The Aspect-Aware Design and Implementation of the CiAO Operating-System Family

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

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

Abstract. CiAO is the first operating-system family that has been developed with AOP concepts from the very beginning. By its *aspect-aware* design and implementation, CiAO reaches excellent configurability, separation of concerns, and low footprints in the resulting systems that outperform leading commercial implementations. CiAO implements the automotive operating-system standard OSEK/AUTOSAR OS and provides configurability of all fundamental system properties by means of AOP.

We describe the aspect-aware design approach and implementation idioms that led to this efficiency and flexibility. On the example of three larger case studies from CiAO, we demonstrate how AOP can be employed in this respect on different levels of complexity: From highly configurable, yet efficient low-level hardware abstractions over the implementation of central kernel policies up to the decomposition of a complete operating-system specification.

Our results show that by a consequent application of the aspect-aware approach, AOP becomes a promising technology to reach configurability, separation of concerns, *and* runtime/memory efficiency on all levels of operating-system development.

1 Introduction

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 [24]. AOP is appealing for this domain, as fundamental operating-system concerns, such as *synchronization, preemption, prefetching,* or *monitoring*, seem to be inherently crosscutting. Their clear separation into dedicated aspect modules would facilitate better *evolvability* and *configurability* of operating-system policies [8,11,1]. 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 [44] and extensive studies with aspects in the PURE and eCos operating system families [43,31].

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

G.T. Leavens et al. (Eds.): Transactions on AOSD IX, LNCS 7271, pp. 168–215, 2012.

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 low resource consumption in the resulting system, which outperforms leading commercial implementations [30].

1.1 Purpose of the Paper

We have described our path to CiAO and its underlying design approach of *aspect-aware operating-system development* in two conference publications [30,29]. From the anonymous reviews to these papers as well as from personal feedback after our AOSD '11 presentation, we learned that there is a strong interest in further details about the design and implementation of CiAO. The reviews especially encouraged us to include "more examples from CiAO" and "more real aspect code" – which, however, was not possible within the given page limits. The purpose of this paper is to complement [30,29] in this spirit.

1.2 Structure and Contributions of the Paper

In this paper, we provide an extensive overview of CiAO: its design goals and architecture (Sect. 2), the resulting aspect-oriented design principles and development idioms (Sect. 2.3), and three larger case studies that exemplify the application of aspect-aware development (and its benefits with respect to configurability) in CiAO on three different levels of granularity:

1. In the "Continuation" study (Sect. 4), we exemplify the aspect-aware decomposition and implementation of a **configurable low-level system abstraction** – from features down to near-hardware code.
2. The "Interrupt synchronization" study (Sect. 5) demonstrates the design (and partly also the implementation) of a **configurable architectural policy** – from nonfunctional requirements down to aspect-aware design. (An early draft of this study has also been presented in [32].)
3. The "CiAO-AS" study (Sect. 6) presents results from the aspect-aware development of a complete **configurable operating system** – from requirements and specification down to evaluation results. (Some results from this study have also been presented in [30,29].)

We discuss the results of these studies, which make the main contribution of the paper, in Sect. 7, related work in Sect. 8, and, eventually, conclude our findings in Sect. 9.

2 An Overview of CiAO

CiAO is a family of research operating systems that has been developed using AOP and software product line concepts from scratch. CiAO targets the domain of embedded systems, such as automotive applications. The *CiAO-AS* family member implements the automotive AUTOSAR-OS standard with configurable protection policies (memory protection, timing protection, and service protection) as defined in [3].

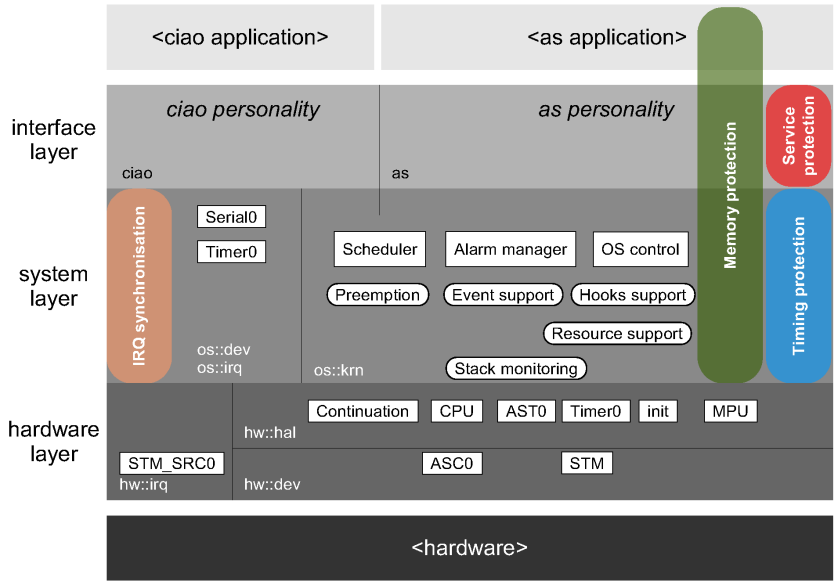

Fig. 1. Layered structure of CiAO. Depicted are the three fundamental layers of the CiAO architecture with a selection of their sublayers, components, abstractions, and aspects (depicted with rounded corners). Each subsystem and sublayer defines a separate namespace. Configurable architectural properties may have an effect across multiple layers.

2.1 Goals and Approach

> *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)

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

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

2.2 General Structure

Figure 1 gives an overview of CiAO's architecture. Like most operating systems, CiAO is designed with a *layered architecture*, in which each layer is implemented using the

functionality of the layers below (Figure 1). The only exceptions from this are the aspects implementing architectural policies, which may take effect across multiple layers.

