Aspect-Aware Operating System Development*

Daniel Lohmann[1] Wanja Hofer[1] Wolfgang Schröder-Preikschat[1] Olaf Spinczyk[2]

[1]Friedrich–Alexander University Erlangen–Nuremberg
[2]Technische Universität Dortmund
{lohmann,hofer,wosch}@cs.fau.de
olaf.spinczyk@tu-dortmund.de

ABSTRACT

The domain of operating systems has often been mentioned as an "ideal candidate" for the application of AOP; fundamental policies we find in these systems, such as *synchronization* or *preemption*, seem to be inherently cross-cutting in their implementation. Their clear separation into dedicated aspect modules should facilitate better evolvability and – the focus of this paper – *configurability.* Our experience with applying AOP to the domain of highly configurable embedded operating systems has shown, however, that these advantages can by no means be taken for granted. To reveal maximum configurability of central system policies, aspects and their potential interactions with the system have to be taken into account much earlier, that is, "from the very beginning". We propose the analysis and design process of *aspect-aware development*, which leads to such an "aspect-friendly" system structure and demonstrate its feasibility on the example of CiAO, an AUTOSAR-OS-compliant operating system that provides configurability of all fundamental system policies by means of AOP.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and Interfaces*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Measurement, Experimentation, Design

Keywords

Aspect-Aware Design, Aspect-Oriented Programming (AOP), AspectC++, CiAO

*This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4, SCHR 603/7-1, SP 968/2-1, SP 968/4-1, and LO 1719/1-1.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
AOSD'11, March 21–25, 2011, Pernambuco, Brazil.
Copyright 2011 ACM 978-1-4503-0605-8/11/03 ...$10.00.

1. INTRODUCTION

> *Throughout the entire operating-system design cycle, we must be careful to separate policy decisions from implementation details (mechanisms). This separation allows maximum flexibility if policy decisions are to be changed later.* (Silberschatz et al., "Operating System Concepts", p. 72, 2005)

When, more than a decade ago, the advent of aspect-oriented programming (AOP) promised a new dimension of separation of concerns in software systems, operating systems were among the targets that were first mentioned for the new approach [16]. AOP is appealing for this domain, as fundamental operating-system concerns, such as *synchronization, preemption, prefetching,* or *monitoring* seem to be inherently cross-cutting. Their clear separation into dedicated aspect modules would facilitate better *evolvability* and *configurability* of operating-system policies [5, 8, 1]. As operating-system engineers in the domain of embedded systems – a domain for which configurability is of utmost importance – we immediately became excited when we first heard about AOP at ECOOP '97. This triggered the design and development of the AspectC++ language and tool suite [26] and extensive studies with aspects in the PURE and eCos operating system families [25, 21].

Now, ten years later, our research activities on applying AOP to the domain of configurable operating systems have culminated in the development of CiAO (CiAO is Aspect-Oriented) – the first operating system family that has been designed and developed with AOP concepts from scratch. By the application of AOP, CiAO reaches excellent configurability, a good separation of concerns, and very low resource consumption in the resulting systems, which outperforms leading commercial implementations [20]. On the path to CiAO, however, we had to learn a lot. The separation and configuration of fundamental operating system policies by aspects turned out to be surprisingly challenging. To reveal maximum benefits, the incorporation of AOP (as a programming paradigm) had to be reflected in the system's design much deeper than we had initially expected; the no-overhead integration and configuration of even low-level operating system concerns by aspects required a decent level of pragmatism.

About This Paper

In this paper, we describe our *experiences* with applying AOP to the domain of configurable operating systems for resource-constrained embedded devices – and how they led to the analysis and design method of *aspect-aware operating system development* that we came up with for CiAO. In particular, we make the following contributions:

- We describe, on the example of two case studies, *typical particularities* of system software that in practice hinder better configurability by AOP and *analyze the fundamental issues* behind them (Section 2).

- We provide a new *analysis and design method* and a set of *fundamental design principles* to improve on the situation (Section 3). We have evaluated our method with AUTOSAR OS [4], the dominant operating system standard for automotive applications.

Our contribution is rounded up by a detailed discussion of the particularities of and requirements on AOP in general for the development of configurable system software (Section 4). In Section 5 we describe related work and finally conclude with the pros and cons that AOP offers for *this particular domain* (Section 6).

2. ANALYSIS AND BACKGROUND

System software provides no business value of its own. Its sole purpose is to ease the development and integration of applications – that is, to serve application developers and integrators with the "right" set of abstractions for *their* particular problems.

2.1 Embedded Operating Systems

This is a challenge especially in the domain of small ("deeply") embedded systems, which are subject to an enormous hardware-cost pressure. System software for this domain has to cope not only with strict resource constraints, but also with a broad variety of application requirements and platforms. For instance, power-train applications are typically safety-critical and have to deal with real-time requirements, while car body systems are far less critical. Hardware platforms range from 8-bit to 32-bit systems. Some applications require a task model with synchronization and communication primitives, whereas others are much simpler control loops. Thus, to allow for reuse, an operating system for the embedded systems domain has to be designed and developed as a software family – that is, for configurability (provide alternatives) and tailorability (leave out as much as possible). Furthermore, resource-saving static configuration mechanisms are strongly favored over dynamic (re-)configuration.

This necessity for best-possible configurability and tailorability was the reason we considered AOP to be so promising: It facilitates separation of many more concerns than the traditional implementation techniques. We especially sought for a technique to implement even fundamental internal "architectural" policies of an operating system kernel in a configurable and tailorable manner. Some examples for such fundamental policies are: Synchronization of kernel components (explicit vs. implicit, fine-grained vs. coarse-grained, hardware-supported), Interaction between kernel components (message-based vs. procedure-based), Preemption of control flows inside the kernel (fully-preemptive, at dedicated preemption points, no preemption until the kernel is left), and Protection of kernel components against invalid access and behavior (coarse-grained vs. fine-grained vs. no memory protection, deadline monitoring, parameter validation).

Such fundamental internal policies define what is commonly referred to as the *architecture* of an operating system, like *micro-kernel* (message-based interaction, implicit synchronization, fine-grained protection) or *monolith* (procedure-based interaction, explicit synchronization, coarse-grained protection). Their implementation, however, is notoriously cross-cutting and, hence, often hard-wired into the system – our call for AOP.

2.2 Early AOP Experiences

Our initial experiences with employing aspect to improve on the situation in the PURE and eCos operating system families, were, despite many success stories [25, 21], double edged: When it comes to the actual implementation, apparently orthogonal concerns (such as Interaction, Synchronization, and Preemption) often turned out to induce hidden functional dependencies and unexpected ambiguities on the join-point level. The following two examples from PURE and eCos illustrate some of the more problematic cases.

2.2.1 Device-Driver Invocation in PURE

In the late nineties, our research group developed the PURE family of operating systems for deeply embedded devices [6]. With more than 250 configurable features and a kernel memory footprint between 434 B and >100 KiB, PURE offers excellent scalability. We achieved this scalability without a single `#ifdef` in the C++ code by a design approach that put old ideas from HABERMANN and PARNAS (*functional dependencies* and *functional hierarchies* [13, 23]) to an extreme and that mapped functional layers to C++ classes.

PURE, however, did not offer configurability of fundamental system policies. Later, we applied AspectC++ to improve on the configurability of its architectural system policies, among them Interrupt Synchronization and Interaction between application code and device drivers [25]. This worked successfully in the first case; however, we ran into unexpected difficulties in the second case.

The scenario was as follows: In the default configuration, the invocation of device-driver services (such as `FloppyDriver::readBlock()`) is implemented by plain method calls, explicitly synchronized on a per-driver basis by mutex objects. A (more micro-kernel–like) architectural alternative for this implementation of Interaction and Synchronization is to employ message passing and active servers: Each device driver is a mini server that runs a message loop in its own thread. In [25], the `ServerSync` aspect implements this alternative (well encapsulated and transparently for application and device-driver developers) by the introduction of a `Thread` object into each driver class plus a piece of around-advice that intercepts all noninternal calls to device-driver services (`call("% FloppyDriver::%(...)") && !within("FloppyDriver")`) to transform them into messages that are sent to and dispatched by the introduced thread.

