are also affected by memory protection considerations; the kernel shall be enabled to access its memory space, while being protected from access by the applications.

Impact on OS-Managed State. The memory protection properties are to be stored in the corresponding OS application object type or task and ISR object types, depending on the protection granularity level. Since they are only configurable and not modified at run time, they can be stored in the static part of the affected types.

■ 3.4.3 Stack Monitoring

For platforms that do not provide any kind of memory protection hardware (i.e., an MPU or an MMU), AUTOSAR recommends to implement a stack monitoring facility [AUT06b, p. 28]. It prescribes to check for a stack overflow at context switch time, and to shut down the OS if a stack fault was detected. An extension to that feature would be to check for a stack overflow each time the OS is invoked, that is, each time a system service is called. This way, stack faults can be detected even earlier, though not as immediate as with hardware memory protection support (see Section 3.4.2).

Impact on Services. Basically none. Only the additional feature to check on each OS service invocation would extend every system service to check the stack before executing.

Impact on Internal Behavior. A stack overflow check is to be performed at the context switch point in the operating system.

Impact on OS-Managed State. Depending on the implementation, the bottom of the stack may need to be stored additionally to the top of stack in the task OS object structure, which is a static configuration information.

■ 3.4.4 Timing Protection

Timing protection is the AUTOSAR feature to enforce previously determined timing characteristics of control flows [AUT06b, p. 34]. These include the overall run time, the execution time while holding resources, the execution time while locking interrupts, and the arrival rate.

Impact on System Services. Whenever a system service to lock interrupts (`DisableAllInterrupts ()`, `SuspendAllInterrupts ()`, or `SuspendOSInterrupts ()`) or to acquire a resource (`GetResource ()`) are called, an internal timing facility has to be started to measure the locking/holding time, or to set a watchdog when the budget will be depleted. This watchdog has to be deactivated when interrupts are unlocked (`EnableAllInterrupts ()`, `ResumeAllInterrupts ()`, or `ResumeOSInterrupts ()`) or resources are released (`ReleaseResource ()`) before expiration of the budget.

Impact on Internal Behavior. The measurements and watchdog activation/deactivation for the overall run-time monitoring and the arrival rate monitoring are to be adjusted at the internal task switch point and whenever an ISR of category 2 is executed and ended.

Impact on OS-Managed State. The timing protection properties are to be stored in the corresponding task and ISR object types. These include the static limits configured at compile time as well as the current (dynamic) budgets.

■ 3.4.5 Service Protection: Invalid Object Parameters

If a system service is called with an OS object parameter not defined in the OIL configuration file, this call is to be intercepted [AUT06b, p. 37].

Impact on System Services. Before each call to a service with at least one OS object parameter, all OS object parameters need to be checked for correctness. The services that are affected are `ActivateTask ()`, `ChainTask ()`, `GetTaskState ()`, `SetEvent ()`, `GetEvent ()`, and `CheckTaskMemoryAccess ()`, which have a task parameter; `GetResource ()` and `ReleaseResource ()`, which have a resource parameter; `StartOS ()`, which has an application mode parameter; all alarm services, which have an alarm parameter; `CheckISRMemoryAccess ()`, `DisableInterruptSource ()`, and `EnableInterruptSource ()`, which have an ISR parameter; `IncrementCounter ()`, which has a counter parameter; all schedule table functions, which have a schedule table parameter; and `CheckObjectAccess ()`, which has an OS application parameter.

Impact on OS-Managed State. None.

■ 3.4.6 Service Protection: Out of Range Values

A check facility related to checking for invalid OS object parameters (see Section 3.4.5) is the check for values that might be out of the statically configured range [AUT06b, p. 37].

Impact on System Services. Statically configured ranges are only used with the alarm and schedule table OS objects. The alarm services `SetRelAlarm ()` and `SetAbsAlarm ()` are to be checked if their cycle and increment or start parameters, respectively, are within the ranges configured through the alarm base parameters `mincycle` and `maxallowedvalue`. The schedule table services `StartScheduleTableRel ()` and `StartScheduleTableAbs ()` get their tick argument checked if it is smaller than the configured `maxallowedvalue`.

Impact on OS-Managed State. The alarm OS object type only needs its statically configured subcomponents `mincycle` and `maxallowedvalue` if these ranges are checked. Therefore, they are to be introduced with the out of range values check feature. The same claim holds for the schedule table type and the static `maxallowedvalue` component.

■ 3.4.7 Service Protection: Wrong Context

AUTOSAR comprises a comprehensive matrix of allowed contexts for calls of every API service of the operating system [AUT06b, p. 37]. If an application issues a service call from a wrong context, the service is not to be performed and an error status code is to be returned where possible (non-void services).

Impact on System Services. Each and every system service is affected by this feature. The context of the call is to be compared to the matrix of the specification and in case of a mismatch the alternative behavior is to be taken.

Impact on OS-Managed State. None.

■ 3.4.8 Service Protection: Missing Task End

In OSEK, when a task does not end in a controlled way with `TerminateTask ()` or `ChainTask ()`, the resulting system behavior is not defined. AUTOSAR checks for these situations [AUT06b, p. 39].

Impact on System Services. None.

Impact on Internal Behavior. All task functions are appended an internal termination call, releasing all resources, enabling all interrupts, and calling the error hook appropriately.

Impact on OS-Managed State. None.

■ 3.4.9 Service Protection: Enable Without Disable

AUTOSAR specifies that the enabling/resuming of interrupts is not to be performed if no corresponding disable/suspend was issued before [AUT06b, p. 39].

Impact on System Services. Additional checks are introduced before the system services `EnableAllInterrupts ()`, `ResumeAllInterrupts ()`, and `ResumeOSInterrupts ()`. The services are not performed if applicable.

Impact on OS-Managed State. None.

■ 3.4.10 Service Protection: Service Calls With Interrupts Disabled

Calling any OS service (except the interrupt services) outside hook routines when interrupts are disabled is not supposed to happen according to AUTOSAR [AUT06b, p. 40].

Impact on System Services. Before any service except the interrupt services is called, the interrupt enable status is to be checked. If interrupts are disabled, the OS shall not provide the service and return a specific error code. Calls from within hook routines are to be excluded from that mechanism.

Impact on OS-Managed State. None.

■ 3.4.11 Service Protection: Shut-Down from Within Non-Trusted Code

The OS shall ignore calls to `ShutdownOS ()` by non-trusted code [AUT06b, p. 40].

Impact on System Services. The system service `ShutdownOS ()` is to be prepended by a context check. If the call context is non-trusted, the call is to be ignored.

Impact on OS-Managed State. None.

■ 3.4.12 Service Protection: Service Calls on Objects in Different OS Applications

If system services are called with OS object parameters from a foreign OS application, and no sufficient access rights were assigned in the configuration, the service call is invalid [AUT06b, p. 41].

Impact on System Services. Every system service with an OS object parameter (for a list of those, see Section 3.4.5) needs an additional check up front that determines if sufficient access rights are given and returns an error code if not.

Impact on Internal Behavior. Since access rights are bound to an *OS application*, the switches of those are points of interest to this concern. Nevertheless, since access rights are only needed on demand (see description above), an alternative is the deduction of the currently running OS application through the evaluation of the currently running task or ISR.

Impact on OS-Managed State. All OS object types are given additional fields for the statically configured access rights of applications other than the owning OS applications.

■ 3.5 Callback Concerns

The ability to activate a user callback routine upon the occurrence of a specific system event bears problems when timing protection or memory protection are enabled (see [AUT06b, p. 21], and Sections 3.4.2 and 3.4.4). That is why concerns introducing such callbacks are to be configurable separately—for instance, the support for alarm callback routines is explicitly deactivated in most AUTOSAR scalability classes.

■ 3.5.1 Alarm Callbacks

An AUTOSAR alarm can be configured to execute a callback routine upon alarm expiration [OSE05a, p. 36].

Impact on Services. None.

Impact on Internal Behavior. The operating system has to provide for the activation of the callback routine upon alarm expiration. This effectively corresponds to an up-call to the application, which has to be bound in some way.

Impact on OS-Managed State. Alarm callback support adds a static configuration option to the alarm OS object type denoting which callback to activate if desired.

■ 3.5.2 Hooks

OSEK specifies five different application hooks that are called at specific points in the operating system: a pre-task hook, a post-task hook, a start-up hook, a shut-down hook, and an error hook [OSE05a, p. 39].

AUTOSAR furthermore defines *application-specific* hooks [AUT06b, p. 46], which can be provided on a per-OS-application basis if OS applications are supported (see Section 3.4.1). The application-specific start-up hooks and shut-down hooks are called after and before the system-wide ones, respectively. The application-specific error hook, however, is only called if it was the application that *caused* the error; the system-specific error hook is always called before it.

Additionally, AUTOSAR also defines an new hook type not needed in OSEK: the protection hook [AUT06b, p. 43]. It is called on protection violation errors, like memory, timing, or service protection errors.

The hooks concern is a special concern since it is a functional one, yet it does not become manifest in the system API. In contrast, AUTOSAR defines the hooks as a an *application interface*, which is called back by the operating system at specific points of execution.

Impact on Services. The OS services are not directly affected by the introduction of the first four hook routines and the protection hook. The error hook, however, leads to an additional check after the call of any system service returning a status code. It is executed whenever a system service returns an error status code, but not if the service was called from *within* the error hook.

Application-specific error hooks need an additional query for the application that was responsible for the occurrence of the error.

Impact on Internal Behavior. The introduction of the first four hooks affects distinct points in the OS internals that are exactly defined in the specification. The start-up hook is to be called after the OS start-up, but before the scheduler. The shut-down hook is called whenever the system is shut down from within the OS or by user request. The post-task hook is called directly before the old task leaves the running state, the pre-task hook directly after the new task is set running.

Protection errors lead to the execution of the provided protection hook.

Impact on OS-Managed State. The configuration of application-specific hooks pertains to the OS application type, which is to be extended accordingly in its static part (see also Section 3.4.1).

3.6 Summary of the Impact Analysis

API and OS Object Types

Table 3.1 on the facing page summarizes the impact of the detected concerns on the AUTOSAR API, whereas Table 3.2 on page 32 subsumes the influence on the AUTOSAR OS object types in its upper part.

Pointcut Candidates

A further analysis of the investigated impact on OS internals reveals distinct points in the control flow of the operating system that classify as *pointcut candidates*. If the concern interested in a specific point is mapped to an aspect entity in the engineering process later, it will be enhanced to a full *pointcut* designating a set of *join points*. In the lower part, Table 3.2 on page 32 also lists the identified pointcut candidates together with the concerns that are interested in them.

3.7 Discussion of the Impact Analysis

3.7.1 Impact on AUTOSAR System Services

Horizontal Analysis (Service-Oriented)

It can be observed that every AUTOSAR system service is introduced by exactly one concern. Most of these concerns are the system abstraction concerns, which is obvious since every service operates on some kind of system abstraction. The exceptions are the two fault isolation concerns bringing in support for distinguishable OS applications and protection of memory areas; they introduce five and two more system services, respectively. It is only them that are concerned with supplying services that presume an additional application and memory boundary, respectively.

Furthermore, every system service is affected by 2–7 modifying concerns; that is, there are up to 8 different concerns that cross-cut at a single service (counting the one that introduces its original functionality).

Vertical Analysis (Concern-Oriented)

Having a look at the concerns beyond the system abstraction concerns, it can be seen that those having an impact on a lot of services (i.e., more than 10) can be assigned simple "verbal" pointcut expressions to define the impact location:

	System Abstractions (Functional)							Internal		Fault Isolation (Super-Functional)												Callbacks	
	OS Control	Tasks	ISRs Category 1	ISRs Category 2	Resources	Events	Alarms	Preemption	Kernel Sync	OS Applications	Memory Protection	Stack Monitoring	Timing Protection	Invalid Parameters	Out of Range	Wrong Context	Missing Task End	Enable w/o Disable	Interrupts Disabled	Non-Trusted Shut-Down	Foreign OS Objects	Alarm Callbacks	Hooks
`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 ()`											⊕			◐		◐			◐		◐		

Table 3.1: Impact of configurable concerns on AUTOSAR system services (⊕ = API extension, ◑ = modification after service execution, ◐ = modification before service execution).

	System Abstractions (Functional)							Internal		Fault Isolation (Super-Functional)												Callbacks	
	OS Control	Tasks	ISRs Category 1	ISRs Category 2	Resources	Events	Alarms	Preemption	Kernel Sync	OS Applications	Memory Protection	Stack Monitoring	Timing Protection	Invalid Parameters	Out of Range	Wrong Context	Missing Task End	Enable w/o Disable	Interrupts Disabled	Non-Trusted Shut-Down	Foreign OS Objects	Alarm Callbacks	Hooks
Application Mode	⊕	⊛					⊛																
Task		⊕			⊛	⊛		⊛		⊛	⊛	⊛	⊛								⊛		
ISR Category 2				⊕						⊛			⊛								⊛		
Resource					⊕					⊛											⊛		
Alarm / Schedule Table		⊛				⊛	⊕			⊛					⊛						⊛	⊛	
OS Application										⊕	⊛												
Alarm Expiry		◑				◑		◑														◑	
Cat. 2 ISR Execution								◑			●		●										
System Start-Up		◑					◑																◑
System Shut-Down																							◐
Protection Violation																							◑
Task Switch											◑	◐	●										●
Application Switch[a]											◑										◑		
Uncontrolled Task End																	◑						
Application–Kernel Transition									◐		◑												
Kernel–Application Transition									◑		◑												

Table 3.2: Impact of configurable concerns on OS-managed state in the form of AUTOSAR object types (upper part of the table; ⊕ = introduction of a new type, ⊛ = modification/extension of an existing type) and on designated OS-internal pointcut candidates (lower part of the table; ◑ = modification after OS-internal pointcut, ◐ = modification before OS-internal pointcut, ● = modification before *and* after OS-internal pointcut).

[a] As pointed out in Section 3.4.2, this is basically a subset of all task switches and category 2 ISR dispatch points.

- The concern checking for invalid parameters as well as the concern for calling services with foreign parameters affect all services with an OS object type parameter.
- The concern checking the service call context affects all services.
- The concern checking for disabled interrupts on a service call affects all services except the interrupt recognition services.
- The hooks concern (or the *error hook* concern, to be precise) affects all services returning a `StatusType`.

All of these concerns have a similar impact on the affected services; for instance, they all modify the behavior before *or* after the regular service execution. These points are a clear yet not sure indication for an aspect-oriented mapping in the later design and implementation stages.

3.7.2 Impact on OS-Managed State

OS Object Type Introduction

The state managed by the operating system is partitioned into different types since it needs to be instantiable in different quantities, depending on the configuration. Each of these OS object types is introduced by exactly one concern, namely the one that also introduces the notion of this OS object type *conceptually*.

Heavily Cross-Cutting Types

Focusing on the object types, both OS-managed control flow abstraction types (i.e., tasks and ISRs of category 2) are extensively affected by other concerns. They constitute the fundamental abstraction of operating systems in general, and need to provide the per-object configuration of the different protection concerns. Most of this information to be supplied is constant, however, and does not change dynamically at run time.

Heavily Cross-Cutting Concerns

The two concerns with a particularly striking effect on the OS-managed state are the ones introducing the OS application grouping, and the one checking for access rights when invoking service calls with a foreign OS object parameter. Both of them introduce static configuration information fields to all core OS object types: the owning OS application, and the accessing OS applications, respectively.

3.7.3 Impact on OS-Internal Pointcut Candidates

The OS-internal pointcut candidates identified during the analysis are affected by 1–4 concerns each. In the following, the candidates with more than one stakeholder concern are discussed more thoroughly.

Dispatch Points

The points that the concerns are most interested in are the points in the operating system where a new control flow is dispatched. This includes switching the active task as well as the dispatching from a task *to* an ISR and back *from* the execution of an ISR to the task level. This is what makes an operating system software special compared to application software: It provides and manages *different* control flows at once, which each have specific characteristics. That is why the dispatch points are important for several concerns. Since OS applications merely group specific ISR and task instances, the application switches are only a subset of all ISR and task dispatch points (see also Section 3.4.2).

Kernel Enter/Leave A second speciality of system software is the distinction of at least two different inherent contexts: the application context, and the system kernel context. The points of transition between these two levels (bearing different privileges) are always controlled by the operating system: The execution of kernel code is either done through an explicit API service invoked by the application, or it is implicitly invoked by the operating system itself (e.g., the execution of kernel code when an interrupt occurs); the kernel is left only at distinct points. All of these points are important for more than one concern also; the analysis yielded the kernel synchronization and the memory protection concerns. Hence, the memory protection concern is an architectural property that affects both special types of points in an operating system.

Alarm Expiration The third point of multiple concern interest is the expiration of an alarm. A task, an event, or an alarm callback might be activated upon the reaching of this point, and thus the corresponding concerns need to be provided this point in execution. They are directly related to the alarm concern and concept, though, and therefore rather tightly than loosely coupled to it. This is different with the preemption policy concern, which, depending on its configuration, might need to trigger a rescheduling upon an alarm expiry.

Hook Points The rest of the system-internal points are only used by the AUTOSAR user hooks; the needed pointcut candidates overlap with the ones discussed, though (e.g., before a task switch, the pre-task hook needs to be invoked).

3.7.4 Concern Hierarchy

Hierarchy Notation Figure 3.5 on the next page shows a hierarchy of the analyzed concerns, visualizing both dependencies and influences. The used notation is based on the one proposed by Spinczyk et al. [SLSP06]: The "uses" relationship constitutes a hard dependency; that is, the using concern needs the used concern in order to work properly. In contrast, the "influences" relationship induces a rather loose and optional coupling: If some or all of the influenced components do not exist, this does not constitute an error. The resulting hierarchy is a step into the direction of a design, being a part of the solution space.

Protection Concerns It can be seen that the concerns that influence the most of the other concerns are from the protection domain; and thereof especially some of the service protection concerns are highly cross-cutting the system. Those service protection concerns have an impact on the group of concerns containing user-provided code—since only those can invoke illegal system services or invoke legal ones in an illegal way, thereby possibly corrupting the system.

System Abstraction Concerns Furthermore, it becomes obvious that the basic system abstractions are rather independent of each other, except for the task concept, which is needed by both the resources concept and the events concept. The dependencies of the alarm abstraction depend on the actual refinement type of the alarm; a common base alarm management is needed in every case.

Kernel-Internal Concerns Interestingly, both OS-internal policies influence only the task management concern—preemption and kernel synchronization both refer to the task scheduling mechanism.

