Configurable Memory Protection by Aspects[*]

Daniel Lohmann, Jochen Streicher, Wanja Hofer,
Olaf Spinczyk, and Wolfgang Schröder-Preikschat

Friedrich-Alexander University Erlangen-Nuremberg
Department of Computer Science 4
{lohmann,streicher,hofer,spinczyk,wosch}@cs.fau.de

ABSTRACT

We describe the implementation of memory protection by means of *aspect-oriented programming (AOP)* in CiAO, an AUTOSAR-like family of embedded operating systems. The use of AOP was originally motivated by the fact that memory protection is a cross-cutting policy, which, furthermore, has to be configurable at build-time in AUTOSAR. We learned, however, that besides switching between full protection and no protection, an AOP-based approach also makes it easy to apply completely different models of protection. For the domain of statically configured embedded systems, where certain failure scenarios can often be excluded by means of code analysis or even probability, this facilitates tailored and light-weight "pay-as-you-use" protection strategies.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Experimentation, Design

Keywords

Aspect-Oriented Programming (AOP), AspectC++, CiAO, Configurability, Aspect-Aware Operating System, Memory Protection

1. INTRODUCTION

Software and electronics have become the driving factor for innovation in the automotive industry. Today, a typical premium class car (e.g., BMW 7, Audi A8) already offers more than 2,000 software-based features, implemented by 10,000,000 lines of code. On the hardware side, this software is run by more than 70 electronical control units (ECUs), which are interconnected by up to 5 different bus systems [5]. The rapidly increasing number of ECUs and bus systems leads to significant problems with respect to wiring, complexity, and scalability. Thus, car manufacturers are striving towards a consolidation of μ-controllers; they want to switch from the high number of individual ECUs (mostly 8- and 16-bit) to a small number of more powerful 32-bit ECUs running multiple applications.

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

However, the parallel execution of multiple applications from different vendors on the same hardware requires a new generation of automotive system software. With respect to safety and liability issues, effective mechanisms for fault isolation and detection, especially memory protection (MP), have to be enforced.

These requirements are reflected in a new OS specification adopted by AUTOSAR [1], a consortium founded by all major players in the automotive industry in order to specify a new system software standard for car applications. The AUTOSAR OS specification [4] defines four *scalability classes*, which (among other things) specify different levels of MP. Whereas scalability classes 1 and 2 offer no protection of the OS core and application domains, MP has to be enforced in scalability classes 3 and 4. Thus, to be competitive on all market segments, OS vendors have to understand and implement MP as a *configurable* and *optional* feature.

On the implementation side, MP is a highly cross-cutting feature. It affects all system calls, the dispatcher, the access to system resources, and so on – a situation that is especially problematic if MP shall be configurable. System calls, for instance, have to be invoked by some trap mechanism if protection applies; if, however, protection is not an issue, system calls can be provided as a simple linker library or even be inlined into the application code.

About this Paper

We present and discuss an implementation approach for MP by means of *aspect-oriented programming* (*AOP*), a programming paradigm that provides specific language support to deal with cross-cutting concerns by the notion of *aspects*.

The paper is organized as follows: Sec. 2 starts with a short analysis of MP in the domain of embedded systems and its issues. Sec. 3 describes our AOP-based implementation in CiAO; first results are discussed in Sec. 4. Finally, Sec. 5 gives an overview of related work and the paper is concluded in Sec. 6.

2. ANALYSIS

In most embedded systems, all code and data sections are allocated statically and mapped to physical addresses at link-time. On the hardware side, MP is implemented by a simple *memory protection unit* (*MPU*). An MPU offers a simple segmentation of the physical address space and is relatively cheap to implement; only a few memory range registers to specify size, location, and access privileges (read/write/execute) to one or more memory regions are required. This is different from page-based virtual memory implemented by most general-purpose operating systems (such as Windows or Linux), which requires a full-featured MMU. However, in order to switch between the different trust levels in a controlled manner, the MPU registers are programmable only if the CPU is running in supervisor mode. Hence, like in a general-purpose operating system, a trap mechanism has to be used to switch to supervisor mode when entering the kernel.

In the software model of most embedded OSes that provide protection, communication between protection domains is typically restricted to message passing only. This is comparable to most μ-kernels, where IPCs are the only provided abstraction for process interaction.

In short: MP in embedded operating systems is, overall, implemented by the same concepts as are used in systems from the domains of “big” computing.