A significant side effect of the `ServerSync` aspect is that (by employing threads) it induces a new *functional dependency* `FloppyDriver` $\rightarrow$ `Scheduler`, which had not been reflected in the original PURE design. Such *ex post* changes to the functional hierarchy of an operating system are risky; they may induce dependency cycles or otherwise invalidate correctness assumptions of the system's design [13, 23].

In our initial tests and analyses for [25], this new dependency was apparently compatible to the existing design; `FloppyDriver` and `Scheduler` were completely unrelated before. Later, however, we realized that both concerns *indirectly* interact with each other – via *another* policy we had *not* explicitly considered before: Initialization. If device drivers employ threads, the `Scheduler` has to be initialized *before* the driver objects – whereas in the default configuration the order of initialization does not matter. The latter is the correctness assumption that is invalidated by the `ServerSync` aspect. The problem: PURE implements the Initialization of system components (such as `Scheduler` and `FloppyDriver`) by means of C++ global instance construction, for which the order is undefined across different compilation units; it cannot be influenced by aspects, such as `ServerSync`. In the end it turned out to be *technically* impossible to turn Interaction (and several other architectural poli-

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

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

Figure 1: Join point ambiguity in the eCos kernel. Because `Cyg_Scheduler::unlock()` is not only used to enforce Synchronization, but also Preemption, the related execution join points are ambiguous.

cies) into fully configurable features by means of aspects due to (1) *hidden dependencies* to *other* policies that (2) were designed in an *"AOP-unfriendly"* way.

2.2.2 Synchronization and Preemption in eCos

eCos, the *embedded Configurable operating system* [9, 22] is an industry-strength and broadly accepted open-source operating system family for the embedded systems domain. Including all optional packages, eCos offers more than 750 configuration options; the kernel itself consists of 5,000 lines of C++ code and offers nearly 100 configuration options, which are technically implemented by means of the C preprocessor – an "`#ifdef` hell".

As part of a larger case study about the run-time and memory effects of AOP, we refactored 16 eCos configuration options and kernel policies from conditional compilation into aspects – and thereby achieved a much better separation of concerns *without* extra run-time and memory costs [21]. One of these polices was Synchronization, which in eCos enforces mutual exclusion between threads and in-kernel interrupt handlers (called *deferred service routines, DSRs*) in order to ensure consistency of kernel state. Although only required if both threads *and* DSRs are actually employed by the application (many embedded applications use only either one), Synchronization is a mandatory feature in eCos that always causes run-time and memory costs. This is probably due to its implementation, which homogeneously cross-cuts large parts of the kernel source base: Each kernel function (as shown in Figure 1) is wrapped by calls to `Cyg_Scheduler::lock()` and `Cyg_Scheduler::unlock()` (187 invocations in total).

Extracting and separating Synchronization into an aspect was straightforward and clearly improved the clarity of the code [21]. However, our further attempts to thereby also *improve* the tailorability of eCos (turning Synchronization into a truly optional feature should be simple if implemented by an aspect) have failed. The eCos developers exploited the fact that `Cyg_Scheduler::unlock()` is called by all kernel functions immediately before the kernel is left to piggyback the enforcement of another central kernel policy on it. `Cyg_Scheduler::unlock()` does not only re-enable DSR propagation (Synchronization), it also activates the scheduler to possibly preempt the running thread (Preemption). The result of this "clever optimization" is ambiguity: Apparently, all 101 invocations of `Cyg_Scheduler::unlock()` that can be found in the kernel sources represent a point of Synchronization (like in `Cyg_Alarm::enable()` in Figure 1), but only 51 of them also represent a point of Preemption, for which the scheduler activation is actually necessary (like in `Cyg_Mutex::unlock()`). However, both concerns are *not distinguishable* on the join-point level; leaving out the enforcement of Synchronization would partly remove Preemption as well – even though both policies are conceptually independent concerns and all Synchronization code has been well encapsulated into an aspect.

2.3 Lessons Learned – A Summary

The respective "show stoppers" we encountered in the described PURE and eCos problem cases may appear to be very specific. Nevertheless, they exemplify some *general issues* we have found over the years in our attempts to achieve the configurability of even fundamental system policies in operating systems:

Hidden concerns caused by correctness assumptions that manifest only *implicitly* in the specification, in the design and – especially – in the implementation of the operating system.

Missing join points caused by optimizations, low-level code, and a generally "aspect-unfriendly" design and implementation.

The Initialization and Preemption policies in PURE and eCos, respectively, are examples for (partly) *hidden concerns*. Conceptually orthogonal to the rest of the system, their *concrete realization* causes *hidden functional dependencies* with respect to other concerns. However, even though eventually revealed and *theoretically resolvable*, it turned out as technically impossible to actually resolve these issues by the aspects, because of *missing join points* in the system's *design* and *implementation*.

2.3.1 Hidden Concerns

Hidden concerns can probably be found in any type of software, but operating systems are particularly prone to them. In our experience, they are often caused by the (comparatively complex) internal control-flow interaction schemes: With *interrupts*, *DSRs*, and *threads*, even small operating systems, like PURE and eCos, support at least three different *types* of control flows, all of which bear specific (and often subtle) interaction constraints. Interrupts and DSRs, for instance, must never block, whereas threads have to be aware of preemption and interruption at any time. Features that have been designed and implemented with *implicit assumptions* in this respect (like "this code is {never | always} invoked on {interrupt | thread | ...} level") can be found in every operating system. These assumptions, however, are often invalidated if we implement fundamental system policies as configurable features: A developer implementing the Threading concern, for instance, typically does this under the assumption that the context-switching code is never invoked from the interrupt level; hence, it does not have to be interrupt-safe. Now consider a `IRQThreads` policy aspect that reduces interrupt latencies by mapping interrupt requests to thread activations (a strategy implemented, for instance, by Solaris [17]). If `IRQThreads` is applied, the original assumption is no longer valid. This is not *per se* an issue if (1) the aspect developer is aware of this fact, *and* (2) it is possible to resolve the new dependency by

augmenting the context-switching code by an aspect in order to make it interrupt-safe.

2.3.2 Missing Join Points

The latter could be achieved, for instance, by a piece of advice that disables (re-enables) interrupts before (after) each context switch – but only if *all* respective thread transitions are exposed as unambiguous join-points to which aspects safely can bind. In theory, missing join-points should never be an issue: "Just program like you always do, and we'll be able to add the aspects later" [11]. In practice, the *obliviousness principle* in the form as postulated by FILMAN and FRIEDMAN has probably not passed the reality check for any type of software. However, in our experience, the problem of missing join-points is *particularly* challenging in operating systems: The context-switching code, for instance, may be scattered over the scheduler implementation for optimization purposes (the case in eCos). Parts of it are typically written in assembly language, which does not expose join-points for aspects written in a high-level language, such as AspectC++. In other cases, join-points are exposed by fragile *near-hardware code* – to which aspects better do not bind, because the transformations performed by the aspect weaver will probably break this code. A further issue are *join-point ambiguities* like we found in eCos.

Join-point ambiguities seem to be a general problem of optimized systems code: ÅBERG and colleagues found a similar situation in the Linux scheduling code and had to enhance their pointcut language by temporal logic [1] to disambiguate scheduling events at run time with stateful aspects (very similar to the later *tracematches* [2]). Besides the *additional* run-time overhead such approach induces it also effectively results in some sort of late binding of the respective features (such as Preemption and Synchronization). This spoils dead-code elimination and, thereby, the original goal: To remove unneeded functionality from the resulting binary. Embedded systems engineers consider such overheads as unacceptable – especially if a static solution *would* be possible.

We expected all these problems to become even more severe if we want to configure *many* fundamental policies.

3. THE CIAO APPROACH

Based on our experiences with using AOP in the PURE and eCos operating systems, we have developed an integrated analysis and development approach for operating systems. Our approach provides for (1) the *early identification* of hidden concerns and (2) their *aspect-aware* design and implementation.

The overall goal is to reach *configurability* even *of fundamental system policies*, whose implementation is highly cross-cutting and interacting in traditional operating systems. Furthermore, by using aspects for the implementation of system configurability, we want to reach a *more fine-grained level of configuration* possibilities – without trading off efficiency in terms of resource usage. Since hardware resource usage is crucial in most embedded systems, our precondition here is to aim at overhead-free configurability, as it would be possible with traditional conditional compilation.