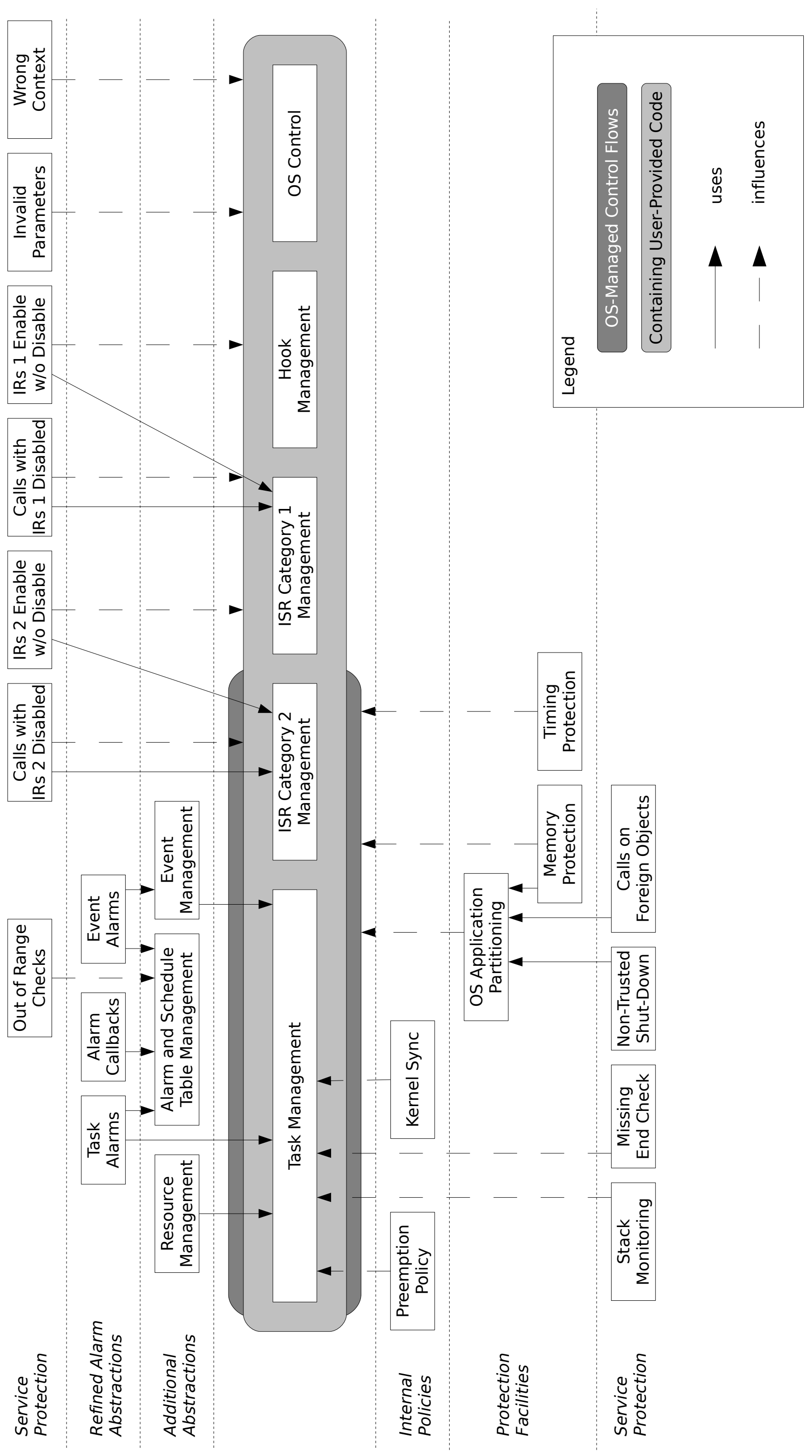

Figure 3.5: A hierarchy of the analyzed concerns present in an AUTOSAR-like operating system.

CHAPTER 4

Design

Taking the concerns and their properties as identified in the analysis as an input (see Chapter 3), the design of the CiAO kernel is developed.

This chapter first gives an overview of the CiAO operating system design (see Section 4.1), and then details the structure of the operating system kernel (see Section 4.2). It then shows how AOP patterns are explicitly used in the kernel design (see Section 4.3) and implementation (see Section 4.4).

4.1 Overview of the CiAO Operating System Design

Layered Architecture

CiAO is designed with a *layered architecture*. Each layer is implemented using the functionality of the layer below it. Design exceptions that lead to an up-call from a lower layer to one located above are clearly denoted and integrated by a special mechanism (see Section 4.3.2). Figure 4.1 on the following page provides an overview of the CiAO design; in the following, it is presented shortly.

AUTOSAR Layer

In order to bring CiAO close to conformity with the OSEK and AUTOSAR OS standards, an AUTOSAR layer is deployed. It provides the application programmer with the complete interface to the operating system necessary according to the standard. Hence, it is the one-and-only means for the application to communicate with the kernel, and itself relies on services that are internal to the OS. It adapts the core CiAO services to the interface and the semantics demanded by the specification, including the adaptation of the specified data types. Technically, it consists of a single class providing static interface methods.

AUTOSAR Error Checks

The AUTOSAR layer is also the single place that is affected by additional plausibility error checks denoted *extended status* in the OSEK specification and *service protection* in AUTOSAR. Depending on the result of a condition that is additionally checked, the affected services return an alternative status code representing the error. This code can then be evaluated in the application or in the triggered error hook if support for it is configured. Beneath the AUTOSAR layer, the kernel supposes correct identifiers and execution conditions.

Kernel Layer

CiAO's kernel is at the heart of the operating system and determines its functionality. Its subdesign is detailed in Section 4.2.

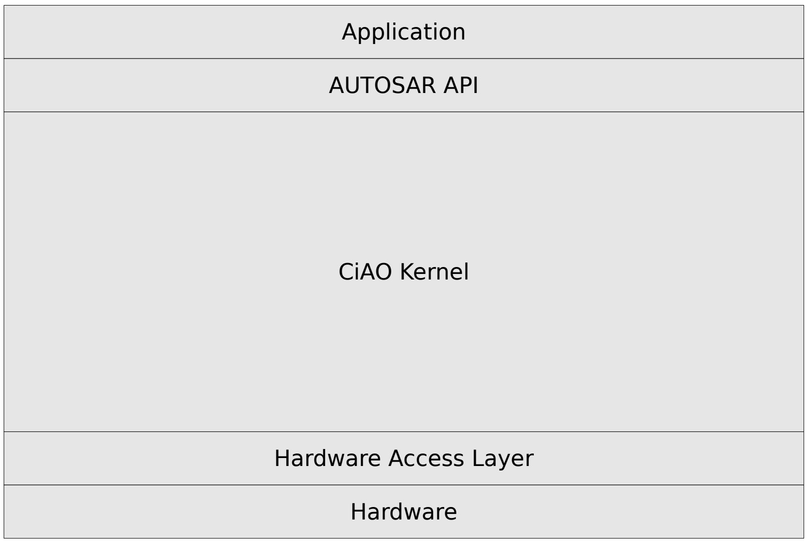

Figure 4.1: A coarse overview of CiAO's layered architecture.

HAL Since CiAO is targeted to support various hardware platforms, it has a defined interface to the hardware. This interface is provided by a *hardware access layer* (HAL), which is different for each supported platform, depending on the functionality *offered* by the platform. However, the abstractions commonly provided by *all* platforms (such as a continuation, CiAO's basic control flow abstraction) constitute the *hardware abstraction layer* that the therefore platform-independent kernel is based on.

■ 4.1.1 OOP and AOP in CiAO

CiAO is designed using both paradigms of object orientation and aspect orientation.

OOP Restrictions Nevertheless, object orientation is not deployed in an uncontrolled way; it is only used when the corresponding implementation bears no overhead compared to a similar procedural one since performance is a critical issue especially in operating system design. Hence, the concept of classes and inheritance is mostly used to provide modularization and inherent namespaces that can be tackled by AOP.

AOP Obliviousness The AOP notion as it is understood by CiAO also differs to some extent from the one commonly described. For instance, the property of *obliviousness* of the affected code, which is considered fundamental to AOP by some [FF00], is explicitly abandoned. Since CiAO's design is aspect-aware, it is explicitly *non-oblivious* to most aspects and their advice. This is most obvious at the definition of *explicit join points* (see Section 4.4.1), which are deployed for the sole purpose of aspect-binding. Furthermore, methods are kept small and manageable not only for reasons of decomposition, but also in order to provide more join points in the kernel for advice to act upon.

Modularization and Patterns

Aspects are incorporated in the design for two main reasons: modularization, and as part of patterns. Modularization of cross-cutting concerns is one of the main advantages of AOP in the first place and leads to a further separation of concerns, implying better maintainability and evolvability. Furthermore, AOP is used in CiAO for specific design and implementation patterns (see Section 4.3 and Section 4.4) that bear advantages of their own.

Separation of Concerns

The goal regarding configurability is that each configurable feature is represented by a distinct number of design and implementation artifacts in a 1:n relationship; that is, no artifact belongs to more than one feature. Since AOP is also about *quantification* [FF00], it is especially useful for that purpose: Advice is only enabled when the target join point is available; if it is not, this piece of advice is just not applied. Hence, different features can be further decoupled and separated.

4.1.2 Aspect Implementations in CiAO

Code Bases

The binary image that is eventually loaded on the target microcontroller platform is compiled from two parts: the operating system code, and the application code. Both code bases may comprise aspects that can affect the whole system. That is, an operating system aspect may contain join points in the application (and therefore modify the application), and an aspect belonging to the code base of the application may affect join points located in the operating system kernel.

Application Aspects

Of course, the latter possibility is restricted to selected explicit join points only; the aspects from the application are generated and not hand-written by the user, anyway. These application aspects, generated depending on the configuration, are used for efficient binding of up-calls from the kernel into the application (see Section 4.3.2).

4.1.3 Overview of the Used Design Notation

Diagram Notation

The following description of the CiAO kernel design is provided both textually, and, to facilitate the overview, by design diagrams. These diagrams are depicted in an own, UML-like notation that is oriented at the syntax and features of AspectC++. It is a simplified model of what the actual implementation looks like, and it is supposed to provide a focused view of the important parts while leaving out unnecessary details.

Notation Introduction

The used notation focuses on the classes and aspects introduced by the feature that is to be depicted. Classes are shown in standard UML fashion, while aspects are depicted class-like with an `<<aspect>>` stereotype; the different pieces of advice are denoted as member elements. The class of advice—introduction, execution, call, or order advice—is denoted by the sequence in parentheses before the pointcut: `(I)`, `(A-exec)`, `(A-call)`, and `(A-order)`, respectively. The pointcut to be advised is denoted in an AspectC++-like syntax using the wildcard characters where appropriate and needed. The dashed arrows target the components influenced by an aspect, where the components are mostly depicted as stubs without their member variables and functions. Furthermore, aspects can inherit pointcut definitions (depicted as member variables of type `pointcut`) and advice[1] from super aspects.

Related Notations There are also other suggestions for notations of aspect-oriented designs, none of which were found suitable to describe CiAO:

Composition Patterns Clarke et al. present composition patterns [CW01] and Theme/UML [BC04] in order to model re-usable aspects. These approaches model aspects as UML templates, which can be parameterized. These parameters are then bound by a binding composition to the component that is advised. This notation, however, would rapidly overflow the CiAO diagrams with information that is not relevant at such detailed level, especially when designing homogeneously cross-cutting aspects.

Work by Stein et al. The aspect-oriented design notation as proposed by Stein et al. [SHU02] explicitly targets AspectJ and its concepts. Hence, it contains clauses such as `instantiation = perJVM`, which make no sense in the context of AspectC++. Furthermore, it is also too detailed and cluttered for multi-aspect diagrams as needed in this case. This is partly due to the goal to be completely compatible to standard UML (in order to be able to use standard CASE tools), which hinders the application of more appropriate but incompatible notation elements.

AML The AML (aspect modeling language) by Groher and Baumgarth [GB04] also aims to be UML-conformant and targets aspect-oriented architecture design. Once more, its level of detail is too high to be able to concisely describe a comprehensive architecture; it is even so detailed that it was possible to develop a code generator taking AML diagrams as its input.

■ 4.2 Basic Design of the CiAO Kernel

In its full configuration, the CiAO kernel source bears *classes* for basic OS facilities (providing a basic interface) and for the OS objects managed internally. All further behavior is modularized in *aspects* that affect the functionality of these classes and their interface.

Core OS Facilities The basic OS facilities are singletons by definition and comprise:

- The scheduler: This is the entity that cares about the dispatching of tasks and the scheduling strategy. It is also the single point that needs to be synchronized with the execution of ISRs of category 2.
- The alarm manager: Cares about the management of alarms and the underlying counters, which may be shared by several alarms. Particularly takes care of hardware alarm facilities to provide a hardware-based system alarm.
- The OS control facility: Provides services for the controlled start-up and shut-down of the system and attends to the management of application modes.

OS Objects Furthermore, the OS objects corresponding to the system abstractions offered in the OS configuration and instantiated in the configuration by the application are held in separate arrays. Each of the OS objects bears the state necessary for the attendant type.

[1] Note that the inheritance of advice from a super aspect is only possible *conceptually* in order to extract advice common to several aspects into a common base aspect.

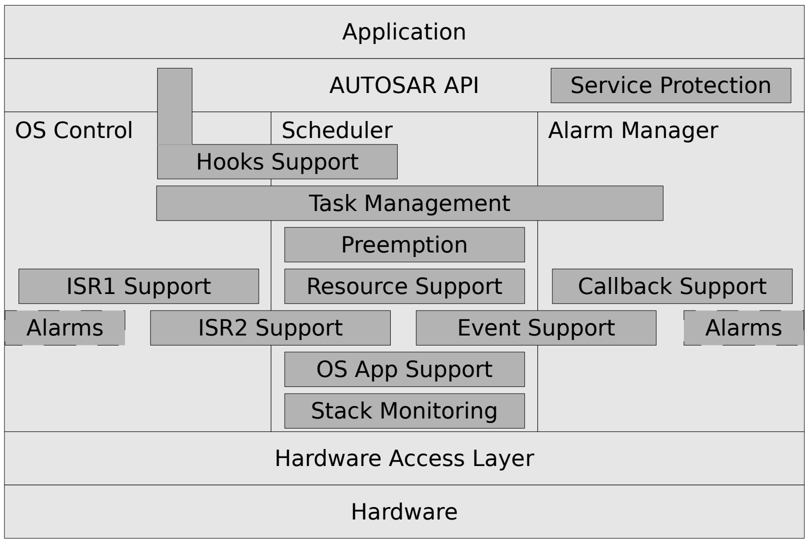

Figure 4.2: The CiAO kernel design and influencing concerns designed as aspects.

Further Functionality

All functionality beyond the core mentioned above is provided via advice given by distinct aspects. These pieces of advice affect both static and dynamic parts in the basic facilities, and add the appropriate state information to the OS object types where necessary. Moreover, they extend the AUTOSAR API offered to the applications accordingly in order to provide a complete encapsulation of the represented concern (see Section 4.3.1 for details).

Figure 4.2 gives an overview of the basic architecture of the CiAO kernel—its core facilities and the aspects supporting further functionality and their targets. The following sections provide detailed information.

■ 4.2.1 The OS Control Facility and Application Modes

The OS control singleton encapsulates three functions that are interrelated: the OS start-up, the OS shut-down, and the management of application modes.

OS Start-Up

The start-up routine is the one that is called by the AUTOSAR service `StartOS ()`, which itself is to be called at the end of a user-provided start routine in the form of a `main ()` function. The goal of the user routine is to determine an application mode, which is then passed as a parameter to the OS control start-up function. This function is responsible for initializing the system and hardware accordingly.

OS Shut-Down

Furthermore, an entry point for a controlled shut-down of the system is provided, where hardware-specific de-initialization routines can be bound to. This shut-down routine is called either on qualified user request through the system service `ShutdownOS ()`, or internally on the occurrence of a non-recoverable error.

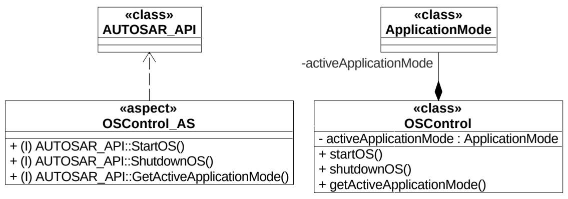
«class»
AUTOSAR_API
«class»
ApplicationMode
-activeApplicationMode
«aspect»
OSControl_AS
+ (I) AUTOSAR_API::StartOS()
+ (I) AUTOSAR_API::ShutdownOS()
+ (I) AUTOSAR_API::GetActiveApplicationMode()
«class»
OSControl
- activeApplicationMode : ApplicationMode
+ startOS()
+ shutdownOS()
+ getActiveApplicationMode()

Figure 4.3: The OS control and application mode classes.

Application Mode Management

The application mode supplied to the OS by the user can be queried at run time in order to write mode-dependent application code. Hence, the application mode passed to the start-up routine is stored in the OS control singleton and can be retrieved through the API service `GetActiveApplicationMode ()`.

A simple depiction of the OS control and application mode classes is presented in Figure 4.3.

■ 4.2.2 The Scheduler, Continuations, Tasks, and Preemption

The scheduler is at the heart of the kernel; it is responsible for the dispatching and execution of the application code in the form of tasks.

Tasks and Continuations

Internally, CiAO's control flow abstraction is a *continuation*. A continuation stores the context of a control flow in its control block. A task does not necessarily correspond to exactly one internal continuation, though; continuations might be shared among tasks. This is possible, for instance, when it is guaranteed that two tasks never run at the same time, or when they only preempt each other in one direction—then the interrupting task can run on the remaining stack of the interrupted task. The sharing of context is an optimization that is especially valuable on architectures with a big context, like, for instance, the ARM architecture.

Control Flow Semantics

Hence, there are two notions of a "task" with respect to its semantics: the internally managed control flow (a continuation), and the externally running task, which has a queryable task identification. This difference is to be respected in the design of features cross-cutting the scheduler (see Section 5.5.2 for a thorough evaluation of the problem).

Scheduler Hierarchy

The scheduler singleton object basically consists of two layers: one encapsulating the scheduling strategy (`SchedStrategy`), and one providing the actual dispatching implementation (`SchedImpl`).

