CiAO: An Aspect-Oriented Operating-System Family for Resource-Constrained Embedded Systems*

Daniel Lohmann, Wanja Hofer, Wolfgang Schröder-Preikschat
{lohmann, hofer, wosch}@cs.fau.de
FAU Erlangen–Nuremberg

Jochen Streicher, Olaf Spinczyk
{jochen.streicher, olaf.spinczyk}@tu-dortmund.de
TU Dortmund

Abstract

This paper evaluates aspect-oriented programming (AOP) as a first-class concept for implementing configurability in system software for resource-constrained embedded systems. To compete against proprietary special-purpose solutions, system software for this domain has to be highly configurable. Such fine-grained configurability is usually implemented "in-line" by means of the C preprocessor. However, this approach does not scale – it quickly leads to "#ifdef hell" and a bad separation of concerns. At the same time, the challenges of configurability are still increasing. AUTOSAR OS, the state-of-the-art operating-system standard from the domain of automotive embedded systems, requires configurability of even fundamental architectural system policies.

On the example of our CiAO operating-system family and the AUTOSAR-OS standard, we demonstrate that AOP – if applied from the very beginning – is a profound answer to these challenges. Our results show that a well-directed, pragmatic application of AOP leads to a much better separation of concerns than does #ifdef-based configuration – without compromising on resource consumption. The suggested approach of aspect-aware operating-system development facilitates providing even fundamental system policies as configurable features.

1 Introduction

The design and implementation of operating systems has always been challenging. Besides the sheer size and the inherent asynchronous and concurrent nature of operating-system code, developers have to deal with lots of crucial nonfunctional requirements such as performance, reliability, and maintainability. Therefore, researchers have always tried to exploit the latest advances in programming languages and software engineering (such as object orientation [6], meta-object protocols [26], or virtual execution environments [14]) in order to reduce the complexity of operating system development and to improve the systems' nonfunctional properties.

1.1 Operating Systems for Small Embedded Systems

This paper focuses on small ("deeply") embedded systems. More than 98 percent of the worldwide annual production of microprocessors ends up in small embedded systems [24] – typically employed in products such as cars, appliances, or toys. Such embedded systems are subject to an enormous hardware-cost pressure. System software for this domain has to cope not only with strict resource constraints, but especially with a broad *variety* of application requirements and platforms. So to allow for reuse, an operating system for the embedded-systems domain has to be developed as a system-software product line that is highly configurable and tailorable. Furthermore, resource-saving static configuration mechanisms are strongly favored over dynamic (re-)configuration.

A good example for this class of highly configurable systems with small footprint is the new embedded operating-system standard specified by AUTOSAR, a consortium founded by all major players in the automotive industry [3]. The goal of AUTOSAR is to continue the success story of the OSEK-OS specification [19]. OSEK-compliant operating systems have been used in almost all European cars over the past ten years, which led to an enormous productivity gain in automotive software development. AUTOSAR extends the OSEK-OS specification in order to cover the whole system-software stack including communication services and a middleware layer.

Even in this restricted domain, there is already a huge variety of application requirements on operating systems. For instance, power-train applications are typically safety-critical and have to deal with real-time requirements,

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

```
Cyg_Mutex::Cyg_Mutex() {
  CYG_REPORT_FUNCTION();
  locked      = false;
  owner       = NULL;
#if defined(CYGSEM_PRI_INVERSION_PROTO_DEFAULT) && \
    defined(CYGSEM_PRI_INVERSION_PROTO_DYNAMIC)
#ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_INHERIT
  protocol    = INHERIT;
#endif
#ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_CEILING
  protocol    = CEILING;
  ceiling     = CYGSEM_PRI_INVERSION_PROTO_DEFAULT_PRI;
#endif
#ifdef CYGSEM_PRI_INVERSION_PROTO_DEFAULT_NONE
  protocol    = NONE;
#endif
#else // not (DYNAMIC and DEFAULT defined)
#ifdef CYGSEM_PRI_INVERSION_PROTO_CEILING
#ifdef CYGSEM_DEFAULT_PRIORITY
  ceiling = CYGSEM_DEFAULT_PRIORITY;
#else
  ceiling = 0;   // Otherwise set it to zero.
#endif
#endif
#endif // DYNAMIC and DEFAULT defined
  CYG_REPORT_RETURN();
}
```

Figure 1: "#ifdef hell" example from eCos [18]

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. In order to reduce the number of electronic control units (up to 100 in modern cars [5]), some manufacturers have the requirement to run multiple applications on the same unit, which is only possible with guaranteed isolation; others do not have this requirement. To fulfill all these varying requirements, the AUTOSAR-OS specification [2] describes a family of systems defined by so-called *scalability classes*. It not only requires configurability of simple functional features, but also of all *policies* regarding temporal and spatial isolation. To achieve this within a single kernel implementation is challenging. The decision about fundamental operating-system policies (like the question if and how address-space protection boundaries should be enforced) is traditionally made in the early phases of operating-system development and is deeply reflected in its architecture, which in turn has an impact on many other parts of the kernel implementation. In AUTOSAR-OS systems, these decisions have to be postponed until the application developer configures the operating system.

1.2 The Price of Configurability

In a previous paper [17], we analyzed the implementation of static configurability in the popular eCos operating system [18], which also targets small embedded systems. The system implements configurability in the familiar way with #ifdef-based conditional compilation (in C++). Even though eCos does not support configurability of architectural concerns as required by AUTOSAR (such as the memory or timing protection model), we have found an "#ifdef hell", which illustrates that these techniques do not scale well enough. Maintainability and evolvability of the implementation suffer significantly. As an example, Figure 1 shows the "#ifdef hell" in the constructor of the eCos mutex class, caused by just four variants of the optional protcol for the prevention of priority inversion. However, the configurability of this protocol does not only affect the constructor code – a total of 34 #ifdef-blocks is spread over 17 functions and data structures in four implementation files.

As a solution, we proposed aspect-oriented programming (AOP) [15] and analyzed the code size and performance impact of applying AOP to factor out the scattered implementation of configurable eCos features into distinct modules called aspects.

1.3 Aspect-Oriented Programming

AOP describes a programming paradigm especially designed to tackle the implementation of *crosscutting concerns* – concerns that, even though conceptually distinct, overlap with the implementation of other concerns in the code by sharing the same functions or classes, such as the mutex configuration options in eCos.

In AOP, *aspects* encapsulate pieces of code called *advice* that implement a crosscutting concern as a distinct module. A piece of advice targets a number of *join points* (points in the static program structure or in the dynamic execution flow) described by a predicate called *pointcut expression*. Pointcut expressions are evaluated by the *aspect weaver*, which weaves the code from the advice bodies to the join points that are matched by the respective predicates.

As pointcuts are described declaratively, the target code itself does not have to be prepared or instrumented to be affected by aspects. Furthermore, the same aspect can affect various and even unforeseen parts of the target code. In the AOP literature [10], this is frequently referred to as the *obliviousness* and *quantification* properties of AOP.