In order to keep the investigation of the suitability of AOP for operating system engineering as independent as possible, we decided to start with a publicly available standard in the domain – AUTOSAR OS [4, 3]. This way, the choice of OS abstractions and system services and their functionality is not biased by the intended AOP implementation. In fact, AUTOSAR OS is very C-focused, and most implementations are therefore also in C, configured by some kind of code preprocessor. We briefly introduce AUTOSAR in Section 3.1, followed by a description of how we analyzed the specification for crosscutting with a method called Concern Impact Analysis (see Section 3.2). After that, we present our concept of aspect-aware operating system development in Section 3.3.

3.1 AUTOSAR OS

AUTOSAR is an initiative formed by all major automotive manufacturers and suppliers like BMW, Ford, Toyota, and Bosch. Their goal is to standardize the interfaces and functionality of the operating system and drivers in automotive microcontrollers in order to facilitate application development in the domain. The operating system standard, AUTOSAR OS [4, 3] describes a kernel that is completely statically configured; the overall system configuration is known at compile time.

AUTOSAR OS offers different kinds of abstractions to the application programmer. Among the control flows, there are tasks (named threads in other operating systems) and hooks, which are callback functions invoked when the corresponding internal point in the system is reached (e.g., upon a task switch, or upon a protection violation). Interrupt services routines (ISRs) are invoked asynchronously by the hardware; ISRs of category 1 must not use OS services, whereas ISRs of category 2 are allowed to invoke the kernel and must therefore be synchronized with the kernel in order not to corrupt kernel state. Tasks and ISRs themselves can synchronize by acquiring and releasing AUTOSAR resources; AUTOSAR events can be used for task and ISR notification. AUTOSAR alarms allow the application to take action after a specified amount of time has elapsed.

The main point that distinguishes AUTOSAR OS from other operating systems in the domain is its configurable support for properties of architectural kinds. These include the decision to make the system fully-, mixed-, or non-preemptable, and different levels of protection between AUTOSAR applications. Protection entails memory protection to prevent memory corruption, timing protection to ensure that applications will not miss their deadlines because of another, misbehaving application, and service protection, which checks for correct usage and context of system-service invocation.

3.2 Concern Impact Analysis

In order to be able to design an aspect-aware system, the developer not only has to scope the system's functionality in the analysis phase, but he also needs to assess the effect of configurable concerns on the system. This raised awareness of the different concerns in the system and their relationships to each other are the main goal of our specialized analysis process named Concern Impact Analysis (CIA; see Figure 2). CIA eventually aims to provide the information necessary to map the initially abstract concerns (c and d in Figure 2) to design and implementation artifacts such as classes or aspects (g–l in Figure 2).

In a first step, the developer scrutinizes the given requirements in the form of a specification or abstract requirements list, mining it for distinguishable concerns in the target domain. This includes concerns that will be kept configurable (and therefore omissible) in the final program family, but also concerns that are fundamental to all system variants. To some extent, this subprocess requires the knowledge of a domain expert, who will also be able to identify concerns that are internal to a system. Such internal concerns are rarely mentioned explicitly in a requirements or specification document; nevertheless, many of them are vital to a working software system. The overall outcome of the first step is a list of identified *explicit* concerns plus a preliminary list of identified *internal* concerns, both of which serve as a basis for the following impact analysis. Obviously, the concern list is a super set of all features that *could* be present in a system; it is subject to tailoring by the system configurator by excluding features from the configuration.

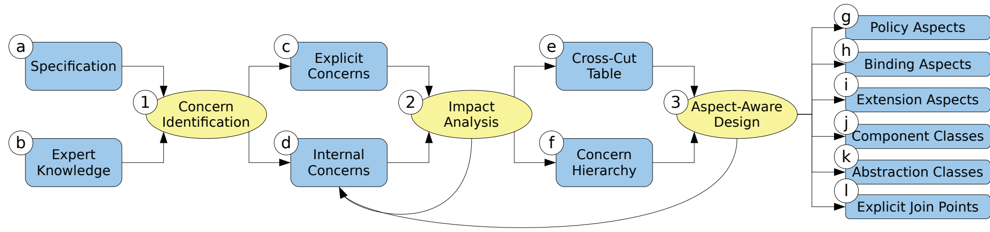

Figure 2: The process of Concern Impact Analysis (CIA) to aid aspect awareness in the system design

	System abstractions (functional)							Callbacks		Protection facilities (architectural)												Internal	
	OS control	Tasks	ISRs category 1	ISRs category 2	Resources	Events	Alarms	Alarm callbacks	Hooks	OS applications	Memory protection	Stack monitoring	Timing protection	Invalid parameters	Out of range	Wrong context	Missing task end	Enable w/o disable	Interrupts disabled	Nontrusted shutdown	Foreign OS objects	Preemption	Kernel sync
`GetActiveApplicationMode()`	⊕								◑							◐			◐				
`StartOS()`	⊕													◐		◐			◐		◐		
`ShutdownOS()`	⊕								◑							◐			◐	◐			
`ActivateTask()`		⊕							◑					◐		◐			◐		◐	◑	
`TerminateTask()`		⊕							◑							◐			◐			◑	
`ChainTask()`		⊕							◑					◐		◐			◐		◐	◑	
`Schedule()`		⊕							◑							◐			◐			◑	
`GetTaskID()`		⊕							◑							◐			◐				
`GetTaskState()`		⊕							◑					◐		◐			◐		◐		
`EnableAllInterrupts()`			⊕										◐			◐		◐					
`DisableAllInterrupts()`			⊕										◑			◐							
`ResumeAllInterrupts()`			⊕										◐			◐		◐					
`SuspendAllInterrupts()`			⊕										◑			◐							
`ResumeOSInterrupts()`				⊕									◐			◐		◐					
`SuspendOSInterrupts()`				⊕									◑			◐							
`GetISRID()`				⊕												◐			◐				
`DisableInterruptSource()`				⊕					◑					◐		◐			◐		◐		
`EnableInterruptSource()`				⊕					◑					◐		◐			◐		◐		
`GetResource()`					⊕				◑				◑	◐		◐			◐		◐		
`ReleaseResource()`					⊕				◑				◐	◐		◐			◐		◐	◑	
`SetEvent()`						⊕			◑					◐		◐			◐		◐	◑	
`ClearEvent()`						⊕			◑							◐			◐				
`GetEvent()`						⊕			◑					◐		◐			◐		◐		
`WaitEvent()`						⊕			◑							◐			◐			◑	
`IncrementCounter()`							⊕		◑					◐		◐			◐		◐	◑	
`GetAlarmBase()`							⊕		◑					◐		◐			◐		◐		
`GetAlarm()`							⊕		◑					◐		◐			◐		◐		
`SetRelAlarm()`							⊕		◑					◐	◐	◐			◐		◐		
`SetAbsAlarm()`							⊕		◑					◐	◐	◐			◐		◐		
`CancelAlarm()`							⊕		◑					◐		◐			◐		◐		
`StartScheduleTableRel()`							⊕		◑					◐	◐	◐			◐		◐		
`StartScheduleTableAbs()`							⊕		◑					◐	◐	◐			◐		◐		
`StopScheduleTable()`							⊕		◑					◐		◐			◐		◐		
`NextScheduleTable()`							⊕		◑					◐		◐			◐		◐		
`SetScheduleTableAsync()`							⊕		◑					◐		◐			◐		◐		
`SyncScheduleTable()`							⊕		◑					◐		◐			◐		◐		
`GetScheduleTableStatus()`							⊕		◑					◐		◐			◐		◐		
`GetApplicationID()`										⊕						◐			◐				
`TerminateApplication()`									◑	⊕						◐			◐				
`CallTrustedFunction()`									◑	⊕						◐			◐				
`CheckObjectAccess()`										⊕				◐		◐			◐		◐		
`CheckObjectOwnership()`										⊕						◐			◐				
`CheckISRMemoryAccess()`											⊕			◐		◐			◐		◐		
`CheckTaskMemoryAccess()`											⊕			◐		◐			◐		◐		
`AppModeType`	⊕	⊛					⊛																
`TaskType`		⊕			⊛	⊛				⊛	⊛	⊛	⊛								⊛	⊛	
ISR category 2				⊕						⊛			⊛								⊛		
`ResourceType`					⊕					⊛											⊛		
`AlarmType/ScheduleTableType`		⊛				⊛	⊕	⊛		⊛					⊛						⊛		
`ApplicationType`										⊕	⊛												
alarm expiry		◑				◑		◑														◑	
category 2 ISR execution											●		●									◑	
system startup		◑					◑		◑														
system shutdown									◐														
protection violation									◑														
task switch									●		◑	◐	●										
application switch											◑										◑		
uncontrolled task end																	◑						
user $\mapsto$ kernel transition											◑												◐
kernel $\mapsto$ user transition											◑												◑

Table 1: Influence of configurable concerns (columns) on system services, system types, and internal events (rows) in AUTOSAR OS. Kind of influence: ⊕ = introduction of a service or type, ⊛ = impact on a type, ◐ / ◑ / ● = impact before / after / around a service or internal event.