`SchedStrategy`

The class `SchedStrategy` stores the list of ready tasks, the running task pointer, and the thread control blocks (i.e., the continuations) configured for the tasks (see above). Since it encapsulates the strategy, it implements the function making the scheduling decisions, `schedule ()`. Furthermore, it provides functions to set and query the state of the managed tasks, including the ready tasks and the running one.

`SchedImpl`

The scheduler implementation `SchedImpl` founds on the strategy encapsulation and cares about the actual dispatching of the managed continuations. Depending on the type of dispatch, it saves the context of the currently running continuation ("real" dispatch), or switches to another continuation discarding the old context (dispatch upon termination of the old continuation).

Preemption

Task management implicitly comprises the commitment to a preemption

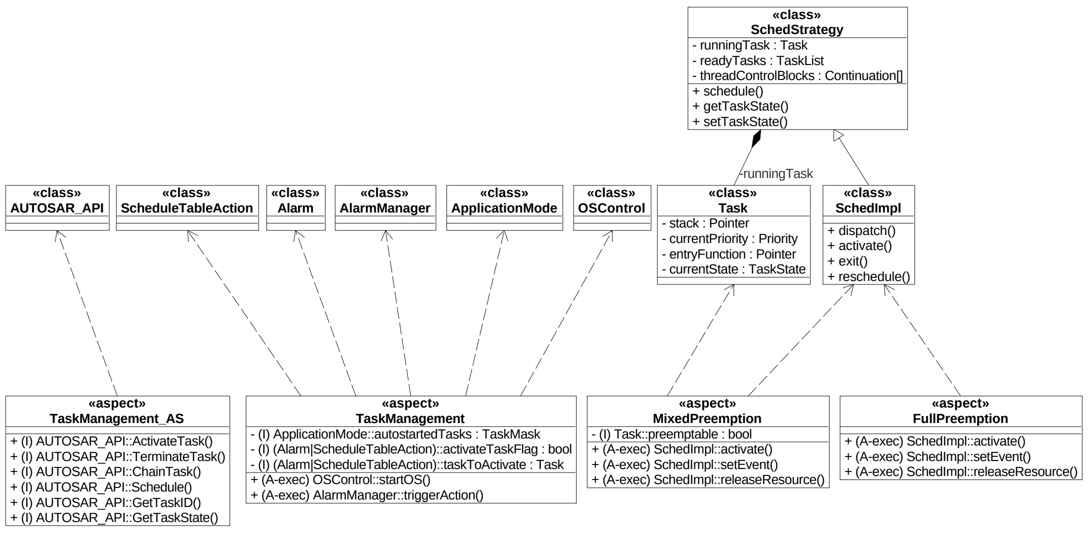
«class» SchedStrategy
- runningTask : Task
- readyTasks : TaskList
- threadControlBlocks : Continuation[]
+ schedule()
+ getTaskState()
+ setTaskState()
-runningTask
«class» AUTOSAR_API
«class» ScheduleTableAction
«class» Alarm
«class» AlarmManager
«class» ApplicationMode
«class» OSControl
«class» Task
- stack : Pointer
- currentPriority : Priority
- entryFunction : Pointer
- currentState : TaskState
«class» SchedImpl
+ dispatch()
+ activate()
+ exit()
+ reschedule()
«aspect» TaskManagement_AS
+ (I) AUTOSAR_API::ActivateTask()
+ (I) AUTOSAR_API::TerminateTask()
+ (I) AUTOSAR_API::ChainTask()
+ (I) AUTOSAR_API::Schedule()
+ (I) AUTOSAR_API::GetTaskID()
+ (I) AUTOSAR_API::GetTaskState()
«aspect» TaskManagement
- (I) ApplicationMode::autostartedTasks : TaskMask
- (I) (Alarm|ScheduleTableAction)::activateTaskFlag : bool
- (I) (Alarm|ScheduleTableAction)::taskToActivate : Task
+ (A-exec) OSControl::startOS()
+ (A-exec) AlarmManager::triggerAction()
«aspect» MixedPreemption
- (I) Task::preemptable : bool
+ (A-exec) SchedImpl::activate()
+ (A-exec) SchedImpl::setEvent()
+ (A-exec) SchedImpl::releaseResource()
«aspect» FullPreemption
+ (A-exec) SchedImpl::activate()
+ (A-exec) SchedImpl::setEvent()
+ (A-exec) SchedImpl::releaseResource()

Figure 4.4: The scheduler and classes and aspects related to task management.

policy. Non-preemptable tasks comprise rescheduling points only when explicitly requested (system service `Schedule ()`), or when not ready to run anymore. This is the default and implicitly implemented in the scheduler. Full preemption additionally makes scheduling decisions whenever the ready list of the scheduler is altered. This is to be checked after activating a task, setting an event for a task, and releasing a resource, and can be encapsulated in a single piece of advice in an aspect; since the corresponding *internal* services are advised, indirect invocations of those through the expiry of an alarm or schedule table are also comprised. A per-task setting of the preemption policy furthermore checks the characteristics of the currently running task before deciding if it is preempted. Hence, the mixed-preemption advice introduces a preemption configuration flag to every task.

OS Start-Up

If the task abstraction is provided and used by the application programmer, the system start-up routine is extended to start the scheduler at the end of the initialization. Before that, all tasks configured to be auto-started at boot time per application mode are set ready. Hence, the configuration of the application mode is extended by an auto-start task mask.

Alarm and Schedule Table Actions

When user tasks are maintained by the operating system, both alarm objects and schedule table objects are given the possibility to activate a task upon alarm expiry. Hence, this configuration is to be provided, and the alarm manager's method that is triggered upon alarm expiration is given execution advice by the task management aspect to consider the activation of a task, depending on the configuration.

All before-mentioned classes and aspects and their relationships are depicted in Figure 4.4.

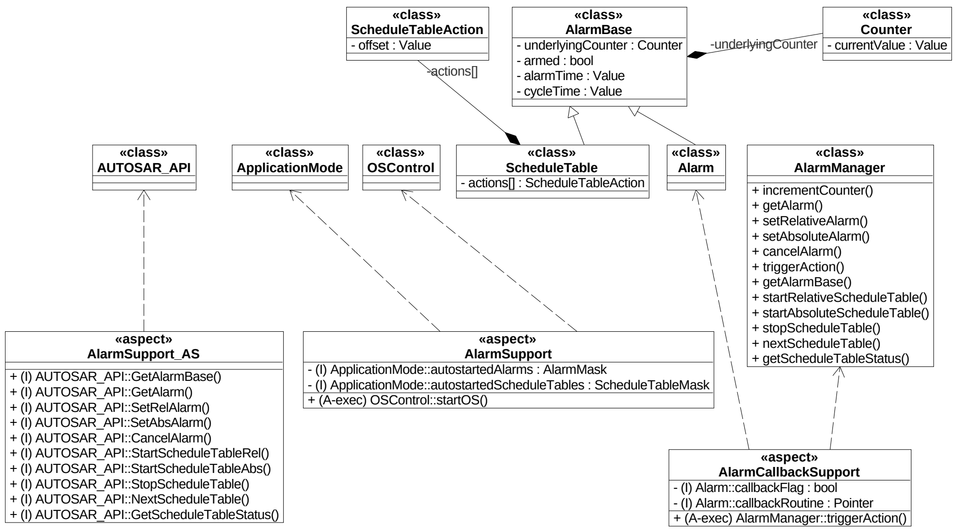
«class» ScheduleTableAction
- offset : Value
«class» AlarmBase
- underlyingCounter : Counter
- armed : bool
- alarmTime : Value
- cycleTime : Value
-underlyingCounter
«class» Counter
- currentValue : Value
-actions[]
«class» AUTOSAR_API
«class» ApplicationMode
«class» OSControl
«class» ScheduleTable
- actions[] : ScheduleTableAction
«class» Alarm
«class» AlarmManager
+ incrementCounter()
+ getAlarm()
+ setRelativeAlarm()
+ setAbsoluteAlarm()
+ cancelAlarm()
+ triggerAction()
+ getAlarmBase()
+ startRelativeScheduleTable()
+ startAbsoluteScheduleTable()
+ stopScheduleTable()
+ nextScheduleTable()
+ getScheduleTableStatus()
«aspect» AlarmSupport_AS
+ (I) AUTOSAR_API::GetAlarmBase()
+ (I) AUTOSAR_API::GetAlarm()
+ (I) AUTOSAR_API::SetRelAlarm()
+ (I) AUTOSAR_API::SetAbsAlarm()
+ (I) AUTOSAR_API::CancelAlarm()
+ (I) AUTOSAR_API::StartScheduleTableRel()
+ (I) AUTOSAR_API::StartScheduleTableAbs()
+ (I) AUTOSAR_API::StopScheduleTable()
+ (I) AUTOSAR_API::NextScheduleTable()
+ (I) AUTOSAR_API::GetScheduleTableStatus()
«aspect» AlarmSupport
- (I) ApplicationMode::autostartedAlarms : AlarmMask
- (I) ApplicationMode::autostartedScheduleTables : ScheduleTableMask
+ (A-exec) OSControl::startOS()
«aspect» AlarmCallbackSupport
- (I) Alarm::callbackFlag : bool
- (I) Alarm::callbackRoutine : Pointer
+ (A-exec) AlarmManager::triggerAction()

Figure 4.5: Entities representing the alarm management functionality.

■ 4.2.3 The Alarm Manager, Counters, Alarms, Schedule Tables, and Alarm Callbacks

The alarm functionality defined in AUTOSAR is completely encapsulated in the alarm manager singleton; it provides the complete interface to it. Both the configured underlying software counters and the system hardware counter are managed by it. Since it is possible to attach several alarms to a single counter, its main functionality is the multiplexing of those. For an overview of the entities involved, see Figure 4.5.

Hardware Alarms In the case of hardware alarms, the alarm manager uses the system timer offered by CiAO's hardware abstraction layer and sets it to the earliest deadline of all armed alarms. This information is updated whenever the user manipulates alarm properties using the alarm system services, or when an alarm expires. Upon alarm expiration, of course, the configured action is triggered, that is, the scheduler is contacted to activate the configured task (see Section 4.2.2), to set an event (see Section 4.2.7), or to activate an alarm callback (see below).

Software Alarms Alarms relying on software counters are different since the counters are directly incremented through an interface of the alarm manager; there are no timers that need to be set and that notify the expiration asynchronously. Hence, software counters and alarms can be managed completely internally in the alarm manager.

Alarm Callbacks The alarm callback extension extends the alarm structure to distinguish the configured behavior, and introduces a link to the callback function. The alarm manager is given advice in its alarm expiry method to check this behavior flag and execute the configured callback if applicable. *Unlike* the callback to the basic user hook functions (see Section 4.2.12) and *like* the application-specific error hook callback, the alarm callbacks can not be bound directly to a join point since they need to be dispatched at run time, depending on the triggered alarm.

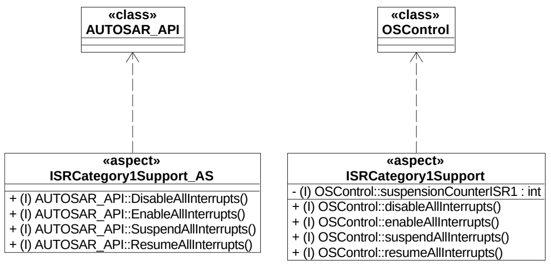
«class»
AUTOSAR_API
«class»
OSControl
«aspect»
ISRCategory1Support_AS
+ (I) AUTOSAR_API::DisableAllInterrupts()
+ (I) AUTOSAR_API::EnableAllInterrupts()
+ (I) AUTOSAR_API::SuspendAllInterrupts()
+ (I) AUTOSAR_API::ResumeAllInterrupts()
«aspect»
ISRCategory1Support
- (I) OSControl::suspensionCounterISR1 : int
+ (I) OSControl::disableAllInterrupts()
+ (I) OSControl::enableAllInterrupts()
+ (I) OSControl::suspendAllInterrupts()
+ (I) OSControl::resumeAllInterrupts()

Figure 4.6: Kernel aspects for supporting ISRs of category 1.

Schedule Tables

A more comprehensive API to the alarm mechanism is provided through the notion of *schedule tables*. A schedule table encapsulates a set of expiry points upon which a configured action is taken (i.e., activating a task (see Section 4.2.2), or setting an event (see Section 4.2.7)). This "capsule" can then be treated as one alarm-like entity by the user, and can be divided up into its single components by the alarm manager. Therefore, alarms and schedule tables share a common base.

OS Start-Up

Since the application mode determined by the user start-up routine also specifies which alarms and schedule tables are to be auto-started, this piece of configuration information is attached to the application mode structure. Additionally, the OS control start-up routine is advised to check this information and auto-start the alarms and schedule tables depending on the application mode.

■ 4.2.4 ISR Category 1 Support

Category 1 ISR Recognition

Since category 1 ISRs are implementation-specific and not managed by the operating system, the kernel is not affected by them and does not need a special synchronization mechanism. If, however, category 1 ISRs are used by the application, *the application* may need to synchronize critical sections by disabling/enabling or suspending/resuming the recognition of those interrupts.

The corresponding aspect gives advice to the OS control singleton, extending it by this functionality, which itself needs to track the nesting level in a piece of state (see Figure 4.6).

■ 4.2.5 ISR Category 2 Support

Category 2 ISR Management

ISRs of category 2 can also be suspended by the application for synchronization purposes. Furthermore, the interrupt source pertaining to an ISR can be disabled and re-enabled. The corresponding functions and the state that needs to be held to track nesting are introduced by the supporting aspect, while the ISR–source binding configuration is provided via an ISR object type.

Kernel Synchronization

Furthermore, since category 2 ISRs have comprehensive access to system services and are therefore able to manipulate kernel structures, the kernel needs additional synchronization points when ISRs of category 2 are supported. The kernel constitutes its own synchronization domain, which is marked by a control flow executing `enterKernel ()` when entering the kernel and `leaveKernel ()` when leaving it. The marking is provided via an aspect that triggers before and

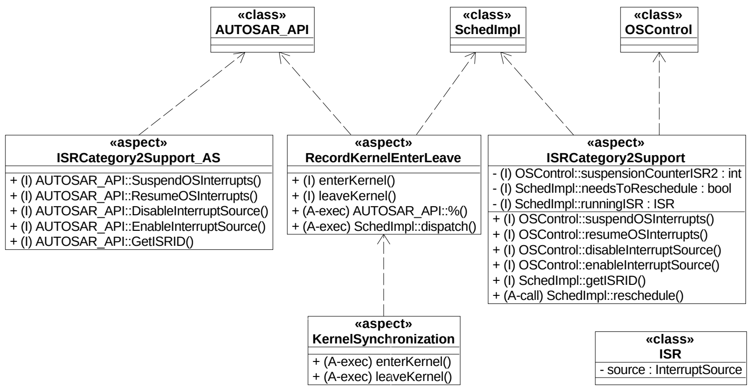
«class»
AUTOSAR_API
«class»
SchedImpl
«class»
OSControl
«aspect»
ISRCategory2Support_AS
+ (I) AUTOSAR_API::SuspendOSInterrupts()
+ (I) AUTOSAR_API::ResumeOSInterrupts()
+ (I) AUTOSAR_API::DisableInterruptSource()
+ (I) AUTOSAR_API::EnableInterruptSource()
+ (I) AUTOSAR_API::GetISRID()
«aspect»
RecordKernelEnterLeave
+ (I) enterKernel()
+ (I) leaveKernel()
+ (A-exec) AUTOSAR_API::%()
+ (A-exec) SchedImpl::dispatch()
«aspect»
ISRCategory2Support
- (I) OSControl::suspensionCounterISR2 : int
- (I) SchedImpl::needsToReschedule : bool
- (I) SchedImpl::runningISR : ISR
+ (I) OSControl::suspendOSInterrupts()
+ (I) OSControl::resumeOSInterrupts()
+ (I) OSControl::disableInterruptSource()
+ (I) OSControl::enableInterruptSource()
+ (I) SchedImpl::getISRID()
+ (A-call) SchedImpl::reschedule()
«aspect»
KernelSynchronization
+ (A-exec) enterKernel()
+ (A-exec) leaveKernel()
«class»
ISR
- source : InterruptSource

Figure 4.7: ISR category 2 support aspects and related kernel synchronization aspects.

after the invocation of an AUTOSAR system service, and after dispatching of a control flow (`leaveKernel ()`). Depending on the synchronization strategy, distinct actions to reach synchronization are performed at these points. Hard synchronization, for instance, would merely disable all interrupt recognition on entering the kernel and enable it on leaving it. This way, ISR execution is deferred.

Scheduler Adaptation

Since the kernel is aware of category 2 ISRs, it provides a service to query the ISR identifier of the currently executing ISR (`GetISRID ()`); this is stored in the adapted scheduler. When binding the user-provided ISR to the low-level interrupt handler, this state is updated accordingly before and after the actual ISR. Furthermore, if category 2 ISRs are present in a system, rescheduling is to be delayed when triggered by an ISR instead of a task. Hence, rescheduling requests from within ISRs are recorded in a boolean variable in the scheduler while simultaneously registering an asynchronous system trap (AST). Only when the ISR is left, the AST will perform the rescheduling if applicable.

The aspects configuring the kernel for the support of category 2 ISRs are depicted in Figure 4.7.

■ 4.2.6 Resource Support

Scheduler Extension

If the deployed applications want to make use of the AUTOSAR resources concept, support for it is needed in the kernel. Basically, this comprises an extension of the scheduler to provide services for a task to get and release a resource and the accompanying AUTOSAR API extension (see Figure 4.8 on the next page). Depending on the priority inversion avoidance protocol that is deployed, these services are implemented in different ways. Furthermore, different pieces of state need to be held in the OS object structures that are relevant (see Figure 4.9 on the facing page).