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

In CiAO, however, layers do not just serve conceptual purposes, but also are a means of aspect-aware development. To provide cross-layer control-flow transitions (especially into and out of `os::krn`) as potential join points, 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 OStoHW() = call("% hw::...::%(...)") && within("% os::...::%(...)");
```

Control-flow transitions down the layer hierarchy are established by function calls; aspects can interfere with these transitions by giving advice to a pointcut like `OStoHW`. Transitions up the hierarchy (*upcalls*) are *only* established by aspects.

2.3 CiAO Design Principles

As pointed out in Sect. 2.1, the basic idea behind *aspect-aware operating system development* is to use aspects to achieve a clear separation between policies and mechanisms in the implementation. This leads 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 joinpoint level as statically evaluable, unambiguous join points.

The Principle of Minimal Extensions. Make sure that the overall system is extensible by minimal feature increments. System components and system abstractions should be fine-grained and spares, that is, provide only a minimal implementation of some feature, on order to be extensible by aspects on a fine granularity.

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

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

2.4 Roles and Types of Classes and Aspects

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*).[1]

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.

However, the distinction between the three types is not strict; a policy aspect, for instance, may also extend some class if this is part of the policy. We will see examples for all three types of aspects in the following sections.

3 Aspect-Aware Development Idioms

In the design and implementation of the CiAO system, we can find the recurring application of three development idioms, which we can understand as the operationalization of the CiAO design principles discussed in Sect. 2.3:

[1] The design decision to model system components and system abstractions as classes is also motivated by a technicality of the AspectC++ implementation language: Aspects cannot give advice to other pieces of advice.

Advice-Based Binding. Essentially, advice inverts the direction in which control-flow relationships are specified. By employing advice as the primary binding mechanism, we achieve an inherently loose *self-integration* of optional features into the control flows of the base system. We shall detail this further by concrete examples from CiAO in Sect. 3.1.

Explicit Join Points. Many semantically important control-flow transitions inside the kernel are not available as join points / not advisable because of technical reasons one has to deal with in low-level system software. An **explicit join point** is a named join point in the kernel control flow that bears a precisely defined semantics and can safely be advised. We shall describe this by concrete examples from CiAO in Sect. 3.2.

Extension Slices. The classes that represent CiAO's system components and system abstractions are generally sparse: they are either completely empty or implement only the minimal base of some feature. Optional features are implemented as **extension slices** and introduced by extension aspects into these classes. We shall give concrete examples from CiAO in Sect. 3.3.

Figure 2 gives a quick overview on the diagram notation we use for class-and-aspect diagrams in the subsequent sections. Several attempts have been published to extend UML with a notation of AOP elements.[2] However, the existing notations tend to be either too formal, too close to AspectJ, or both. Hence, we developed our own notation that offers a suitable level of detail. The notation is related on UML, but intentionally abstains from the official but verbose UML extension system (e.g., stereotypes) to bring in the relevant AOP concepts.

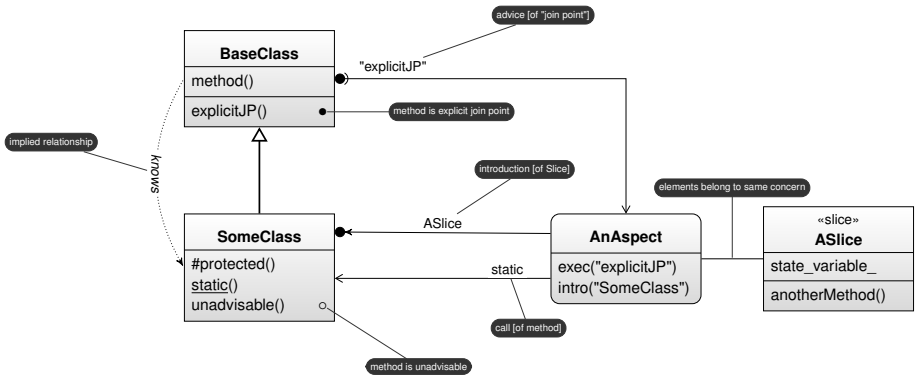

Fig. 2. Diagram notation for class-and-aspect diagrams. The notation is related to the notation of UML class diagrams, but uses several nonstandard style elements to depict aspects, advice, and introductions.

[2] Well-known examples are: *composition patterns* [10], *Theme/UML* [5], the notation by STEIN and colleagues [46], and *AML* [19].

3.1 Loose Coupling by Advice-Based Binding

With **advice-based binding** components and policies *integrate themselves* into the system.[3] This is the most fundamental idiom for the implementation of loose coupling. It exploits the effect that aspects effectively invert the direction in which control-flow relationships between components are established. Thereby, optional components and policies can easily hook into the system's control flows.

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

Component Self-integration. The canonical example of self-integration by advice-based binding is component initialization: Every CiAO component includes an accompanying `_Init` aspect that gives advice to the system initialization handler `hal::init()` to invoke the component's `init()` method at system startup time (see Figure 3). 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.

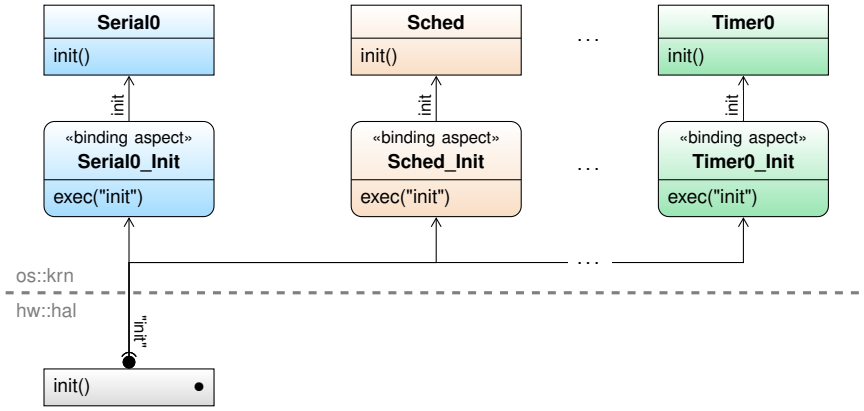

Fig. 3. Self-integration of components by advice-based binding. Depicted is the CiAO component initialization scheme. Every CiAO component integrates itself into the global system initialization handler `hal::init()` by an accompanying `_Init` aspect.

Figure 3 also demonstrates another advantage of advice-based binding, namely the **transparent support of** $1 : n$ **relationships**. Without any further preparations, multiple clients can bind to the same join point by several aspects giving advice for it. The result is sequential activation of the respective advice implementations at run time.

[3] Note that we understand both as logical concepts. They may technically be implemented by a set of classes and aspects.

However, with respect to loose coupling and aspect awareness, the most important benefit of advice-based binding is that we can *further* influence it by additional aspects, for instance with respect to the activation order. Consider an (optional) aspect `Serial0Ext` that extends the serial driver from Figure 3 by a task of its own (e.g., for some background protocol handling). This aspect effectively inserts a new functional dependency between the serial driver and the scheduler; the serial driver now *uses* the scheduler. The consequence for the implementation is that now the scheduler has to be initialized *before* the serial driver. This new constraint can easily be realized with **order-advice**. Additional to the extension of the class `Serial0`, the aspect `Serial0Ext` can specify a relative invocation order for the *foreign* aspects `Sched_Init` and `Serial0_Init` at the join point `execution( "void hw::hal::init()" )` as follows:[4]

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

Essentially, the aspect thereby re-establishes the *uses—hierarchy* of the system.[5]

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

Figure 4 demonstrates self-integration of policies by the example of two variants of the CiAO preemption policy. Generally, system components report the need for rescheduling (and, thus, potential preemption of the running task) by calling `Sched::setNeedReschedule()`. The actual activation of the scheduler is, however, delayed:

The aspect `Sched_LeaveBinding` in Figure 4.a implements a simple delayed activation policy for a cooperative system; with this policy, preemption is only possible at the return from some system service.

The aspect `Sched_ASTBinding` in Figure 4.b implements a more sophisticated delayed activation policy for an interruptive system; with this policy, preemption can also take place after interrupt termination. Technically, this is realized by binding the scheduler activation (`Sched::reschedule()`) to the function `AST0::ast()`, which is the handler of an *asynchronous system trap* (*AST*).[6] Additionally, the triggering of the AST is bound to `setNeedReschedule()`. The fact that the scheduler is now activated from

[4] A detailed explanation of the syntax of *order-advice* can be found in [44].

[5] The relationship *uses* describes dependencies between components of a system with respect to their functional correctness [37]. The *uses hierarchy* thereby describes the order in which components can be tested and integrated in a bottom-up development process.

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

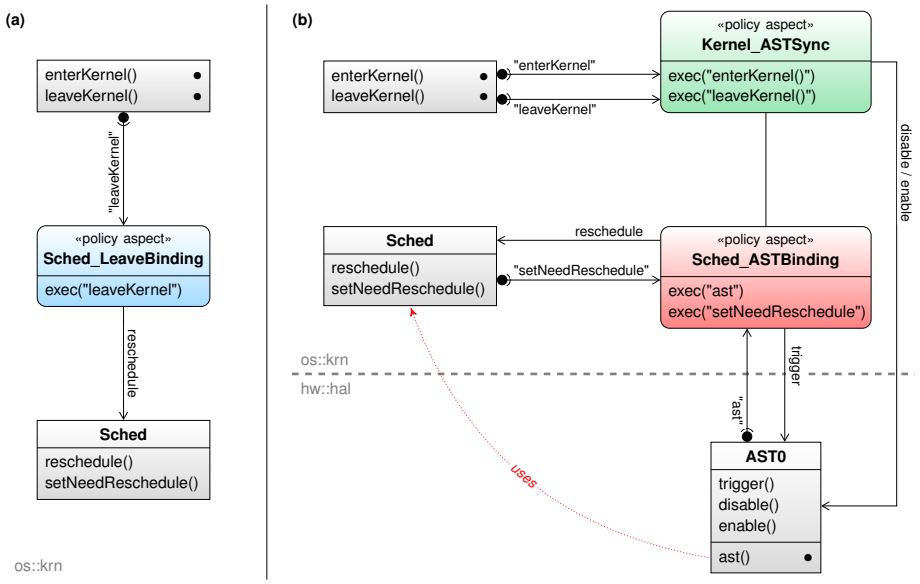

Fig. 4. Self-integration of policies by advice-based binding. Depicted are two alternatives for the delayed preemption policy in CiAO. (a) The aspect `Sched_LeaveBinding` binds to `leaveKernel()` to activate the scheduler when some task leaves the kernel. (b) The aspect `Sched_LeaveBinding` binds to the handler of an *asynchronous system trap* (*AST*) to activate the scheduler when all (potentially nested) interrupt handlers have terminated. This has the consequence that the scheduler now runs on the AST level, which leads to a new functional dependency between the (otherwise unrelated) system components `AST0` and `Sched`. The aspect `Kernel_ASTSync` re-establishes a correct *uses* hierarchy by synchronizing AST propagation with other control flows in the kernel.

`AST0::ast()` leads to a new functional dependency, which has the consequence that the kernel now has to be synchronized on the AST level. We can, however, easily enforce this constraint with additional pieces of advice given by the `Kernel_ASTSync` aspect.

3.2 Visible Transitions by Explicit Join Points

By consequent application of the three principles of aspect-aware development (Sect. 2.3) 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 on joinpoint level because they do not occur on the boundary of function calls or executions.

Table 1. Explicit join points in CiAO. Listed is a selection of *upcall* (U) and *transition* (T) join points offered by the different layers (respectively components in these layers) of CiAO. The actual set of available explicit join points is configuration dependent.

	type	representing function or method	description
os::krn	U	`interalErrorHook()`	Explicit join points for the support and binding of OSEK OS and AUTOSAR OS user-level hook functions, as specified in [36, p. 39] and [4, p. 46]. Triggered in case of an *error*, a *protection* violation, before (*pre*) and after (*post*) at high-level task switch, and at operating-system startup and shutdown time.
	U	`internalProtectionHook(StatusType error)`	
	U	`internalPreTaskHook()`	
	U	`internalPostTaskHook()`	
	U	`internalStartupHook()`	
	U	`internalShutdownHook()`	
	T	`enterKernel()`	Triggered when a control flow enters (respectively leaves) the kernel domain.
	T	`leaveKernel()`	
		…	
hw::hal	U	`ThreadFunc()`	Entry point of a new thread (continuation).
	T	`before_CPURelease(Continuation*& to)`	Triggered immediately before the running continuation is deactivated or terminated; `to` is going to become the next running continuation.
	T	`before_LastCPURelease(Continuation*& to)`	
	T	`after_CPUReceive()`	Triggered immediately after the (new) running continuation got reactivated or started.
	T	`after_FirstCPUReceive()`	
	U	`AST<#>::ast()`	Entry point of the respective AST.
	U	`init()`	Triggered during system startup after memory busses and stack have been initialized.
		…	
hw::irq	U	`<IRQ_NAME>::handler()`	Entry point of the respective interrupt handler. (Interrupts are still disabled.)
		…	

In other cases, their place of occurrence is configuration-dependent, or there are multiple places of occurrence. For example, *application* $\mapsto$ *kernel* transitions might occur if a kernel function is called, when a trap handler is activated, or during task switching to another task. However, as CiAO is designed as family of operating systems, this is not fixed, but a matter of configuration of the actual family member.

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. In the diagram notation, an open dot ($\circ$) is used to mark such unadvisable methods.

For these reasons, many CiAO components and layers provide furthermore a well-defined **explicit join-point interface** that defines one or several *explicit join points*.

An **explicit join point** is a named join point in the kernel control flow that bears a precisely defined semantics and can safely be advised. Technically, explicit join points are implemented as empty methods – provided for the sole purpose that aspects can bind to them. The joinpoint provider invokes these methods at run time, either directly or indirectly by component-specific adapter aspects. In the diagram notation, a filled dot (●) marks a method that represents an explicit join point.

We have already seen several examples: The methods `hal::init()` and `AST0::ast()`, for instance, are actually explicit join points. Both have an empty implementation, but represent the occurrence of a well-defined, semantically important run-time event. They are explicitly triggered by their providing components (the startup code; and the hardware-based or software-based implementation of the AST facility, both of which are platform-dependent).

Conceptually, explicit joinpoint 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, as explained in the previous section on the example of `hal::init()`.

Table 1 lists a selection of the explicit join points provided by CiAO. We distinguish between **upcall join points** and **transition join points**. This differentiation is not strict (depending on the client, an upcall can also represent a transition and vice versa); however, it underlines the primary purpose of the respective explicit join point.

Explicit Upcall Join Points. For the sake of configurability, the processing of most system-internal events is postponed to a higher layer than the layer on which they occur. By representing these events as explicit join points, higher layers (up to the application itself) can subscribe to them with advice-based binding.

The methods `hal::init()`and `AST0::ast()` are examples for (internally used) upcall join points. Other examples include interrupt handlers (`<IRQ_NAME>::handler()`), signal handlers (AUTOSAR OS hook functions), the `TCBUser::ThreadFunc()` start function of a new coroutine (will be further detailed in Sect. 4), or the method `Sched::idle()` that is called from the scheduler idle loop. Besides the direct binding (and potential inlining) of the event-processing code, the kernel can also exploit these join points to implement configuration-dependent upcall policies. An upcall-policy aspect may, for instance, filter events, or translate them into virtual function invocations, thread activations, or send message operations.

Explicit Transition Join Points. Control flow transitions inside the kernel, such as the transition from application level to kernel level, from thread level to interrupt level, or the context switch from one thread to another one, are important events for the implementation of many policies. Many of these events, however, have multiple sources ($m : n$ relationships); or they occur in fragile, low-level parts of the implementation. By representing them as explicit join points, providers, and publishers of transition events can be decoupled.

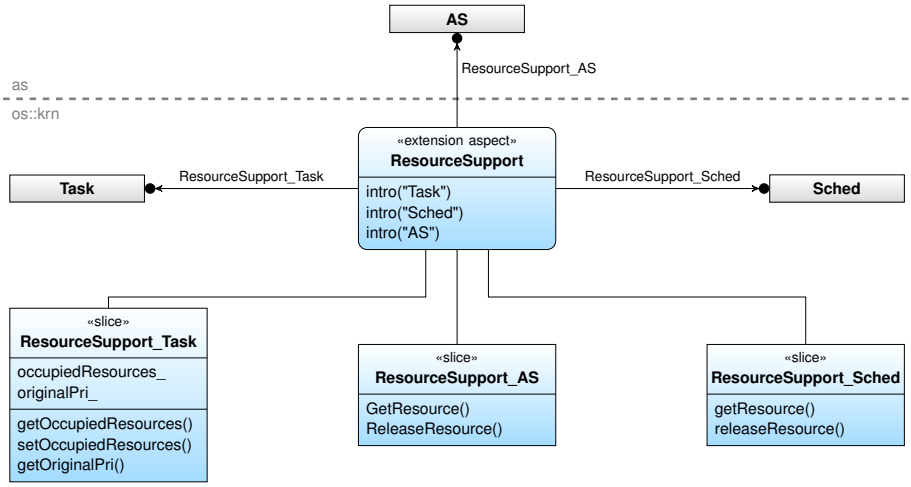

Fig. 5. Integration of an optional feature by extension slices. The aspect `ResourceSupport` adds support for AUTOSAR-OS resources to the CiAO kernel. This requires the introduction of respective *extension slices* into the `Task` system abstraction and the `Sched` system component from the *os::krn* layer, as well as into CIAO-AS API, which is contained in the class `AS` on the interface layer.

An already mentioned example of transition join points are the *application* $\mapsto$ *kernel* and *kernel* $\mapsto$ *application* transitions, which are represented in CiAO by the explicit join points `krn::enterKernel()` and `krn::leaveKernel()`. Another example are the transition join points provided by the CiAO dispatcher (the class `Continuation`), which are further detailed in Sect. 4.

3.3 Minimal Extensions by Extension Slices

The use of *extension slices* is the most relevant idiom for the implementation of minimal extensions: The implementation of optional features does usually not affect a single component or abstraction, but crosscuts with the implementation of several other concerns – often even across multiple layers. This is, for instance, always the case if the extension is also to become visible in the API provided by the interface layer.

Figure 5 demonstrates this by the example of the `ResourceSupport` extension aspect that adds support for AUTOSAR-OS resources to the CiAO-AS kernel. The actual implementation is introduced as methods and state variables into the `os::krn` classes `Task` and `Sched`. However, to be accessible from applications, the CIAO-AS API on the interface layer has to be adapted as well – which requires the introduction of the respective methods into the class `AS`.

With extension slices, collateral adaptations of several kernel components, abstractions, and the API can be separated and grouped into a single logical module – the extension aspect. Basically all optional system services provided by the CiAO-AS kernel are implemented this way. This ensures that services and abstractions that have not been configured for the kernel are not reflected in the API either; hence, many configuration errors can be detected early at compile time.

We can understand extension slices as the static counterpart of advice-based binding (see Sect. 3.1); the latter is used for loose coupling and self-integration with respect to the system's control flows, whereas the former fulfills the same task for the static system structure, that is, abstractions and components. Therefore, extension slices and advice-based binding are often used together. We will see examples for this in the "Continuation" study in Sect. 4.

3.4 Summary

The CiAO design principles of *loose coupling, visible transitions*, and *minimal extensions,* are to a high degree implemented by three development idioms: *advice-based binding*, *explicit join points*, and *extension slices.* Technically, these idioms exploit the mechanisms AOP provides for obliviousness: code advice and introductions. Here, they facilitate the self-integration (and thereby decoupling) of mechanisms and policies in the implementation — all "glueing" is done by advice.

The following sections further elaborate on the application of the discussed principles and idioms by three larger case studies from CiAO:

"Continuation". In the "Continuation" study we exemplify the aspect-aware decomposition and implementation of a **configurable system abstraction** – from features down to code. The purpose of this study is to demonstrate the need of explicit join points in even the fundamental low-level system abstractions of the operating system to achieve extensibility by visible transitions and loose coupling (Sect. 4).

"Interrupt Synchronization". The "Interrupt synchronization" study demonstrates the aspect-aware design (and partly also the implementation) of a **configurable architectural policy**. The purpose of this study is to show how architectural configurability can be achieved by means of aspect-aware development (Sect. 5).

"CiAO AS". The "CiAO-AS" study finally presents results from the aspect-aware development of a complete **configurable operating system** – from requirements and specification down to evaluation results. The purpose of this study is to show that the approach does scale up and is applicable for the development of a complete kernel (Sect. 6).

4 Case Study "Continuation"

In this section we present a detailed example of the aspect-aware design and implementation of a system abstraction. The Continuation concept is CiAO's system abstraction for the instantiation of preemptable control flows (*Coroutines, Threads*). It is provided by the *hw::hal* layer and serves as the base for other subsystems, especially the kernel, to implement higher-level thread or task abstractions.

We begin with a description of the intended variability of the Continuation concept in Sect. 4.1 and describe the resulting aspect-aware design in Sect. 4.2. This is followed by a detailed discussion of an actual implementation in Sect. 4.3, where the technical interactions and subtleties between aspects and near-hardware code are explored. Finally, Sect. 4.4 sums up our results.

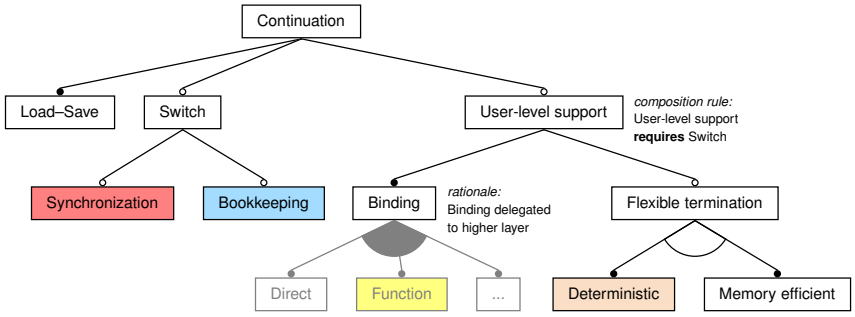

Fig. 6. Feature diagram of CiAO's control flow abstraction. Feature types include mandatory features (filled circle), optional features (hollow circle), minimum-one feature sets (filled arc), and exactly-one feature sets (hollow arc). The Continuation concept provides mechanisms to load, save, and initialize a control flow context (Load–Save); extension stages on top of this are mechanisms for dispatching (Switch) and the support for user-level control flows (User-level support). Context switches can be atomic (Synchroniziation) and tracked (Bookkeeping); user-level control flows can be allowed to terminate while executing subfunctions (Flexible termination). (The coloring is for the purpose of feature traceability with Figure 7.)

4.1 Continuation Features

Figure 6 depicts the variability of the Continuation concept as a feature diagram [12]. A CiAO continuation provides at least mechanisms to load, save, and initialize a control-flow context (Load–Save). Optional extension stages (features) are the mechanisms to dispatch from one continuation to another (Switch) and the interface for the kernel to implement user-level control flows (User-level support). User-level control flows can have an explicit entry point where they begin their execution; however, the actual representation of the entry point (Binding) is left open and postponed to a higher layer, which might implement it, for instance, as C-style function address, virtual function pointer, or a macro that is directly inlined into the Continuation concept. The extension stages can be further specialized by additional features to keep track of the running continuation (Bookkeeping), ensure atomicity of context switches (Synchronization), and permit user-level control flows to terminate from subfunctions (Flexible termination).

4.2 Continuation Design

The Continuation abstraction is provided by the *hw::hal* layer as a set of classes and aspects around the class `Continuation`. Figure 7 shows the resulting class-and-aspect diagram. The three extension stages Load–Save, Switch, and User-level support have been implemented as classes `ContinuationBase`, `Contination`, and `TCBUser`, respectively. In theory it would also have been possible to implement Switch and User-level support as extension aspects of `ContinuationBase`; the class-based design was chosen to support control-flow instances of different extension stages to coexist in the system (e.g., to execute interrupt handlers as `Continuation` instances while user-level threads are instances of `TCBUser`).

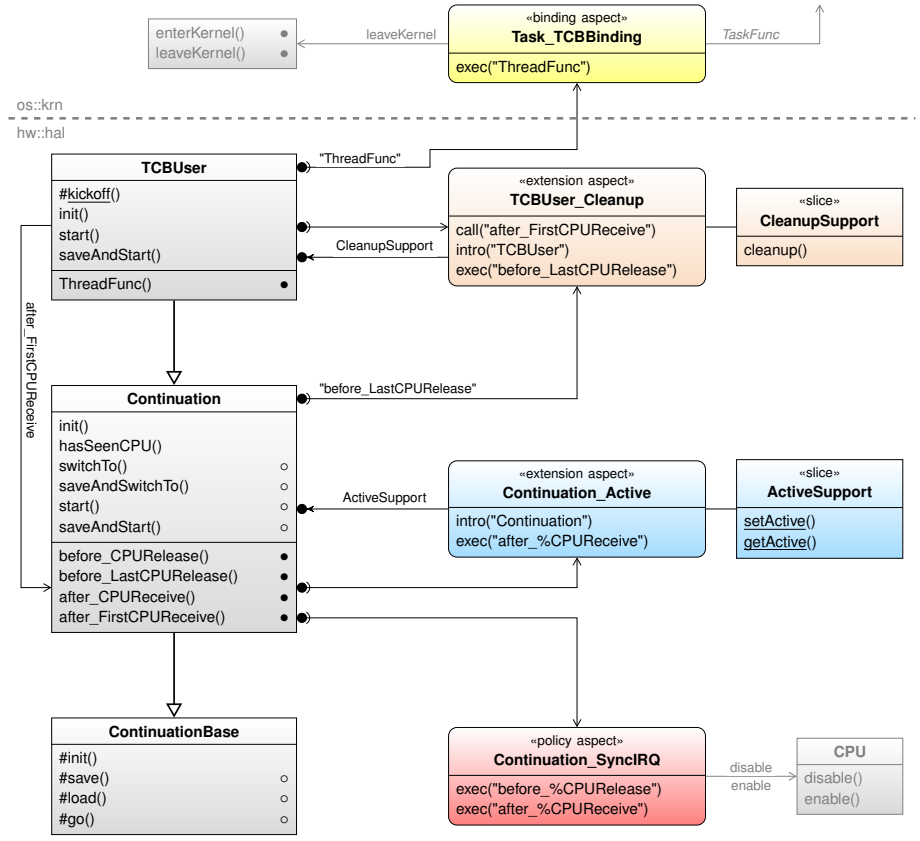

Fig. 7. Design of CiAO's control flow abstraction. Depicted is the static structure of classes and aspects that implement the features from Figure 6. Central element is the class `Continuation`, which provides an interface of four explicit transition join points; the class `TCBUser` extends `Continuation` by an additional upcall join point for the kernel. All other features are modeled as policy, binding, or extension aspects that bind to these join points.

Besides the relevant mechanisms, the classes `Continuation` and `TCBUser` each provide certain events by an explicit join-point interface (methods marked with ●) for potential aspects to bind to. Thereby, all other functions could be implemented as loosely coupled policy, extension, or binding aspects that use advice-based binding and extension slicing to integrate themselves into the respective abstraction.

The explicit join-point interface is of particular importance here. As pointed out in Sect. 3.2, implicit join points in the implementation of low-level mechanisms can be fragile or completely invisible and, hence, have to be considered as unadvisable (methods marked with ○). Even if they are visible, their semantics can bear subtle differences across platforms and implementations. The four explicit transition join points provided by the class `Continuation`, on the other hand, make all relevant transitions in the life cycle of a control flow (*start*, *deactivation*, *reactivation*, and *termination,* see also Table 1) visible on the join point level with well-defined semantics. In a similar manner the class