The AOP language and weaver used in the eCos study and in the development of CiAO is AspectC++ [22], a source-to-source weaver that transforms AspectC++ sources to ISO C++ code, which can then be compiled by any standard-compliant C++ compiler.

Figure 2 illustrates the syntax of aspects written in AspectC++. The (excerpted) aspect `Priority_Ceiling` implements the priority ceiling variant of the eCos mutex class. For this purpose, it *introduces* a *slice* of additional elements (the member variable `ceiling`) into the

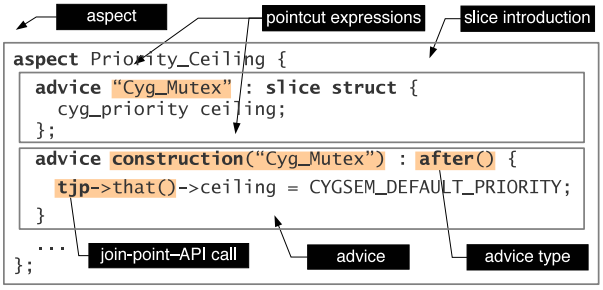

Figure 2: Syntactical elements of an aspect

class `Cyg_Mutex` and gives a piece of *advice* to initialize `ceiling` *after* each *construction* of a `Cyg_Mutex` instance. The targets of the introduction and the piece of construction advice are given by *pointcut expressions*.

In AspectC++, pointcut expressions are built from *match expressions* and *pointcut functions*. The match expression `"Cyg_Mutex"`, for instance, returns a pointcut containing just the class `Cyg_Mutex`. Match expressions can also be fed into pointcut functions to yield pointcuts that represent events in the control flow of the running program, such as the event where some function is about to be *called* (`call()` advice) or an object instance is about to be *constructed* (see `construction("Cyg_Mutex")` in Figure 2). In most cases, the join points for a given pointcut can be derived statically by the aspect weaver so that the respective advice is also inserted statically at compile time without any run-time overhead.

The construction pointcut in the example is used to specify some *after* advice – that is, additional behavior to be triggered after the event occurrence. Other types of advice include *before* advice (speaks for itself) and *around* advice (replaces the original behavior associated with the event occurrence).

Inside the advice body, the type and pointer `JoinPoint *tjp` provide an interface to the event context. The aspect developer can use this *join-point API* to retrieve (and partly modify) contextual information associated with the event, such as the arguments or return value of the intercepted function call (`tjp->arg(`*i*`)`, `tjp->result()`). The `tjp->that()` in Figure 2 returns the `this` pointer of the affected object instance, which is used here to initialize the `ceiling` member variable (which in this case was introduced by the aspect itself).

1.4 Contribution and Outline

The results of applying AOP to eCos were very promising [17]. The refactored eCos system was much better structured than the original; the number of configuration points per feature could be drastically reduced. At the same time, we found that there is no negative impact on the system's performance or code size.

However, we also found that not all configurable features could be refactored into a modular aspect-oriented implementation. The main reason was that eCos did not expose enough unambiguous join points. We took this as a motivation to work on "aspect-aware operating system design". This led to the development of fundamental design principles and the implementation of the CiAO[1] OS family for evaluation purposes. The idea was to build an operating system in an aspect-oriented way from scratch, considering AOP and its mechanisms *from the very beginning* of the development process. The resulting CiAO system is *aspect-aware* in the sense that it was analyzed, designed, and implemented with AOP principles in mind. In order to avoid evaluation results biased by the eCos implementation, CiAO was newly designed after the AUTOSAR-OS standard introduced above [2].

Our main goal is to evaluate the suitability of aspect-oriented software development as a first-class concept for the design and implementation of highly configurable embedded system software product-lines. The research contributions of this work are the following:

- Deeper insights on reasons for the #ifdef hell and the value of AOP in this context (Section 2).
- Design principles for aspect-aware operating system development (Section 4).
- CiAO: The first complete implementation of an operating system kernel developed with AOP concepts[2] (Section 5).
- A discussion of our results from CiAO (Section 6) and general experiences with the approach (Section 7).

For each of the topics, there is a dedicated section in the remaining part of this paper. In addition to that, Section 3 discusses relevant related work. The paper ends with our conclusions in Section 8.

2 Problem Analysis

Why exactly do state-of-the-art configurable systems like eCos exhibit badly modularized code termed as "#ifdef hell"? Is this an inherent property of highly configurable operating systems or just a matter of implementation means? In order to examine these questions, we took a detailed look at an abstract system specification, namely the AUTOSAR-OS standard introduced in Section 1.

[1] CiAO is Aspect-Oriented

[2] The CiAO-OS family is freely available for research purposes from the authors.

	System abstractions (functional)							Callbacks		Protection facilities (architectural)						Internal	
	OS control	Tasks	ISRs category 1	ISRs category 2	Resources	Events	Alarms	Hooks	...	Timing protection	Invalid parameters	Wrong context	Interrupts disabled	Foreign OS objects	...	Preemption	...
... <3 OS services>	⊕							◑	...		◐	◐	◐	◐	...		...
`ActivateTask()`		⊕						◑	...		◐	◐	◐	◐	...	◑	...
`TerminateTask()`		⊕						◑	...			◐	◐		...	◑	...
`Schedule()`		⊕						◑	...			◐	◐		...	◑	...
... <3 more task services>		⊕						◑	...		◐	◐	◐	◐	...	◑	...
`ResumeAllInterrupts()`			⊕						...	◐		◐			...		...
`SuspendAllInterrupts()`			⊕						...	◑		◐			...		...
... <7 more ISR services>			⊕	⊕				◑	...	●	◐	◐	◐	◐	...		...
`GetResource()`					⊕			◑	...	◑	◐	◐	◐	◐	...		...
❶ `ReleaseResource()`					⊕			◑	...	◐	◐	◐	◐	◐	...	◑	...
... <4 event services>						⊕		◑	...		◐	◐	◐	◐	...	◑	...
... <6 alarm services>							⊕	◑	...		◐	◐	◐	◐	...	◑	...
... <7 schedule table services>							⊕	◑	...		◐	◐	◐	◐	...		...
... <7 OS application services>								◑	...		◐	◐	◐	◐	...		...
❷ `TaskType`		⊕			⊛	⊛			...	⊛				⊛	...	⊛	...
`ResourceType`					⊕				...					⊛	...		...
... <4 more structures>	⊕	⊛		⊕		⊛	⊕		...	⊛				⊛	...		...
System startup		◑					◑	◑	...						...		...
Task switch								●	...	●					...		...
Protection violation								◑	...						...		...
... <4 more internal points>		◑				◑		◐	...	●				◑	...	◑	...