No Protocol

If no resource protocol is configured, the implementation is similar to a regular mutex: If the requested resource is already in use, the task is blocked; if not, it occupies it. On releasing a resource, it is marked unused and one of the blocked tasks is set ready by the scheduler (depending on the strategy) and acquires the resource. Hence, each resource OS object needs to hold its current

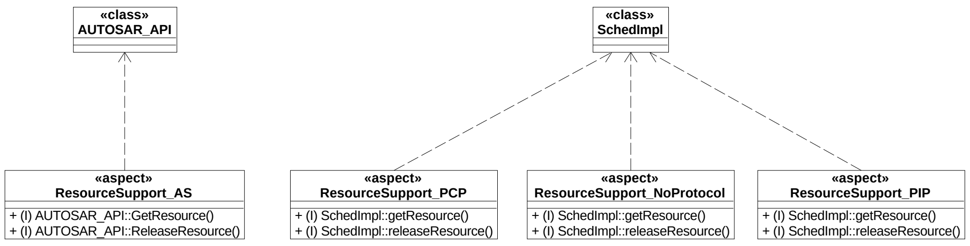
«class» AUTOSAR_API
«class» SchedImpl
«aspect» ResourceSupport_AS
+ (I) AUTOSAR_API::GetResource()
+ (I) AUTOSAR_API::ReleaseResource()
«aspect» ResourceSupport_PCP
+ (I) SchedImpl::getResource()
+ (I) SchedImpl::releaseResource()
«aspect» ResourceSupport_NoProtocol
+ (I) SchedImpl::getResource()
+ (I) SchedImpl::releaseResource()
«aspect» ResourceSupport_PIP
+ (I) SchedImpl::getResource()
+ (I) SchedImpl::releaseResource()

Figure 4.8: Resource support aspects affecting OS facilities.

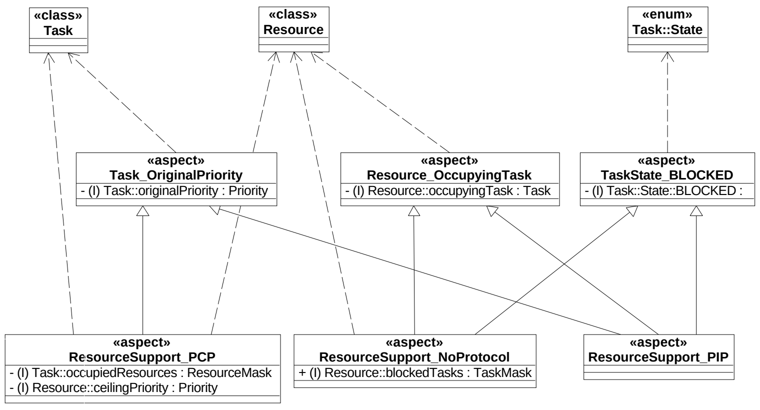
«class» Task
«class» Resource
«enum» Task::State
«aspect» Task_OriginalPriority
- (I) Task::originalPriority : Priority
«aspect» Resource_OccupyingTask
- (I) Resource::occupyingTask : Task
«aspect» TaskState_BLOCKED
- (I) Task::State::BLOCKED :
«aspect» ResourceSupport_PCP
- (I) Task::occupiedResources : ResourceMask
- (I) Resource::ceilingPriority : Priority
«aspect» ResourceSupport_NoProtocol
+ (I) Resource::blockedTasks : TaskMask
«aspect» ResourceSupport_PIP

Figure 4.9: Resource support aspects affecting OS object structures.

used state and a list of the tasks waiting for it to get released. Furthermore, an additional *blocked* state needs to be introduced to the state type of the task type.

Priority Ceiling Protocol

When the problems of priority inversion and deadlocks are addressed by deploying the OSEK priority ceiling protocol, the resource OS object needs to provide the configured ceiling priority, that is, the highest priority of all tasks that are able to acquire it. Furthermore, each task has to keep track of the resources it occupies, since its priority depends on the ceiling priorities of *all* resources currently held. This functionality is bound to the services introduced to the scheduler. Since a task can never be blocked in this scenario, there is no need for a *blocked* task state.

Priority Inheritance Protocol

If the priority inheritance protocol is chosen for resource management, it is also possible for a task to become *blocked* (hence, the introduction of this task state is needed), but not in an uncontrolled way. If the task holding the resource has a lower priority than the one requesting it, it inherits the higher priority. Thus, each resource needs to track the task currently holding it.

■ 4.2.7 Event Support

Scheduler Extension

AUTOSAR uses events for signalization to tasks. Hence, each task OS object needs to save its currently signaled events and the events it is waiting for after the execution of the system service `WaitEvent ()`. If the event is then already

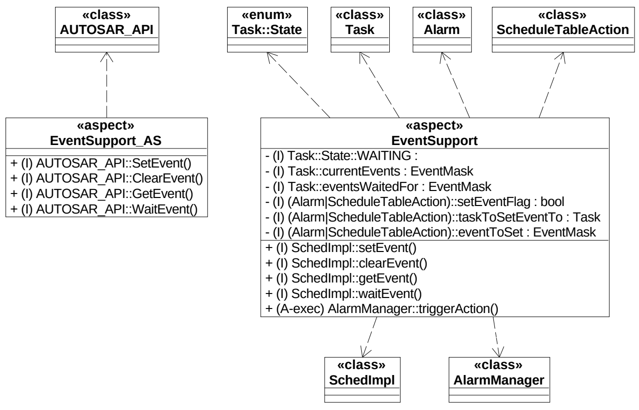
«class»
AUTOSAR_API
«enum»
Task::State
«class»
Task
«class»
Alarm
«class»
ScheduleTableAction
«aspect»
EventSupport_AS
+ (I) AUTOSAR_API::SetEvent()
+ (I) AUTOSAR_API::ClearEvent()
+ (I) AUTOSAR_API::GetEvent()
+ (I) AUTOSAR_API::WaitEvent()
«aspect»
EventSupport
- (I) Task::State::WAITING :
- (I) Task::currentEvents : EventMask
- (I) Task::eventsWaitedFor : EventMask
- (I) (Alarm|ScheduleTableAction)::setEventFlag : bool
- (I) (Alarm|ScheduleTableAction)::taskToSetEventTo : Task
- (I) (Alarm|ScheduleTableAction)::eventToSet : EventMask
+ (I) SchedImpl::setEvent()
+ (I) SchedImpl::clearEvent()
+ (I) SchedImpl::getEvent()
+ (I) SchedImpl::waitEvent()
+ (A-exec) AlarmManager::triggerAction()
«class»
SchedImpl
«class»
AlarmManager

Figure 4.10: The aspects providing the support for events.

signaled, the task returns immediately without leaving the running state, if not, it will be blocked in the *waiting* state, which needs to be added to the task state type. Waiting tasks can be unblocked by setting one of their waited-for events. This is also an extension of the scheduler, since it needs to adapt the task's state accordingly if applicable.

Alarm Manager Extension

The alarm manager's trigger action is extended to be able to set an event to a task if an alarm or a portion of a schedule table is configured that way. This configuration possibility is added to the alarm OS object type and the action type of a schedule table.

All of these different pieces of advice are encapsulated in a single event support aspect (see Figure 4.10).

4.2.8 OS Application Partitioning

Owning Application

The first consequence of deploying separate OS applications is to assign each OS object an owning application. A comprehensive introduction advice performs this addition of configuration information.

`GetApplicationID ()`

The system service to query the running application (`GetApplicationID ()`) can be designed by adding state to the scheduler that is updated at every context switch to represent the assigned application of the newly dispatched control flow. Since this service is not expected to be called frequently, the more economical design would be to query the application pertaining to the control flow dynamically on request. Hence, there is no additional state to be held by the scheduler and no need to update it at every dispatch point.

Object Ownership and Access

A similar query can also be used for the system service `CheckObjectOwnership ()`, which reveals the owning application for an OS object given as a parameter. The basic behavior of the service `CheckObjectAccess ()`, however, can be designed to return all access rights when the queried application is the owning one, and no rights in all other cases. This service is altered only when explicit access rights are introduced by a service protection concern (see Section 4.2.11).

`TerminateApplication ()`

The request for all tasks of the running application to be terminated (through `TerminateApplication ()`) is also designed by querying the application of all

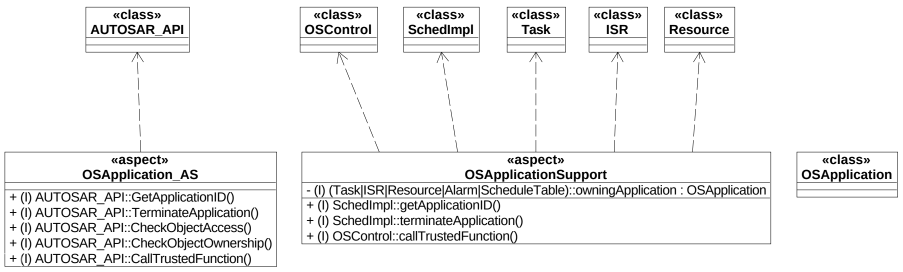
«class»
AUTOSAR_API
«class»
OSControl
«class»
SchedImpl
«class»
Task
«class»
ISR
«class»
Resource
«aspect»
OSApplication_AS
+ (I) AUTOSAR_API::GetApplicationID()
+ (I) AUTOSAR_API::TerminateApplication()
+ (I) AUTOSAR_API::CheckObjectAccess()
+ (I) AUTOSAR_API::CheckObjectOwnership()
+ (I) AUTOSAR_API::CallTrustedFunction()
«aspect»
OSApplicationSupport
- (I) (Task|ISR|Resource|Alarm|ScheduleTable)::owningApplication : OSApplication
+ (I) SchedImpl::getApplicationID()
+ (I) SchedImpl::terminateApplication()
+ (I) OSControl::callTrustedFunction()
«class»
OSApplication

Figure 4.11: The aspects for OS application support.

non-suspended tasks on demand instead of keeping this information available at all times.

Trusted Functions

Furthermore, the introduction of OS applications demands for shared code to be explicitly marked and cleared. This is done via a `CallTrustedFunction ()` service, which accesses a function table configured at compile time to dispatch the execution of a shared function. This functionality is introduced to the OS control singleton.

The introduction advice and the extension of the scheduler and OS control facility by the OS application services is encapsulated in a distinct aspect (see Figure 4.11).

4.2.9 Memory Protection and Timing Protection

Memory Protection

As pointed out in the analysis in Section 3.4.2, the AUTOSAR memory protection concern is out of the scope of this diploma thesis; the thesis by Jochen Streicher [Str07] cares about its issues and implementation in detail. The memory protection design as deployed in CiAO is also described in [LSH+07].

Timing Protection

The timing protection feature as proposed by AUTOSAR is also very comprehensive and therefore the target of future work on the CiAO system (see Section 6.2).

4.2.10 Stack Monitoring

Basic Advice

A simple implementation of stack protection is based on a run-time check of the stack pointer at context switch time. For the comparison, an additional property needs to be retrieved from the currently running task: the bottom of its stack, which can also be calculated when its stack size is given (which is a more common figure for a system designer). Hence, this figure is introduced to the task structure and the dispatch point in the scheduler is advised to perform the comparison and call the internal protection hook when the stack is exceeded.

Extended Advice

A further variant that emerged in the analysis (see Section 3.4.3) is to perform checks whenever the kernel is invoked, that is, before every system service call. Hence, the extended aspect advises the AUTOSAR API singleton.

Detectable Overflows

Note that this does not detect all stack overflows since the stack could have overflowed earlier, but is already reduced above the stack bottom at dispatch time or system call time.

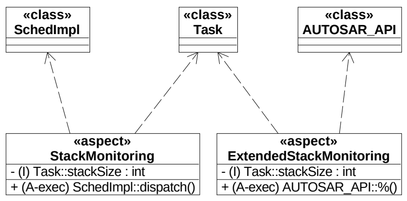
«class»
SchedImpl
«class»
Task
«class»
AUTOSAR_API
«aspect»
StackMonitoring
- (I) Task::stackSize : int
+ (A-exec) SchedImpl::dispatch()
«aspect»
ExtendedStackMonitoring
- (I) Task::stackSize : int
+ (A-exec) AUTOSAR_API::%()

Figure 4.12: The aspects performing stack monitoring checks.

See Figure 4.12 for a schematic overview of the two stack monitoring aspect's advice.

Alternative Implementation

An alternative to the implementation described and designed above is a magic pattern check. Therefore, more stack is allocated below each task's bottom of stack, and filled with a magic pattern. Then, at check time (i.e., at task switch or every system invocation), the consistency of this pattern is checked. This way, more RAM for the stacks is used, but the bottom of stack figure is needed anyway for the comparison. Hence, the join points given advice to are basically the same. This method has the advantage that overflows that have happened earlier can be detected by a modification of the magic pattern.

4.2.11 Service Protection Features

AUTOSAR API Layer

As described in Section 4.1, the application can only access the CiAO kernel through the AUTOSAR API layer. This layer is also the single point for additional plausibility checks denoted *extended status* in OSEK and *service protection* in AUTOSAR. Inside the kernel, correct identifiers, value ranges, and execution conditions are presumed, and no further assertions are made.

Advice

Therefore, all the different pieces of the service protection (analyzed in Section 3.4.5 through Section 3.4.12) give advice to the AUTOSAR API singleton only. Nevertheless, some of them need additional information that they retrieve from the kernel. Depending on the condition if the context of a service call is needed for the service protection feature (e.g., to check for a non-trusted shut-down request), call advice or execution advice is given to the API layer. Since these plausibility checks induce significant overhead, all of them are encapsulated in single aspects and are therefore configurable independently. If there are common pointcuts for different aspects, they can be shared by introducing a base aspect that the other aspects specialize; thereby, they inherit the pointcut definition.

Missing Task End Check

The single service protection aspect that does *not* advise the API singleton is the one checking for a potentially missing task end. It gives static code advice to the end of all user-defined task functions and therefore weaves in the appropriate actions to be taken when a task does not end correctly.

Related Work

In the past, several attempts were done on refactoring a system to separate exception handling concerns from the main functionality of an application or a framework [FCF$^+$06, LL00], even a dedicated error handling aspect pattern was proposed [FGR07]. Most of the work comes to the conclusion that deploying exception handling in the form of separated aspects needs the base design and code not to be oblivious but considered *a priori*. By basically restraining the join points for an exception handling to one well-defined layer, and by making

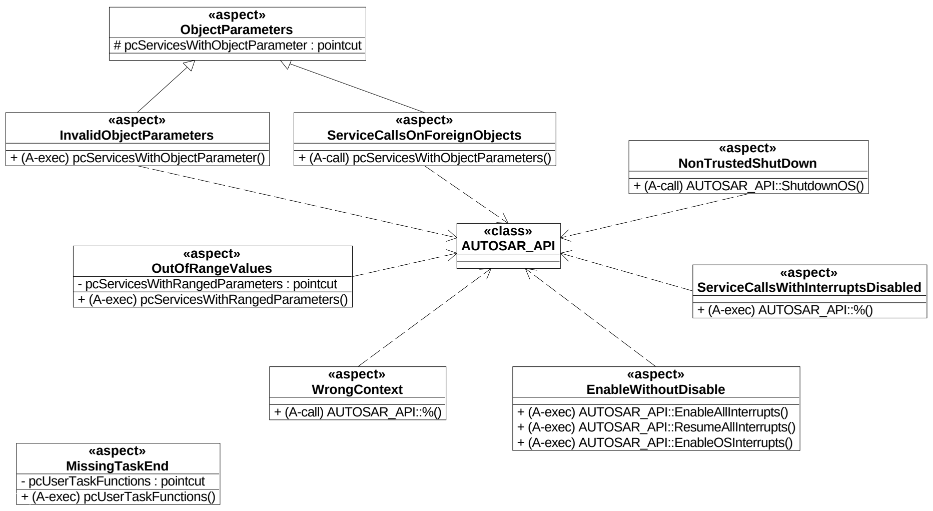
«aspect»
ObjectParameters
# pcServicesWithObjectParameter : pointcut
«aspect»
InvalidObjectParameters
+ (A-exec) pcServicesWithObjectParameter()
«aspect»
ServiceCallsOnForeignObjects
+ (A-call) pcServicesWithObjectParameters()
«aspect»
NonTrustedShutDown
+ (A-call) AUTOSAR_API::ShutdownOS()
«class»
AUTOSAR_API
«aspect»
OutOfRangeValues
- pcServicesWithRangedParameters : pointcut
+ (A-exec) pcServicesWithRangedParameters()
«aspect»
ServiceCallsWithInterruptsDisabled
+ (A-exec) AUTOSAR_API::%()
«aspect»
WrongContext
+ (A-call) AUTOSAR_API::%()
«aspect»
EnableWithoutDisable
+ (A-exec) AUTOSAR_API::EnableAllInterrupts()
+ (A-exec) AUTOSAR_API::ResumeAllInterrupts()
+ (A-exec) AUTOSAR_API::EnableOSInterrupts()
«aspect»
MissingTaskEnd
- pcUserTaskFunctions : pointcut
+ (A-exec) pcUserTaskFunctions()

Figure 4.13: The service protection aspects and their advice.

use of the properly defined AUTOSAR specification, the CiAO kernel facilitates this deployment.

Figure 4.13 sums up the advice given by the distinct service protection aspects.

Alternative Implementations

The advice code actually deployed in the service protection aspects could be held configurable as well, providing an alternative that is not AUTOSAR-compliant but more suited for some application scenarios. Some of the service protection features, for instance, can easily be implemented to detect the erroneous system service use at *compile time* instead of at *run time*. For example, the range of allowed OS object IDs given as parameters to services is already known at compile time, as well as most of the actual parameters of the corresponding service calls. Hence, it could be the *compiler* that conveys the error message, providing for a fault isolation that can not be earlier in time. Anyway, this type of error can only occur when directly using numerical identifiers for OS objects instead of the symbolical ones as suggested, or when the configuration of the application does not match the one of the OS.

■ 4.2.12 Hook Support

Basic System-Wide Hooks

The support of the four basic system-wide hooks (start-up, shut-down, pre-task, and post-task) mainly induces the designation of the specified points of execution in the kernel. The application hooks can then be bound to those points by giving execution advice to them (see also Section 4.3.2). The points are either on the boundary of an internal kernel method (and, hence, can be given *before* advice or *after* advice), or are marked explicitly using an explicit join point (see also Section 4.4.1).

Task Change Hooks

Special attention needs to be payed to the two task hooks. They can not be designed to merely be called before and after a dispatch. That is because the dispatch function is special in the way that it is not exited like any other regular function; depending on the platform-dependent implementation, the context is altered and the return address is modified such that the control flow emerges at a different point. The return from the original dispatch happens only when another dispatch *to* the original control flow occurs, but not when a task is dispatched *for the first time*. That is why the pre-task hook can not be bound to be executed after a dispatch. Hence, it is bound after the execution of the scheduler-internal method that sets a new task to be running; this way, the requirement of the specification for `GetTaskID ()` to yield the ID of the task about to execute can be fulfilled. The post-task hook, however, can be bound before the dispatch function in a straight-forward kind of way.