2.1 Properties and Issues of Memory Protection in Embedded Systems

However, the development constraints of the typical ECU differ a lot from the domain of desktop computing:

Hardware costs. ECUs are produced and sold in very large quantities. A strict tailoring of the per-unit hardware costs is crucial; a few cents can decide over market success and failure and especially SRAM is relatively expensive. System software for this domain has offer an excellent “pay-as-you-use” scalability by means of static configuration and tailoring.

Safety, not security. MP is motivated by *safety concerns*, not by *security issues*. The software installed on an ECU is generally considered as trustworthy – but not as bug-free, though.

Fault class constraints. Fault situations that impair dependability are inherently limited. Code, for instance, is immutable at run-time if placed into ROM/flash memory sections. Certain classes of faults (e.g., those resulting from pointer arithmetic) can sometimes be excluded by offline analysis of the source/binary code or by an official certification of the software with respect to safety standards such as MISRA C [11].

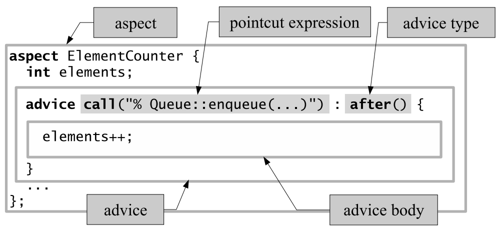
Applying the traditional MP paradigms to the embedded systems domain in an unabrigded form bears several problems, especially with respect to tailoring and, thus, hardware costs.

System calls. The common implementation through traps and a system service table effectively results in a late binding of system services; this inhibits the elimination of unused code by the linker. Furthermore, services cannot be inlined, which disables another important source of cost optimization.

Message-based interaction. Even though conceptually considered advantageous and well understood, message-based IPC between protection domains induces costs and complexity. The necessity for extra server threads increases the number of system objects and invocations of the dispatcher. The inherent synchronization of message-based IPC impacts determinism and worst case execution time (WCET) calculations.

2.2 Summary

Existing MP mechanisms neither support the specific constraints of embedded systems nor exploit the available *a-priory* knowledge about the applications. Potential for cost reduction is given away by understanding fault isolation by MP as a “hundred percent solution”. Whereas “hundred percent” is certainly desirable for ECUs with functionality relevant to human safety, such as the ABS in a car, most ECUs actually implement (dispensable) comfort functions. For these systems, there is a trade-off between the user-perceivable dependability gain and the hardware cost increase. The handling of just the most probable fault situations (e.g., stack overflows) might already increase dependability by an order of magnitude with



```
aspect ElementCounter {
  int elements;
  advice call("% Queue::enqueue(...)") : after() {
    elements++;
  }
  ...
};
```

Figure 1: Syntactical Elements of an Aspect in AspectC++

minimal extra costs. Thus, protection mechanisms should scale with respect to fault class tolerance and be tailorable to the actual application requirements.

3. MEMORY PROTECTION IN CIAO

3.1 The CiAO Embedded OS

In the CiAO project (CiAO is Aspect-Oriented), our group has been developing a family of *aspect-aware* operating systems for embedded applications. The system is aspect-aware in the sense that it has been developed with the idea of configurability by aspects from the very beginning. The goal is to come up with a system design that provides means to configure even fundamental and highly cross-cutting OS policies, like synchronization and protection strategies. In a previous paper [13], we demonstrated the approach on interrupt synchronization; the focus of this paper is memory protection.

Primary development platform for CiAO is the Infineon TriCore, an architecture of 32-bit μ-controllers mostly used in the automotive industry that also serves as a reference platform for AUTOSAR.

3.2 AspectC++

The implementation language for CiAO is AspectC++ [15], an AOP language extension for C++. The AspectC++ weaver `ac++` transforms AspectC++ code into ISO C++ code, which can then be compiled by any standard-compliant C++ compiler.

The most relevant AspectC++ language concepts are *join-points* and *advice*. An *advice* definition describes a transformation to be performed at specific positions either in the static program structure (*static cross-cutting*) or in the run-time control flow (*dynamic cross-cutting*) of a target program. A *join-point* denotes such a specific position in the target program. Advice is given by *aspects* to sets of join-points called *pointcuts*. Pointcuts are defined declaratively in a *join-point description language*. The sentences of the join-point description language are called *pointcut expressions*. An *aspect* encapsulates a cross-cutting concern and is otherwise very similar to a class. Besides advice definitions, it may contain class-like elements such as methods or state variables.