3.2.1 Cross-Cut Table

The main CIA step is the actual analysis of the impact each concern has on the system to be built (see step 2 in Figure 2). The impact is classified and visualized in a cross-cut table matrix (e in Figure 2) as exemplified in Table 1 with the analysis of the AUTOSAR OS specification and the design of CiAO.

As a starting point for the cross-cut table (which is, to some degree, comparable to a design structure matrix [7]), the columns list the concerns as identified in the first step, whereas the rows above the double line list the API of the system to be built, in this case AUTOSAR OS. The API includes both system services (listed in semantic groups in Table 1) as well as instantiable system types. The events listed below the double line include system-internal transitions that are of importance to one concern or the other. This list of transitions does not form until the actual analysis of the concerns takes place, which is when the need to list these transitions emerges.

In order to populate the cross-cut table, in an iterative process, each concern is analyzed for its impact on the system to be. This mainly entails three types of impact:

1. Extension of the system API by a service or a type. Some concerns are reflected in a system's interface to the using components, since without them, certain services cannot be offered by the system. Thus, those concerns *introduce* API services and types, denoted by a ⊕ sign in the cross-cut table.

2. Modification of a system service or its functionality. Several concerns will not affect the system's external interface, but they will alter or adapt its functionality. Usually, this adaptation only affects the execution or invocation of well-selected services, which is denoted by a ◐, ◑, or ● sign in the corresponding row and column in the table. If besides that, a concern needs to be notified of additional events that are internal to the system, that event is listed in an additional row below the double line and marked to be influenced by the given concern.

3. Extension of a system type. In some cases, a concern will not introduce a new type to be able to fulfill its duty, but it will instead need to extend an existing API type as introduced by another concern. This type-internal extension is denoted by a ⊛ sign in the cross-cut table.

We are aware of the fact that producing the cross-cut table means thinking about the concerns' *implementation* to some degree already in the analysis phase. However, we think that this is crucial to be able to design a complex configurable software system in an aspect-aware manner, because it is the cross-cut table that enables the developer to make informed decisions about the system architecture. Furthermore, forcing the developer to think about the impact of each concern will reveal additional internal concerns that were previously hidden in the requirements (see feedback loop from step 2 in Figure 2). A typical example from the operating systems domain is kernel synchronization, which is rarely mentioned but vital to keeping kernel state consistent if interrupt service routines are allowed to call system services. Revealing and analyzing such concerns in the early analysis stage makes the subsequent design process respect them explicitly.

Consider, for instance, the different sets of concerns as depicted in Table 1 (vertical analysis view). The system abstraction concerns each canonically extend the system API by the corresponding system services and types of the given abstraction, since they extend the system *functionally*. Note that each system service is introduced by exactly one concern (i.e., exactly one ⊕ sign per service/type row). The architectural concerns, however, which all implement some form of protection mechanism, mostly influence and enhance *existing* system services. Some of them are highly cross-cutting (e.g., consider the column corresponding to the concern "wrong context", which checks for the correct invocation context of a system service call), whereas others have a very selective influence on the system (e.g., consider the concern "nontrusted shutdown", which prevents nontrusted application from shutting down the whole system). By making the type of influence and its locality and dimension explicit in the cross-cut table diagram, the concerns can be directly considered to be modeled as a class or an aspect in the aspect-aware design step (see Section 3.3). Highly cross-cutting concerns, for instance, will probably benefit the most from AOP quantification mechanisms and are likely candidates for an aspect module implementation.

On the other hand, consider the system services and types as depicted in Table 1 (horizontal analysis view). Note that not a single service is influenced by only one concern; in fact, system services such as ReleaseResource() are influenced by as many as eight concerns! Hence, such "hot spot" services require special attention in the design, and potential aspect implementations of influencing concerns need to be carefully ordered not to break each other's functionality.

Ultimately, a comprehensive analysis with a comprehensive impact table like the one in Table 1 will provide an ideal basis for the aspect-aware design of the system concerns, making the implementation a straight-forward step.

3.2.2 Concern Hierarchy

The second artifact to be output by the impact analysis step besides the cross-cut table is a concern hierarchy (f in Figure 2). Due to space constraints, we have omitted the resulting concern hierarchy of CiAO from this paper. However, a concern hierarchy is basically a functional hierarchy [13] enriched by *influence* relationships between the different (sub) concerns. It describes, on the one hand, which concern *uses* which other concerns: This means that the respective concerns are tightly coupled – the functional correctness of the using concern depends on the one of the used concern. On the other hand, an extension to functional hierarchies, a concern might only *influence* other concerns: This indicates a *loose* coupling; if one of the target concerns is not included in a given system configuration, the source concern will still be able to fulfill its specification semantically.

We found that a concern that extends one or more system types (denoted by a ⊛ sign in the cross-cut table) typically *uses* the concerns that introduce the respective types. In AUTOSAR OS, for instance, events are generally task bound, hence, the Events concern *uses* the Tasks concern. This is already indicated in Table 1 by the fact that Events extends `TaskType`, which is introduced by Tasks.

A concern that modifies system services only, on the other hand, usually *influences* the respective target concerns. The Interrupts disabled concern, for instance, ensures for a number of system services (among them `GetResource()` and `ReleaseResource()`) that they are not invoked while running on interrupt level (with interrupts disabled). If some *influenced* concern (e.g., Resources) is not present in the current configuration, this property is implicitly true for its services.

3.3 Aspect-Aware Design

Eventually, the system's concerns and their interactions have been identified and described as far as possible. The goal of the following step 3 (Figure 2) is then to compose the gained knowledge into a model of *classes*, *aspects*, and, where necessary, *explicit join points*

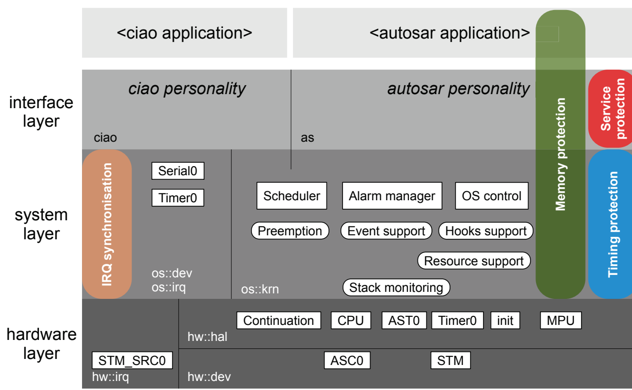

Figure 3: Layered structure of CiAO. Depicted are the three fundamental layers of the CiAO architecture with a selection of their sublayers, components, abstractions, and aspects (depicted with rounded corners).

(documents g–l). Based on our experience with PURE and eCos, this process is guided by three fundamental principles of aspect-aware software development:

The principle of loose coupling. Make sure that aspects can hook into all facets of the static and dynamic integration of system components. The *binding* of components, but also their *instantiation* (e.g, placement in a certain memory region) and the time and order of their *initialization* should all be established (or at least be influenceable) by aspects.

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

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

All subsequent design and implementation decisions are evaluated with respect to these three principles.

3.3.1 CiAO Architecture

The first steps towards *aspect-awareness* are already made in the architecture of the system: Like most operating systems, CiAO is based on a layered architecture, in which each layer is implemented using the functionality of the layers below (Figure 3). The only exceptions from this are the aspects implementing architectural policies, which may take effect across multiple layers.