System-Wide Error Hook The error hook is different, since it is not activated at a single point of execution, but at multiple ones. It can be designed as an aspect giving comprehensive *after* advice to all system services that can return an erroneous status. Depending on the run-time value of that status, an internal (empty) error hook is executed, which serves as an explicit join point (see Section 4.4.1). The user error hook(s) can then be bound to that join point (see Section 4.3.2).

Protection Hook When a protection error occurs, an internal protection error handling routine is activated by the operating system. This handling routine then calls an internal protection hook, which the protection hook provided by the application is bound to in the same way the other hooks are. Depending on the return value of the user routine, the handling routine then takes the appropriate actions (e.g., kill the faulty task, or shut-down the OS).

Basic Application-Specific Hooks The application-specific start-up and shut-down hooks as defined by AUTOSAR are basically further instances of the regular hooks, just the execution order compared to the system-wide hooks is defined. An aspect giving order advice encapsulates this requirement.

Application-Specific Error Hooks Once again, the application-specific error hooks are different. Their activation points are not the same as the ones of the system-wide error hook, but are effectively a subset of those, namely the ones caused by the corresponding application. Thus, the same points as in the system-wide error hook are advised, but the user functions are not bound directly to them since they need to be dispatched at run time, depending on the currently running (and therefore error-causing) application. The application-specific error hook functions are therefore introduced as function pointers pertaining to the OS application objects.

Hence, the design of the basic application-specific hooks is different than was presumed in Section 3.5.2. This way, it is more efficient and does not induce the introduction of additional configuration information to the OS application type.

Figure 4.14 on the next page summarizes the advice of the hook support aspect.

4.3 Aspect-Oriented Design Schemes

Having described the design of the CiAO kernel, it is possible to extract design schemes that are re-used in several related situations; they make use of aspect-oriented constructs in order to reach their goal.

4.3.1 API Slices and OS Object Type Slices

Slice Introduction In aspect-oriented programming, slices are used to introduce members bearing

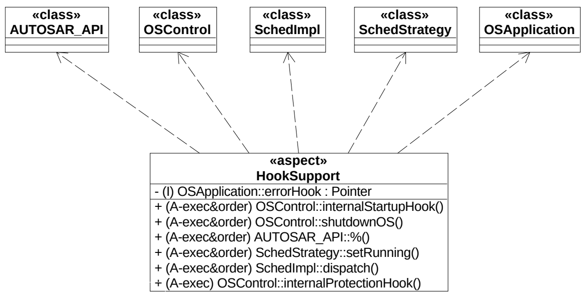
«class»
AUTOSAR_API
«class»
OSControl
«class»
SchedImpl
«class»
SchedStrategy
«class»
OSApplication
«aspect»
HookSupport
- (I) OSApplication::errorHook : Pointer
+ (A-exec&order) OSControl::internalStartupHook()
+ (A-exec&order) OSControl::shutdownOS()
+ (A-exec&order) AUTOSAR_API::%()
+ (A-exec&order) SchedStrategy::setRunning()
+ (A-exec&order) SchedImpl::dispatch()
+ (A-exec) OSControl::internalProtectionHook()

Figure 4.14: The hook support aspect and its pieces of advice.

state (member variables) and code (member functions) into classes. Concerns are better encapsulated when state and code that does not *conceptually* belong to the class it *technically* has to be attached to is introduced in an aspect, which may also contain other advice representing that concern and affecting the rest of the system.

CiAO API Slicing

The API of CiAO that is offered to the user (see Section 4.1) is also composed of slices. If support for a specific feature of the OS kernel is encapsulated in an aspect (e.g., support for events), the extension of the user API is also performed in it. Hence, both the modifications of OS internals and the adaptation of the API are encapsulated in one module: the corresponding aspect. This way, not only are the concerns clearly separated, but also errors resulting from trying to combine an application with an incompatible configuration are detected early at compile time since services that are not configured are not reflected in the API either. Thus, calls to services that are not enabled in the configuration result in a compiler/linker error.

OS Object Type Slicing

Another use of AOP slices in CiAO are introductions made to OS object types like task or alarm structures. Every configurable feature that needs to store (and retrieve) additional state in an OS object bears slice introduction advice to the target OS object structures. Since only fields needed in the current configuration are reflected in the structures, the objects are as memory-efficient as possible with the code base being separated by concerns at the same time.

■ 4.3.2 Up-Call Binding

ISRs and Hooks

At some points, the kernel needs to execute application code other than the tasks that are scheduled. This additional application code comprises interrupt service routines (ISRs), and hook routines, which are called upon specific events, and corresponds to an up-call from the OS into the application. Since an up-call breaks the common call direction in the layered CiAO system (see Section 4.1), it is designed in a special way.

Aspectual Binding

In CiAO, the binding of the user routines to the operating system is performed through several *binding aspects*. A binding aspect advises a specific explicit join point in the kernel to execute the configured user routine (for an example, see Figure 4.15 on the following page). The induced pattern is an *aspect-oriented, static observer pattern*. The designated, explicit join point constitutes the publisher part, while the binding aspects represent the subscribers

```
#include "user_hooks.h"

aspect PreTaskHook {
    pointcut internalPreTaskHook () =
        "% os::krn::SchedImpl::internalPreTaskHook (...)";

    advice execution (internalPreTaskHook ()) : before () {
        UserPreTaskHook ();
    }
};
```

Figure 4.15: An example hook function binding aspect in AspectC++.

```
aspect os_krn_PreTaskHook {
    advice "os::krn::SchedImpl" : slice class {
    public:
        internalPreTaskHook () {
            // empty, aspects bind here
        }
    };

    pointcut setRunning () =
        "% os::krn::SchedStrategy::setRunning (...)";

    advice execution (setRunning ()) : after () {
        os::krn::SchedImpl::Inst ().internalPreTaskHook ();
    }
};
```

Figure 4.16: An example OS hook introduction aspect in AspectC++.

(observers) of that join point. The pattern effectively decouples the provider of an event and its bound consumers, which get notified upon the occurrence of the event.

Classic Observer Pattern

The classic observer pattern as described by Gamma et al. [GHJV95] is different in the way that it is both object-oriented (compared to the aspect-oriented version here) and dynamic; that is, the constellation of observers can change at run-time. This is not the scope of the posed scenario, though, and besides would only be possible with dynamic aspect weaving.

Binding Join Points

The explicit binding join points (see also Section 4.4.1) are themselves introduced in OS aspects that weave the join points to the places in the operating system that they are supposed to be called (see Figure 4.16). Since only designated explicit join points are used in application binding aspects, they define an additional, stable interface to the operating system, which bears an internal structure that is subject to change.

Advantages

This binding scheme has several advantages over a traditional (procedural or object-oriented) approach using external symbols to reference a user function:

1. The user routine does not have to be named like the operating system

implementor prescribes it. (The AUTOSAR specification, for instance, defines the hook functions to have standardized names.) Traditionally, a linker program resolves the external symbols and inherently does that by referencing the *name* of the symbol. The advice given by the binding aspect can be adjusted to the application, though, and can be generated out of a user-supplied configuration.

2. Advice code can be inlined. If the woven routine is considered suitable for inlining by the compiler, it is directly embedded into the kernel at the designated join points. Binding using external function symbols always results in a function call with the associated overhead, even if the user function is very short and even if it only referenced at a single point (which is the case with all hook routines and ISRs).

3. More than one user function can be bound to a join point. This is done independently from other binding aspects affecting the same join point, thereby effectively facilitating the integration of different applications with their specific hook and interrupt service routines. No further integration combining the different functions is needed—which would be the case with traditional binding mechanisms—, and all of them can be inlined if applicable. This feature is especially useful when binding ISRs to interrupt sources: If several devices share one interrupt line, their handlers are implicitly chained—oblivious of the other devices and their handlers. If the execution of the handlers shall be adherent to a specific order, additional *order advice* can be given to describe that order.

■ 4.4 Aspect-Oriented Implementation Schemes

Besides common schemes that can be identified in the kernel *design*, there are also common aspect-oriented techniques that are deployed in the *implementation* of the kernel.

■ 4.4.1 Explicit Join Points

Semantical Importance

When designing an aspect-aware operating system from scratch, an important structural feature is the concept of an *explicit join point*. An explicit join point is a point in the execution control flow inside the operating system that bears a semantical importance. It is therefore necessary to define its meaning and context in a precise and explicit manner.

Deployment Reasons

Explicit join points are deployed for three different reasons, and *only* for those. (In general, the kernel is designed in an *aspect-aware* way to provide the necessary join points (see also Section 1.3).)

1. The first one is the aim to provide join points of importance without the need to artificially split a method. Since code advice can only be given on method and call boundaries, this would otherwise be necessary.

2. The second reason is the provision of a stable and explicit "join point interface" to the applications. Since applications can bear well-defined aspects (see Section 4.3.2), the join points they are allowed to bind to are clearly specified and documented. As the internal structure of the

operating system is subject to change, system-internal aspects perform the transformation of internal to external, explicit join points.[2]

3. The third reason is a very technical one: In operating system implementations, there are points that can not be reached by giving regular advice to methods. An example is the OS-internal dispatch function, which cannot be given correct *after advice* since it does not return in the control flow it was called within. Another example includes low-level functions written in assembly, which can not be advised directly but call explicit join points instead, which themselves then *can* be advised.

Technical Realization From a technical point of view, explicit join points are calls to functions with empty bodies, where aspects can give advice. The introduction of explicit join points into the operating system code for the second reason (to provide a stable interface) can be realized by an aspect itself (see, e.g., the introduction of explicit join points for hook function binding in Section 4.3.2).

State Transitions Explicit join points are mainly used for parts in the kernel where interesting state transitions take place that aspects encapsulating independent concerns may be interested in. Such an example transition can be the dispatching of continuations (OS-internal on low level) or of tasks (on high level of the AUTOSAR interface). The former deploys explicit join points for the third reason, while the latter does so to provide a stable interface (reason two); both of them are described in the following.

`Continuation` The class `Continuation` comprises four explicit join points with distinct semantics:

1. `before_CPURelease ()` is called immediately before dispatching to another continuation. The bound advice is guaranteed to be running in the context of the old continuation.

2. `before_LastCPURelease ()` is called immediately before terminating a continuation control flow. It is also called in the old context.

3. `after_CPUReceive ()` is called before the first instruction of a newly dispatched (probably continued) continuation. The bound advice runs in the context of the new control flow.

4. `after_FirstCPUReceive ()` is called before the very first instruction of a started continuation control flow and also runs in the new context.

`SchedImpl` The scheduler class `SchedImpl` bears similar join points for transitions on task level:

1. `internalPreTaskHook ()` is called whenever a high-level task is about to be executed. Advice that binds to that method executes in the high-level context of the newly dispatched task (i.e., querying its ID returns the ID of the task that is about to run).

[2]An alternative approach would be to provide the applications with a named pointcut that is then adapted on internal structure changes. However, this would be inconsistent with the explicit join points deployed for the first or the third reason—the application should not have to use different concepts if it wants to advise a particular join point in the system.

2. `internalPostTaskHook ()` is executed right before the suspension of a high-level task, which can occur upon termination or because of a scheduling decision leading to a preemption. The high-level execution context is the one of the old task.

Customer Aspects

Customer aspects that make use of these explicit join points include the memory protection aspects and stack monitoring aspects on low level, and the binding of the user hook functions on high level (see also Section 4.3.2).

Other Explicit Join Points in CiAO

More explicit join points deployed in CiAO include the points where the kernel is entered and left again (e.g., needed for memory protection and kernel synchronization), and explicit initialization points for the different CiAO layers, where each component provides an initialization aspect binding the initialization routine statically (also a static observer pattern as described in Section 4.3.2). They are not described further, however, since they are similar to the ones already discussed above.

4.4.2 Generic Advice

Definition

Advice in AOP is called *generic advice* if its implementation depends on join-point-specific static type information [LBS04]. This type of genericity is needed by the aspect-oriented implementation of CiAO in several places since it allows for the adaptation of advice to the context it is applied to (i.e., the specific join point). The presented examples include the aspect encapsulating the service protection concern checking invalid object parameters (see also Section 3.4.5) and the aspect mapped from the OS application concern (see also Section 3.4.1).

Generic Advice and AspectC++

Kniesel and Rho define aspect genericity to be the ability to concisely express aspect effects that vary depending on the context of a join point known at weave time, *without falling back to run-time reflection* [KR06]. Since AspectC++ in combination with C++ allows for static typing and provides the context of a statically known join point also at weave/compile time, it can fulfill these criteria. Full aspect genericity is especially important in the investigated domain of operating systems since this piece of system software is expected to be especially efficient. A type of genericity that is only resolvable at run time can lead to a significant and non-acceptable overhead of an operating system. Thus, aspects that can be formulated generically in AspectC++ can be widely re-used while still being feasible for overhead-sensible domains like operating system implementations.

4.4.2.1 Generic Advice for Invalid Object Parameters

Non-Generic Implementation

In AUTOSAR, potential invalid object parameter types passed to system services include the OS object types of a task, a resource, an application mode, an alarm, an ISR, a counter, a schedule table, and an OS application. Without the ability to exploit static type information, all of these affected system services need to be given separate advice—depending on their OS object parameter type—since the range of allowed values differs.

Parameter Types

An implementation giving generic advice can make use of the static information available about the advised join point in form of the type of the first parameter of the corresponding system service. If furthermore the allowed range of values is made statically available for each OS object parameter type, the implementation can be kept completely generic while still resolvable

```
aspect ServiceProtectionInvalidObjectCheck {
    pointcut affectedServices () = [...];

    advice execution (affectedServices ()) : around () {
        if ((* tjp->arg<0> ()) > OSObjectTraits<Arg<0>::Type>::MAXVALUE) {
            invalidateResult<Result> (tjp->result ());
        } else {
            tjp->proceed ();
        }
    }

    template <typename T>
    void invalidateResult (T *result) {
        // not implemented for the general case
    }

    template <>
    void invalidateResult<StatusType> (StatusType *result) {
        // return error code
        *result = E_OS_ID;
    }

    template <>
    void invalidateResult<void> (void *result) {
        // do nothing, just return
    }

    template <>
    void invalidateResult<AccessType> (AccessType *result) {
        // return no memory access rights
        *result = NO_MEMORY_ACCESS;
    }

    template <>
    void invalidateResult<ObjectAccessType> (ObjectAccessType *result) {
        // return no access rights
        *result = NO_ACCESS;
    }
};
```

Figure 4.17: The service protection aspect checking for invalid object parameters using generic advice, implemented in AspectC++.

statically at compile time. Figure 4.17 depicts an implementation sketch in AspectC++, which is explained in the following paragraphs.

Traits Class Since the types that are to be distinguished—the OS object parameter types—are not full classes, but rather enumerations, the necessary static information can not be added directly to them. A further indirection using a *traits class* [Mye96] is to be introduced, which contains the maximum allowed parameter value, and which is specialized for every affected OS object parameter type. The resulting traits class is shown in Figure 4.18 on the next page.

```
template <typename T>
class OSObjectTraits {
public:
    // general case
    enum {MAXVALUE = 0;}
};

template <>
class OSObjectTraits<TaskType> {
public:
    enum {MAXVALUE = cfOS_NUMBER_OF_TASKS - 1;}
};

[...]

template <>
class OSObjectTraits<AlarmType> {
public:
    enum {MAXVALUE = cfOS_NUMBER_OF_ALARMS - 1;}
};
```

Figure 4.18: The traits class necessary for the service protection aspect checking for invalid object parameters using generic advice.

Return Type

Additionally, depending on the return type of the affected system service, an error status code is to be returned or not (in case of a `void` service), or another type-dependent value is to be returned when an invalid parameter is detected. Since the static type of the join point return value is available via the Aspect++ join point API, this behavior can also be discriminated statically at compile-time.

■ 4.4.2.2 Generic Advice for OS Application Partitioning

OS Object Properties Retrieval

The aspect for OS application partitioning, amongst others, introduces an owning OS application to the OS objects managed by the kernel (see also Section 4.2.8). The interface to retrieve properties of OS objects in CiAO is static, and dispatches to the desired OS object *instance* by evaluating an ID parameter (see Figure 4.19 on the following page for an abbreviated example). The identification parameter type, however, depends on the OS object type, of course; that is, the static task services expect a task ID parameter, while the resource services expect a resource ID parameter, for instance.

Generic `Id` Type

The static `getApplication ()` method to be added with the owning OS application field, hence, has to bear different parameter types depending on the OS object type that it is introduced to. This adaptation to the advice context is accomplished by a generic piece of advice; the method slice has a generic `Id` type parameter to overcome this problem. Therefore, the OS object types that are generically advisable have an internal `Id` type definition evaluating to the identification type of that particular OS object type; this local `Id` type is part of the common interface of all OS object types. This way, such aspects can be kept completely generic and re-usable (see Figure 4.20 on the next page).

```
struct Task {
    typedef KernelTaskType Id;

    Priority priority_;

    static Priority getPriority (Id of) {
        [...]
    }

    [...]
};
```

Figure 4.19: Example of the internal interface to OS object properties.

```
slice struct OSApplicationSupport {
    os::krn::Application::Id application_;

    // Id must be defined in the OS object structure
    static os::krn::Application::Id getApplication (Id of) {
        [...]
    }
};
```

Figure 4.20: AspectC++ slice that is introduced to OS object types with OS application support.

4.5 Summary

Kernel Design The CiAO kernel is designed below an artificial AUTOSAR layer, which provides the standardized API and is the target of the service protection features. The internal kernel itself has three main components, namely the scheduler, the OS control facility, and an alarm manager. All other configurable features advise these components to provide further or alternative behavior in an encapsulated way—they are therefore mostly designed by aspects with a few supporting classes.

Schemes The identified common schemes used in the design include slice introductions to improve concern encapsulation, and an aspect-oriented up-call binding design, which provides an overhead-free but concern-separated mechanism to invoke the application from the operating system. Implementation patterns commonly made use of comprise the concept of explicit join points to provide for aspect awareness, and generic advice in order to keep advice applicable even when targeting slightly different contexts.

CHAPTER 5

Evaluation

This chapter presents an evaluation of both the design of the CiAO kernel (see Chapter 4) and its implementation. In Section 5.1, the conditions for the evaluation in Section 5.2 and Section 5.3 are listed; this includes an investigation of CiAO's memory footprint and some facets of its performance, respectively. Section 5.4 takes a look at the pros and cons of AOP in operating system engineering, whereas Section 5.5 traces aspects in the different stages of the CiAO engineering process.