```
1  class ContinuationBase {
2   protected:
3    _tc::PCXI_t_nonv  pcxi_;      // previous context pointer (caller's context)
4    void*             addr_;      // return address (caller)
5
6    void init() {
7      pcxi_.reg = 0;
8    }
9    void CIAO_INLINE save() {
10     addr_     = _getRA();       // save return address (caller)
11     pcxi_.reg = _mfcr( $pcxi ); // save PCXI register (caller's context)
12     _dsync();                   // sync data pipeline
13   }
14   void CIAO_INLINE load() {
15     _mtcr( $pcxi, pcxi_.reg );  // restore PCXI register (caller's context)
16     _setRA( addr_ );            // restore return address (caller)
17     _isync();                   // sync instruction pipeline
18   }
19   void CIAO_INLINE go( void* tos, StartFunc starter ) {
20     _mtcr( $pcxi, 0 );          // new control flow has no caller
21     _setSP( tos );              // set the stack pointer
22     _isync();
23     hw::JUMP1( starter, this ); // jump to start address
24   }
25 };
```

Fig. 8. TriCore implementation of the class `ContinuationBase`

`TCBUser` provides with `ThreadFunc()` an upcall join point with a well-defined semantics for the binding of further kernel policies or abstractions.

In the following, we illustrate these constraints, the join point interfaces, and how they are used by extension, policy, or binding aspects by an actual implementation of the Continuation abstraction, namely for the Infineon TriCore platform [21], a modern 32-bit microcontroller that is used in the automotive domain.

4.3 Implementation for TriCore

On the TriCore platform with G++ as the compiler, all context switch functionality could be implemented in C++ (with utilization of a few assembler intrinsics that are provided as function-like macros, identifiable by their leading underscore).[7]

The Foundation: Continuation, ContinuationBase, and TCBUser

ContinuationBase. Listing 8 shows the TriCore implementation of the class `ContinuationBase`. The purpose of this class is to encapsulate the elementary state of a continuation and to provide the elementary operations to initialize, save, load, and

[7] This is also the reason we use the TriCore as illustration platform here: The respective implementation for other platforms supported by CiAO (IA32, ARM Cortex, ...) require the use of inline assembler statements, which makes them more verbose and less comprehensible.

begin a continuation context (`init()`, `save()`, `load()`, `go()`). As the TriCore CPU automatically saves and restores all non-volatile registers around function calls, the state to be managed in the continuation object itself is relatively small: Only the register that contains the return address and the register that points to the implicitly saved caller context have to be dealt with in the `save()`, `load()`, and `go()` operations.[8]

However, even though these operations are implemented in C++ they must not be advised – they are fragile. As `save()` and `load()` effectively store and restore the *caller's* context (the actual switch after a `load()` operation takes place when the *surrounding* function returns), they *must* be inlined into their invoker, which itself *must not* be inlined. The `go()` operation must be inlined, too – otherwise the call context that was implicitly created for the call to `go()` would never be freed.

The explicit control over inlining versus noninlining depends on compiler-specific language extensions.[9] As pointed out in Sect. 3.2, the nontrivial transformations of the aspect weaver might easily break such code. However, even if it were safely possible to give advice to these implicit join points we should refrain from doing so – they might be unavailable or bear subtle semantical differences in other implementations of the load–save mechanisms.

Continuation. The class `Continuation` implementation-inherits from `ContinuationBase` and uses the elementary operations to provide the four higher-level context-switch operations that are available to clients:

```
class Continuation : public ContinuationBase {
  void ... start( void* tos, StartFunc starter, Continuation* to );
  void ... switchto( Continuation* to );
  void ... saveAndStart( void* tos, StartFunc starter, Continuation* to );
  void ... saveAndSwitchto( Continuation* to );
  ...
```

These operations have to be considered as nonadvisable, too. They are also fragile with respect to inlining and may bear subtle semantical differences in other implementations. However, all control-flow transitions that result from using these operations are made visible on the join-point level by an explicit join-point interface:

```
  ...
  void before_CPURelease( Continuation*& to ) {}
  void before_LastCPURelease( Continuation*& to ) {}
  void after_CPUReceive() {}
  void after_FirstCPUReceive() {}
};
```

[8] A peculiarity of the TriCore platform is that call frames are not managed on the stack, but in linked lists of dedicated *context save areas* (*CSAs*) that are implicitly created and destroyed by the `call` and `ret` instructions. A CSA is a memory block of 128 bytes that represents a function frame including all nonvolatile registers and the pointer to the previous context. The *PCXI* (*previous context information*) CPU register always points to the most recent CSA, which is the head of the CSA list of the currently active thread. Consult [21, pp. 5-1ff] for further details.

[9] On G++ `CIAO_INLINE` and `CIAO_NOINLINE` expand to `__attribute__((always_inline))` and `__attribute__((noinline))`, respectively.

The explicit join points are triggered by the context switch operations. The `start()` and `switchto()` operations, for instance, are used to start, respectively reactivate, another continuation `to` without saving their own context. Both operations never return; the calling continuation terminates. Immediately before termination they trigger `before_LastCPURelease()` to signal this transition to interested aspects. In the implementation of `start()` this looks as follows:

```
void CIAO_INLINE start( void* tos, StartFunc starter, Continuation* to ) {
  before_LastCPURelease( to ); // we are going to leave forever
  to->go( tos, starter );      // start 'to'
}                              // <-- we never come here
```

Thereby, an aspect that gives advice to `before_LastCPURelease()` is activated whenever the current continuation control flow is about to terminate for `to` to receive the CPU. Note that `to` is passed as a *reference* parameter to `before_[Last]CPURelease()` – an aspect could not only inspect, but even influence the ongoing transition by choosing another continuation to receive the CPU.

The `saveAnd...()` context switch operations work similarly, but save the calling's continuation context first. They return when the calling continuation gets reactivated. Hence, they trigger `before_CPURelease()` and `after_CPUReceive()`, which signal these transitions. In the implementation of `saveAndSwitchto()` this looks as follows:

```
void Continuation::saveAndSwitchto( Continuation* to ){
  int_saveAndSwitchto( to );  // call (!) internal implementation
                              // <-- point of reactivation
  after_CPUReceive();         // hello, again
}
void CIAO_NOINLINE Continuation::int_saveAndSwitchto( Continuation* to ) {
  before_CPURelease( to );    // we are going to leave for 'to'...
  save();                     // but not forever
  to->load();                 // load 'to'
}                             // <-- point of 'to' reactivation
```

The explicit join point `after_FirstCPUReceive()` signals that a continuation has been started (activated for the very first time).

TCBUser. Class `Continuation` employ three constraints that have to be obeyed by its clients: (1) Every start function that is passed as parameter `starter` to the `start()` or `saveAndStart()` operations has to trigger the explicit join point `after_FirstCPUReceive()`. (2) `starter` has to adhere to a platform/compiler-dependent signature (that often involves using non-standard compiler features expressed by `#pragma` statements or `__attribute__((...))` qualifiers). (3) The control flow must not terminate (invoke `start()` or `switchTo()`) from a function call level below `starter`.

Whereas the above constraints are perfectly acceptable for system-internal control flows, they might be inappropriate for continuations that start in user-level code. In general, the decision *how* the user-level code has to be shaped, is bound, and gets activated should be understood as a policy decision of the kernel – and not be prescribed by the *hw::hal* layer.

The class TCBUser implements a continuation interface for the kernel that deals with these issues. It is intended as the base for user-level control flows and provides an explicit upcall join-point (ThreadFunc()) to which the kernel can bind its own policy for the binding and activation of user code without having to deal with constraints (1) and (2) imposed by Continuation. (The third constraint is tackled by the TCBUser_Cleanup extension aspect, which will be discussed further below.):

```
class TCBUser : public Continuation {
  ...
  // the internal Continuation start function
  static void cfHAL_STARTFUNC_ATTRIBUTES kickoff( TCBUser* me ) {
    me->after_FirstCPUReceive();   // we are alive
    me->ThreadFunc();              // and this is what we do
    _debug();                      // <-- we should never come here!
  }
 public:
  void inline TCBUser::ThreadFunc() { /* upcall JP for the kernel*/ }
};
```

Technically, TCBUser uses an internal start function (kickoff()) that fulfills the Continuation contract[10] and then triggers the explict upcall join point. As ThreadFunc() can be inlined by the compiler, this indirection does not cause an overhead.

The Aspects: Utilizing the Explicit Join-Point Interfaces. The remaining optional or alternative features have been implemented as aspects; they integrate themselves into the continuation classes with advice-based binding and extension slices (see Figure 7).

Continuation_IRQSync. Continuation_IRQSync is a platform-independent policy aspect that implements the optional Synchronization feature. It ensures atomicity of context switches by disabling interrupts in before_[Last]CPURelease() and reenabling them in after_[First]CPUReceive(). The implementation of this aspect is straightforward:

```
aspect Continuation_IRQSync {
  ...
  advice execution( "void ...::before_%CPURelease( ... )" ) && ... : before() {
    hw::hal::CPU::disable();
  }
  advice execution( "void ...::after_%CPUReceive( ... )" ) && ... : after() {
    hw::hal::CPU::enable();
  }
};
```

Continuation_Active. Continuation_Active is a platform-independent extension aspect that implements the optional Bookkeeping feature. It upgrades the class Continuation to a full-blown dispatcher that "knows" the currently running continuation. For this purpose, an extensions slice is used to introduce the static active_ pointer

[10] The cfHAL_STARTFUNC_ATTRIBUTES macro encapsulates the platform/compiler-dependent signature attributes with respect to constraint (2). On the TriCore platform, no such attributes are required, so it expands to nothing.

along with corresponding accessor functions into class `Continuation`; advice-based binding is employed to update the pointer after a context switch transition:

```
aspect Continuation_Active {
  pointcut pcClass() = "hw::hal::Continuation";
  advice pcClass() : slice class ActiveSupport {
    static ActiveSupport* active_;
   public:
    static void setActive(ActiveSupport* to) {active_ = to;}
    static ActiveSupport* getActive() {return active_;}
  };
  advice execution( "void ...::after_%CPUReceive( ... )" )
        && within( pcClass() )  : before() {
    JoinPoint::That::setActive( tjp->that() );
  }
};
```

TCBUser_Cleanup. `TCBUser_Cleanup` is an extension aspect that implements the Flexible termination feature. For the sake of potential stack sharing, continuation control flows are generally constrained to terminate only from the call-depth level of their start function – otherwise remaining call contexts may not be freed. On platforms that implicitly share call contexts between all control flows (as on the TriCore) this constraint is compulsory. However, as a restriction for the user level it might be inappropriate.[11] With the Flexible termination feature, it becomes permissible for user-level control flows to terminate even while executing some subfunction.

Listing 9 shows the source code of `TCBUser_Cleanup`, which implements the Deterministic alternative of Flexible termination for the TriCore platform. When a continuation terminates, it adds all remaining call contexts to the free list of the CPU (lines 17–22). In order to not have to collect them by walking down the whole list (which would take an indeterministic amount of time), the aspect creates an extra dummy context that serves as a tail pointer (`fcxi_`) at the beginning of a `TCBUser` continuation. By using call-advice instead of execution-advice, the creation of this extra context only affects `TCBUser` continuations (lines 12–16).

Task_TCBBinding. The last aspect under discussion is not provided by the *hw::hal* layer as part of the continuation concept; it is an example of the implementation of the alternative Binding feature by the kernel. `Task_TCBBinding` is the binding aspect that is employed by the CiAO-AS kernel implementation to bind the user-level task implementations to continuations via an AUTOSAR-OS–compatible `TaskFunc` function pointer. As the switch to user level is itself a semantically important transition (for which the kernel employs separate explicit join points, see Table 1), the aspect furthermore signals this transition before activating the user code:

[11] This depends on the kernel personality and, hence, should be configurable. AUTOSAR OS, for instance, constrains the allowed termination points of user-level task functions in a similar manner; however, other operating systems do not impose such a restriction.

```
aspect Task_TCBBinding {
  ...
  advice execution( "% hw::hal::TCBUser::ThreadFunc(...)" ) : around() {
    os::krn::leaveKernel();            // we are going up
    hw::JUMP ( tjp->that()->start_ );  // execute user-level code
  }
};
```

4.4 "Continuation" Summary

The CiAO control flow abstraction is a good example of the aspect-aware design and implementation of a low-level system abstraction by applying the idioms discussed in Sect. 3. Visible transitions by explicit join-points and advice-based binding facilitate the loose coupling of the core abstraction, its optional extensions, and the related policies. The result is a perfect one-to-one mapping from features to implementation components. New policies or extensions – either platform-independent or for some particular hardware platform – are easy to add. A good example of the latter would be an extension with respect to the amount of context information. If, for instance, a CPU architecture provides dedicated floating-point registers, saving and restoring these registers could be left to an extension aspect.

Especially the four explicit join points provided by class `Continuation`, which make all relevant transitions from the life cycle of a control flow visible on the join-point level, turned out as particularly useful. Besides the aspects depicted in Figure 7, we can find customer aspects that make use of them in many other parts of the kernel. This includes policy aspects from the implementation of the architectural policies memory protection and timing protection, but also extension aspects that implement optional kernel features, such as the support for AUTOSAR-OS hook functions.