On the coarse level, we have three layers. From bottom up these are: the *hardware layer* (`hw`, hardware programming interface), the *system layer* (`os`, the operating system itself), and the *interface layer* (`ciao` or `as`, the (configurable) application programming interface).

This architecture is aspect-aware in the sense that layers do not only serve as conceptual levels of abstraction, but also as a means to provide cross-layer control-flow transitions on the join-point level (*visible transitions*). Each layer is modeled as a top-level C++ namespace or class, which makes it easy to grasp such transitions by pointcuts, like the following AspectC++ pointcut yields all join points where a system-layer component (namespace `os`) accesses the hardware (namespace `hw`):

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

Control-flow transitions *down* the layer hierarchy (such as the invocation of some system service) are established by method calls; aspects can interfere with these transitions by giving advice to a pointcut like `OStoHW`. Transitions *up* the hierarchy (*upcalls,* such as a thread start or a signal delivery) are modeled as explicit join-point shadows and *only* established by aspects (*loose coupling*). In the case of CiAO, aspects thereby can hook into all transitions into and out of the *system layer* that are visible on the static join-point level (*visible transitions*).

3.3.2 Classes and Aspects

With respect to the three design principles: Which concerns are best to be implemented as classes and which as aspects? With respect to *loose coupling*, we came up with the following general rule: Some concern is implemented as a class if – and only if – it represents a *distinguishable run-time–instantiable concept* of the system, otherwise it is realized as an aspect.

In the case of CiAO, this holds in particular for the **system abstractions** taken from the system specification document and identified during concern analysis. System abstractions (`AppModeType`, `TaskType`, and so on) are directly listed in the cross-cut table (Table 1, column 1) and represent the OS-managed entities that are instantiated on behalf of the application. Furthermore modeled as classes are the **system components**, which horizontally subdivide the architectural layers and represent its functional sub-domains (such as the `Scheduler` or the `AlarmManager` in the system layer, see Figure 3). Their identification is guided by the concern hierarchy, but also requires a decent amount of expert knowledge regarding (potential) synchronization and protection domains. The point, however, is: All classes that represent system abstractions and system components are *sparse* or even *empty*, that is, they implement only the *minimal base* of the respective concern (*minimal extensions*). Their major purpose is to provide a *distinct scope* for introductions of cross-component interactions (*visible transitions*). All further features are "filled in" by the aspects. During the development of CiAO we came up with three idiomatic roles of aspects:

1. **Extension aspects** add additional features to a system abstraction or component (*minimal extensions*), such as extending the scheduler by means for task synchronization (e.g., AUTOSAR OS resources).

2. **Policy aspects** "glue" otherwise unrelated system abstractions or components together to implement some kernel policy (*loose coupling*), such as activating the scheduler from a periodic timer to implement time-triggered preemptive scheduling.

3. **Upcall aspects** bind behavior defined by higher layers to events produced in lower layers of the system, such as binding a driver function to interrupt events.

Extension aspects can be identified in the cross-cut table by the fact that they affect especially the static structure, typically by introducing some system services. Most extension aspects accompany some system abstraction (e.g., `ResourceType`); they integrate the actual *implementation* of the respective concern (Resources) into the system components (`Scheduler`) and extend the interface layer by the corresponding services (`GetResource()`, `ReleaseResource()`).

Policy aspects, in contrast, lead to a different system behavior. In the cross-cut table they can be identified by seeking concerns that (mostly) affect the dynamic structure of the system, like Preemption.

concern	extension	policy	upcall	advice	join points	extension of | advice-based binding to
ISR cat. 1 support	1		m	$2+m$	$2+m$	API, OS control | m ISR bindings
ISR cat. 2 support	1		n	$5+n$	$5+n$	API, OS control, scheduler | n ISR bindings
Resource support	1	1		3	5	scheduler, API, task | PCP policy implementation
Resource tracking		1		3	4	task, ISR | monitoring of Get/ReleaseResource
Event support	1			5	5	scheduler, API, task, alarm | trigger action JP
Full preemption		1		2	6	| 3 points of rescheduling
Mixed preemption		1		3	7	task | 3 points of rescheduling for task / ISR
Wrong context check		1		1	s	| s service calls
Interrupts disabled check		1		1	30	| all services except interrupt services
Invalid parameters check		1		1	25	| services with an OS object parameter
Error hook			1	2	30	scheduler | 29 services
Protection hook	1	1		2	2	API | default policy implementation
Startup / shutdown hook			1	2	2	| explicit hooks
Pre-task / post-task hook			1	2	2	| explicit hooks

Table 2: Selected CiAO-AS kernel concerns implemented as aspects with number of affected join points. Listed are selected kernel concerns that are implemented as *extension*, *policy*, or *upcall aspects*, together with the related pieces of *advice* (not including order advice), the affected number of *join points*, and a short explanation for the purpose of each join point (separated by "|" into *introductions of extension slices | advice-based binding*).

Upcall aspects realize *loose coupling* with respect to upcalls, as described in Section 3.3.1. They are invisible in the cross-cut table, as most of them do not manifest before the implementation phase.

Table 2 displays an excerpt of the list of AUTOSAR OS concerns that are implemented as aspects in CiAO. The first three columns list for each concern the number of *extension*, *policy*, and *upcall aspects* that implement the concern. (The resource-support aspect and the protection-hook aspect have both an extension and a policy facet.) The majority of concerns contribute to the set of *policy aspects* (12 aspects), which is followed by the set of *extension aspects* (9 aspects). The number of *upcall aspects* ($3+n+m$) differs from these in so far as it does not only depend on the system configuration, but also on the application configuration: Each specified ISR in the application is bound with the respective interrupt source in the kernel or hardware access layer (HAL) by its own upcall aspect. These aspects are, however, not to be provided by the application developer; they are generated automatically from the application configuration.

3.3.3 Explicit Join Points

By consequent application of the fundamental principles of aspect-aware development in the architecture and design of the system, CiAO already offers a rich join-point interface "by structure". Nevertheless, in many cases the implicit join-point interface is not ample enough. This has conceptual as well as technical reasons:

1. Implicit join points are inherently implementation dependent. Their amount – but especially their semantics – may be inconsistent between different implementations of the same concept. This is absolutely acceptable for component-specific extension aspects, as these aspects have to know the component they extend anyway. It is, however, not satisfying for system aspects that implement more general policies.

2. Some semantically important control-flow transitions are not visible at the join-point level because they do not occur on the boundary of function calls or executions. In other cases, their place of occurrence is configuration-dependent, or there are multiple places of occurrence. For example, *user* $\mapsto$ *kernel* transitions might occur if a kernel function is called, when a trap handler is activated, or during task switching to another task. However, in CiAO, this is a matter of configuration.

3. Several semantically important control-flow transitions are not available as join points because of technical reasons. This is often the case with low-level system abstractions, such as interrupt handlers or the implementation of the context switch mechanism.

For these reasons, many CiAO components and layers provide furthermore a well-defined **explicit join-point interface** that defines one or several *explicit join points*. An **explicit join point** is a named join point in the kernel control flow that bears precisely defined semantics and can safely be advised. Technically, explicit join points are implemented as empty methods – provided for the sole purpose that aspects can bind to them. The join-point provider invokes these methods at run time, either directly or indirectly by component-specific adapter aspects.

Conceptually, explicit join-point interfaces can be compared to hooks or interceptor interfaces in other component models. An advantage of explicit join points is, however, their low overhead. In most cases (that is, when they do not have to be triggered from parts written in assembly language) they can be implemented as empty inline methods, which get optimized away by the compiler if no aspect binds to them. Another advantage is the inherent support for $1:n$ relationships – handler chaining for shared interrupt sources, for instance, is supported "out of the box".

We distinguish between **upcall join points** and **transition join points**. The former are the interface that *upcall aspects* bind to; the incarnations of hardware interrupts or threads, for instance, are provided in this realm. Another example is the system initialization handler `hw::hal::init()`, which is invoked during system startup. Upcall join points manifest naturally in a bottom-up development process.