5.1 Evaluation Environment

Environment

The CiAO kernel was prototypically implemented for the Infineon TriCore platform, the concrete derivative used is the TC1796b. The back-end compiler for the ISO C++ code generated by AspectC++ used is `tricore-g++` by Hightec, version 3.4.3.1; all sources were compiled with the following options: `-O3 -fno-rtti -funit-at-a-time -ffunction-sections -Xlinker --gc-sections`. Hence, all figures listed in this chapter are the results of measurements of the CiAO AspectC++ code compiled for the TC1796b by the Hightec toolchain. The execution-time figures were measured using a Lauterbach hardware debugger with a trace unit.

Implemented Features

The following of the analyzed and designed features are currently implemented:

- OS control facility
- Task management
- Support for ISRs of category 1
- Support for resources with the OSEK priority ceiling protocol
- Support for events
- Support for alarms
- Preemption policies: non-, mixed-, and fully-preemptive

- Stack monitoring
- Support for service protection (without features relying on OS application partitioning)
- Support for alarm callbacks
- Support for hooks

5.2 Memory Footprint Scalability

Footprint Analysis The memory footprint of a fully linked CiAO system in different feature configuration variants was investigated in order to get a notion of its scalability. The analysis was performed using the map file produced by the linker and by comparing disassembled code parts where necessary; the results are depicted in Table 5.1 on the facing page.

Investigated Data The analysis includes all code and data put into the text, data, and BSS segments. It does not include the debug information, of course; neither does it respect the stack usage, which will eventually be important to be fully and fairly comparable to other AUTOSAR OS implementations (see also Section 5.3 for a trade-off example involving stack usage).

The following sections shortly explain the caveats associated with each tested feature and discusses the results, highlighting problems to be tackled in the future work on optimizations (see Section 5.2.11) where necessary.

5.2.1 General Caveats and Observations

Function-Level Linking Since the CiAO build process makes use of the compiler's feature of function-level linking, services that are not used are not reflected in the target binary image. Hence, in order to measure the text size of additional functions, they either need to be referenced or attributed for the compiler to keep. Where not stated otherwise, the figures reflect the *full implementation* of the particular feature.

Configuration Dependency Some of the data types (mostly identifiers) used in the OS are of variable length (before compile time); their actual length depends on the number of different system abstractions of that type supported in the OS configuration. The table always shows the memory usage with the *widest data type* possible.

Alignment A lot of the figures actually depend on the alignment of the affected data. Thus, if a feature introduces a field to a structure, it depends on the fields allocated before it if the new field will already be aligned or needs fill bytes. Hence, the *worst-case figures* are reflected in the table.

Aligned Data Access Alignment also has an effect on the code generated by the compiler, when accessing a field in a structure, for example. If the size of the structure is a power of 2, the compiler can generate more efficient code for the TriCore architecture without having to copy the addresses to data registers; therefore, depending on the features available in a configuration, different text segment sizes can be observed although only a data structure type was advised. Here also the *worst-case figures* are listed.

Section	Unit	Sub-Unit	Text	Data	BSS
5.2.2	**Base Size (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
5.2.3	ISR Cat. 1 Support		0	0	0
		per ISR	+ ISR function size + 0	0	0
		per enable () / disable ()	+ 4	0	0
5.2.4	Resource Support		+ 128	0	0
		per Resource	0	+ 4	0
		per Task	0	+ 8	0
5.2.5	Event Support		+ 280	0	0
		with Alarm Support	+ 54	0	0
		per Task	0	+ 8	0
		per Alarm	0	+ 12	0
5.2.6	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
5.2.7	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
5.2.8	Stack Monitoring		0	0	0
		per Join Point	+ 44	0	0
		per Task	0	+ 4	0
5.2.9	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
5.2.10	Error Hook		0	0	+ 4
		per Join Point	+ 54	0	0
	Start-Up / Shut-Down Hook		0	0	0
	Pre-Task / Post-Task Hook		0	0	0

Table 5.1: Scalability of CiAO's memory footprint (text, data, and BSS segments in bytes).

5.2.2 OS Control and Task Management

Base Functionality The CiAO base image used for the evaluation comprises about 3 kilobytes. However, this image not only comprises the task management and OS control features of the kernel, but also facilities like the start-up code and several debug facilities including several assertions, debug abstractions, and, first and foremost, a `printf ()` implementation of the used `libcmini`. Hence, this figure only serves as a base size; all other figures are to be added to the base size depending on the configuration.

Alarm Support The task management feature cross-cuts the alarm support feature (if available) in its action trigger method, leading to the 32 bytes of text segment augmentation.

Tasks and Application Modes Every configured task's function is put in the text segment, its stack in the BSS segment. 20 bytes of data are necessary for the task structure (priority, state, function, stack, interrupted flag), and 16 bytes (BSS) per continuation control block (saved PCXI register, return address, start function, task ID). Since tasks and continuations are basically different concepts (see Section 4.2.2), this reflects the simplified assumptions that every task is configured to be linked to one continuation. Each application mode adds 4 bytes to the data section (the auto-started tasks mask), whereas 8 bytes are added to it per alarm (if configured; task to activate and the corresponding flag).

5.2.3 Category 1 ISRs

ISR Cat. 1 Binding The binding of category 1 ISRs does not induce any costs—since the user ISR is inlined, the image size only increases by the size of the user handler code.

ISR Cat. 1 Management The ISR category 1 management functions (`EnableAllInterrupts ()`, and `DisableAllInterrupts ()`) are inlined and take up 4 bytes each (for the `enable` and `disable` instruction, respectively), as expected.

5.2.4 Resources

The implementation of the resource services and the OSEK PCP protocol takes up 128 bytes of code, and additionally 4 bytes per resource and 8 bytes per task in order to store the ceiling priority, and the currently acquired resources and the original priority, respectively.

5.2.5 Events

Event Management The implementation of the event support feature takes up 280 bytes in total. Besides the implementation of the four event-related AUTOSAR services, this includes the introduction of a block possibility for tasks in the scheduler (when waiting for an event), and the modification of the internal schedule function to check if waiting tasks have become ready to run.

Alarm Support In combination with alarm support, the alarm trigger function is advised at a cost of 54 bytes in the text segment.

Tasks and Alarms Additionally, the task and alarm OS object structures are enlarged by 8 bytes and 12 bytes, respectively, to accommodate the current and waited-for event masks (task structure), and the event setting flag and the event and task to set the event to (alarm structure).

■ 5.2.6 Alarms

Hardware Alarm Management

The current implementation only supports hardware counters and therefore hardware alarms. It needs 568 bytes in the text segment and 24 bytes in the BSS segment since it not only comprises the alarm manager, but also the HAL abstraction for the required system timer with its initialization routines, and associated low-level and higher-level IRQ handlers. Nevertheless, since this figure is strikingly high, it is to be examined further (see Section 5.2.11).

Alarms and Application Modes

Besides this fixed overhead, each deployed alarm comprises a structure of 16 bytes in the data segment (ticks per base, armed flag, trigger time, cycle time). Furthermore, the application mode structure is appended a field for the auto-started alarms (4 bytes).

Alarm Callback Support

Alarm callback support needs each alarm OS object to store a flag if this instance is to activate a callback function, together with the address of the function (8 bytes per alarm in the data segment). The trigger function that is advised to discriminate the configured behavior of the expired alarm at run time is added 24 bytes of code.

■ 5.2.7 Preemption

Preemption Points

The selection of a distinct preemption policy other than the implicit non-preemptive one does not lead to a global overhead per se; the overhead depends on the number of relevant additional *points of rescheduling*, which serve as join points for the preemption policy aspects. These rescheduling points comprise the internal representations of the services `ActivateTask ()`, `ReleaseResource ()`, and `SetEvent ()`. Hence, the total overhead depends on the system abstractions available in the configuration; per join point, it is 12 bytes and 44 bytes for full preemption and mixed preemption, respectively.

Mixed Preemption Figure

The latter figure is to be explained by the need to query the running task and then check the preemptable flag in the task structure to decide whether to reschedule or not; furthermore, the compiler generates inefficient and partly redundant code. Both contributions to the figure are subject to further optimizations (see also Section 5.2.11).

Tasks

The mixed preemption aspect additionally adds 4 bytes to the task structure: the preemptable attribute.

■ 5.2.8 Stack Monitoring

Stack monitoring was measured using an implementation that simply compares the current stack pointer to the bottom of stack pertaining to the running task at specific points of execution (e.g., the dispatch to another task) and shuts down the OS if applicable. Currently, this takes up 44 bytes per join point due to the access to the running task pointer, and the access to the stack bottom property in the running task's structure (which adds 4 bytes to it). Since this figure seems unacceptably high for this simple feature implementation, it provides a point to examine for systematic optimization possibilities (see Section 5.2.11).

5.2.9 Service Protection

Most of the service protection features were implemented, leading to a wide range of overhead from none to distinctly noticeable.

Wrong Context The aspect checking for calls from wrong context merely substitutes the call by nothing (if the system service is `void`), or by the returning of an appropriate error `StatusType` (if the service returns a `StatusType`). The first one does not induce any costs, while the latter costs 8 bytes per join points. A closer look at the disassembled code shows that this is because the compiler unnecessarily uses the stack as an intermediate storage. This fact itself is due to the way AspectC++ stores join-point context information in a structure, and is subject to further optimizations either at the side of the compiler or the weaver (see Section 5.2.11).

Calls with Interrupts Disabled If the interrupt enable status is checked at each service call, each join point is advised additional 64 bytes of code. Again, this is because of unnecessary stack usage, and the figure could be much lower.

Enable Without Disable The check for an enable without a prior disable, however, only induces 14 bytes in the text segment of the image.

Missing Task End When the system is configured to check for a missing task end, this comprises 58 additional bytes at the end of each task. This includes the code to enable OS interrupts and all interrupts if necessary, and to end the task in a controlled manner. Since there is no potential for big optimizations like in some of the other cases, this feature is to be enabled carefully.

Out of Range Values The out-of-range values check only refers to the services `SetRelAlarm ()` and `SetAbsAlarm ()` since schedule tables were not implemented. Both services are join points for the corresponding aspect, which induces 152 bytes of code to each of them. This overhead is almost completely attributable to badly generated code and shows the need to optimize the compiler/weaver toolchain for these situations (see also Section 5.2.11). Additionally, the alarm structure comprises two more fields for the maximum allowed value and the minimum cycle value, totaling to 8 bytes per deployed alarm.

Invalid Object Parameters The aspect checking for invalid object parameters costs 36 bytes per join point, the join points comprising all services having an OS object parameter (see Section 3.4.5). The unpredictedly high cost again results from stack usage by the compiler.

5.2.10 Hooks

No-Overhead Binding The up-call binding of the system-specific start-up, shut-down, pre-task, and post-task hooks and of the application-specific start-up and shut-down hooks does not lead to any overhead at all since they are directly inlined to the appropriate join points.

Error Hook Binding The error hook can also be inlined; however, an additional check for the return value of the advised join point is necessary. The current implementation needs 54 bytes for that. This is for the same reason as stated in Section 5.2.9; the compiler produces inefficient code out of the ISO C++ code generated by AspectC++. First and foremost, this again comprises comprehensive but unnecessary usage of the stack.

Error Hook Flag The 4 bytes in the BSS segment introduced by the error hook aspect pertain to a flag tracking if an error hook is currently executing, necessary to avoid recursive error hook invocations.

5.2.11 Conclusions and Optimizations

Scalability

The analysis of the code and data size induced by the different features shows the scalability of the aspect-oriented configuration approach of CiAO. Depending on the features needed, the target image size will be rather small or big.

Selective Advice

However, it was also observed that some features induce significant costs on a per-join-point basis (e.g., the missing task end check; see Section 5.2.9). Therefore, it would be desirable to enable the advice encapsulated in an aspect more selectively; for instance, a system integrator could wish to apply most of the service protection features only to selected, untrusted applications. Though this is not specified by AUTOSAR, it is easy to implement by means of AOP by merely adapting the aspect pointcuts to reflect untrusted code only (this would best be implemented using annotations, though; see Section 5.4.5).

Fine-Grained Configurability

The *scalability* of the image sizes of a CiAO system also justifies its goal of *fine-grained configurability* (see also Section 1.1). Every feature induces costs that need to be justified in embedded systems development and therefore examined on a trade-off basis. By providing the ability to select features on a fine-grained level, CiAO enables the embedded systems developer to take such a decision. This includes the adherence of the system to the AUTOSAR standard versus what makes sense in a particular scenario or even at all[1].

Bit Fields vs. Structures

A further point that needs to be taken care of is the storage of information in structures. As was noticed during the measurements, depending on the size of relevant structures, code addressing those can be optimized or not. Hence, a first step would be to order slices accordingly so that an ideal internal structure is reached. Going a step further involves the use of *bit fields* instead of structures to allow for the allocation of only the exact amount of memory needed for it. However, a trade-off is involved: The data memory that is saved might be over-amortized by the more sophisticated code needed to address it (which will also take more time to execute). Hence, this optimization should be kept configurable, depending on the needs for execution time, code size, and RAM usage of the system developer.

Frequently Accessed Data

When taking a look at some of the assembly code, the frequency of occurrence of code patterns to access common data is striking. When determining the currently running task, for instance, the sequence of constructing the address of the (singleton) scheduler object and then accessing the member variable bears a lot of overhead for such a frequently accessed piece of data. Therefore, these kinds of data need to be figured out, and can then located in compiler-optimized places such as small data sections or even global registers. This is expected to both decrease the memory footprint and increase the perceivable performance to some extent.

Tool-Chain Optimization

The most important conclusion drawn from the memory footprint analysis,

[1]Consider, for instance, the service protection feature that checks for a prior enable before executing a disable service. While this makes sense with `ResumeAllInterrupts ()` and `ResumeOSInterrupts ()` in order not to corrupt the internal suspension counters, it does *not* with `EnableAllInterrupts ()`. The code-size and run-time overhead induced by the check for the CPU enable bit is completely unnecessary since an `EnableAllInterrupts ()` with interrupts already enabled does not lead to an erroneous state, especially since this “fault” situation is not reported but ignored.

though, is the need for an optimized compiler tool chain. At too many points, the generated assembly code is inefficient, sometimes with unnecessary usage of the stack, which is especially valuable in embedded systems design. Therefore either the compiler or the weaver are to be optimized with respect to this situation (see also Section 5.4.5).

5.3 Execution-Time Comparison with Other OS Kernels

Selected Benchmarks

In order to get a nition of the overhead induced by the aspect-oriented implementation of an AUTOSAR-like operating system, selected execution times were measured in an application deployed on both a CiAO system and a commercial OSEK implementation (ProOSEK by EB). The OSEK specification gives hints for comparable benchmarks for OSEK implementations [OSE05a, p. 71]; out of those, two basic operating system benchmarks were selected: the start-up time, and the task switch time. Although the interrupt latency is another basic embedded operating system characteristic, it was not examined here since it was already investigated in [LSSSP07]. The overhead for the CiAO memory protection mechanism was measured and discussed in [LSH+07].

5.3.1 Benchmark Setup

Task Switch Scenarios

For both benchmarks, both CiAO and ProOSEK were configured with the minimal set of features supporting the (simple) application. For the task switch time measurement, three different scenarios were chosen:

1. A voluntary task switch in a non-preemptive system (see Figure 5.1 on the next page).
2. A forced task switch in a non-preemptive system (see Figure 5.2 on the facing page).
3. A preemptive task switch in a fully-preemptive system (see Figure 5.3 on the next page).

All the instructions executed from the `FROM` label to the `TO` label are taken into account for the resulting figure. Since the scenarios were set-up for the tasks to be executed in an endless loop, the measurement samples were taken at least 10,000 times each.

Start-Up Scenarios

The start-up time measurement as suggested by the OSEK specification measures the time from the `StartOS ()` issue and the execution of the first instruction of user code (see Figure 5.4 on page 70); this scenario needs a user-provided `main ()` function. All the instructions executed from the `FROM` label to the `TO` label are taken into account for the resulting figure. The measurement samples were taken at least 10 times each.

5.3.2 Benchmark Results

The results of all four measurement rows are listed in Table 5.2 on page 70, depicting the average values of the corresponding execution times. However, the distinct measurement results hardly varied so that the standard deviation is negligibly low in all cases. Both systems were tested by supplying two different

```
TASK (Task0) { // low priority
    ActivateTask (Task1);
    asm volatile ("FROM:");
    Schedule ();
    ChainTask (Task0);
}

TASK (Task1) { // high priority
    asm volatile ("TO:");
    TerminateTask ();
}
```

Figure 5.1: Task switch time measurement scenario 1: Voluntary task switch in a non-preemptive system.

```
TASK (Task0) {
    ActivateTask (Task1);
    asm volatile ("FROM:");
    TerminateTask ();
}

TASK (Task1) {
    asm volatile ("TO:");
    ActivateTask (Task0);
    TerminateTask ();
}
```

Figure 5.2: Task switch time measurement scenario 2: Forced task switch in a non-preemptive system.

```
TASK (Task0) { // low priority
    asm volatile ("FROM:");
    ActivateTask (Task1);
    ChainTask (Task0);
}

TASK (Task1) { // high priority
    asm volatile ("TO:");
    TerminateTask ();
}
```

Figure 5.3: Task switch time measurement scenario 3: Preemptive task switch in a fully-preemptive system.

```
int main () {
    asm volatile ("FROM:");
    StartOS (OSDEFAULTAPPLICATIONMODE);
}

TASK (Task0) {
    asm volatile ("T0:");
    TerminateTask ();
}
```

Figure 5.4: Start-up time measurement scenario.

Scenario	CiAO External RAM	ProOSEK External RAM	CiAO Internal RAM	ProOSEK Internal RAM
Voluntary Task Switch	1,710	1,471	174	218
Forced Task Switch	1,311	1,755	118	280
Preemptive Task Switch	1,852	2,001	201	274
Start-Up	1,980	2,450	176	399

Table 5.2: Clock cycles for the task switch and start-up scenarios.

linker scripts each, putting the OS and the application either in the internal RAM or the external RAM of the microcontroller unit. Figure 5.5 on the facing page and Figure 5.6 on the next page show the results in a graphical form; mind the different scale factors of the y axes when attempting to compare them to each other.

5.3.3 Discussion

CiAO vs. ProOSEK

Comparing the results of the execution-time benchmarks from ProOSEK to CiAO, it is obvious that the CiAO implementation is faster in almost all scenarios, independently of the RAM type used. The execution times of the CiAO systems are up to 2.4 times lower than the ones of the comparable ProOSEK systems; this is due to several reasons.