```
1  aspect TCBUser_Cleanup {
2    pointcut pcClass() = "hw::hal::TCBUser";
3    advice pcClass() : slice class CleanupSupport {
4      _tc::PCXI_t_nonv fcxi_;                      // tail pointer (for dummy CSA)
5     public:
6      void CIAO_INLINE cleanup() {
7        _tc::PCXI_t_nonv head = _mfcr( $pcxi );    // get head of CSA list
8        _mtcr( $pcxi, 0 );                         // "forget" it
9        _tc::free_cx_list( head, fcxi_ );          // add CSAs to the CPU's free list
10     }
11   };
12   advice call( "% ...::after_FirstCPUReceive(...)" ) // when a *TCBUser* starts...
13         && within ( pcClass()  ) : after() {
14     _svlcx();                                      // create and link a CSA
15     tjp->target()->fcxi_ = _mfcr( $pcxi );         // remember it in fcxi_
16   }
17   advice execution( "% ...::before_LastCPURelease(...)" )
18         && within ( base( pcClass() ) ) : after() {
19     if( _mfcr( $pcxi) != 0 ) {                     // still CSAs left?
20       ((hw::hal::TCBUser*)(tjp->target()))->cleanup();  //   we need to cleanup
21     }
22   }
23 };
```

Fig. 9. TriCore implementation of the aspect TCBUser_Cleanup

5 Case Study "Interrupt Synchronization"

In the domain of event-triggered systems, *interrupt requests* (*IRQs*) are the common approach to signal events from peripheral devices (such as the expiry of a timer or a level change on a digital I/O line) to the CPU. The CPU deals with the event by the (immediate or delayed) execution of a corresponding *interrupt service routine* (*ISR*). To ensure consistency of system state that is accessed by ordinary control flows as well as by ISRs, the operating system has to apply measures for *interrupt synchronization*. Interrupt synchronization is an architectural policy – the chosen strategy is transparent to the application, but can have a notable influence on non-functional properties, such as *latency* and *performance*. For these reasons, interrupt synchronization is implemented in CiAO as a configurable policy.

We begin with a description of the intended variability of interrupt synchronization in Sect. 5.1 and the resulting aspect-aware design in Sect. 5.2. This is followed by an implementation sketch in Sect. 5.3 and a comparison of the resulting latencies in Sect. 5.4. Finally, Sect. 5.5 sums up our results.

5.1 CiAO Interrupt Synchronization Models

Interrupt handling in CiAO device drivers is explicitly divided into two parts: The first part, called *prologue*, is intended for time-critical actions and restricted with respect to the resources it may access, typically only hardware registers. Before termination, the prologue may request the (potentially delayed) execution of a second part. The second part, called *epilogue* in CiAO, is allowed to access other system components, such as the scheduler. The general idea is to execute the time-critical part immediately on interrupt level and the synchronized second part with a lower priority or at a later time when the required resources are available.

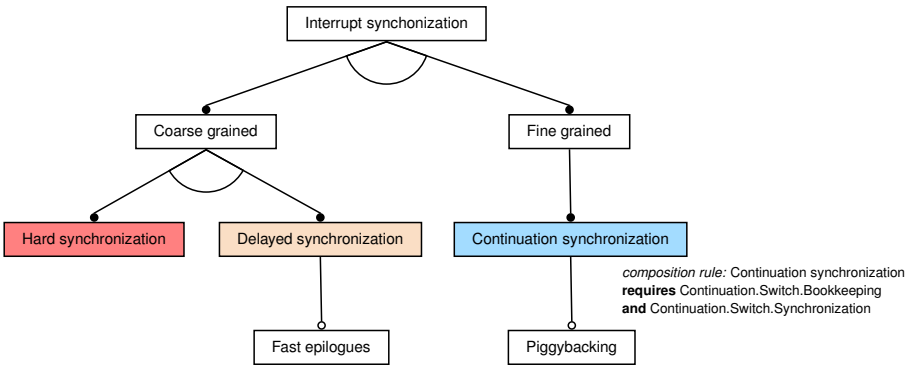

Fig. 10. Feature diagram of the configurable architectural policy *interrupt synchronization*. CiAO supports three different strategies for interrupt synchronization: Hard synchronization, Delayed synchronization, and Continuation synchronization. Continuation synchronization requires, however, certain Continuation features (see Figure 6) to be present. (The coloring is for the purpose of feature traceability with Figure 11.)

The feature diagram in Figure 10 depicts the offered variability for the architectural property interrupt synchronization. CiAO currently provides two different models for coarse-grained interrupt synchronization (Hard synchronization and Delayed synchronization) and one model for fine-grained interrupt synchronization (Continuation synchronization). They are all based on well-known techniques that are also used in other operating systems.

Hard Synchronization. In this configuration, the two parts are actually combined into one. When an interrupt occurs, prologue and epilogue are just executed consecutively on the interrupt level (interrupts remain disabled). Before accessing shared resources, threads have to enter interrupt level as well, that is, they have to disable interrupts.

The advantage of this model is its simplicity and low overhead. It is employed in many proprietary operating systems but also in sensor-network operating systems like TinyOS [6]. However, if interrupts are disabled too long or the interrupt handler has to perform a time-consuming task, latency rises and IRQ signals might be lost.

Delayed Synchronization. The prologue is executed with low latency at interrupt level. Epilogues are executed on their own epilogue level; they are queued until the kernel propagates them for execution, which is the case after all nested prologues have terminated and before the scheduler is activated. Epilogues thereby have priority over threads, but are interruptible by prologues if new IRQ signals come in. Threads inside the kernel can temporarily disable the propagation of epilogues to access shared resources. In this case, epilogue propagation is delayed until the thread finishes its access. Interrupts only have to be disabled if a thread or epilogue operates on prologue-accessible state.

If low latencies for critical handler code are crucial, this is our model of choice, as prologue deferment is rare and short. Many operating systems employ means for such a delayed execution of the major handler code: The term *epilogues* as used in CiAO stems from PEACE [41]; in eCos, epilogues are called *delayed service routines* (*DSRs*) [35], in Linux *Tasklets* [39,33], and Windows calls them *deferred procedure calls* (*DPCs*) [42].

The optional Fast epilogues feature represents an optimization. When this feature is applied, epilogues are – if possible – executed directly without queueing them first.[12]

Continuation Synchronization. In this configuration, the role of the prologue is the same as above. If an epilogue is requested, interrupts are reenabled and a new *continuation* (basic thread abstraction in CiAO, see Sect. 4) is started to execute the epilogue code in its own control-flow context. Epilogues synchronize with other continuation objects (epilogues or threads) via mutex objects using a priority inheritance protocol: The execution of the epilogue continuation can block on such mutex if a shared resource is currently in use by some thread or lower-priority epilogue; in this case, the owning control flow is reactivated until it frees the mutex. Hence, interrupts (respectively epilogues) and threads share (logically) a common priority space.

The major advantage of this model is that it thereby becomes permissible for interrupt control flows to block. This facilitates on-demand and fine-grained locking

[12] This is possible, when (a) there are currently no nested prologues and (b) epilogue propagation has not been temporarily disabled.

of kernel components. The implementation in CiAO was inspired by the interrupt-as-coroutines approach from Solaris [25]; however, the earliest references to this idea can be found in Moose [40] and Mach [2]. Several other systems, such as FreeBSD [34] and L4 [27], also execute interrupt handlers as threads.

The optional Piggybacking feature represents an optimization. When this feature is applied, interrupts "borrow" – if possible – the interrupted continuation control flow for the execution of their epilogue instead of starting their own continuation for this purpose.[13]

5.2 Design

In the following, we sketch the functional layers and their relevant components of interrupt synchronization in a bottom-up manner. The achieved separation of concerns through aspect-aware design will then be further detailed in Sect. 5.2; the implementation in Sect. 5.3.

Functional Layers. Figure 11 shows the structure of interrupt synchronization in CiAO as a layered model:

(1) Interrupt handling starts in the *hw*::*irq* layer (the interface to the underlying hardware), which contains one separate system component (a C++ class) for each (platform-specific) interrupt source. Each interrupt class provides a static `handler()` method as an explicit upcall join point (see Table 1). This join point is triggered when the corresponding interrupt occurs.

(2a) Binding aspects from the *os::dev* layer establish the link from hardware interrupt sources to corresponding system layer components (drivers). As a driver may service more than one IRQ, prologue and epilogue are contained in *virtual IRQ* (*VIRQ*) components inside the driver. VIRQs are the operating system's software abstraction for hardware interrupt sources and the corresponding handlers. Each VIRQ class provides an empty `handler()` method as an explicit transition join point for the execution policy.

(4a) The `Executor` policy aspect from the *policy* layer binds the proper activation of the actual interrupt handler implementation in `VIRQ::prologue()` and `VIRQ::epilogue()` to `VIRQ::handler()`.

(2) The *os* layer and its sublayers (*os::dev*, *os::krn*) contain the functional parts of the operating system, which are independent of the interrupt synchronization policy. Device drivers, but also other system components (such as the scheduler) are placed in this layer. Device drivers implement the interrupt service code as VIRQs (that is, define the behavior of `prologue()` and `epilogue()`), but have neither information nor any influence on the actual circumstances of their execution. Depending on the chosen synchronization model, a VIRQ may also act as a continuation or a delayed execution object. Every system component used by interrupts (either directly or indirectly) is subject to interrupt synchronization and provides an accompanying ..._`IntSync` aspect that describes its synchronization requirements.

[13] This is possible when the interrupted continuation has not aquired a mutex.

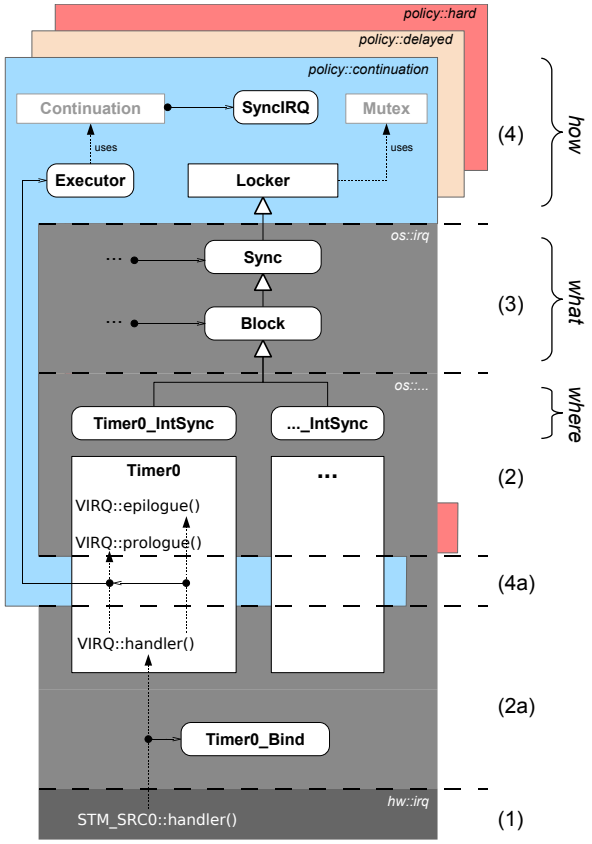

Fig. 11. Design model of the architectural policy interrupt synchronization in CiAO. Depicted is, on the example of a `Timer0` device driver and the Continuation synchronization strategy, how interrupt processing engages with the policy implementation (`Executor`, `Locker`) of the chosen strategy. (The coloring indicates the corresponding feature from Figure 10.)

(3) The *os::irq* layer is responsible for enforcing the synchronization constraints. The aspect `Block` enforces disabling of interrupts when methods are called that operate on prologue-accessible state (explained in the following section). It may be deactivated if we want to combine prologues and epilogues, which means that they are actually synchronized with the same mechanism. This is always the case with the Hard synchronization strategy. The `Sync` aspect enforces protection of all methods that run on the epilogue level.

(4) Finally, a (logical) *policy* layer implements the chosen model of interrupt synchronization. A policy layer contains at least a `Locker` and an `Executor`. Whereas the `Locker` encapsulates the implementation of the locking mechanism, the `Executor` applies the policy-dependent mechanisms for the activation of prologues and epilogues. This responsibility includes the necessary transformations of VIRQs in a way that they are able to act as a continuation or a delayed execution object.

In the shown *policy::continuation* strategy, the `Executor` activates epilogues as new continuations using the `Continuation` abstraction as dispatcher. VIRQs have to be equipped with their own `Continuation` context for this purpose; locking is implemented by mutex objects. As the context switch mechanisms are now also activated from interrupt level (the `Executor` aspect introduces a new functional dependency between interrupts and the class `Continuation`), they have to be synchronized on the interrupt level. For this purpose the `Continuation_SyncIRQ` aspect we discussed in Sect. 4 is employed.

Separation of Policies and Mechanisms. With respect to aspect-aware operating system development, the most interesting point of the sketched design is how the separation of policies and mechanisms is achieved. We can roughly divide the architectural policy of interrupt handling in two concerns:

Synchronization. The *synchronization* concern deals with the question of which mechanism is used for the coordination of interrupt and thread control flows (deferring of interrupts, deferring of epilogues, mutex) and where it has to be applied.

Execution. The *execution* concern deals with the question of which mechanism is used for the activation of interrupt control flows (direct, delayed, as separate continuation) and where it has to be applied.

Both concerns are not independent of each other – if we choose a blocking synchronization mechanism (such as mutex) we also need a preemptable execution mechanism (such as continuation). Hence, they together constitute an implementation of the interrupt synchronization policy.

To be able to develop system components (such as device drivers) in a way that they are transparent with respect to such a policy, a further separation is necessary. It can be expressed as simple questions of *how*, *where*, and *what*:

How. The answer to the question *how* we want to synchronize is the *policy-dependent* part of interrupt synchronization and execution. It defines which mechanisms are to be used for synchronization of components (`Locker`) together with a suitable execution model for prologues and epilogues (`Executor`).

Where. The answer to the question *where* synchronization measures have to be applied is the *component-dependent* part of interrupt synchronization. We need a representation of the synchronization requirements of each operating system component. As this knowledge depends on the component implementation, it has to be provided by the component developer. In the current implementation, synchronization requirements are encoded on a per-method level in the accompanying ...`IntSync` aspects, which provide named pointcuts that specify the members of each synchronization class.[14] We distinguish between three synchronization classes.

[14] The specification of the synchronization requirements on method granularity and by manual maintenance of named pointcuts is not optimal. Ideally, we would be able to tag methods directly with their synchronization class – instead of listing them in a named pointcut. Even better would be the possiblity to apply tags on a finer granularity, such as on the level of code blocks. Such tagging concept is currently being implemented for AspectC++; once it becomes available it will be easy to catch this up in the discussed design as no policy-related parts will be affected.

1. Most methods belong to the class *synchronized*, which means that they get synchronized on the epilogue level.
2. Methods that access prologue-accessible state belong to the class *blocked* instead, which means that they are synchronized on the prologue level.
3. Methods that only perform atomic or interrupt-transparent operations do not need to be subject of any synchronization measures and belong to the class *transparent*.

In principle, the *where* of prologue/epilogue activation is component-dependent, too. However, this knowledge is already encoded in an aspect-aware manner by the common structure and join-point interface of the VIRQ classes. Thereby it is possible to quantify over all points of prologue/epilogue activation with a single pointcut expression. Hence, in the actual implementation the *where* of the execution concern is not component-dependent and can directly be encoded as a pointcut in the `Executor` aspect.

What. Finally, we have to ensure that the chosen synchronization mechanism actually gets applied appropriately at the correct positions in the control flow. This part is independent of both the policy and the component. It is accomplished by pieces of advice given by the aspects `Sync` and `Block` from the *os::irq* layer. For the execution mechanism it is taken care of by pieces of generic advice given by the `Executor` aspect.

The concerns and their corresponding aspects are briefly summarized in Table 2.

Table 2. Concerns of *interrupt handling* and corresponding aspects in CiAO

aspect	#	concern
Binder	per IRQ–VIRQ mapping	upcall binding, hardware decoupling
VIRQ	per VIRQ per component	execution domain specification
Executor	per policy	execution mechanism and enforcement
Locker	per policy	synchronization mechanism
Sync	1	synchronization enforcement
Block	1	synchronization enforcement
IntSync	per component	synchronization domain specification

5.3 Implementation

In the following, we provide a closer look at some selected parts of the implementation. A typical characteristic of architectural policies is that their implementation homogeneously crosscuts with the implementation of a (potentially unknown) number of kernel components. In the actual implementation, this is tackled with generic advice.

Component Implementation. Listing 12.a shows excerpts from the driver implementation for the `Timer0` timer device: The `windupPeriodical()` method arms the timer

(a) Timer0.h