Fig. 1 illustrates the syntax of aspects written in AspectC++. The aspect increments the member variable `elements` *after* each *call* of the function `Queue::enqueue()`. In AspectC++, pointcut expressions are built from *match expressions* and *pointcut functions*. Match expressions are already primitive pointcut expressions and yield a set of *name join-points* which represent elements of the static program structure such as classes or functions. Technically, match expressions are given as quoted strings that are evaluated against the identifiers of a C++ program. The expression `"% Queue::enqueue(...)"`, for instance, returns a name pointcut containing every (member) function of the class `Queue` that is called `enqueue`. *Code join-points*, on the other hand, represent events in the dynamic control flow of a program, such as the execution of a function. Code pointcuts are retrieved by feeding name pointcuts into certain pointcut functions such as `call()` or `execution()`. The pointcut expression `call("% Queue::enqueue(...)")`, for

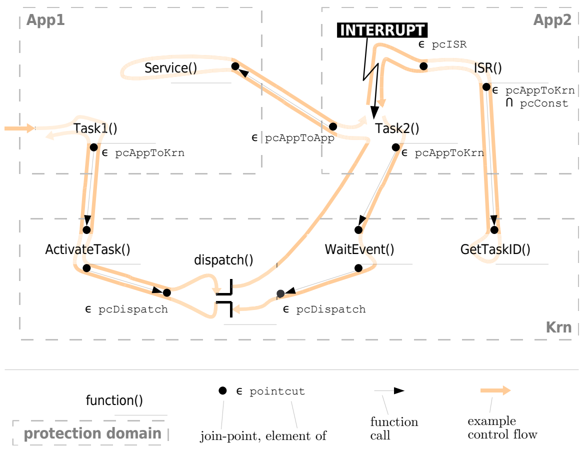


Figure 2: Control Flows and Protection Domains

instance, yields all events in the dynamic control flow where a function `Queue::enqueue()` is about to be called.

As pointcuts are described declaratively, usually the target code itself does not have to be prepared or instrumented to be affected by aspects. Furthermore, the same aspect can affect various and even unforeseen parts of the target code.

3.3 Design

3.3.1 *Basic Design Decisions*

Two design decisions are fundamental to our concept of MP:

Protect the data, not the code. CiAO does not prevent conceptually protected code to be executed, it rather prevents the protected data from being modified. Execution of disallowed code is only trapped when that code modifies protected data. This way, a control flow can only inadvertently modify its *own* data structures, without affecting the other protected domains. This ensures *safety*, not *security* (see Sec. 2.1).

Control flows and protection domains are orthogonal. We clearly separate between the protection domains and the control flows present in a system. A control flow (e.g., a task or an interrupt service routine (ISR)) has a defined initial protection domain, but it can transit into other domains when calling an exported function from a foreign application, for instance. When a control flow is switched to another protection domain, the remaining stack can be used in the new domain; that is, at the time of the transition, the current stack pointer is used as top of stack.

3.3.2 *Protection Models*

Depending on the configuration, MP is applied in different degrees:

No protection does not feature any protection mechanism at all; OS functions are regularly linked to the application and even inlined if applicable.

Kernel protection defines one protection domain for the kernel and one for all applications.

Application protection additionally separates the different applications from each other; each application constitutes an own protection domain. Domain barriers have to be crossed on the invocation of an exported function of another application.

Task protection even protects the task-local data (i.e., its stack and possibly thread-local storage) from modification of other control flows in the same application. Due to space limitations, this model is not further pursued in this paper.

3.3.3 *Join-Points*

The points in the control flow of a CiAO system that are possibly affected by a MP domain switch can be understood as AOP join-points. These join-points can be influenced by advice given by different aspects that, depending on the configured protection model, do or do not switch protection domains in one or the other way.

Fig. 2 depicts an example control flow in a scenario with two distinct application domains `App1` and `App2`, and the kernel domain `Krn`. Domain transitions occur when `Task1` calls the kernel primitive `ActivateTask()`, for instance, or when `Task2` invokes the function `Service()` exported by `App1`. These join-points can be matched by the generic pointcut expressions assigned to `pcAppToKrn` and `pcAppToApp`, respectively:

```
pointcut pcApp1() = "% App1::...::%(...)";
pointcut pcApp2() = "% App2::...::%(...)";
pointcut pcKrn() = "% Krn::...::%(...)";
pointcut pcConst() = "% ...::%(...) const";
pointcut pcAppToApp() =
    (call(pcApp1()) && !within(pcApp1()) ||
    call(pcApp2()) && !within(pcApp2())) &&
    !call(pcConst());
pointcut pcAppToKrn() =
    call(pcKrn()) && !within(pcKrn()) &&
    !call(pcConst());
```

As a further optimization, the domain transition pointcut expressions exclude calls to functions declared as `const` in C++. Since security is not relevant in our context, all of the memory is readable by all control flows. Hence, a protection domain switch is only necessary for state-changing functions.