❸

Table 1: Influence of configurable concerns (columns) on system services, system types, and internal events (rows) in AUTOSAR OS [2, 19]; kind of influence: ⊕ = extension of the API by a service or type, ⊛ = extension of an existing type, ◑ = modification after service or event, ◐ = modification before, ● = modification before *and* after

2.1 Why #ifdef Hell Appears to Be Unavoidable

The AUTOSAR-OS standard proposes a set of *scalability classes* for the purpose of system tailoring. These classes are, however, relatively coarse-grained (there are only four of them) and do not clearly separate between conceptually distinct concerns. CiAO provides a much better granularity; each AUTOSAR-OS concern is represented as an individual feature in CiAO, subject to application-dependent configuration.

In order to be able to grasp all concerns and their interactions, we have developed a specialized analysis method termed *concern impact analysis* (CIA) [13]. The idea behind CIA is to consider requirement documents together with domain-expert knowledge to develop a matrix of concerns and their influences in an iterative way. In the analysis of the AUTOSAR-OS standard, CIA yielded a comprehensive matrix, which is excerpted in Table 1.

The rows show the AUTOSAR OS system services (API functions) and system abstractions (types) in groups that represent distinct features. AUTOSAR OS is a statically configured operating system with static task priorities; hence, at run time, only services that alter the status of a *task* (e.g., setting it ready or suspended) are available. *Interrupt service routines* (ISRs), in contrast, are triggered asynchronously; the corresponding system functionality allows the application to prohibit their occurrence collectively or on a per-source basis. AUTOSAR OS distinguishes between two categories of ISRs that are somewhat comparable to top halves and bottom halves in Linux: Category-1 ISRs are scheduled by the hardware only and must not interact with the kernel. Category-2 ISRs, in contrast, run under the control of the kernel and may invoke other AUTOSAR-OS services. The third type of control flows supported by the AUTOSAR-OS kernel are *hooks*. Hooks define a callback interface for applications to be notified about kernel-internal events, such as task switch events or error conditions (see Column 8 in Table 1).

Resources are the means for AUTOSAR applications to ensure mutual exclusion to synchronize concurrent access to data structures or hardware periphery. They are comparable to mutex objects in other operating systems. In order to avoid priority inversion and deadlocks, AUTOSAR prescribes a stack-based priority ceiling protocol, which adapts task priorities at run time. Hence, a task never blocks on `GetResource()`. The only way for application tasks to become blocked is by waiting for an AUTOSAR-OS *event*; another task or ISR that sets that event can unblock that task.

Alarms allow applications to take action after a specified period of time; a *schedule table* is an abstraction that encapsulates a series of alarms. Finally, tasks, ISRs, and data can be partitioned into *OS applications*, which define a spatial and temporal protection boundary to be enforced

by the operating system.

The table lists selected identified concerns of AUTOSAR OS (column headings) and how we can expect them to interact with the named entities of the specification (row headings); that is, the 44 system services (e.g., `ActivateTask()`) and the relevant system abstractions (e.g., `TaskType`) as specified in [19, 2]. Furthermore, the lower third lists how we can expect concerns to impact *system-internal* transitions, which are not visible in the system API that is specified by the standard. Table 1 thereby provides an overview of how we can expect AUTOSAR-OS concerns to crosscut with each other in the structural space (abstractions, services) and behavioral space (control flow events) of the implementation.

The comprehensive table shows that a system that is built according to that specification will *inherently* exhibit extensive crosscutting between its concern implementations, leading to code tangling (many different concerns implemented in a single implementation module) and scattering (distribution of a single concern implementation across multiple implementation artifacts). This is because services like `ReleaseResource()` (see Table 1, row ❶) and types like `TaskType` (see Table 1, row ❷), for instance, are affected by as many as nine different concerns! That means that these implementations will exhibit at least nine #ifdef blocks – in the ideal case that each concern can be encapsulated in a single block, completely independent of the other concerns (which is unrealistic, of course). In fact, there is not a single AUTOSAR-OS service that is influenced by only one concern, which means that a straight-forward implementation using the C preprocessor will have numerous #ifdefs in every implementation entity. Thus, "#ifdef hell" seems unavoidable for the class of special-purpose, tailorable operating systems.

2.2 Why AOP Is a Promising Solution

There are several properties inherent in AOP that are promising with respect to overcoming the drawbacks in #ifdef-based configuration techniques that were detailed above.

First, AOP introduces a new kind of binding between modules. In traditional programming paradigms, the caller module P (event producer) knows and *has to know* the callee module C (event consumer); that is, its name and interface (see Figure 3.a):

```
void C::callee() {
  <additional feature>
}
void P::caller() {
  ...
  C::callee(); // has to know C to bind feature
}
```

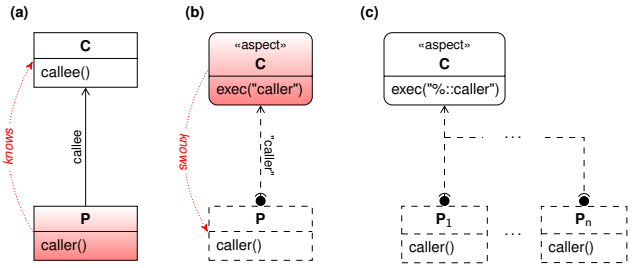

Figure 3: The mechanisms offered by AOP: advice-based binding and implicit per–join-point instantiation of advice

The advice-based binding mechanism offered by AOP can effectively invert that relationship: The callee (i.e., the aspect module C) can *integrate itself* into the caller (i.e., the base code P) without the caller having to know about the callee (see Figure 3.b):

```
advice execution("void P::caller()") : after() {
  <additional feature>
}
void P::caller() {
  ...
  // feature binds "itself"
}
```

If module C is optional and configurable, this loose coupling is an ideal mechanism for integration, because the call is implicit in the callee module. Using the traditional mechanisms, the call has to be included in the base module P and therefore has to be explicitly omitted if the feature implemented by module C is not in the current configuration. This configurable omission is realized by #ifdefs in state-of-the-art systems, bearing the significant disadvantages described above. A similar advantage of advice-based binding applies to configurable static program entities like classes or structures; aspects can integrate the state and operations needed to implement the corresponding feature into those entities *themselves* through slice introductions.