```
class Timer0 ... {
 ... // state
public:
 void windupPeriodical( long time );
 long value() const;
 void addEvent( const EventCB* cb );
private:
  void tick();
  void processEvents();
  class VIRQ ... {
    void handler();
    void prologue() {
      tick();
    }
    void epilogue() {
      process_events();
    }
  };
  ...
};
```

(b) Timer0_IntSync.ah

```
aspect Timer0_IntSync : public IntSync {
 pointcut pcClass = "os::dev::Timer0";

 pointcut virtual pcSynchronized() =
   within( pcClass() ) && (
      "% addEvent(...)"
   || "% processEvents()"
   );

 pointcut virtual pcBlocked() =
   within( pcClass() ) && (
      "% windupPeriodical(...)"
   || "% tick()"
   );

  pointcut virtual pcTransparent() =
    within( pcClass() ) && (
       "% value()"
    );
};
```

Fig. 12. A CiAO device driver with corresponding ..._IntSync aspect. (a) Class `Timer0` with inner class `Timer0::VIRQ` implements the driver for an interrupt-driven timer device. (b) The accompanying aspect `Timer0_IntSync` encodes the synchronization requirements of `Timer0` methods by assigning them to one of the pointcuts `pcSynchronized()`, `pcBlocked()`, or `pcTransparent()`.

device to request an interrupt after the specified `time`. However, for the timer to do this periodically, the `tick()` method has to be invoked from the interrupt handler to rearm the timer. This is considered time-critical, so it is done in the prologue. Therefore, the timer hardware registers and the period belong to *prologue-accessible state*. Consequently, `windupPeriodical()` and `tick()` belong to the synchronization class *blocked*. The callback functions, which are registered by `addEvent()` and eventually get triggered by `processEvents()`, are held in a queue. This queue belongs to the *epilogue-accessible s*tate; hence `addEvent()` and `processEvents()` are members of the synchronization class *synchronized*. The `value()` method simply reads the timer value; this happens atomically on this hardware architecture and does not need to be synchronized, so `value()` belongs to the synchronization class *transparent*. The accompanying `Timer0_IntSync` aspect in Listing 12.b encodes these decisions by specifying a named pointcut for each class. The named pointcuts `pcSynchronized()`, `pcBlocked()`, and `pcTransparent()` are actually definitions of pure virtual pointcuts from the aspects `Block` and `Sync` (Sect. 5.3), from which (indirectly) every ..._IntSync aspect inherits.[15]

[15] An aspect may give advice to an yet undefined (pure virtual) pointcut, which turns the aspect into an *abstract aspect*; the pure virtual pointcut eventually has to be defined by some derived aspects [44].

Policy Implementation. The `Locker` is simply a type alias to a class which provides methods to be called in order to protect critical method calls. In the case of Hard synchronization that class looks like this:

```
struct Hard {
  static void enter() {
    hw::hal::CPU::disable();
  }
  static void leave() {
    hw::hal::CPU::enable();
  }
};
```

For the other models, more sophisticated actions are to be performed by these two methods. With Continuation synchronization, for instance, `Locker` is aliased to the kernel mutex class `IMutex`, which has to deal with priority inheritance and potential dispatching in `enter()` and `leave()`.

With respect to configurability, the more interesting part of model implementation is the `Executor` aspect, which again looks relatively simple in the case of Hard synchronization:

```
aspect Executor_Hard {
  advice execution("% os::...::VIRQ%::handler(...)") : after() {
    if (JoinPoint::That::prologue())  // execute prologue
      JoinPoint::That::epilogue();    // execute epilogue
  }
};
```

The `Executor` aspect is quantified over all VIRQ classes; it uses *generic advice* [28] to bind `prologue()` and `epilogue()` via their static type (given by `JoinPoint::That`), so they can be inlined.

If we want to run the Delayed synchronization policy, the aspect has to enqueue the VIRQ for later execution of the epilogue. For this, it is necessary to transform VIRQ classes into queueable objects by introducing a `Queueable` base class:

```
aspect Executor_Delayed  {
  advice "os::...::VIRQ%" : slice class Gate : public Queueable {};
  advice execution("...::VIRQ%::handler(...)") : after() {
    if ( JoinPoint::That::prologue() ) {          // execute prologue
        Guard::relay( JoinPoint::That::Inst() );  // enqueue for later exec.
  } }
};
```

To be queueable, an actual instance is needed for each VIRQ class, even though the VIRQ classes contain just static elements. To provide such instance the introduced slice `Gate` also transforms VIRQs into singletons.

The realization of Continuation synchronization requires one separate continuation context per VIRQ class to which the `Executor` may switch for epilogue activation:

```
aspect Executor_Continuation  {
  advice "...::VIRQ%" : slice class {     // introduce its own Continuation
    static Continuation ctx;              // into every VIRQ
    static char *stack[cfIRQ_EPISTACK];
    static void cfHAL_STARTFUNC_ATTRIBUTES entry( Continuation* me) {
      me->after_FirstCPUReceive();
      epilogue();                         // run epilogue
  } };

  advice execution("...::VIRQ%::handler(...)") : after() {
    ...
    typedef JoinPoint::That VIRQ;
    if ( VIRQ::prologue() ) {                   // execute prologue
      Continuation::getActive()->saveAndStart(  // save current context and
        &VIRQ::stack[cfIRQ_EPISTACK],           //  start epilogue in its own
        VIRQ::entry, &VIRQ::ctx);               //  continuation
  } ... }
};
```

No object instance for the VIRQ is needed in this case as all members can be static.

Enforcement of Synchronization. To accomplish the task of combining *what*, *how* and *where*, the aspects of this layer use pure virtual pointcuts to which they give pieces of advice that contain the synchronization code. These pointcuts are later defined by the component-specific ..._IntSync aspects. The `pcExclude()` pointcut protects *synchronized* but magic code (for example the epilogue itself) from being affected by the piece of advice:

```
aspect Sync : Locker {

  pointcut virtual pcSynchronized() = 0;
  pointcut pcToSync() = call(pcSynchronized()
                        && !pcExclude())
                        && !within(pcSynchronized());