The second location where a domain switch must be considered is the dispatching to another control flow; the newly activated control flow might belong to a different protection domain than the one that was interrupted or terminated. These points in the control flow of the system are represented by the pointcut `pcDispatch`:

```
pointcut pcDispatch() = execution
        ("% Krn::dispatch(Task current, Task next)");
```

3.3.4 *Basic Operations*

The aspects that give advice to these join-points make use of the following conceptual operations, which are provided by CiAO's MP system:

enterKernel() enters the kernel's protection domain. It either disables the MP hardware or configures it for full memory access. If the CPU supports privilege modes, the supervisor mode is entered and privileged instructions may be executed; that is, in particular instructions that reconfigure the MP hardware are now available. `switchApplication()` can only be called from within the kernel's protection domain.

leaveKernel() leaves the kernel's protection domain and enters the domain of the currently active application. It leaves the supervisor mode and configures the MP hardware for memory write access restrictions according to the currently active control flow and application, if that degree of protection is demanded.

switchApplication() changes the currently active application and updates the MP system for the activated application's domain, which is then entered upon the invocation of `leaveKernel()`.

3.3.5 *Aspects*

Depending on the configured protection model, different aspects are deployed to give advice to the previously described pointcuts.

If only the kernel is to be separated from the applications, the advice code is straightforward when making use of the basic operations described above. Upon the dispatching of a new task, the kernel domain is left for the first time:

```
aspect KernelProtection {
  advice pcAppToKrn() : before() {
    enterKernel();
  }
  advice pcAppToKrn() : after() {
    leaveKernel();
  }
  advice pcDispatch() : after() {
    if (tjp->arg<1>()->firstRun_)
      leaveKernel();
  }
};
```

If application protection is desired, cross-application calls are also advised to switch the protection domain. Dispatches that cause a change of application are also targeted:

```
aspect ApplicationProtection {
  advice pcAppToApp() : before() {
    enterKernel();
    switchApplication(JoinPoint::That::AppId)
    leaveKernel();
  }
  advice pcAppToApp() : after() {
    enterKernel();
    switchApplication(JoinPoint::Target::AppId)
    leaveKernel();
  }
 advice pcDispatch() : before() {
    if (   tjp->arg<0>()->owningApp_
        != tjp->arg<1>()->owningApp_)
      switchApplication(tjp->arg<1>()->owningApp_);
  }
};
```

3.4 Implementation Variants

Our implementation targets the Infineon TriCore platform (TC1796b). It currently offers two variants that implement the actual MP switch:

Standard Variant. This configuration represents a "traditional" implementation. Protection domains are write-protected from each other; system calls trap into the kernel to get the CPU privilege level required to reprogram the MPU.

Semi-Trusted Variant. An interesting peculiarity of the TriCore is that MP is not implicitly disabled when the processor runs in supervisor mode; the supervisor mode just *permits* to reprogram the MPU. This variant exploits this peculiarity by running even the application code in supervisor mode. Thereby, only the MPU has to be reprogrammed by the OS in case of a domain switch; no costly traps into the kernel are required. As a consequence, short system services can be inlined, and unused services are automatically eliminated by the linker by means of dead code elimination (see Sec. 2.1).

The *standard variant* is probably more expensive but also offers more safety. In case of the *semi-trusted variant*, the application code may theoretically reprogram the MPU by itself, which would compromise safety. On the TriCore, however, it is relatively easy to perform an offline analysis which proves that application code does *not* reprogram the MPU, as this requires dedicated instructions. The Harvard architecture effectively prevents code modifications at runtime. Furthermore, the TriCore uses an extra (protectable) stack to manage call frames that is independent from the stack used by local variables; hence, an unsafe jump caused by an accidental modification of the return address is not possible. Only function pointers bear a categorical danger; an accidentally overwritten function pointer may result in a situation where the application directly jumps into parts of the kernel that modify protection registers. Hence, this variant is not a "hundred percent" solution.

protection	GetTaskID()	ActivateTask()	dispatch()	Service()
none	3	24	88	-
semi-trusted	3	43	148	89
standard	3	86	148	174

Table 1: Number of CPU Cycles for Characteristic Functions and Protection Implementation Variants

3.5 Preliminary Evaluation

We measured the clock cycles needed to perform characteristic functions with both application and kernel protection enabled. Tab. 1 shows first results[1] of the two protection implementation variants (*standard*, *semi-trusted*), and the basic overhead of the respective function with protection disabled (*none*). `GetTaskID()` does not induce any overhead in any implementation since it is a read-only kernel function (declared `const`) and therefore not affected by protection advice. `ActivateTask()` is a service that modifies kernel structures and, thus, is protected. The semi-trusted variant, which only temporarily disables MP, costs 43–24=19 extra clock cycles for the `enterKernel()` / `leaveKernel()` round-trip, while the overhead of a full kernel trap is three times as much with 86–24=62 cycles.