Second, by offering the mechanism of quantification through pointcut-expression matching, AOP allows for a modularized implementation of crosscutting concerns, which is also one of its main proclaimed purposes. This mechanism provides a flexible and implicit instantiation of additional implementation elements at compile-time (see Figure 3.c), ideally suited for the integration of concern implementations into configurable base code where the number of junction points (i.e., AOP join points) is flexible, ranging from zero to n:

```
advice execution("void %::caller()") : after() {
  <additional feature> // binds to any "caller()"
}
```

As we have seen in Table 1, most concerns in an AUTOSAR-OS implementation have a crosscutting im-

pact on many different points in the system in a similar way. An example is the policy that system services must not be called while interrupts are disabled (see Table 1, column ❸). In the requirements specification of AUTOSAR OS, this policy is defined by 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`. [2, p. 40]

This requirement can be translated almost "literally" to a single, modularized AspectC++ aspect:

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

For convenience and the sake of separation of concerns, the aspect uses predefined *named pointcuts*, which are defined separately from the aspects in a global header file and specify which AUTOSAR-OS service belongs to which group:

```
pointcut pcOSServices() = "% ActivateTask()" || ...
pointcut pcISRServices() = ...
...
```

Using these named pointcuts, the aspect gives advice to all points in the system where any OS service but not the interrupt services are called:

```
call(pcOSServices() && !pcISRServices()) ...
```

The resulting set of join points is further filtered to exclude all events from within a hook routine:

```
... && !within(pcHooks())
```

Thus, we eventually get all calls outside of hook routines that are made to any service that is not an ISR service.
The piece of around advice given to these join points performs a test whether the interrupts are currently disabled: If positive, the return code is set to the prescribed error code and the call is aborted; if negative, the call is performed as normal. (Around advice *replaces* the original processing of the intercepted event; however, it is possible to invoke the original processing explicitly with `tjp->proceed()`.)
The complete concern is encapsulated in this single aspect. The result is an enhanced separation of concerns in the system implementation. Layered, configurable systems can especially benefit from AOP mechanisms by being able to flexibly omit parts of the system without breaking caller–callee relationships.

3 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 [7], 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 is analyzed retroactively. The results show that an aspect-oriented implementation would have led to significantly *better evolvability* of these concerns. Around the same time, our own group experimented with AspectC++ in the PURE OS product line and later with aspect-refactoring eCos [17]. Our results from analyzing the AspectC++ implementation of various previously hard-wired crosscutting concerns show that this new paradigm leads to *no overhead* in terms of resource consumption per se.

Not a general-purpose AOP language but an AOP-inspired language of temporal logic is used in the Bossa project to integrate the Bossa scheduler framework into the Linux kernel [1]. Another example for a special-purpose AOP-inspired language is C4 [12, 21], which is intended for the application of kernel patches in Linux. The same goal of smarter patches (with a focus on "collateral evolutions" – changes to the kernel API that have to be caught up in dozens or hundreds of device drivers) is followed by Coccinelle [20]. Although the input language for the Coccinelle engine "SmPL" is not called an AOP language, it supports the modular implementation of crosscutting kernel modifications (i.e., quantification). 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 [9]; KLASY is used for aspect-based dynamic instrumentation in Linux [25].

All these studies demonstrate that there are good cases for aspects in system software. However, both Bossa and our own work on eCos 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). So far no study exists that analyzes the effects of using AOP for the development of an operating-system kernel from the very beginning. This paper explores just that.

4 Aspect-Aware Operating-System Development

The basic idea behind aspect-aware operating-system development is the strict separation of concerns in the *implementation*. Each implementation unit provides exactly one feature; its mere presence or absence in the config-

ured source tree decides on the inclusion of the particular feature into the resulting system variant.

Technically, this comes down to a strict decoupling of policies and mechanisms by using aspects as the primary composition technique: Kernel mechanisms are glued together and extended by aspects; they support aspects by ensuring that all relevant internal control-flow transitions are available as unambigious and statically evaluable join points.

However, this availability cannot be taken for granted. Improving the configurability of eCos even further did not work as good as expected because of join-point ambiguity [17]. For instance, eCos does not expose a dedicated user API to invoke system services. This means that, on the join-point level, *user*$\rightleftharpoons$*kernel* transitions are not statically distinguishable from the kernel-internal activation and termination of system services. The consequence is that policy aspects that need to hook into these events become more expensive than necessary – for instance, an aspect that implements a new *kernel-stack* policy by switching stacks when entering/leaving the kernel. The ideal implementation of the kernel-stack feature had a performance overhead of 5% for the actual stack switches, whereas the aspect implementation induced a total overhead of 124% only because of unambiguous join points. The aspect had to use dynamic pointcut functions to disambiguate at run time: It used `cflow()`, a dynamic pointcut function that induces an extra internal control-flow counter that has to be incremented, decremented, and tested at run time to yield the join points.However, in other cases it was not possible at all to disambiguate, rendering an aspect-based implementation of new configuration options impossible.

We learned from this that the exposure of all relevant gluing and extension points as statically evaluable and unambigious join points has to be understood as a primary design goal from the very beginning. The key premise for such *aspect awareness* is a component structure that makes it possible to influence the composition and shape of components as well as all run-time control flows that run through them by aspects [16].

4.1 Design Principles

The eCos experience led us to the three fundamental principles of aspect-aware operating-system 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 points.

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.

Aspect awareness, as described by these principles, means that we moderate the AOP ideal of obliviousness, which is generally considered by the AOP community as a defining characteristic of AOP [11]. CiAO's system components and abstractions are *not* totally oblivious to aspects – they are supposed to provide explicit support for aspects and even depend on them for their integration.

4.2 Role and Types of Classes and Aspects

The relationship between aspects and classes is asymmetrical in most AOP languages: Aspects augment classes, but not vice versa. This gives rise to the question which features are best to be implemented as classes and which as aspects and how both should be applied to meet the above design principles.

The general rule we came up with in the development of CiAO is to provide some feature as a class if – and only if – it represents a *distinguishable instantiable concept* of the operating system. Provided as classes are:

1. **System components**, which are instantiated on behalf of the kernel and manage its run-time state (such as the `Scheduler` or the various hardware devices).

2. **System abstractions**, which are instantiated on behalf of the application and represent a system object (such as `Task`, `Resource`, or `Event`).

However, the classes for system components and system abstractions are sparse and to be further "filled" by *extension slices*. The main purpose of these classes is to provide a distinct scope with unambiguous join points for the aspects (that is, *visible transitions*).

All other features are implemented as 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.

The effect of *extension aspects* typically becomes visible in the API of the affected system component or abstraction. *Policy aspects*, in contrast, lead to a different system behavior. We will see examples for extension and policy aspects in the following section. *Upcall aspects* do not contribute directly to a design principle, but have a more technical purpose: they exploit advice-based binding and the fact that AspectC++ inlines advice code at the respective join point for flexible, yet very efficient upcalls.

5 Case Study: CiAO-AS

CiAO is designed and implemented as a family of operating systems and has been developed from scratch using the principles of aspect-aware operating-system development. Note, however, that the application developer does not have to have any AOP expertise to use the OS. A concrete CiAO variant is configured statically by selecting features from a feature model in an Eclipse-based graphical configuration tool [4].

The *CiAO-AS* family member implements an operating-system kernel according to the AUTOSAR-OS standard[3] [2], including configurable protection policies (memory protection, timing protection, service protection). The primary target platform for CiAO is the Infineon TriCore, an architecture of 32-bit microcontrollers that also serves as a reference platform for AUTOSAR and is widely used in the automotive industry.

5.1 Overview

Figure 4 shows the basic structure of the CiAO-AS kernel. Like most operating systems, CiAO is designed with a *layered architecture*, in which each layer is implemented using the functionality of the layers below. The only exceptions to this are the aspects implementing architectural policies, which may affect multiple layers.