  advice pcToSync() : around() {
    enter();
    tjp->proceed();
    leave();
  }
};
```

For fine-grained locking as used by the Continuation synchronization strategy, every component has to be synchronized independently. This is achieved by the fact that a separate instantiation of the whole synchronization hierarchy is performed for each (component-specific) ..._IntSync aspect, resulting in one `IMutex` per component. In this case the

`Sync` aspect instruments all calls into "foreign" synchronization domains to obtain and release the respective `Mutex` instance around the call.

With coarse-grained locking as used by the Hard synchronization and Delayed synchronization strategies, all components share a single synchronization domain. This is realized by combining the component-specific definitions of the virtual pointcuts `pcSynchronized()`, `pcBlocked()`, and `pcTransparent()`.

5.4 Interrupt Latency Comparison

Table 3 compares the relative prologue and epilogue activation overhead of the three implementations.[16] The numbers represent the latency in the optimal case: no other control flow is in the kernel that blocks or delays the execution of prologues and epilogues. It is therefore not surprising that the implementation of Hard synchronization performs best as this model involves the lowest ground overhead. With Delayed synchronization, the prologue activation time is identical, however the potentially delayed execution of the epilogue causes some overhead. As expected, the overhead is highest in the implementation of Continuation synchronization. For the later context switch out of interrupt state, the TriCore CPU requires some additional processing before entering the prologue, which causes the higher latency for its activation. The context switch to activate the epilogue itself comes at a price, too, even though 60 cycles can still be considered as a fairly small overhead for the gained flexibility of fine-grained locking.

Although the latency of Continuation synchronization is highest in Table 3 (the ideal case without any delays), it can quickly pay off: If the length of epilogue locks caused by other interrupt handlers or kernel components exceeds the amount of 20 cycles, we already could have a break-even to Delayed synchronization. This, however, always depends on particularities of the concrete application (event frequency, deadlines, processor utilization, ...) and, thus, should be configurable.

The last two rows show the interrupt latency of category 1 and category 2 ISRs in a commercial OSEK implementation.[17] In OSEK OS, ISRs cannot be split into two parts; instead they run either, as a prologue, outside of the kernel (category 1 ISR) or, similar to an epilogue, synchronized with the kernel (category 2 ISR) [36, p. 25].

5.5 "Interrupt Synchronization" Summary

Interrupt synchronization is a good example of the decoupling of policies and mechanisms of even architectural policies in CiAO. Whereas the enforcement of a usual kernel

[16] All measurements in this paper base on variants that 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 were retrieved byte-exact from the linker-map files. Run-time numbers were measured with a high-resolution hardware trace unit (Lauterbach PowerTrace TC1796).

[17] 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.

Table 3. Latencies for non-delayed interrupts in CiAO and OSEK. Depicted numbers are the elapsed time [*cycles*] from the begin of the hardware interrupt handler to the first *prologue* instruction, *epilogue* instruction, and until interrupt termination (*iret*).

	t_{prologue}	t_{epilogue}	t_{iret}
hard	*8*	*8*	*16*
delayed	*8*	*40*	*60*
continuation	*16*	*60*	*108*
OSEK category 1 ISR	*12*	–	*20*
OSEK category 2 ISR	–	*12*	*20*

policy (such as the preemption strategy presented in Figure 4) affects only a small selection of well-known components, it is characteristic for an architectural policy that its implementation crosscuts with the implementation of a (potentially unknown) number of kernel components. Hence, it has to be quantifiable – which requires some preparations on the side of the affected components. CiAO's kernel components are *aware* of interrupt synchronization – they adhere to a common driver model for VIRQs and explicitly specify their synchronization requirements. However, they do not have to know the concrete strategy. Thanks to generic advice and static typing, this architectural transparency can be implemented in way that leads to quite efficient yet flexible and concern-separated code.

6 Case Study "CiAO-AS"

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.

The main point that distinguishes AUTOSAR OS from its predecessor OSEK OS [36] and other operating systems in the domain of statically configured embedded systems 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 a misbehaving component, and *service protection*, which checks for correct usage and context of system-service invocation.

The purpose of the "CiAO-AS" study was to evaluate the identified principles and idioms of aspect-aware operating-system development on a larger scale, that is, by construction of a complete kernel. The AUTOSAR OS standard is a particularly interesting test subject for this kind of evaluation:

- AUTOSAR OS is not a concrete system but a standard, described by a set of requirements and a detailed specification of the system services (API) and abstractions.

This makes it possible to evaluate the approach with a top-down implementation of real-world requirements.
- The suggested protection facilities (*memory protection*, *timing protection*, and *service protection*) make AUTOSAR OS a convincing case for architectural configurability by aspects.
- AUTOSAR is currently a "hot topic" in the embedded systems domain.

CiAO-AS is an AUTOSAR-OS operating system based on the CiAO kernel. In the following, we present and discuss some results from the "CIAO-AS" study. As we already have discussed several examples for the aspect-aware design and implementation of CiAO's system abstractions on a relatively high level of detail, we shall concentrate more on the achieved general results. This includes the global analysis of AUTOSAR-OS concerns in Sect. 6.1 and their interdependencies in Sect. 6.2, the implementation of these concerns by aspect-aware operating-system development in CiAO (Sect. 6.3), and the memory and execution-time footprint of the resulting kernel in Sect. 6.4. Finally, Sect. 6.5 summarizes our results.

6.1 AUTOSAR OS Abstractions in a Nutshell

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* (comparable to signal handlers in other operating systems), which are invoked in case of certain control-flow events (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.

6.2 Analysis Results – From Requirements to Concerns

The AUTOSAR-OS standard proposes a set of *conformance-and-scalability classes* for the purpose of system tailoring [36,4]. These classes are, however, relatively coarse-grained and do not clearly separate between conceptually distinct concerns. As CiAO aims at a much better granularity (see Sect. 2.1), *every* AUTOSAR-OS concern is represented as an individual feature in CiAO-AS.

Table 4 presents – in a condensed form – an excerpt of the results of the analysis of the AUTOSAR-OS concerns.[18] It lists some of the identified concerns of AUTOSAR OS (column headings) and how we can expect them to interact with the named entities of the implementation (row headings), that is, the 44 system services (e.g., `ActivateTask()`) and the relevant system object types (e.g., `TaskType`) specified in [36,4]. Additionally examined in Table 4 are some *internal concerns* (Preemption) and

[18] For the complete table please consult our AOSD '11 paper [29].

Table 4. Influence of configurable concerns (columns) on system services, system types, and internal events (rows) in AUTOSAR OS [4,36]. Kind of influence: ⊕ = extension of the API by a service or type, ⊛ = extension of an existing type, ◑ = modification after service or event, ◐ = modification before, ● = modification before *and* after.

	System abstractions (functional)							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()`					⊕			◑	...	◑	◐	◐	◐	◐	...		...
`ReleaseResource()`					⊕			◑	...	◐	◐	◐	◐	◐	...	◑	...
... <4 event services>						⊕		◑	...		◐	◐	◐	◐	...	◑	...
... <6 alarm services>							⊕	◑	...		◐	◐	◐	◐	...	◑	...
... <7 schedule table services>							⊕	◑	...		◐	◐	◐	◐	...		...
... <7 OS application services>								◑	...		◐	◐	◐	◐	...		...
`TaskType`		⊕			⊛	⊛			...	⊛				⊛	...	⊛	...
`ResourceType`					⊕				...					⊛	...		...
... <4 more structures>	⊕	⊛		⊕		⊛	⊕		...	⊛				⊛	...		...
System startup		◑					◑	◑	...						...		...
Task switch								●	...	●					...		...
Protection violation								◑	...						...		...
... <4 more internal points>		◑				◑		◐	...	●				◑	...	◑	...

internal transitions (events) that are not mentioned explicitly in the AUTOSAR-OS specification, but that are nevertheless of high relevance. These concerns were examined by experience and deduction [29].

Table 4 thereby provides an idea of how we can expect AUTOSAR-OS concerns to crosscut with each other in the structural space (types, services) and behavioral space (control flows events) of the implementation. We can see, for instance, that the design of the API is canonical – each system service and object type is motivated (introduced) by exactly one concern. The state to be maintained in the listed object types, however, is influenced by several concerns, so is the behavior that is associated with the execution of a system service. In fact, there is not a single AUTOSAR-OS service that is influenced by only one concern!

Some concerns are "highly crosscutting", in the sense that their implementation is expected to touch a high number of system services or object types. The Hooks facility, for instance, includes support for several application-specific signal handlers, among them the `ErrorHook` that is to be invoked in case of an error. In the implementation, this touches every system service that may return with an error code (i.e., those that return a `StatusType`).

As expected, some of the architectural protection properties classify as "highly crosscutting", too. This is particularly true for the various Protection constraints to be checked for – the enforcement of these constraints is naturally associated with the execution of system services.

The identified system internal events are of particular importance with respect to an aspect-aware development as they reflect relevant transitions that are *not* implicitly provided by a system service. Instead, we had to deal with these transitions explicitly in the design and implementation, for example, model them as explicit join-points. We have already seen several examples for this in Sect. 3.2.

Overall, the concerns defined by the AUTOSAR-OS specification documents [4,36] bear a surprisingly high amount of crosscutting in the specified services and abstractions. A concrete implementation should profit significantly from the aspect-aware development approach.

6.3 Development Results – From Concerns to Classes and Aspects

In its full configuration, the CiAO-AS kernel contains three basic system components (cf. Sect. 2.4) that are singletons by definition and implemented as classes:

1. The *scheduler* (`Scheduler`) takes care of the dispatching of tasks and the scheduling strategy.
2. The *alarm manager* (`AlarmManager`) takes care of the management of alarms and the underlying (hardware / software) counters.
3. The *OS control facility* (`OSControl`) provides services for the controlled startup and shutdown of the system and the management of OSEK-OS / AUTOSAR-OS application modes.

Also represented as classes are the system abstractions (the types that represent instantiable system objects, such as `TaskType`, `ResourceType`, and so on) and the namespace of the API (`AS`).

However, as described in Sect. 2.4, these classes are sparse or even empty. If at all, they implement only a minimal base of their respective concern. All further features and variants are brought into the system by aspects. The class `Scheduler`, for instance, provides only the minimal base of the scheduling facility, which is nonpreemptive scheduling with single task activations. Support for more sophisticated preemption or activation modes is provided by additional *policy aspects* and *extension aspects*. Table 5 displays an excerpt of the list of AUTOSAR-OS concerns that have been implemented as aspects in CiAO-AS.

The first three columns list for each concern the number of *extension*, *policy*, and *upcall aspects* that implement the concern. For all concerns, the implementation could be realized as a single aspect. (The *further* separation into an extension aspect and a policy aspect in two cases (Resource support and Protection hook) is owed to the goal of strict decoupling of mechanisms and policies suggested by the CiAO approach, see Sect. 2.1.)

The majority of concerns from Table 5 contributes to the set of *policy aspects* (12 aspects), which by extension 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 HAL by its own upcall aspect. These aspects are, however, not to be provided

Table 5. CiAO-AS kernel concern implemented as aspects with number of affected join points. Listed are kernel concerns that are implemented as *extension*, *policy*, or *upcall aspects* (not including aspects for memory protection, timing protection, and *hw::hal*-bindings), together with the related pieces of *advice* (not including order-advice), the affected *join points*, and a short explanation for the purpose of each join point (separated by "|" into *introductions of extension slices* | *advice-based binding*).

concern	extension	policy	upcall	advice	join points	extension of | advice-based binding to
ISR cat. 1 support	1		m	$2+m$	$2+m$	API, OS control | m ISR bindings
ISR cat. 2 support	1		n	$5+n$	$5+n$	API, OS control, scheduler | n ISR bindings
ISR abortion support	1			2	$1+m+n$	scheduler | $m+n$ ISR functions
Resource support	1	1		3	5	scheduler, API, task | PCP policy implementation
Resource tracking		1		3	4	task, ISR | monitoring of Get/ReleaseResource
Event support	1			5	5	scheduler, API, task, alarm | trigger action JP
Alarm support	1			1	1	API
OS application support	1			2	3	scheduler, task, ISR
Full preemption		1		2	6	| 3 points of rescheduling
Mixed preemption		1		3	7	task | 3 points of rescheduling for task / ISR
Multiple activation support	1			3	3	task | binding to scheduler
Stack monitoring		1		2	3	task | CPU-release JPs
Context check		1		1	s	| s service calls
Disabled interrupts check		1		1	30	| all services except interrupt services
Enable w/o disable check		1		3	3	| enable services
Missing task end check		1		1	t	| t task functions
Out of range check		1		1	4	| alarm set and schedule table start services
Invalid object check		1		1	25	| services with an OS object parameter
Error hook			1	2	30	scheduler | 29 services
Protection hook	1	1		2	2	API | default policy implementation
Startup / shutdown hook			1	2	2	| explicit hooks
Pre-task / post-task hook			1	2	2	| explicit hooks

by the application developer; they are generated automatically during the build process from the application configuration.

Another interesting point is the realization of synergies by *quantification*. If for some concern the number of pieces of advice is lower than the number of affected join points, (displayed in the last two columns of Table 5) we have actually profited from the AOP concept of quantification. For 14 out of the 22 concerns listed in Table 5 this is the case.

The net amount of this profit, however, depends on the type of the concern and aspect. *Extension aspects* typically crosscut *inhomogeneously* with the implementation of other

Table 6. Distribution analysis of CiAO base code and aspect code. Listed are the number of files and the lines of code (LOC) in the C++ base code (headers and implementation) and the AspectC++ aspect code, counted using CLOC [13].

	Base code		Aspect code	
	Files	LOC	Files	LOC
CiAO kernel only	*423*	*21,086*	*333*	*5,923*
CiAO COM	*112*	*8,689*	*297*	*5,552*
CiAO IP stack	*45*	*5,038*	*96*	*3,230*
CiAO overall	*580*	*34,813*	*726*	*14,705*

concerns, which does not leave much potential for synergies by quantification. *Policy aspects* on the other hand – especially those for architectural policies – tend to crosscut *homogeneously* with the implementation of other concerns; here quantification creates significant synergies. Note, however, that in many cases the necessary homogeneity of the affected join points could only be achieved by using generic advice.

Table 6 shows the distribution of aspects and base code in the CiAO operating system; its communication stack (COM) and the IP stack are listed separately since they constitute large modules of the system. Overall, CiAO exhibits about one line of aspect code per 2.4 lines of C++ base code in the system, showing the focus on aspect orientation in the implementation. We discuss implications of this fact on CiAO developers in Sect. 7.

Overall, the identified AUTOSAR-OS concerns (cf. Sect. 6.2) could be well separated into distinct implementation units in terms of class modules and aspect modules by applying the principles and idioms of aspect-aware operating system development.

6.4 Evaluation Results – From Configurations to Cost

In the following, we present some results from the performance and memory footprint evaluation of the CiAO-AS implementation that demonstrate the achieved granularity.

Execution Time. The effects of the achieved configurability also become visible in the CPU overhead. Table 7 displays the execution times of a number of tasking related micro-benchmark scenarios **(a)** – **(j)** and a comprehensive application **(k)** on CiAO and the commercial OSEK implementation. For each scenario, we first configured both systems to support the smallest possible set of features (*min* columns in Table 7). 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 "artificially enriched" version of CiAO that provides the same amount of unwanted functionality as OSEK (column *full* in Table 7). This reduces the performance differences; however, CiAO is still faster in 6 out of 11 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.

Table 7. Performance measurement comparison from CiAO and OSEK. Listed numbers represent execution times [*cycles*] taken by CiAO and OSEK .

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*

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 are depicted in Table 8. Listed are the deltas in code, data, and BSS section size per feature that are 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, and 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.

Table 8. Scalability of CiAO's memory footprint. Listed are the increases in static memory demands [bytes] of selected configurable CiAO features.

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*

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. This effect underlines again the flexibility of loose coupling by advice-based binding.

6.5 "CiAO-AS" Summary

The results from the "CiAO-AS" study show that the approach of aspect-aware operating-system development is both feasible and benefitial for the implementation of real-world operating systems. The concerns, services, and abstractions defined by the AUTOSAR-OS standard bear a noticeable amount of internal crosscutting. Nevertheless, by applying the principles and idioms of aspect-aware operating system development, they could be implemented in a well separated and fine grained manner in CiAO-AS.

7 Discussion of Results

With respect to the goals described in Sect. 2.1, the approach of aspect-aware operating system development was quite successful. CiAO reaches the primary research goal of **architectural configurability**. The system combines a good separation of concerns in the implementation with excellent granularity and configurability, and – thereby – a quite competitive efficiency regarding hardware resources.

In the following, 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.

7.1 AOP and Operating Systems

Aspects as First-Class Citizens. 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 [45]: "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 4) 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.

Quantification and Obliviousness. The `DisabledIntCheck` aspect is also a good example of the benefits of *quantification* because of homogeneous cross cutting. Given that other studies [22] 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 5, this is the case.

With respect to *obliviousness*, the situation is less clear. In [16], 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 independent 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.

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 silently dropped. This property is useful for the implementation of *interacting* optional features, which are difficult to tackle with other decomposition approaches [23].

Extensibility. We are convinced that the three design principles of aspect-aware operating system development (*loose coupling*, *visible transitions*, and *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 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 [14]. That kind of priority inversion occurs when low-priority control flows are executed in interrupt handlers, which can disturb execution of high-priority control flows executed in