Transition join points, in contrast, mark events that are important for the implementation of system policies. They are typically identified during concern analysis and can be taken directly from the bottom of the cross-cut table. An example we can find there (Table 1) are the already mentioned *user* $\mapsto$ *kernel* transitions, which in CiAO are provided as explict join points `os::krn::enterKernel()` and `os::krn::leaveKernel()`. Other examples include transitions

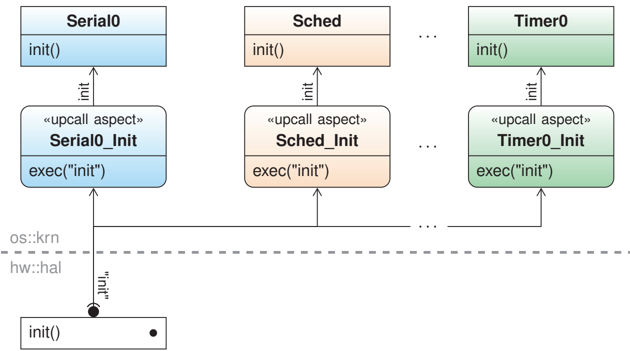

Figure 4: Self-integration of components. Depicted is the CiAO component initialization scheme. Every system component integrates itself into the system initialization handler `hw::hal::init()` by an accompanying `_Init` upcall aspect.

from thread level to interrupt level, or the context switch from one thread to another. These transitions often have *multiple* and *implementation-dependent* sources ($m : n$ relationships); or they occur in fragile, low-level parts of the implementation. By representing them as explicit join points, providers and publishers of transition events can be decoupled.

3.4 Problems Revisited in CiAO

The following two examples from CiAO resemble some of the issues we encountered in PURE and eCos (namely, *hidden concerns* and *missing join points*, see Section 2.3), in the sense that they demonstrate how such problems can be avoided by applying the principles of aspect-aware software development.

3.4.1 Self-Integration of Components

The key to *loose coupling* of policies and components is to provide the necessary explicit join points and then establish *all* bindings by advice. Figure 4 shows this on the example of component initialization: Every system component (which are singletons by definition) has an accompanying `_Init` aspect that gives advice to the system initialization handler `hal::init()` (an explicit join point) to invoke the component's `init()` method at system startup time. Thereby, the startup code does not have to know which components are present in the actual CiAO configuration. Nevertheless this flexibility does not come at a price, as all initialization code gets bound and inlined at compile time. This is not only more efficient than the initialization concept used in PURE (which was based on global instance construction, see Section 2.2.1), it also is a lot more flexible. Component initialization thereby becomes a *visible transition*, which we can *further* influence it by additional aspects: Consider, for instance, an (optional) extension aspect `Serial0Ext` that extends the serial driver from Figure 4 by a task of its own (e.g., for some background protocol handling). Similar to the `ServerSync` aspect in the PURE study (Section 2.2.1), this aspect effectively inserts a new functional dependency between the serial driver and the scheduler; the serial driver now *uses* the scheduler. The consequence for the implementation is that the scheduler has now to be initialized *before* the serial driver. In AspectC++, we can realize this new constraint relatively easy by employing *order*-advice [26]. Additional to the extension of the class `Serial0`, the aspect `Serial0Ext` can specify a partial invocation order for the *foreign* aspects `Sched_Init` and `Serial0_Init` at the join point `execution( "void hw::hal::init()" )`:

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

Essentially, the aspect thereby re-establishes a correct functional hierarchy of the system. This is possible because of the application of the principles of *loose coupling* and *visible transitions.*

3.4.2 Self-Integration of Policies

Another common use case for advice-based binding in CiAO is the self-integration of policies. Self-integration of policies is crucial for the aspired decoupling of policies and mechanisms. Most policy implementations induce new interactions between (otherwise unrelated) components. This may, again, lead to new functional dependencies that we also have to deal with. Figure 5 demonstrates self-integration of policies by the example of two variants of the CiAO preemption policy (which, to some degree, resembles the issues we found in eCos, see Section 2.2.2):

Generally, system components report the need for rescheduling (and, thus, potential preemption of the running task) by calling `Sched::setNeedReschedule()`. The actual activation of the scheduler is, however, delayed:

(a) The aspect `Sched_LeaveBinding` in Figure 5.a implements a simple delayed activation policy for a cooperative system; with this policy, preemption is only possible at the return from some system service. Technically, this is realized by binding the scheduler activation (`Sched::reschedule()`) to the explicit tranistion join point `leaveKernel()`, which is guaranteed to be triggered if some thread returns from the kernel.

(b) The aspect `Sched_ASTBinding` in Figure 5.b implements a more sophisticated delayed activation policy for an interruptive system; with this policy, preemption can also be triggered by interrupts. Technically, this is realized by binding the scheduler activation to the function `AST0::ast()`, which is the handler of an *asynchronous system trap*[1] (*AST*). Additionally, the triggering of the AST is bound to `setNeedReschedule()`. The fact that the scheduler is now activated from `AST0::ast()` leads to a new functional dependency, which has the consequence that the kernel now has to be synchronized on AST level. We can, however, easily enforce this constraint with additional pieces of advice that are given by the `Kernel_ASTSync` aspect:

```
aspect Kernel_ASTSync {
  advice execution( "os::krn::enterKernel()" ) : before() {
    AST0::Inst().disable(); // delay scheduling
  }
  advice execution( "os::krn::leaveKernel()" ) : after() {
    AST0::Inst().enable();  // point of rescheduling
  }
};
```

By *visible transitions* and advice-based binding we have achieved a completley *loose coupling* of the scheduler component and the preemption policy in the *implementation*. This makes it very easy to provide numerous variants of either concern an embedded systems engineer can choose from.

4. DISCUSSION

We discuss the combination of AOP and operating systems in general, how our approach can be applied to other system software, and the lessons learned with respect to language and tooling.

[1] An AST is a low-priority interrupt that can be triggered by higher-prority interrupts or the kernel to delay activities, such as scheduling, to a later point in time (e.g., when the kernel is left).

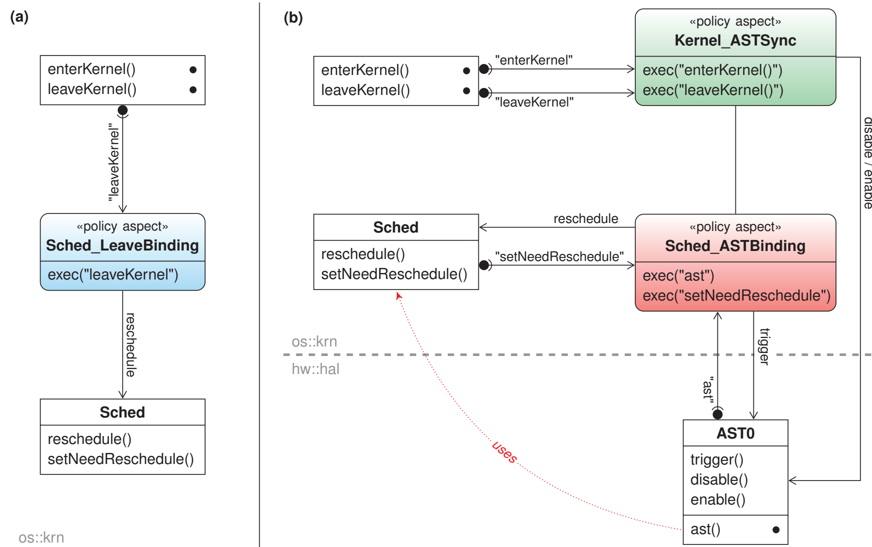

Figure 5: Self-integration of policies. Depicted are two alternatives for the delayed preemption policy in CiAO. **(a)** The aspect `Sched_LeaveBinding` binds to `leaveKernel()` to activate the scheduler when some task leaves the kernel (cooperative system). **(b)** The aspect `Sched_LeaveBinding` binds to the handler of an *asynchronous system trap* (*AST*) to activate the scheduler when all (potentially nested) interrupt handlers have terminated (interruptive system).

4.1 AOP and Operating Systems

4.1.1 Aspects as First-Class Entities

AOP has been facing much critique in the sense that aspects (in contrast to classes) do not represent real *domain concepts,* but (only) "aspects *of programming*". STEIMANN details this in [27]: "literally all aspects discussed in the literature are technical in nature: authentication, caching, distribution, logging, persistence, synchronization, transaction management, etc."