On the coarse level, we have three layers. From bottom up these are: the *hardware access layer,* the *system layer* (the operating system itself), and the *API layer.*

[3]Because of legal issues, we do not claim full conformance; we have not performed any formal conformance testing.

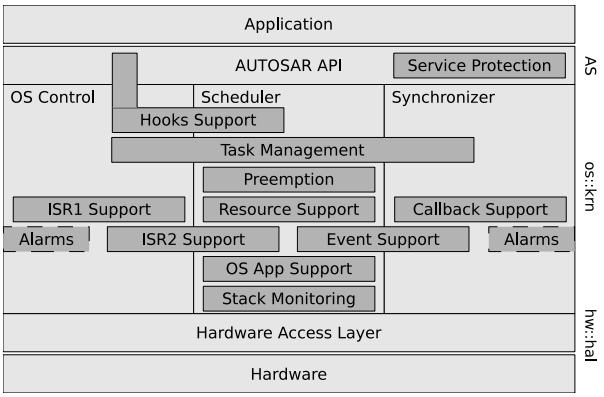

Figure 4: Structure of the CiAO-AS kernel

In CiAO, however, layers do not just serve conceptual purposes, but also are a means of aspect-aware development. With regard to the principle of *visible transitions*, each layer is represented as a separate C++ namespace in the implementation (`hw::hal`, `os::krn`, `AS`). Thereby, cross-layer control-flow transitions (especially into and out of `os::krn`) can be grasped by statically evaluable pointcut expressions. The following expression, for instance, yields all join points where a system-layer component accesses the hardware:

```
pointcut pcOStoHW() = call("% hw::hal::%(...)")
        && within("% os::krn::%(...)");
```

5.2 The Kernel

In its full configuration, the system layer bears three logical *system components* (displayed as columns in Figure 4):

1. The *scheduler* (`Scheduler`) takes care of the dispatching of tasks and the scheduling strategy.

2. The *synchronization facility* (`Synchronizer`) takes care of the management of events, alarms, and the underlying (hardware / software) counters.

3. The *OS control facility* (`OSControl`) provides services for the controlled startup and shutdown of the system and the management of OSEK/AUTOSAR application modes.

However, as pointed out in Section 4.2, these classes are sparse or even empty. If at all, they implement only a minimal base of their respective concern. All further concerns and variants (depicted in dark grey in Figure 4) are brought into the system by aspects, most of which touch multiple system components and system abstractions.

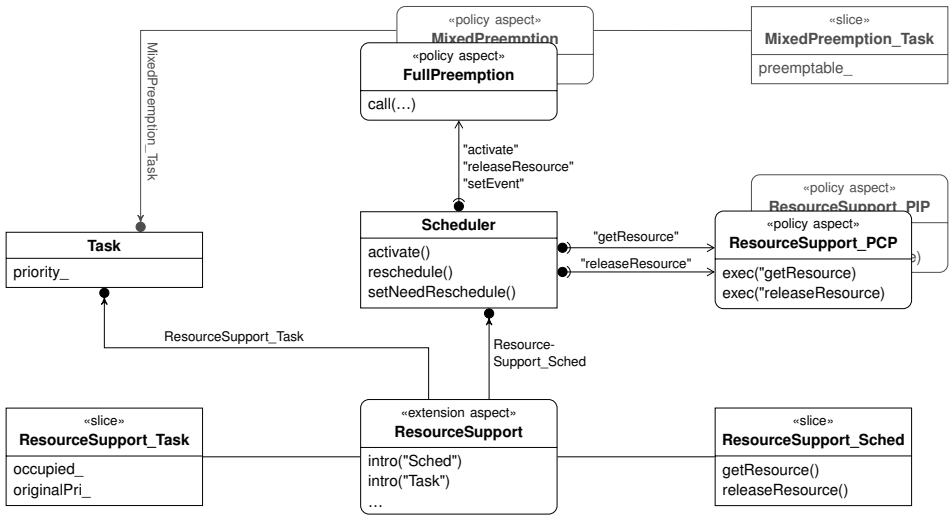

Figure 5: Interactions between optional policies and extensions of the CiAO scheduler

5.3 Aspect-Aware Development Applied

Figure 5 demonstrates how components, abstractions, and aspects engage with each other on a concrete example. The central element is the system component `Scheduler`. However, `Scheduler` provides only the minimal base of the scheduling facility, which is nonpreemptive scheduling:

```
class Sched {
  Tasklist    ready_;
  Task::Id    running_;
 public:
  void activate(Task::Id whom);
  void reschedule();
  void setNeedReschedule();
  ...
};
```

Support for preemption and further abstractions is provided by additional *extension aspects* and *policy aspects*.

`ResourceSupport` is an example for an *extension aspect*. It extends the `Task` system abstraction `Scheduler` system component with support for resources. For this purpose, it introduces some state variables (`occupied_`, `originalPri_`) and operations (`getResource()`, `releaseResource()`).[4] The elements to introduce are given by respective *extension slices*:

```
slice struct ResourceSupport_Task {
  ResourceMask occupied_;
  Pri originalPri_;
};

slice struct ResourceSupport_Sched {
  void getResource(Resource::Id resid) {...}
  void releaseResource(Resource::Id resid) {...}
};

aspect ResourceSupport {
  advice "Task" : slice ResourceSupport_Task;
  advice "Scheduler" : slice ResourceSupport_Sched;
};
```

`FullPreemption` is an example for a *policy aspect*. It implements the full-preemption policy as specified in [19], according to which every point where a higher-priority task may become ready is a potential point of preemption:

```
pointcut pcPreemptionPoints() =
  "% Scheduler::activate(...)" ||
  "% Scheduler::setEvent(...)" ||
  "% Scheduler::releaseResource(...)";

aspect FullPreemption {
  advice execution(pcPreemptionPoints()) : after() {
    tjp->that()->reschedule();
} };
```

The named pointcut `pcPreemptionPoints()` (defined in a global header file) specifies the potential preemption points. To these points, *if present*, the aspect `FullPreemption` binds the invocation of `reschedule()`. This demonstrates the benefits of *loose coupling* by the AOP mechanisms, which makes it easy to cope with conceptually different, but technically interacting features: In a fully-preemptive system without resource support, `Scheduler::releaseResource()` is just not present, thus does not constitute a join point for `FullPreemption`. However, if the `ResourceSupport` extension aspect is part of the current configuration, `Scheduler::release-`

[4] `ResourceSupport` furthermore extends the API on the interface layer (it introduces the respective AUTOSAR-OS services `Get-/ReleaseResource()` and the `ResourceType` abstraction) so that applications can use the new functionality. For the sake of simplicity, this cross-layer extension is omitted here.

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*).

`Resource()` implicitly triggers the advice. The separation of policy invocation from mechanism implementation makes it easy to integrate additional features, such as the `ResourceSupport_PCP` aspect, which implements a stack-based priority ceiling protocol for resources. As AspectC++ inlines advice code at the matching join point, this flexibility does not cause overhead at run time.