```
1  aspect PCP_Extension {
2    advice execution("void hw::init()") : after() {
3      PCP::init();
4    }
5    advice execution("% Scheduler::setRunning(...)") :
6    before() {
7      PCP::setPrio(os::krn::Task::getPri(tjp->args<0>()));
8    }
9    advice execution("% enterKernel(...)") : after() {
10     // wait until PCP has left kernel (Peterson)
11     PCP_FLAG0 = 1; PCP_TURN = 1;
12     while ((PCP_FLAG1 == 1) && (PCP_TURN == 1)) {}
13   }
14   advice execution("% leaveKernel(...)") : before() {
15     PCP_FLAG0 = 0;
16   }
17   advice execution("% AST0::ast(...)") : around() {
18     // AST0::ast() is the AST handler that activates
19     // the scheduler (bound by an upcall aspect)
20
21     // wait until PCP has left kernel (Peterson)
22     PCP_FLAG0 = 1; PCP_TURN = 1;
23     while ((PCP_FLAG1 == 1) && (PCP_TURN == 1)) {}
24
25     // proceed to aspect that activates scheduler
26     tjp->proceed();
27     PCP_FLAG0 = 0;
28   }
29   advice execution("% Scheduler::schedule(...)") : after() {
30    // write priority of running task to PCP memory
31    PCP::setPrio(Task::getPri(
32       Scheduler::Inst().getRunning()));
33   }
34 };
```

Fig. 13. PCP co-processor extension aspect. The listing shows the complete implementation for integrating the PCP into CiAO, except for 10 lines of initialization code and the PCP code itself (which has to be programmed in assembly language).

threads. With the CiAO PCP extension, the CPU is only interrupted when there is actually 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 13. 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 unambiguous join points to which the aspect can bind.

7.2 Language and Tooling – Lessons Learned

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++ [28].

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 (implemented, for instance, in LogicAJ [26]) that can be very beneficial for a further decoupling of aspects and base code [20]. 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.

NonJoin 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.

User Experience – AOP for "Hackers". More than a dozen master students were involved in the development of CiAO and our previous work with aspectizing the PURE and eCos operating system [43,31,29], 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 3 h 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 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.

Another consequence in this direction is that we migrated CiAO's configuration management from the commercial Eclipse-based PURE::VARIANTS variant-management tool [7] to a (less convenient and powerful, but more "hacker-compatible") tool chain based on Linux KCONFIG as front end and a set of PERL scripts for the variant generation process.

It would be interesting to look more systematically into these (in its present form only anecdotical) experiences regarding the longer term use of AOP in systems software. Such an empirical study remains a topic for further research.

8 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 [11], 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 *better evolvability* of these concerns.

C4 [17,38] is an example of 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 [15]; KLASY is used for aspect-based dynamic instrumentation in Linux [47].

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 [31] show that a useful application of AOP to *existing* operating systems often requires additional AOP expressivity that results in run-time overheads (e.g., temporal logic or dynamic instrumentation). This paper shows that such overhead is not inevitable; by the *aspect-aware* design and development "from scratch" it is even possible to outperform existing systems with respect to their CPU and memory footprint..

Even though the Flux OSKit [18] is not aspect-oriented, it shares common goals with CiAO. Both aim at being highly flexible OS construction kits with an overhead-free composition mechanism. Also related is the approach of Event-Based Systems [9]. This is a popular architectural style that lets components interact by generating or receiving *event notifications*. A component can select events, for instance, by the event type, by the event message content, or even by pattern matching. This is very similar to AOP where events are join points and selection is done with the pointcut language. Both approaches lead to *loose coupling*; the advantage of advice-based binding in CiAO is a technical one: As AspectC++ inlines advice code, loose coupling can be implemented very efficiently.

9 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. 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.

Acknowledgments. We would like to thank the anonymous reviewers of this paper for their helpful and detailed feedback! This work was partly supported by the German Research Council (DFG) under grants no: LO 1719/1-1, SFB/TRCRC 89 "Invasive Computing" (subproject C1), SP 968/4-1, SP 968/5-1, and SFB 876 "Providing Information by Resource-Constrained Data Analysis" (subprojects A1 and A4).

References

1. Åberg, R.A., Lawall, J.L., Südholt, M., Muller, G., Le Meur, A.-F.: On the automatic evolution of an OS kernel using temporal logic and AOP. In: Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), pp. 196–204. IEEE Computer Society Press, Montreal (2003)
2. Accetta, M., Baron, R., Golub, D., Rashid, R., Tevanian, A., Young, M.: MACH: A New Kernel Foundation for UNIX Development. In: Proceedings of the USENIX Summer Conference, pp. 93–113. USENIX Association (June 1986)
3. AUTOSAR. Requirements on operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR (June 2006)
4. AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR (June 2006)
5. Baniassad, E., Clarke, S.: Theme: An approach for aspect-oriented analysis and design. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), pp. 158–167. IEEE Computer Society Press, Washington, DC (2004)
6. UC Berkeley. TinyOS homepage, `http://www.tinyos.net`
7. Beuche, D.: Variant management with pure:variants. Technical report, pure-systems GmbH (2006), `http://www.pure-systems.com/fileadmin/downloads/pv-whitepaper-en-04.pdf`(visited November 12, 2011)
8. Beuche, D., Fröhlich, A.A., Meyer, R., Papajewski, H., Schön, F., Schröder-Preikschat, W., Spinczyk, O., Spinczyk, U.: On architecture transparency in operating systems. In: Proceedings of the 9th ACM SIGOPS European Workshop Beyond the PC: New Challenges for the Operating System, pp. 147–152. ACM Press (September 2000)
9. Carzaniga, A., Di Nitto, E., Rosenblum, D.S., Wolf, A.L.: Issues in supporting event-based architectural styles. In: Proceedings of the 3rd International Workshop on Software Architecture (ISAW 1998), pp. 17–20. ACM Press, New York (1998)
10. Clarke, S., Walker, R.J.: Composition patterns: An approach to designing reusable aspects. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), pp. 5–14. IEEE Computer Society Press, Washington, DC (2001)

11. Coady, Y., Kiczales, G.: Back to the future: A retroactive study of aspect evolution in operating system code. In: Akşit, M. (ed.) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp. 50–59. ACM Press, Boston (2003)
12. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Methods, Tools and Applications. Addison-Wesley (May 2000)
13. Al Danial. CLOC – count lines of code homepage, `http://cloc.sourceforge.net/`
14. del Foyo, L.E.L., Mejia-Alvarez, P., de Niz, D.: Predictable interrupt management for real time kernels over conventional PC hardware. In: Proceedings of the 12th IEEE International Symposium on Real-Time and Embedded Technology and Applications (RTAS 2006), pp. 14–23. IEEE Computer Society Press, Los Alamitos (2006)
15. Engel, M., Freisleben, B.: TOSKANA: A Toolkit for Operating System Kernel Aspects. In: Rashid, A., Aksit, M. (eds.) Transactions on AOSD II. LNCS, vol. 4242, pp. 182–226. Springer, Heidelberg (2006)
16. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced SoC (OOPSLA 2000) (October 2000)
17. Fiuczynski, M., Grimm, R., Coady, Y., Walker, D.: Patch(1) considered harmful. In: Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS 2005). USENIX Association (2005)
18. Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., Shivers, O.: The Flux OSKit: A substrate for kernel and language research. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP 1997). ACM SIGOPS Operating Systems Review, pp. 38–51. ACM Press (October 1997)
19. Groher, I., Baumgarth, T.: Aspect-orientation from design to code. In: Proceedings of the 2004 AOSD Early Aspects Workshop, AOSD-EA 2004 (March 2004)
20. Gybels, K., Brichau, J.: Arranging language features for pattern-based crosscuts. In: Akşit, M. (ed.) Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), pp. 60–69. ACM Press, Boston (2003)
21. Infineon Technologies AG, St.-Martin-Str. 53, 81669 München, Germany. TriCore 1 User's Manual (V1.3.5), Volume 1: Core Architecture (February 2005)
22. Kästner, C., Apel, S., Batory, D.: A case study implementing features using AspectJ. In: Proceedings of the 11th Software Product Line Conference (SPLC 2007), pp. 223–232. IEEE Computer Society Press (2007)
23. Kästner, C., Apel, S., Saif ur Rahman, S., Rosenmüller, M., Batory, D., Saake, G.: On the impact of the optional feature problem: Analysis and case studies. In: Muthig, D., McGregor, J.D. (eds.) Proceedings of the 13th Software Product Line Conference (SPLC 2009), Carnegie Mellon University, Pittsburgh (2009)
24. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Aksit, M., Matsuoka, S. (eds.) ECOOP 1997. LNCS, vol. 1241, pp. 220–242. Springer, Heidelberg (1997)
25. Kleiman, S., Eykholt, J.: Interrupts as threads. ACM SIGOPS Operating Systems Review 29(2), 21–26 (1995)
26. Kniesel, G., Rho, T.: A definition, overview and taxonomy of generic aspect languages. L'Objet, Special Issue on Aspect-Oriented Software Development 11(2-3), 9–39 (2006)
27. Liedtke, J.: On μ-kernel construction. In: Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995). ACM Press (December 1995)
28. Lohmann, D., Blaschke, G., Spinczyk, O.: Generic Advice: On the Combination of AOP with Generative Programming in AspectC++. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 55–74. Springer, Heidelberg (2004)

29. Lohmann, D., Hofer, W., Schröder-Preikschat, W., Spinczyk, O.: Aspect-aware operating-system development. In: Chiba, S. (ed.) Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD 2011), pp. 69–80. ACM Press, New York (2011)
30. Lohmann, D., Hofer, W., Schröder-Preikschat, W., Streicher, J., Spinczyk, O.: CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In: Proceedings of the 2009 USENIX Annual Technical Conference, pp. 215–228. USENIX Association, Berkeley (2009)
31. Lohmann, D., Scheler, F., Tartler, R., Spinczyk, O., Schröder-Preikschat, W.: A quantitative analysis of aspects in the eCos kernel. In: Berbers, Y., Zwaenepoel, W. (eds.) Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys 2006), pp. 191–204. ACM Press, New York (2006)
32. Lohmann, D., Streicher, J., Spinczyk, O., Schröder-Preikschat, W.: Interrupt synchronization in the CiAO operating system. In: Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS 2007), ACM Press, New York (2007)
33. Love, R.: Linux Kernel Development, 2nd edn. Novell Press (2005)
34. Neville-Neil, G.V., McKusick, M.K.: The Design and Implementation of the FreeBSD Operating System. Addison-Wesley (2004)
35. Massa, A.: Embedded Software Development with eCos. New Riders (2002)
36. OSEK/VDX Group. Operating system specification 2.2.3. Technical report, OSEK/VDX Group (February 2005), http://portal.osek-vdx.org/files/pdf/specs/os223.pdf (visited August 17, 2011)
37. Parnas, D.L.: Some hypothesis about the "uses" hierarchy for operating systems. Technical report, TH Darmstadt, Fachbereich Informatik (1976)
38. Reynolds, A., Fiuczynski, M.E., Grimm, R.: On the feasibility of an AOSD approach to Linux kernel extensions. In: Proceedings of the 7th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS 2008), pp. 1–7. ACM Press, New York (2008)
39. Rubini, A., Corbet, J.: Linux Device Drivers. O'Reilly (2001)
40. Schröder, W.: Eine Familie von UNIX-ähnlichen Betriebssystemen – Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes beim strukturierten Betriebssystementwurf (December 1986)
41. Schröder-Preikschat, W.: The Logical Design of Parallel Operating Systems. Prentice Hall PTR (1994)
42. Solomon, D.A., Russinovich, M.: Inside Microsoft Windows 2000, 3rd edn. Microsoft Press (2000)
43. Spinczyk, O., Lohmann, D.: Using AOP to develop architecture-neutral operating system components. In: Proceedings of the 11th ACM SIGOPS European Workshop, pp. 188–192. ACM Press, New York (2004)
44. Spinczyk, O., Lohmann, D.: The design and implementation of AspectC++. Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software 20(7), 636–651 (2007)
45. Steimann, F.: Domain Models Are Aspect Free. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 171–185. Springer, Heidelberg (2005)
46. Stein, D., Hanenberg, S., Unland, R.: A UML-based aspect-oriented design notation for AspectJ. In: Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002), pp. 106–112. ACM Press, New York (2002)
47. Yanagisawa, Y., Kourai, K., Chiba, S., Ishikawa, R.: A dynamic aspect-oriented system for OS kernels. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE 2006), pp. 69–78. ACM Press, New York (2006)