There might be some truth in this for the kind of software STEIMANN had in mind when writing his paper, but for the domain of system software, we have to clearly rebut this argument: System software *is* very technical in nature, too; the above mentioned "technical" aspects are text-book examples for *the* dominant concerns of system-software development! In the *specification* of AUTOSAR OS [3], for instance, we can find the *requirement* OS093:

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

This requirement (which maps to the Interrupts disabled concern in Table 1) translates almost "literally" to an AspectC++ aspect:

```
aspect DisabledIntCheck {  // implements OS093
  advice call( pcOSServices() && !pcInterruptServices() )
  && !within( pcHookRoutines() ) : around() {
    if( interruptsDisabled() )
      *tjp->result() = E_OS_DISABLEDINT;
    else
      tjp->proceed();
} };
```

So for our domain, we can assess that aspects lead to a much more natural separation of domain-specific *concepts* – if considered as first-class design elements from the very beginning.

4.1.2 Quantification and Obliviousness

The `DisabledIntCheck` aspect is also a good example for the benefits of *quantification* because of homogeneous cross-cutting. Given that other studies [14] about applying AOP for the fine-grained configuration of system software (in this case embedded databases) came to the conclusion that quantification is "rarely applicable", these benefits seem to be domain-specific to a certain degree. However, for the implementation of operating system policies, especially architectural ones, quantification clearly creates synergies. For 8 out of the 14 aspects listed in Table 2 this is the case.

With respect to *obliviousness*, the situation is less clear. In [11], FILMAN and FRIEDMAN describe the obliviousness *ideal* of AOP, according to which obliviousness can be a *bidirectional* relationship between components and aspects: The programmers of the base system and the aspect developers can work completely independently of each other. However, in actual applications of AOP, obliviousness is usually understood to be *unidirectional*: The components of the base system are kept oblivious of aspects – at the price that the aspects have to be perfectly aware of the components they affect. This often involves knowledge about certain implementation details, which in turn leads to fragile pointcuts if the component *developers* are kept oblivious of the aspects, too. Furthermore, this approach hits its limits when the base code just does not offer the required join-point shadows. The ambiguity problems we found in eCos are a good example here.

Aspect-aware operating system development moderates these issues by pragmatically considering *obliviousness* and *awareness* as two ends of a continuum: The more oblivious a component should be of the aspects that potentially engage with it, the more aware the aspects have to be of the component – and vice versa. Much of the flexibility and configurability of CiAO stems from the freedom to decide for each relationship about the placement on this continuum.

In our opinion, the advantage of the *advice*-mechanism of AOP is not so much quantification and obliviousness, but *loose coupling*: Essentially, advice inverts the direction in which control-flow relationships are specified. This facilitates the self-integration of the implementation of optional features into the control flows of the base system. Furthermore, advice-based binding is inherently loose – if the addressed join point is not present, the binding is sliently dropped. This property is useful for the implementation of *inter-*

acting optional features, which are difficult to tackle with other decomposition approaches [15].

4.2 Applicability to Other Domains

The presented methodology emerged from experiences in a special domain – highly-configurable system software for resource-constrained embedded systems. Nevertheless, it is at least partly applicable to a wider range of domains. For example, PUMA is a product line of C/C++ code analysis and transformation frameworks [28]. In this project we have not conducted the *concern identification* and *impact analysis* steps, but the principles of *aspect-aware design* and the underlying AspectC++ idioms including the three roles of aspects were applied and turned out to be generic and helpful.

4.3 Language and Tooling – Lessons Learned

4.3.1 AspectC++ – How the Language Is Evolving

When we started with the development of AspectC++ it seemed "natural to use AspectJ as a foundation when creating a set of extensions for the C/C++ language". This led to many similarities between the two languages such as advice code that is anonymous and, thereby, cannot be overridden by a derived aspect or the explicit interface for accessing join-point context information within advice code (thisJoinPoint-API).

However, it turned out that there are more differences between C++ and Java than initially expected, and also our application domain of deeply embedded systems forced us to rethink the language design with resource consumption in mind. In contrast to the beginning, AspectC++ now has a much stronger focus on static typing and language features that can be implemented completely at compile time. Run-time mechanisms such as the dynamic thisJoinPoint-API, which is typically used in combination with run-time reflection, are too expensive and, thus, have been mostly replaced by a static counterpart. For instance, the "join point API" of AspectC++ provides static type information for advice code. As a consequence, multiple variants of the same advice code can be instantiated at compile time, which depend on the matched set of join points. Additionally the advice can use the type information to instantiate C++ templates or even template meta-programs. Thereby, a complex chain of code generation steps can be triggered. It turned out that this combination of aspects and C++ templates is a very powerful mechanism that is a unique feature of AspectC++ [19].

Currently, a complete static introspection mechanism for all program entities – and not only join points – is under development. This will, for instance, allow generic aspects to very efficiently marshall/unmarshall any objects in order to transparently perform remote method invocations or to manage a persistent state. In the context of CiAO this feature shall be used to transparently copy objects between address spaces when isolation is turned on and tasks in different address spaces interact.

Even though AspectC++ is already very useful, we identified the following missing features, which are on the agenda for future enhancements:

Free variables in pointcut expressions. This is a language feature that is already known from LogicAJ [18]. It would significantly enhance the expressiveness of AspectC++ pointcut expressions.

Extensible pointcuts. Self integration of components such as device drivers would be easier if named pointcuts could be extended or composed from collected fragments. For instance, a driver has certain properties: It services interrupts, it handles a block device, and it needs a helper thread. Aspects should be able to affect all components with a specific property. However, the system configuration – including the set of configured drivers – is unknown before compile time. AspectJ 5 users can achieve this goal by exploiting Java 5 annotations. For AspectC++ a similar mechanism shall be integrated.

More control over code generation. When low-level assembler code and AspectC++ are combined it is often necessary to control the code generation very precisely. For instance, in a function or advice that implements a context switch between tasks and that contains inline assembler code, it is crucial to know whether the function will be inlined by the compiler. If the compiler behaves unexpectedly, a machine crash will be unavoidable.

Non join points. Some parts of the CiAO operating system should simply be guaranteed to never be touched by any aspect. We aim at providing mechanisms to specify these parts in a modular manner and a weaver extension that obeys these rules.

4.3.2 User Experience – AOP for "Hackers"

More than a dozen master students were involved in the development of PURE, the aspectized version of eCos, and CiAO, and contributed a significant amount of the aspect code to these systems. All of them were advanced C/C++ hackers, the majority already had some experience in low-level kernel programming, and all of them carried on with R&D in the domain of low-level system software after finishing their studies. So, to a certain degree this group represents the typical "kernel hacker", whose take on AOP might be interesting to the AOSD community. While we have not evaluated this in a systematic way, we nevertheless observed some recurring peculiarities:

AOP semantics is generally easy to grasp. To our (pleasant) surprise, the students generally had, after a brief introduction into the topic (a three hour lecture plus a "toy" exercise), little to no problems in understanding AOP concepts, the AspectC++ language, and the particularities of its application to embedded systems. They grasped the CiAO development idioms and application patterns by examining the existing code and were quickly able to contribute their own aspects.

Technical side effects of aspect weaving are more challenging. In theory, aspect weaving should be a transparent process, but in practice it is not – due to technical side effects. A frequent and always challenging issue, for instance, was the understanding and resolving of *`#include` cycles*. Such a cycle appears if two header files (indirectly) `#include` each other, which in most cases leads to uncompilable code. Unexpected `#include` cycles are a tough problem for any larger C/C++ project. The point is that they appear *a lot* more frequently with aspect weaving: An aspect that itself `#includes` some external module (a property that holds for any nontrivial aspect) thereby also contributes to the list of `#include` files of the modules it affects in the weaving process, which often results in `#include` cycles that are very hard to hunt down. As a consequence, we have improved the AspectC++ weaver to detect and report `#include` cycles caused by aspects already at weaving time. While this has certainly improved on the situation, it is still up to the developer to resolve the conflict (e.g., by means of forward declarations or by splitting larger aspects into smaller pieces).