6 Discussion of Results

By following the principles of aspect-aware operating system development, policies and mechanisms are cleanly separated in the CiAO implementation. This separation is a golden rule of system-software development, but in practice difficult to achieve. While on the design level it is usually possible to describe a policy in a well-separated manner from underlying mechanisms, the implementation often tends to be crosscutting. The reason is that many system policies, such as the preemption policy, not only depend on decisions but also on the specific points in the control flow where these decisions are made. Here, the modularization into aspects shows some clear advantages.

6.1 Modularization of the System

Table 2 displays an excerpt of the list of AUTOSAR-OS concerns that are implemented as aspects in CiAO-AS. The first three columns list for each concern the number of *extension*, *policy*, and *upcall aspects* that implement the concern. (The resource-support aspect and the protection-hook aspect have both an extension and a policy facet.)

An interesting point is the realization of synergies by means of AOP *quantification.* If for some concern the number of pieces of advice is lower than the number of affected join points, we have actually profited from the AOP concept of quantification by being able to reuse advice code over several join points. For 8 out of the 14 concerns listed in Table 2, this is the case.

The net amount of this profit depends on the type of concern and aspect. *Extension aspects* typically crosscut *heterogeneously* with the implementation of other concerns, which means that they have specific pieces of advice for specific join points. These kinds of advice do not leave much potential for synergies by quantification. *Policy aspects* on the other hand – especially those for architectural policies – tend to crosscut *homogeneously* with the implementation of other concerns, which means that a specific piece of advice targets many different join points at once. In these cases, quantification creates significant synergies.

For all concerns, however, the implementation is realized as a distinct set of aspect modules, thereby reaching complete encapsulation and separation of concerns. Thus, any given feature configuration demanded by the application developer can be fulfilled by only including the implementation entities belonging to that configuration in the configured source tree to be compiled.

6.2 Scalability of the System

Execution Time. The effects of the achieved configurability also become visible in the CPU overhead. Table 3 displays the execution times of the micro-benchmark sce-

narios[5] **(a)** to **(j)** and the comprehensive application **(k)** on CiAO and a commercial OSEK implementation[6]. For each scenario, we first configured both systems to support the smallest possible set of features (*min* colums in Table 3). The differences between CiAO and OSEK are considerable: CiAO is noticeably faster in all test scenarios.

One reason for this is that CiAO provides a much better configurability (and thereby granularity) than OSEK. As the micro-benchmark scenarios utilize only subsets of the OSEK/AUTOSAR features, this has a significant effect on the resulting execution times. The smallest possible configurations of the commercial OSEK still contained a lot of unwanted functionality. The scheduler is synchronized with ISRs, for instance; however, most of the application scenarios do not include any ISRs that could possibly interrupt the kernel.

To judge these effects, we performed additional measurements with an "artifically enriched" version of CiAO that provides the same amount of unwanted functionality as OSEK (column *full* in Table 3). This reduces the performance differences; however, CiAO is still faster in six out of eleven test cases. This is most notable in test case **(k)**, which is a comprehensive application that actually *uses* the full feature set.

Another reason for the relative advantage of CiAO is that OSEK's internal thread-abstraction implementation is less efficient. This is mainly due to particularities of the TriCore platform, which renders standard context-switch implementations ported to that platform very inefficient. CiAO, however, has a highly configurable and adaptable thread abstraction, therefore not only providing for an upward tailorability (i.e., to the needs of the application), but also downward toward the deployment platform.

Memory Requirements. In embedded systems, tailorability is crucial – especially with respect to memory consumption, because RAM and ROM are typically limited to sizes of a few kilobytes. Since system software does not directly contribute to the business value of an embedded system, scalability is of particular importance here. Thus, we also investigated how the memory requirements of the CiAO-AS kernel scale up with the number of selected configurable features; the condensed results

test scenario		CiAO		OSEK
		min	full	min
(a)	voluntary task switch	*160*	*178*	*218*
(b)	forced task switch	*108*	*127*	*280*
(c)	preemptive task switch	*192*	*219*	*274*
(d)	system startup	*194*	*194*	*399*
(e)	resource acquisition	*19*	*56*	*54*
(f)	resource release	*14*	*52*	*41*
(g)	resource release with preemption	*240*	*326*	*294*
(h)	category 2 ISR latency	*47*	*47*	*47*
(i)	event blocking with task switch	*141*	*172*	*224*
(j)	event setting with preemption	*194*	*232*	*201*
(k)	comprehensive application	*748*	*748*	*1216*

Table 3: Performance measurement results [clock ticks]

are depicted in Table 4. Listed are the deltas in code, data, and BSS section size per feature that is added to the CiAO base system.

Each `Task` object, for instance, takes 20 bytes of *data* for the kernel task context (priority, state, function, stack, interrupted flag) and 16 bytes (*bss*) for the underlying CiAO thread abstraction structure. Aspects from the implementation of other features, however, may extend the size of the kernel task context. Resource support, for instance, crosscuts with task management in the implementation of the `Task` structure, which it extends by 8 bytes to accommodate the occupied resources mask and the original priority.

The cost of several features does not simply induce a constant cost, but depends on the number of affected join points, which in turn can depend on the presence of *other* features, as explained in Section 5.3 with the example of full preemption and resource support. This effect underlines again the flexibility of loose coupling by advice-based binding.

7 Experiences with the Approach

The CiAO results show that the approach of aspect-aware operating-system development is both feasible and beneficial for the class of configurable embedded operating systems. The challenge was to implement a system in which almost everything is configurable. In the following, we describe our experience with the approach.

7.1 Extensibility

We are convinced that the three design principles of aspect-aware operating-system development (*loose coupling*, *visible transitions*, *minimal extensions*) also lead to an easy extensibility of the system for new, unanticipated features. While it is generally difficult to prove the soundness of an approach for unanticipated change, we have at least some evidence that our approach has clear benefits

[5] All variants were woven and compiled for the Infineon TriCore platform with AC++-1.0PRE3 and TRICORE-G++-3.4.3 using -O3 -fno-rtti -funit-at-a-time -ffunction-sections -Xlinker --gc-sections optimization flags. Memory numbers are retrieved byte-exact from the linker-map files. Run-time numbers are measured with a high-resolution hardware trace unit (Lauterbach PowerTrace TC1796).