Configurability

First of all, ProOSEK is less configurable than CiAO—which has configurability as one of its major design goals (see Section 1.1). As one of the benchmarking conditions (see above), ProOSEK was configured to provide as minimal functionality as possible for each application scenario, of course, but this is not enough. The scheduler is synchronized with ISRs, for instance; however, there are no ISRs in the application scenario possibly interrupting the kernel. The corresponding code leads to part of the noticed task switch overhead. In order to provide a broader and, hence, fairer comparison, future measurements will include more sophisticated application scenarios requiring more complex configurations.

RAM Usage Optimization

Furthermore, ProOSEK is optimized for RAM usage, and therefore for the stack and data used by the kernel (the code is stored in a read-only flash memory part in production systems), which were not considered in this evaluation. Since this is effectively a trade-off, this optimization is at the cost of code size (also not measured here) and therefore to some extent of execution times.

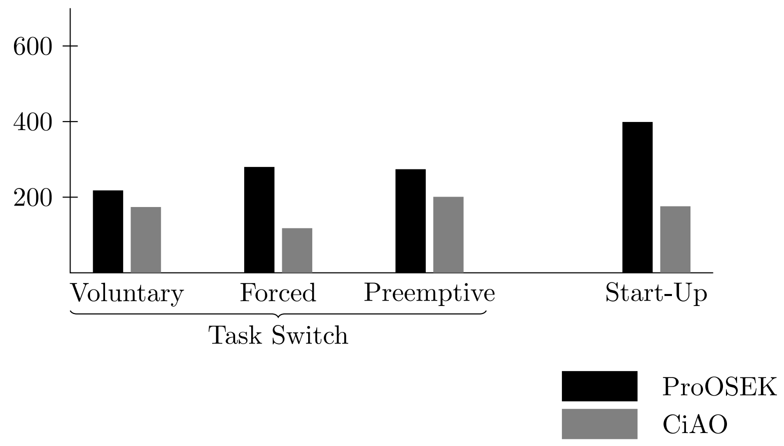

Figure 5.5: Bar chart: clock cycles for the task switch and start-up scenarios (code and data in *internal* RAM).

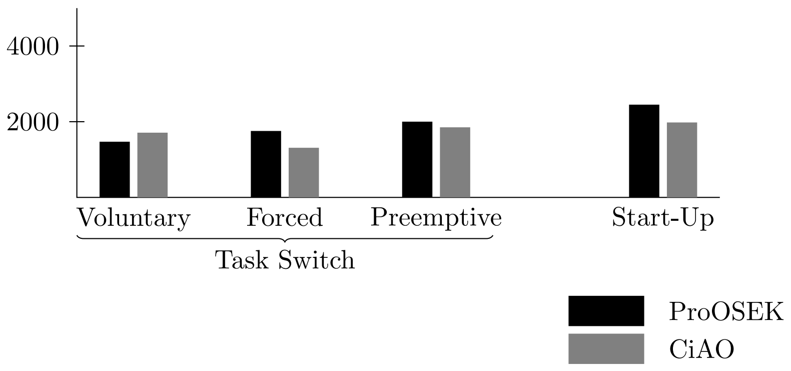

Figure 5.6: Bar chart: clock cycles for the task switch and start-up scenarios (code and data in *external* RAM).

Nevertheless, a difference of this magnitude is unexpected and shows that a research-oriented AspectC++ implementation can stand the comparison with a commercial C implementation of the OSEK standard.

Internal vs. External RAM

As expected, the execution times measured in external RAM systems are much higher than the corresponding ones located in the internal RAM; the factor varies from 9.2–11.3 in CiAO systems, and from 6.1–7.3 in ProOSEK systems. In general, the high discrepancy between internal and external RAM values is explicable by the fact that the internal RAM consists of static RAM and bears no wait states, while the external RAM modules are dynamic and need refresh cycles. Also, the external RAM access time is much higher than the one of the internal RAM. As for the different factors between CiAO and ProOSEK systems, it can be deduced that the CiAO implementation bears more relative memory instructions than the ProOSEK implementation, leading to a higher execution-time increase when placed into external RAM. This might also be the result of the fact that ProOSEK is trimmed for RAM usage and therefore does not need to hold as much state in RAM, where the access is expensive when located on *external* RAM.

5.4 The AOP Approach in OS Design and Implementation

5.4.1 General AOP Benefits

An operating system can benefit from several well-documented advantages of aspect-oriented design and implementation.

Separation of Concerns

Using AOP, it is possible to separate the high number of concerns present in an OS (see Chapter 3) from the beginning. This way, the encapsulated concerns are better maintainable and evolvable. Especially, particular concerns are thereby easily configurable by exchanging the implementing aspects; this is an important point in an operating system, which allows for the configuration of several global concern policies. Otherwise, conditional compilation techniques have to be used, which impair maintainability, evolvability, and understandability of the design and the source code.

Quantification

Through the AOP property of *quantification*, it is possible to apply a piece of advice to several join points at once. These join points are designated via a matching mechanism of the pointcut expressions that a piece of advice is connected to. A flexible match and wildcard grammar allows for a fine-grained selection of join points contained in a pointcut, which is necessary in order not to insert code to unnecessary points (for an example excluding `const` functions from memory protection aspects, consider [LSH+07]). Quantification allows for re-use of code, namely the implementation of an advice. Especially the fault isolation concerns of an AUTOSAR OS are highly (homogeneously) crosscutting (see Section 3.6 and Section 5.5.1), and therefore ideally designed and implemented using aspects.

Separation of What and Where

Furthermore, the configurable features of an OS most often only differ in *either* the implementation *or* the points where they are effective. By making use of quantification, these two dimensions of configuration can be kept separate and are therefore also configurable separately. An example is the stack monitoring aspect (see Section 4.2.10), whose implementation is fixed, but depending on the configuration the target pointcut is either the context switch or all system service invocations.

Configurability

Through the before-mentioned AOP properties of quantification and better separation of concerns, it is easier to reach a high level of *configurability* for an operating system, even on a fine-grained level. As evaluated in Section 5.3, this can lead to a significant advantage especially in embedded operating systems. The configurability is particularly well reached by the flexibility of aspects: If a declaratively described join point does not exist in the system (due to a limited configuration), the aspect just does not give advice to it, resulting in neither a compiler error nor a semantical one. The same thing holds for the quantification of aspects in *order* advice, for example; a single comprehensive order advice is also suitable for kinds of configurations not providing every mentioned aspect.

Straight-Forward Design

Some parts of the AUTOSAR and OSEK specification texts can be formulated by an aspect almost verbatim. For instance, take requirement OS093 of the AUTOSAR OS specification [AUT06b, p. 40]:

> 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`.

This can almost directly be mapped to an AspectC++ aspect:

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

Vice versa, the specification requirement can also be easily formulated by taking a look at the implementing aspect. This is mostly due to the flexible mechanism of pointcut expressions, which can be inter-connected using boolean operators, and because of the comprehensive catalog of pointcut functions (such as `within ()`) available.

Many concerns in the requirement form are already formulated in a way that cross-cuts other concerns—AOP allows to keep these requirements encapsulated also in the design and implementation phase.

■ 5.4.2 AOP Benefits for Safety Concerns

Since one important concern in operating systems is always *safety*—especially in embedded operating systems like OSEK and AUTOSAR OS—, AOP properties having an impact on safety are particularly of interest.

API Slicing

One design pattern having an impact on fault isolation that was already introduced in Section 4.3.1 is the slicing of the AUTOSAR API. By extending the API depending on the configuration, problems resulting from a mismatch between the application and the OS configuration can be detected early in the system deployment process.

No Forgotten Join Points

Furthermore, careful aspect-oriented design and implementation allows for less programming errors when tackling a highly cross-cutting concern compared to a manual and, hence, scattered one. For instance, rarely used exit paths are likely to be forgotten in the manual implementation of a cross-cutting feature, which can not happen when encapsulated in a carefully quantified aspect. Not to respect *all* possible impact points is a type of human error that is rather frequent and can impact even complex system software.

Error Detection and Handling

Lastly, AOP is especially useful to implement *error handling concerns*, since those can most often take advantage of its flexible quantification property (see also Section 5.4.1). Pointcut expressions can use pointcut functions and match expressions to discriminate particular parameter types or return types, for instance, which often determine the applicability of a fault isolation and handling concern. Several pieces of work were dedicated to the combination of error handling and AOP [FCF+06, LL00, FGR07], coming to the conclusion that those concerns can be well separated from the base code using AOP, but partly result in a non-negligible overhead compared to the manual, scattered implementation. However, these experiments were all conducted using AspectJ; AspectC++ does not induce a significant overhead per se in most cases (see Section 5.2 and Section 5.3, and also [LST+06]).

Exclusion of Distinct Scenarios

Moreover, aspects can be used to exclude specific, undesirable situations. Under the premise that the aspect weaver implementation is correct, this exclusion can then be guaranteed by design. Consider, for example, the aspect excluding the shut-down of the operating system from a non-trusted application (see Section 4.2.11).

5.4.3 AOP Benefits Pertaining to the AspectC++ Weaver

There are AOP features that the CiAO kernel benefits from that are special to the AspectC++ weaver implementation.

Advice Inlining

Aspects giving advice in AspectC++ are transformed in a way that makes it possible for the back-end compiler to inline the advice code directly in the affected join points, depending on the size of the advice and the number of join points it affects. This decision is made solely by the compiler and according to the criteria for regular function inlining. Hence, a piece of advice affecting a single join point can be inlined as if put there manually, saving the costs of a function call and return.

Boundary-Crossing Advice

In the CiAO design, this is especially important for those aspects crossing the application–kernel boundary (see also Section 4.1.2). They can be localized in separated code bases according to the concerns they represent, and still influence the join points in the counter part in an efficient way. Hence, up-calls from the OS to the application, for instance, do not induce any overhead (see also Section 5.2).

However, the current AspectC++ weaver implementation is overly zealous in this respect; see Section 5.4.5 for more about that problem.

5.4.4 Limitations of the AOP Approach

General-Purpose AOP

As shown in this chapter, it is definitely possible to reach a complete separation of the concerns present in an operating system by means of AOP. However, since AOP is a general-purpose paradigm, it is not ideally suited to reflect some of the particularities found in operating system software.

Control Flow Context

The fundamental specialty of an operating system compared to other kinds of software is that the functionality of its services depends on the control flow context that it is invoked from; that is, depending on the type of control flow (e.g., a task, ISR category 1, or ISR category 2 in the case of AUTOSAR OS), and on the concrete instance. This fact is not reflected in the current AOP languages and the pointcut functions they offer. The root of the problem are the base programming languages, though, since most of those used for OS implementations have no self-contained concept of a control flow; they merely have control flow *functions* building schemes that are then instantiated by the OS as distinct control flows. Therefore, the control flow dispatch functions in operating systems are mostly "hacks" on the programming language level, or can not even be expressed in them and need to be formulated in assembly language.

5.4.4.1 The Vision

Desired Pointcut Concept

Hence, a dedicated pointcut concept hereby named `cflowtype` would be conceptually helpful for operating systems design. This `cflowtype` pointcut function filters all join points in the control flow of a specific control flow *type* (e.g., a task, an ISR of category 2, or an ISR of category 1). An example use is the preemption aspect (see Section 4.2.2 and Section 4.2.5), which either reschedules directly after a call to a preemption point if invoked from within a *task*, or registers an AST that performs the rescheduling if called from within an *ISR*. This would also be advantageous with refined types; for example, a preemptable task, or a non-preemptable task.

Weaver Implementation

The implementation of such a language and weaver feature would need to be provided further internal information about the operating system (e.g., how to query the running control flow and how to differentiate between control flow types) or need run-time support by it (e.g., updating special variables upon dispatching to enable the run-time part of the weaver system to determine the running control flow and its type). Hence, this part of the code generation engine of the weaver would be needed to generate different code for every supported OS.

5.4.4.2 The Status Quo

Current Problems

AspectC++ offers two pointcut functions that might be suited for the job at first sight. `within ()` together with call advice captures all calls from within a defined set of functions. However, there are two problems with that approach:

1. The definition of the set of functions is based on their *names*. Therefore, in order to distinguish different types of control flows namespaces or another kind of artificial naming convention have to be used.

2. Calls from within sub-procedures of the control flow's entry function are not matched by this pointcut function.

Solutions

In order to overcome problem number one annotations could be used, which are not yet implemented in AspectC++ (see also Section 5.4.5). Problem number two is addressed by another pointcut function, `cflow ()`, which captures all calls from within the *control flow* of a function, including calls from subprocedures. It is implemented by setting a flag on the entry of the target function, and re-setting it upon exiting it.

`cflow ()` Problem The problem in the operating systems domain is that control flows can be *dispatched*—by special, platform-specific functions. After a dispatch, `cflow ()` will yield that the path of execution is still in the context of the *original* control flow, which is *technically* right (the first control flow function was not yet exited) but is *semantically* wrong from the perspective of the operating system (a different control flow is executing).

Workaround The current workaround is to omit an additional pointcut function and to query the current control flow type or control flow instance manually in the advice code if needed.

5.4.4.3 Differences to General-Purpose AOP

Data Type Workaround In general, the mapping from control flows to data types and their distinct object instances is a workaround to make them visible on the level of the programming language in order to be able to advise them using AOP. This, however, bears the drawbacks discussed above.

Client–Server Software Entities The concept of control flows and their kinds of contexts is significantly different from the client–server relationship found in entities of classical software, which are addressed by general-purpose AOP languages. These client and server data types, objects, and their relationships can be described by the pointcut language pertaining to the AOP language (consider, e.g., `that`, `target`, and `call ()`). The corresponding description mechanisms for control flows of an operating system, as described above, are to be provided by an enhanced AOP language and weaver.

5.4.5 Limitations of the AspectC++ Weaver

There are some features that could conceptually be available in an AOP language, but are not in AspectC++ due to restrictions of the aspect weaver.

`enum` Slicing Currently, it is not possible to introduce a new named integer constant to an existing enumeration by means of an introduction advice. *Technically*, this does not make a difference since an additional enumeration constant does not need additional memory. However, *conceptually*, this possibility would provide for a complete separation of some affected concerns. For instance, the `WAITING` task state conceptually pertains to the event concern (see also Section 3.2.6), and, hence, is designed as a slice introduction advice (see Section 4.2.7), but can not be implemented this way.

Annotations Pointcut definitions in AspectC++ are composed of pointcut functions and basic match expressions. These expressions match the *name* of code constructs like classes and their methods. In some situations, however, it is necessary to attach additional properties to these constructs, which can then be evaluated in the match expression of a piece of advice. For instance, the synchronization properties of a method inherently belong to that method, and not the synchronization aspect, which only *evaluates* the properties. The current implementation classifies the methods according to their synchronization properties in a separate aspect, and is therefore fragile when methods are added or changed [LSSSP07]. Another use of annotations would be to tag the classes belonging to a specific layer, since some advice affects exactly one layer. The current workaround is to assign classes to hierarchical namespaces and use those as classifiers in the match expressions.

Extended Join-Point IDs

An additional desirable feature for AspectC++ is the provision of an extended notion of *join-point IDs* in its API. AspectC++ already provides join-point IDs, but they are only accessible within advice referring to that join point, and not outside. This is needed for situations as specified by OSEK for its error hook, where inside the hook the ID of the failed function should be queryable. Since this is a truly cross-cutting piece of advice affecting many different join points (namely, all services returning a `StatusType`), the concrete function causing the invocation of the error hook can indeed be identified by storing the join-point ID. The problem is that the user does not know about these artificial ID numbers and, hence, uses symbolic names to identify the services (e.g., `OSServiceId_ActivateTask`). However, currently there is no way to bind the ID of a join-point to a variable or constant by specifying it. For performance reasons, this implementation should be able to provide the ID as a *constant expression* so that the symbolic name does not have to use any memory.

If, however, the user implementation of the error hook merely constitutes some kind of logging of the failed service, it would be easier anyway to use the AspectC++ join-point function `signature ()` in combination with `arg<i> ()` and `ARGS` to provide the signature and the actual values of the parameters.

Code Transformation

As evaluated in Section 5.2, the machine code produced by the toolchain of the AspectC++ weaver and the back-end `tricore-g++` compiler is unacceptably inefficient at some points; especially the stack is used unnecessarily, bloating the code and the RAM usage. Hence, this problem could be tackled at the C++ code generation step of the weaver to be optimized for the back-end compiler.

Advice Inlining

Though the inlining of advice into the target code is useful in many cases, it is counter-productive in other ones. In general, this depends on the code size of the advice, the number of affected join points, and the use of join-point context information within the advice. Currently, the AspectC++ weaver inlines *every* piece of advice, thereby multiplying it even when the advice is big, needs no context information, and affects many join points. Future versions of the weaver should allow to specify *advice attributes* to allow the designer to determine whether the advice is to be always inlined, never inlined, or inlined depending on the factors stated above; the weaver could then make that decision since (unlike the compiler) it has a global view of the system. The current workaround is to code an artificial function call in order to disable inlining where needed.

■ 5.5 Aspect Traceability

■ 5.5.1 Separation of Concerns by Aspects

Concern Separation

When comparing the concerns extracted from the AUTOSAR requirements in the analysis part (see Chapter 3) to the design of the CiAO kernel (see Section 4.2), it becomes obvious that all of the concerns are encapsulated in a single, non-tangled and non-scattered entity each. The basic kernel facilities (i.e., the scheduler, the OS control facility, and the alarm manager) are encap-

Unit	Advice #	JPs	Explanation
ISR Cat. 1 Support	1+x	1+x	introduction to OS control, x ISR bindings
Resource Support	3	3	introductions to scheduler, API, task
Event Support	6	6	schedule, trigger action introduction to scheduler, API, task, alarm
Alarm Support	1	1	introduction to API
Full Preemption	1	3	3 points of rescheduling
Mixed Preemption	2	4	3 points of rescheduling, introduction to task
Stack Monitoring	2	3	before (last) CPU release, introduction to task
Context Check	1	x	x service calls
Disabled Interrupts Check	1	30	all services except interrupt services
Enable Without Disable Check	3	3	enable services
Missing Task End Check	1	x	x tasks
Out of Range Values Check	1	4	alarm set and schedule table start services
Invalid Objects Check	1	25	services with an OS object parameter
Error Hook	2	30	29 services, introduction to scheduler
Start-Up / Shut-Down Hook	2	2	explicit hooks
Pre-Task / Post-Task Hook	2	2	explicit hooks

Table 5.3: Number of join points and pieces of advice given per concern designed as an aspect (without *order* advice).

sulated in a *class*, while all of the other concerns are formulated as an *aspect* extending the functionality of the objects instantiated from the classes, and the state held by the relevant OS objects. Hence, all concerns that could be separated in the requirements and the analysis phase could also be separated in the design and implementation of the kernel.