"Hackers hate IDEs." Even though all students at some point ran into difficulties with respect to join-point tracking, it turned out to be more than difficult to convince them to use the AspectC++ plug-in for ECLIPSE (ACDT), which provides features (such as join-point visualization) for exactly this kind of problem. Even the majority of students working on CiAO – who *had* to use ECLIPSE anyway to configure the operating system – did not use it for *anything* else. They considered it to be "too clumsy" compared to the shell and their favorite VIM editor, and preferred hunting for join-point mismatches by analyzing the woven source code or by GREP'ing

through the (XML-based) join-point repository that AC++ generates for the ECLIPSE plug-in. We have learned from this that (even in the case of relatively young students) tool support has to fit the – domain-specific – habits of the developers to get accepted. As a consequence, we are now working on a more generic interface to the join-point repository and a set of command-line tools to query and analyze it in a "no-frills" fashion.

5. FURTHER RELATED WORK

There are several other research projects that investigate the applicability of aspects in the context of operating systems. Among the first was the α-kernel project [8], in which the evolution of four scattered OS concern implementations (namely: prefetching, disk quotas, blocking, and page daemon activation) between versions 2 and 4 of the FreeBSD kernel was analyzed retroactively. The results show that an aspect-oriented implementation would have led to significantly *better evolvability* of these concerns.

C4 [12, 24] is an example for a special-purpose AOP-inspired language. It is intended for the application of kernel patches in Linux. Other related work concentrates on dynamic aspect weaving as a means for run-time adaptation of operating system kernels: TOSKANA provides an infrastructure for the dynamic extension of the FreeBSD kernel by aspects [10]; KLASY is used for aspect-based dynamic instrumentation in Linux [29].

All of these studies demonstrate that there are good cases for aspects in system software. However, the work of ÅBERG in Linux [1] and our own work on eCos [21] show that a useful application of AOP to existing operating systems requires additional AOP expressivity that results in run-time overheads (e.g., temporal logic or dynamic instrumentation).

6. SUMMARY AND CONCLUSIONS

The CiAO project contributes a large-scale case study for the application of aspect technology in the domain of system software. From a systems researcher's perspective, the properties (such as code size, performance, and especially configurability) of the resulting systems are convincing [20, 21]. This paper has focused on the development methodology, which evolved over years. Two main insights can be learned: (1) Operating systems for the domain of resource-constrained embedded systems have to be highly configurable. Our analysis of the AUTOSAR OS specification revealed that these effects can already be found in the requirements; they are an *inherent phenomenon* of complex systems. (2) AOP is very well suited for the design and implementation of such systems under the premise that it is applied with the aspect awareness principles in mind. This paper has shown how this *aspect awareness* can be put into practice.

7. REFERENCES

[1] ÅBERG, R. A., LAWALL, J. L., SÜDHOLT, M., MULLER, G., AND MEUR, A.-F. L. On the automatic evolution of an OS kernel using temporal logic and AOP. In *ASE '03* (Mar. 2003), IEEE, pp. 196–204.

[2] ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. Adding trace matching with free variables to AspectJ. In *OOPSLA '05* (Oct. 2005), ACM, pp. 345–364.

[3] AUTOSAR. Requirements on operating system (version 2.0.1). Tech. rep., Automotive Open System Architecture GbR, June 2006.

[4] AUTOSAR. Specification of operating system (version 2.0.1). Tech. rep., Automotive Open System Architecture GbR, June 2006.

[5] BEUCHE, D., FRÖHLICH, A. A., MEYER, R., PAPAJEWSKI, H., SCHÖN, F., SCHRÖDER-PREIKSCHAT, W., SPINCZYK, O., AND SPINCZYK, U. On architecture transparency in operating systems. In *9th SIGOPS European Workshop* (Sept. 2000), ACM, pp. 147–152.

[6] BEUCHE, D., GUERROUAT, A., PAPAJEWSKI, H., SCHRÖDER-PREIKSCHAT, W., SPINCZYK, O., AND SPINCZYK, U. The PURE family of object-oriented operating systems for deeply embedded systems. In *ISORC '99* (May 1999), pp. 45–53.

[7] CAI, Y., AND SULLIVAN, K. J. Modularity analysis of logical design models. In *ASE '06* (2006), IEEE, pp. 91–102.

[8] COADY, Y., AND KICZALES, G. Back to the future: A retroactive study of aspect evolution in operating system code. In *AOSD '03* (Mar. 2003), ACM, pp. 50–59.

[9] eCos homepage. `http://ecos.sourceware.org/`.

[10] ENGEL, M., AND FREISLEBEN, B. TOSKANA: a toolkit for operating system kernel aspects. In *TAOSD II* (2006), no. 4242 in LNCS, Springer, pp. 182–226.

[11] FILMAN, R. E., AND FRIEDMAN, D. P. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA '00 Workshop on Advanced SoC* (Oct. 2000).

[12] FIUCZYNSKI, M., GRIMM, R., COADY, Y., AND WALKER, D. patch(1) considered harmful. In *HotOS '05* (2005), USENIX.

[13] HABERMANN, A. N., FLON, L., AND COOPRIDER, L. W. Modularization and hierarchy in a family of operating systems. *CACM 19*, 5 (1976), 266–272.

[14] KÄSTNER, C., APEL, S., AND BATORY, D. A case study implementing features using AspectJ. In *SPLC '07* (2007), IEEE, pp. 223–232.

[15] KÄSTNER, C., APEL, S., UR RAHMAN, S. S., ROSENMÜLLER, M., BATORY, D., , AND SAAKE, G. On the impact of the optional feature problem: Analysis and case studies. In *SPLC '09* (2009), IEEE.

[16] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C. V., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *ECOOP '97* (June 1997), vol. 1241 of *LNCS*, Springer, pp. 220–242.

[17] KLEIMAN, S., AND EYKHOLT, J. Interrupts as threads. *ACM OSR 29*, 2 (Apr. 1995), 21–26.

[18] KNIESEL, G., AND RHO, T. A definition, overview and taxonomy of generic aspect languages. *L'Objet, Special Issue on Aspect-Oriented Software Development 11*, 2–3 (Sept. 2006), 9–39.

[19] LOHMANN, D., BLASCHKE, G., AND SPINCZYK, O. Generic advice: On the combination of AOP with generative programming in AspectC++. In *GPCE '04* (Oct. 2004), vol. 3286 of *LNCS*, Springer, pp. 55–74.

[20] LOHMANN, D., HOFER, W., SCHRÖDER-PREIKSCHAT, W., STREICHER, J., AND SPINCZYK, O. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *USENIX '09* (June 2009), USENIX, pp. 215–228.

[21] LOHMANN, D., SCHELER, F., TARTLER, R., SPINCZYK, O., AND SCHRÖDER-PREIKSCHAT, W. A quantitative analysis of aspects in the eCos kernel. In *EuroSys '06* (Apr. 2006), ACM, pp. 191–204.

[22] MASSA, A. *Embedded Software Development with eCos*. New Riders, 2002.

[23] PARNAS, D. L. Some hypothesis about the "uses" hierarchy for operating systems. Tech. rep., TH Darmstadt, Fachbereich Informatik, 1976.

[24] REYNOLDS, A., FIUCZYNSKI, M. E., AND GRIMM, R. On the feasibility of an AOSD approach to Linux kernel extensions. In *AOSD-ACP4IS '08* (Mar. 2008), ACM, pp. 1–7.

[25] SPINCZYK, O., AND LOHMANN, D. Using AOP to develop architecture-neutral operating system components. In *11th SIGOPS European Workshop* (Sept. 2004), ACM, pp. 188–192.

[26] SPINCZYK, O., AND LOHMANN, D. The design and implementation of AspectC++. *Knowledge-Based Systems 20*, 7 (2007), 636–651.

[27] STEIMANN, F. Domain models are aspect free. In *MoDELS '05* (2005), vol. 3713 of *LNCS*, Springer, pp. 171–185.

[28] URBAN, M., LOHMANN, D., AND SPINCZYK, O. PUMA: An aspect-oriented code analysis and manipulation framework for C and C++. In *TAOSD VIII* (2011), no. 6580 in LNCS, Springer. To appear.

[29] YANAGISAWA, Y., KOURAI, K., CHIBA, S., AND ISHIKAWA, R. A dynamic aspect-oriented system for OS kernels. In *GPCE '06* (Oct. 2006), ACM, pp. 69–78.