[6] ProOSEK is the leading commercial implementation of the OSEK standard and part of the BMW and Audi/VW standard cores. We compare CiAO against ProOSEK since (1) AUTOSAR is a true superset of OSEK and (2) we do not yet have access to a complete AUTOSAR implementation.

feature	with feature or instance	text	data	bss
Base system (OS control and tasks)				
	per task	*+ func*	*+ 20*	*+ 16 + stack*
	per application mode	*0*	*+ 4*	*0*
ISR cat. 1 support		*0*	*0*	*0*
	per ISR	*+func*	*0*	*0*
	per disable–enable	*+ 4*	*0*	*0*
Resource support		*+ 128*	*0*	*0*
	per resource	*0*	*+ 4*	*0*
	per task	*0*	*+ 8*	*0*
Event support		*+ 280*	*0*	*0*
	per task	*0*	*+ 8*	*0*
	per alarm	*0*	*+ 12*	*0*
Full preemption		*0*	*0*	*0*
	per join point	*+ 12*	*0*	*0*
Mixed preemption		*0*	*0*	*0*
	per join point	*+ 44*	*0*	*0*
	per task	*0*	*+ 4*	*0*
Wrong context check		*0*	*0*	*0*
	per void join point	*0*	*0*	*0*
	per StatusType join point	*+ 8*	*0*	*0*
Interrupts disabled check		*0*	*0*	*0*
	per join point	*+ 64*	*0*	*0*
Invalid parameters check		*0*	*0*	*0*
	per join point	*+ 36*	*0*	*0*
Error hook		*0*	*0*	*+ 4*
	per join point	*+ 54*	*0*	*0*
Startup hook or shutdown hook		*0*	*0*	*0*
Pre-task hook or post-task hook		*0*	*0*	*0*

Table 4: Scalability of CiAO's memory footprint. Listed are the increases in static memory demands [bytes] of selected configurable CiAO features.

here:

In a specific real-time application project that we implemented using CiAO, minimal and deterministic event-processing latencies were crucial. The underlying hardware platform was the Infineon TriCore, which actually is a heterogeneous multi-processor-system-on-chip that comes with an integrated peripheral control processor (PCP). This freely-programmable co-processor is able to handle interrupts independently of the main processor. We decided to extend CiAO in a way that the PCP pre-handles all hardware events (interrupts) in order to map them to activations of respective software tasks, thereby preventing the real-time problem of rate-monotonic priority inversion [8]. This way, the CPU is only interrupted when there actually is a control flow of a higher priority than the currently executing one ready to be dispatched.

This relatively complex and unanticipated extension could nevertheless be integrated into CiAO by a single *extension aspect,* which is shown in Figure 6. The `PCP_Extension` aspect is itself a *minimal extension*; its implementation profited especially from the fact that all other CiAO components are designed according to the principle of *visible transitions.* This ensures here that all relevant transitions of the CPU, such as when the kernel is entered or left (lines 9 and 14, respectively) or when the running CPU task is about to be preempted (line 17), are available as statically evaluable and unambigious join points to which the aspect can bind.

Note, that the aspect in Figure 6 is basically the complete code for that extension, except for some initialization code (10 lines of code) and the PCP code, which is written in assembly language due to the lack of a C/C++ compiler for the PCP instruction set.

7.2 The Role of Language

We think that the expressiveness of the base language (in our case C++) plays an important role for the effectiveness of the approach. Thanks to modularization through namespaces and classes, C++ has some clear advantages over C with respect to *visible transitions*: the more of the base program's purpose and semantics is expressed in its syntactic structure, the more unambigious and "semantically rich" join points are available to which the aspects can bind.

Note, however, that even though CiAO is using C++, it is not developed in an object-oriented manner. We used C++ as a purely static language and stayed away from any language feature that induces a run-time overhead, such as virtual functions, exceptions, run-time type information, and automatic construction of global variables.

7.3 Technical Issues

Aspects for Low-Level Code. A recurring challenge in the development of CiAO was that the implementation of fundamental low-level OS abstractions, such as interrupt handlers or the thread dispatcher, requires more control over the resulting machine code than is guaranteed by the semantics of ISO C++. Such functions are typically (1) written entirely in external assembly files or (2) use a mixture of inline assembly and nonstandard language extensions (such as `__attribute__((interrupt))` in `gcc`). For the sake of *visible transitions*, we generally opted for (2). However, the resulting join points often have to be considered as fragile – if advice is given to, for instance, the context switch function, the transformations performed by the aspect weaver might break the programmer's implicit assumptions about register usage or the stack layout. The workaround we came up with for these cases is to provide *explicit join points* to which the aspects can bind instead. Technically, an explicit join point is represented by an empty inline function that is invoked from the fragile code when the execution context is safe. CiAO's context switch functionality, for instance, exposes four explicit join points to which aspects can bind: `before_CPURelease()`, `before_LastCPURelease()`, `after_CPUReceive()`, and `after_FirstCPUReceive()`. Because of function inlin-

```
aspect PCP_Extension {
  advice execution("void hw::init()") : after() {
    PCP::init();
  }
  advice execution("% Scheduler::setRunning(...)") :
  before() {
    PCP::setPrio(os::krn::Task::getPri(tjp->args<0>()));
  }
  advice execution("% enterKernel(...)") : after() {
    // wait until PCP has left kernel (Peterson)
    PCP_FLAG0 = 1; PCP_TURN = 1;
    while ((PCP_FLAG1 == 1) && (PCP_TURN == 1)) {}
  }
  advice execution("% leaveKernel(...)") : before() {
    PCP_FLAG0 = 0;
  }
  advice execution("% AST0::ast(...)") : around() {
    // AST0::ast() is the AST handler that activates
    // the scheduler (bound by an upcall aspect)

    // wait until PCP has left kernel (Peterson)
    PCP_FLAG0 = 1; PCP_TURN = 1;
    while ((PCP_FLAG1 == 1) && (PCP_TURN == 1)) {}

    // proceed to aspect that activates scheduler
    tjp->proceed();
    PCP_FLAG0 = 0;
  }
  advice execution("% Scheduler::schedule(...)") : after() {
   // write priority of running task to PCP memory
   PCP::setPrio(Task::getPri(
      Scheduler::Inst().getRunning()));
  }
};
```

Figure 6: PCP co-processor extension aspect

ing, this does not induce an overhead and the aspect code is still embedded directly into the context switch functionality.

Aspect–Aspect Interdependencies. In several cases we had to deal with subtle interdependencies between aspects that affect the same join points. For instance, the following aspect implements the *ErrorHook* feature, which exempts the application developer from manually testing the result code of OS services:

```
aspect ErrorHook {
  advice execution(pcOSServices() ... ) : after() {
    if(*tjp->result() != E_OK)
      invokeErrorHook(*tjp->result());
} };
```

Later we figured that, depending on the configuration, there are also other *aspects* that modify the result code. To fulfill its specification, `ErrorHook` has to be invoked after these other aspects. Whereas detecting such interdependencies was sometimes tricky (especially those that emerge only in certain configurations), they were generally easy to resolve by *order* advice:

```
advice execution(pcOSServices() ... ) : order(
  "ErrorHook", !"ErrorHook");
```

This type of advice allows the developer to define a (partial) order of aspect invocation for a pointcut. The precedence of aspects is specified as a sequence of match expressions, which are evaluated against all aspect identifiers. In the above example, the aspect yielded by the expression `"ErrorHook"` has precedence (is invoked *last* of all aspects that give *after* advice to the pointcut) over all other aspects (the result of `!"ErrorHook"`). Very helpful was that order advice does not necessarily have to be given by one of the affected aspects, instead it can be given by any aspect. This made it relatively easy to encapsulate and deal with configuration-dependent ordering constraints.