One Aspect per Concern

Since an aspect can comprise an arbitrary number of pieces of advice, the aspectual concerns are designed to be encapsulated in a single comprehensive aspect giving the pieces of advice necessary to implement it[2]. More interesting numbers are therefore the *number of pieces of advice* given by a concern designed as an aspect, and the *number of join points* affected by these aspects. Table 5.3 shows these numbers for those concerns designed as aspects that were actually implemented; it does not include *order* advice since this type of advice is not attributable to a single concern but shared among those affecting the same join points.

Types of Cross-Cutting

Obviously, the basic aspects providing additional system abstractions to the kernel are *heterogeneously cross-cutting*; that is, the affected join points are advised in different ways each [CC04]. This can be observed by the conformance of the pieces of advice and the number of join points of the corresponding aspects. The error hook aspect and most of the service protection aspects, however, exhibit an extremely *homogeneously cross-cutting* behavior. It is mostly a single piece of advice that affects numerous join points at once with the same functionality. Thus, both types of cross-cutting can be observed in an operating system kernel, and both can be designed as aspects.

Homogeneity and Generic Advice

Interestingly, some of the homogeneously cross-cutting aspects are only homogeneous because they are implemented giving *generic advice* (see Sec-

[2]Note that the *implementation*, however, might need more than one *aspect header file* because in some cases AspectC++ slices need to be out-sourced to avoid include cycles.

tion 4.4.2). Though their basic nature is rather heterogeneous in the sense of the definition (i.e., they give slightly different advice depending on the join point affected), this tendency to heterogeneity can be overcome by the adaptation to the join-point context, which is exactly what generic advice does. Take, for instance, the simple example presented in Section 4.4.2: Without generic advice, four different pieces of advice would have to be given to accommodate the return types of all services.

5.5.2 Evaluation of the Identified Pointcut Candidates

When comparing the analysis with the design, it becomes apparent that not all kernel-internal pointcut candidates could be directly mapped to a pointcut in the kernel code.

Task Switch Pointcut Candidate

For instance, the discussion of the analysis of internal pointcut candidates (see Section 3.7.3) brought special attention to the task switch point in the operating system, which is of importance to several concerns at once (see also Table 3.2 on page 32). Comparing this result to the design of the aspects corresponding to these concerns, it becomes obvious that the preliminarily identified task switch pointcut candidate in fact comprises different notions and therefore different pointcuts.

Two Task Switch Semantics

The task switch that the pre-task hook and the post-task hook are interested in is different from the one that, for instance, the memory protection concern is. This is a result of the two different notions of a task: the high-level notion described in the AUTOSAR specification, and the low-level notion used in the kernel implementation. The latter is actually not a task, but a continuation (see Section 4.2.2); hence, the continuation term is used here to distinguish between the two. There are several situations where the task notion and the continuation notion diverge; selected ones are presented here to state an example:

1. When the scheduler has no task that is ready to run, there is still a continuation that is running in idle mode until a productive one becomes ready. Nevertheless, the external semantics of that situation is that there is no valid running task.

2. ISRs of category 2 are always executed in the context of a continuation, even if no high-level task is currently running. Nevertheless, `GetTaskID ()` shall return the invalid task ID during the ISR execution in that case.

When considering the first situation, it is clear that this execution point does not constitute a join point to the memory protection aspect, for example, since without knowing the memory protection properties of the next task it does not make sense to alter them. Nevertheless, it *is* a join point for the post-task hook aspect, for instance, since a high-level task was interrupted (i.e., blocked or ended).

Task Switch Pointcuts

Hence, there are two different sets of join points for the task switch pointcut candidate:

1. High-level task switch: `setRunning ()` is the single point in the kernel where the running high-level task is switched; it is therefore the point to

be advised by the pre-task hook, post-task hook, and foreign OS objects check aspects.

2. Low-level continuation switch: `after_%CPUReceive ()` is called directly before the first instruction of a continuation in a scheduling cycle, `before-_%CPURelease ()` directly after the last instruction of a continuation in a scheduling cycle (see also Section 4.4.1). These are the points to be advised by the memory protection, stack monitoring, and timing protection aspects.

Kernel–Application Transitions

Another point that is not directly graspable is the transition from a control flow of the application into the kernel and vice versa. This was already touched on as an example for the deployment of *explicit join points* (see Section 4.4.1). These are introduced before and after system service invocations (which are processed in the kernel), but additionally need to be cared about when dispatching a task by introducing appropriate explicit join points. For instance, a newly dispatched task starts in the kernel, but then leaves it just before executing the first user instruction. Thereby, the aspects interested in these transitions can advise the artificial, explicit join points to capture the points in the control flow.

One-to-One Mappable Pointcut Candidates

There are also pointcut candidates identified in the analysis that *can* be mapped to a pointcut in a one-to-one way:

- The alarm expiry point explictly corresponds to the action method that is called inside the alarm manager: `AlarmManager::triggerAction ()`.
- Since ISRs of category 2 are serialized and have a higher priority than all tasks, they can never be interrupted by other category 2 ISRs or by tasks. Hence, the start and end of a category 2 ISR execution can be captured by the start and end of the corresponding ISR function. *All* ISR functions are covered by the pointcut `"void functionISR% (void)"`.
- The pointcut candidate named “system start-up” can also be captured concisely, but needs more thorough investigation. The concerns interested in that point of execution are the start-up hook concern and the alarm and task management concerns (which provide the corresponding auto-start feature). The affected points in the start-up process are exactly defined in the OSEK specification [OSE05a, p. 42]; all three refer to the same point, but in a defined order. This single point of interest in the start-up routine of the OS is exposed by deploying an explicit join point (see Section 4.4.1, reason one): `OSControl::internalStartupHook`.
- Since there is a dedicated routine to shut-down the operating system (also reflected in an API function), it is an advisable join point: `OSControl-::shutdownOS ()`.
- When a protection violation occurs, the violation handler calls a dedicated method (with a default implementation) in order to determine the handler action. This inherently is a clearly defined join point: `SchedImpl-::internalProtectionHook ()`.
- The uncontrolled end of a task can be grasped by the straight-forward pointcut expression `"void functionTask% (void)" : after ()`.

- As already noted in Section 3.4.2, an application switch is a subset of all task switches and ISR category 2 dispatch points, which were both already discussed above. Hence, advice interested in application switches can use the discussed pointcuts, and additionally check if the application pertaining to the new control flow is different from the current one.

5.6 Summary of the Evaluation Results

Achieved Goals

The goal to keep the CiAO kernel highly configurable and therefore scalable was reached, which is shown by the analysis of the memory footprint of different CiAO configurations. The implementation does not induce an overhead per se due to its aspect-oriented design as proven by the comparison with a commercial OSEK implementation; AspectC++ even allows to deploy generic advice without falling back to run-time reflection in many cases.

Advantages of the Approach

Additionally, the aspect-oriented design provides significant advantages for the programmer and maintainer of the CiAO system. Even homogeneously cross-cutting features are encapsulated in a single aspect, and many requirements can be designed straight-forwardly and in an encapsulated way. Programming errors can be reduced by several measures designed with AOP; many properties of features targeting the scope of their deployment can be made explicit via pointcut declarations.

Problems to Be Tackled

The evaluation also yielded problems in the implementation and the tool chain used, which are both target for future optimizations. This also includes work on concepts to be provided by the aspect weaver that are useful for operating system engineering, but are not yet available.

CHAPTER 6

Summary and Outlook

This chapter concludes the thesis by giving a summary of its results in Section 6.1; furthermore, different directions of future work are presented in Sections 6.2 through 6.4. In the end, the final conclusions are given in Section 6.5.

6.1 Summary

Goals The goal of this thesis was to develop a kernel for the CiAO operating system, and both design and implement it by making use of aspect-oriented programming. The approach was to be evaluated in order to get an idea of the suitability of AOP techniques in the domain of operating system engineering.

Analysis During the analysis, the concerns to be respected in the kernel design were identified and defined. These concerns were extracted from the AUTOSAR OS standard specification as part of the thesis' requirements; both explicit ones and implicit ones were stated. The resulting concern set was classified depending on their goals and properties into system abstraction concerns, kernel-internal concerns, fault isolation concerns, and callback concerns. These classes were then analyzed regarding their impact on the behavior of the offered AUTOSAR system services, on the state managed by the OS, and on kernel-internal points of execution (designated as pointcut candidates). The discussion of the analysis yielded that the impact of those concerns highly cross-cutting the rest of the kernel can be described by simple "verbal" pointcuts (likely to be mapped to an AOP pointcut), and that the points of particular interest to many concerns are the control flow abstraction types, the control flow dispatch points, and the kernel–application context switch points.

Design The basic design of the CiAO kernel was deducted from the results of the analysis and defined to include three core facilities: the scheduler, the alarm manager, and an OS control facility. All previously identified concerns were kept separate and independent by deploying encapsulating classes and aspects, which extend and affect the core facilities. Furthermore, API slicing, OS object type slicing, and an aspect-oriented up-call binding mechanism were identified as specific aspect-oriented design schemes deployed in CiAO, bearing distinct advantages. The identified aspect-oriented *implementation* schemes comprise the deployment of explicit join points and generic advice in order to reach special goals by means of AOP.

Evaluation

Finally, the design and implementation of the CiAO kernel were evaluated in different ways. The analysis of the memory footprint induced by different CiAO configurations yielded a good scalability of the kernel as a result of the fine-grained configurable features. It also hinted at several points to be targeted by optimizations, and at the desire to have more selective advice for expensive functionality, which would be beyond the AUTOSAR standard. An execution-time comparison with a commercial OSEK operating system showed that there is no basic overhead induced by the AOP approach; instead, in the selected simple scenarios a better performance by CiAO could be observed due to its better tailorability. Concerning the aspect-oriented design, general AOP benefits especially applicable to the OS domain were identified as well as benefits relevant for safety concerns in operating systems. This part of the evaluation also includes the identification of limitations of AOP in that domain, which would need special pointcut functions for filtering join-point contexts by the type of the control flow contexts, and for filtering points in control flows with defined properties. The evaluation was concluded by an investigation of the aspect traceability, which showed that thanks to AOP a clear separation of concerns could be reached, with the deployed aspects showing different kinds of cross-cutting nature, though; moreover, pointcut candidates as identified in the analysis could not be mapped to a single pointcut in every case.

6.2 AUTOSAR Timing Protection

Comprehensive Architectural Property

The AUTOSAR timing protection feature, which was presented and analyzed in Section 3.4.4, constitutes a comprehensive architectural property. It can therefore serve as another use case for the *configurability* of architectural properties by aspect-oriented techniques, which is the main goal of CiAO (see Section 1.1). When taking a look at the corresponding specification part and the analysis, it seems promising that this feature can be designed and implemented very well with the AOP paradigm—the timing protection concern cross-cuts many of the other basic system concerns at clearly defined points.

Support by the TriCore Platform

Furthermore, the TriCore platform used for the CiAO prototype (see Section 5.1) bears different support possibilities for this kind of feature, which could be evaluated and designed to be configurable. Depending on the accuracy requirements of the target system, the active control flow to be accounted the execution-time (or resource acquirement time, or interrupt locking time) can be determined regularly by a polling mechanism or by setting up a timer preempting the control flow when its corresponding time budget is depleted. The timer could be implemented using the TriCore watchdog, which triggers an NMI that definitely interrupts the time-depleted control flow. The regular polling mechanism could either also be implemented by the watchdog, or be out-sourced to a second processor of the TriCore platform, the *peripheral control processor*. This processor could then check the running control flow on a regular basis and update the corresponding execution budget in the double-ported RAM.

The actual and detailed design and implementation of the AUTOSAR timing protection is to be tackled in future work.

■ 6.3 Hardware- vs. Software-Based Memory Protection

KESO

The KESO project by Wawersich et al. [WSSP07] can be connected to the CiAO project since it is based on an OSEK system, providing a Java environment for embedded applications. Thereby, the type safety of the Java programming language is exploited to provide memory protection by software mechanisms.

Memory Protection Comparison

Since the CiAO kernel is OSEK-compliant, it can be used as an evaluation base for a KESO system. Furthermore, since (hardware-based) memory protection is configurable in CiAO, hardware-based and software-based memory protection can be directly compared to each other. This comparison is assumed to be fair since the base kernel is in both cases the same: the CiAO kernel.

Mixed-Protection Systems

Moreover, when support for OS applications and their different trust levels is fully integrated in CiAO, it is possible to inter-connect both projects even more. For instance, a certified KESO system with potentially *untrusted* KESO applications running on top of it can nevertheless be designated a *trusted* OS application for the underlying CiAO AUTOSAR kernel. This way, both hardware- and software-based memory protection domains can co-exist in a single embedded system. The allocation of an application to either one of these protection classes can therefore solely depend on its demands and the different properties evaluated before.

■ 6.4 Static Application Analysis

CiAO Configuration Space

With CiAO being designed as a fine-grained *software product line* from the very beginning (see also Section 1.1), the configuration space of the CiAO system is fairly big. Hence, a system designer has to make many configuration decisions before the deployment of the applications with CiAO.

Code Analysis in CiAO

Nevertheless, some of these decisions can be derived from the application itself by a detailed code analysis. In CiAO, this includes the configuration of support for events, for instance, which is only needed if event services are used within the application mode. Further examples comprise resources, alarms, and hooks, whose support need can easily be determined by taking a look at the referenced symbols. These are only trivial examples; further deduction of configuration options is possible through more detailed analysis and more complex predicates (e.g., the necessity of a dedicated stack for a task depends on the resource protocol, the number of tasks per priority possible, and if it uses events).

Static Application Analysis

The corresponding scientific discipline is called *static application analysis*, and it aims for the automatic configuration of a product line where possible. Thereby, the software is tailored to the demands of the application, leaving aside only strategic configuration options for the system deployer.

■ 6.5 Conclusions

The cost pressure in the embedded systems market forces the system deployers to choose operating systems that are highly configurable and therefore tailorable to the needs of the applications. As shown in this thesis, aspect-oriented programming can satisfy this demand on the system software, while providing a clear separation of different concerns at the same time. This has a positive impact on the evolvability of the operating system kernel, which is particularly important for implementations of the AUTOSAR OS standard since the specifications are not yet finalized and therefore still subject to changes. CiAO is ready to face them.

Bibliography

[AUT06a] AUTOSAR. Methodology (version 1.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[AUT06b] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[AUT06c] AUTOSAR. Specification of RTE software (version 1.0.1). Technical report, Automotive Open System Architecture GbR, July 2006.

[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.

[BGP+99] 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.

[CC04] Adrian Colyer and Andrew Clement. Large-scale AOSD for middleware. In Karl Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 56–65, Lancaster, UK, March 2004. ACM Press.

[CE00] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* Addison-Wesley, May 2000.

[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.

[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.

[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.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Communications of the ACM*, 15(10):859–866, 1972.

[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.

[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.

[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.

[FF00] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced SoC (OOPSLA '00)*, October 2000.

[FGR07] Fernando Castor Filho, Alessandro Garcia, and Cecília Mary F. Rubira. Error handling as an aspect. In *Proceedings of the 2nd Workshop on Best Practices in Applying Aspect-Oriented Software Development (BPAOSD '07)*, New York, NY, USA, 2007. ACM Press.

[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.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[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.

[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.

[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.

[LL00] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 418–427, New York, NY, USA, 2000. ACM Press.

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

[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)*, New York, NY, USA, October 2007. ACM Press.

[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.

[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 EuroSys 2006 Conference (EuroSys '06)*, pages 191–204, New York, NY, USA, April 2006. ACM Press.

[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-OOOSWS '02)*, pages 49–54, Malaga, Spain, June 2002.

[Mye96] Nathan Myers. *A new and useful template technique: "traits"*, pages 451–457. C++ gems. SIGS Publications, Inc., New York, NY, USA, 1996.

[OSE01] OSEK/VDX Group. *Time Triggered Operating System Specification 1.0*. OSEK/VDX Group, July 2001. http://www.osek-vdx.org/.

[OSE04a] OSEK/VDX Group. *OSEK Implementation Language Specification 2.5*. OSEK/VDX Group, 2004. http://www.osek-vdx.org/.

[OSE04b] OSEK/VDX Group. *OSEK/VDX Communication 3.0.3*. OSEK/VDX Group, July 2004. http://www.osek-vdx.org/.

[OSE04c] OSEK/VDX Group. *OSEK/VDX Network Management 2.5.3*. OSEK/VDX Group, July 2004. http://www.osek-vdx.org/.

[OSE05a] OSEK/VDX Group. *Operating System Specification 2.2.3*. OSEK/VDX Group, February 2005. http://www.osek-vdx.org/.

[OSE05b] OSEK/VDX Group. *OSEK Run Time Interface (ORTI), Part A*. OSEK/VDX Group, November 2005. http://www.osek-vdx.org/.

[OSE05c] OSEK/VDX Group. *OSEK Run Time Interface (ORTI), Part B*. OSEK/VDX Group, November 2005. http://www.osek-vdx.org/.

[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.

[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.

[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.

[SLSP06] Olaf Spinczyk, Daniel Lohmann, and Wolfgang Schröder-Preikschat. Concern hierarchies. In *1st GPCE Workshop on Aspect-Oriented Product Line Engineering (GPCE-AOPLE '06)*, October 2006.

[Str07] Jochen Streicher. Aspektorientierte Entwicklung konfigurierbarer Speicherschutzverfahren für die CiAO Betriebssystemfamilie. Diplomarbeit, Friedrich-Alexander-Universität Erlangen-Nürnberg, September 2007.

[WE07] Sean Walton and Eric Eide. Resource management aspects for sensor network software. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS '07)*, New York, NY, USA, October 2007. ACM Press.

[WSSP07] Christian Wawersich, Michael Stilkerich, and Wolfgang Schröder-Preikschat. An OSEK/VDX-based multi-JVM for automotive appliances. In *Embedded System Design: Topics, Techniques and Trends*, IFIP International Federation for Information Processing, pages 85–96, Boston, 2007.

List of Figures

List of Tables

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Erlangen, den 30.10.2007, ________________________________