Join-Point Traceability. An important factor for the development were effective tools for join-point traceability. From the viewpoint of an aspect developer, the set of join points offered by some class implementation constitutes an interface. However, these interfaces are "implicit at best" [23]; a simple refactoring, such as renaming a method, might silently change the set of join points and thereby break some aspect. To prevent such situations, we used the Eclipse-based AspectC++ Development Toolkit (ACDT[7]), which provides a join-point–set delta analysis (very helpful after updating from the repository) and visualizes code that is affected by aspects. Thereby, unwanted side effects of code changes could be detected relatively easy.

8 Summary and Conclusions

Operating systems for the domain of resource-constrained embedded systems have to be highly configurable. Typically, such configurability is implemented "in line" by means of the C preprocessor. However, due to feature interdependencies and the fact that system policies and system mechanisms tend to crosscut with each other in the implementation, this approach leads to "#ifdef hell" and a bad separation of concerns. 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. If fundamental architectural policies have to be provided as configurable features, "#ifdef hell" appears to be unavoidable.

We showed that a pragmatic application of aspect-oriented programming (AOP) provides means for solving these issues: The advice mechanism of AOP effectively reverses the direction of feature integration; an (optional) feature that is implemented as an aspect *integrates itself* into the base code. Thanks to AOP's pointcut expressions, the integration of features through join points is declarative – it scales implicitly with the presense or absence of

[7] `http://acdt.aspectc.org/`

other features. A key prerequisite is, however, that the system's implementation exhibits enough unambigious and statically evaluable join points. This is achieved by the three design principles of *aspect-aware operating-system development*.

By following this design approach in the development of CiAO, we did not only achieve the complete separation of concerns in the code, but also excellent configurability and scalability in the resulting system. We hope that our results encourage developers who start from scratch with a piece of configurable system software to follow the guidelines described in this paper.

Acknowledgments

We wish to thank the anonymous reviewers for EuroSys and USENIX for their helpful comments. Special thanks go to Robert Grimm, whose demanding and encouraging shepherding helped us tremendously to improve content and clarity of this paper.

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 *18th IEEE Int. Conf. on Automated Software Engineering (ASE '03)* (Montreal, Canada, Mar. 2003), IEEE, pp. 196–204.

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

[3] AUTOSAR homepage. `http://www.autosar.org/`, visited 2009-03-26.

[4] BEUCHE, D. Variant management with pure::variants. Tech. rep., pure-systems GmbH, 2006. `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`, visited 2009-03-26.

[5] BROY, M. Challenges in automotive software engineering. In *28th Int. Conf. on Software Engineering (ICSE '06)* (New York, NY, USA, 2006), ACM, pp. 33–42.

[6] CAMPBELL, R., ISLAM, N., MADANY, P., AND RAILA, D. Designing and implementing Choices: An object-oriented system in C++. *CACM 36*, 9 (1993).

[7] COADY, Y., AND KICZALES, G. Back to the future: A retroactive study of aspect evolution in operating system code. In *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)* (Boston, MA, USA, Mar. 2003), M. Akşit, Ed., ACM, pp. 50–59.

[8] DEL FOYO, L. E. L., MEJIA-ALVAREZ, P., AND DE NIZ, D. Predictable interrupt management for real time kernels over conventional PC hardware. In *12th IEEE Int. Symp. on Real-Time and Embedded Technology and Applications (RTAS '06)* (Los Alamitos, CA, USA, 2006), IEEE, pp. 14–23.

[9] ENGEL, M., AND FREISLEBEN, B. TOSKANA: a toolkit for operating system kernel aspects. In *Transactions on AOSD II* (2006), A. Rashid and M. Aksit, Eds., no. 4242 in LNCS, Springer, pp. 182–226.

[10] FILMAN, R. E., ELRAD, T., CLARKE, S., AND AKSIT, M. *Aspect-Oriented Software Development*. AW, 2005.

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

[12] FIUCZYNSKI, M., GRIMM, R., COADY, Y., AND WALKER, D. patch(1) considered harmful. In *10th W'shop on Hot Topics in Operating Systems (HotOS '05)* (2005), USENIX.

[13] HOFER, W., LOHMANN, D., AND SCHRÖDER-PREIKSCHAT, W. Concern impact analysis in configurable system software—the AUTOSAR OS case. In *7th AOSD W'shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)* (New York, NY, USA, Mar. 2008), ACM, pp. 1–6.

[14] HUNT, G. C., AND LARUS, J. R. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 37–49.

[15] KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. Aspect-oriented programming. In *11th Eur. Conf. on OOP (ECOOP '97)* (June 1997), M. Aksit and S. Matsuoka, Eds., vol. 1241 of *LNCS*, Springer, pp. 220–242.

[16] LOHMANN, D. *Aspect Awareness in the Development of Configurable System Software*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, 2009.

[17] LOHMANN, D., SCHELER, F., TARTLER, R., SPINCZYK, O., AND SCHRÖDER-PREIKSCHAT, W. A quantitative analysis of aspects in the eCos kernel. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2006 (EuroSys '06)* (New York, NY, USA, Apr. 2006), ACM, pp. 191–204.

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

[19] OSEK/VDX GROUP. Operating system specification 2.2.3. Tech. rep., OSEK/VDX Group, Feb. 2005. `http://portal.osek-vdx.org/files/pdf/specs/os223.pdf`, visited 2009-03-26.

[20] PADIOLEAU, Y., LAWALL, J. L., MULLER, G., AND HANSEN, R. R. Documenting and automating collateral evolutions in Linux device drivers. In *ACM SIGOPS/EuroSys Eur. Conf. on Computer Systems 2008 (EuroSys '08)* (Glasgow, Scotland, Mar. 2008).

[21] REYNOLDS, A., FIUCZYNSKI, M. E., AND GRIMM, R. On the feasibility of an AOSD approach to Linux kernel extensions. In *7th AOSD W'shop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)* (New York, NY, USA, Mar. 2008), ACM, pp. 1–7.

[22] SPINCZYK, O., AND LOHMANN, D. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software 20*, 7 (2007), 636–651.

[23] STEIMANN, F. The paradoxical success of aspect-oriented programming. In *21st ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '06)* (New York, NY, USA, 2006), ACM, pp. 481–497.

[24] TURLEY, J. The two percent solution. *embedded.com* (Dec. 2002). `http://www.embedded.com/story/OEG20021217S0039`, visited 2009-03-26.

[25] YANAGISAWA, Y., KOURAI, K., CHIBA, S., AND ISHIKAWA, R. A dynamic aspect-oriented system for OS kernels. In *6th Int. Conf. on Generative Programming and Component Engineering (GPCE '06)* (New York, NY, USA, Oct. 2006), ACM, pp. 69–78.

[26] YOKOTE, Y. The Apertos reflective operating system: the concept and its implementation. In *7th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '92)* (New York, NY, USA, 1992), ACM, pp. 414–434.