The `dispatch()` operation is internal to the kernel and performs a `switchApplication()` in both protected variants. A protection domain switch from one application to another costs 148–88=60 clock cycles. The cross-application call `Service()` in the semi-trusted variant has to invoke `switchApplication()` twice, which costs 89 CPU cycles. The standard variant additionally traps into the kernel for each of the protection modifications, which is worth 174 cycles. A configuration without application protection (not shown here) would let the `dispatch()` and the `Service()` functions unaffected, and, hence, would not induce this overhead.

4. DISCUSSION

MP is an inherently cross-cutting strategy of an OS kernel. Its actual implementation affects all cross-domain transitions of control flows. AOP, with its model of *dynamic cross-cutting* and *advice* (see Sec. 3.2), seems to be a natural choice for this kind of problems. In the following, we discuss our approach.

4.1 Applying Memory Protection by AOP

A big advantage of the AOP-based solution is **configurability** by a clear separation of *what* and *where*. The implementation of MP by advice defines only the *what*, whereas the *where* is described by the declarative pointcut mechanism. Thereby, it becomes very easy to adapt either dimension; the protection strategy and its level of granularity can be configured independently of each other. A time-critical but formally certified application could, for example, be considered as fully trusted and be excluded from protection by merely modifying the `pcAppToKrn` and `pcAppToApp` pointcuts.

[1]TC1796b@50MHz, internal no-wait-state RAM, `tricore-gcc` 3.4.2 (`-O1`). Measurements performed and averaged over 10 iterations with a hardware trace analyzer (Lauterbach).

The AOP-based CiAO approach makes it easy to experiment with different protection models.

A potential issue might be that all components have to be available as **source code.** This is required by the source-to-source transformation approach of the AspectC++ weaver and the CiAO approach to understand aspects as part of the OS interface. In the domain of embedded systems, components *are* usually provided as source code. If, however, the source code is not available, a potential solution may be to weave directly into binary code [8, 7]. This remains a topic for further research.

4.2 Light-Weight Memory Protection

Configurability (and, thereby, tailorability) is of particular importance in the domain of **embedded systems**. For this very cost-sensitive domain, MP should not be understood as a one-fits-all property; there is always a trade-off between better dependability and cost increase.

On the other side, embedded systems integrators usually have more knowledge about the actual application code than in general-purpose computing. The illegal use of privileged instructions, for instance, can be ruled out by means of static code analysis. Specific properties of the hardware (such as the dedicated call stack of the TriCore) might rule out other potential run-time faults. Dangerous code constructions, such as the use of function pointers, are forbidden by coding standards like MISRA C. Our semi-trusted variant exploits this extra knowledge to provide a light-weight MP mechanism that still isolates the effects of the most typical errors such as stack overflows and invalid pointers. Of course, one can argue if one should really go for "semi-trusted" MP. In our opinion, this decision should depend only on the specific constraints of the actual ECU and the applications to be deployed. The AOP approach just offers the option to choose.

4.3 Hardware- Versus Software-Based MP

A lot of work has been conducted in the field of MP in operating systems by restricting developers to type-safe languages such as Java [10, 16] or Sing# [3]. AIKEN and colleagues could furthermore show that constructive (language- and compiler-based) MP is generally more efficient than hardware-based approaches [3]. The reason is that MMUs (and MPUs) are well optimized for the standard case (legal memory access), but induce a significant overhead in the cases of detecting and handling the relatively infrequent access violations and domain transitions.

The semi-trusted variant of CiAO's MP can be seen as a **mixture of hardware- and software-based protection**. Control of memory write operations (and thus the isolation of pointer problems in C/C++ code) is still ensured by the MPU. The protection of the MPU itself, however, is ensured constructively. Only the aspects that handle the domain transitions are permitted to use MPU-modifying operations. Thereby, the responsibility for exactly those parts that induce an overhead in hardware-based MP – detection and handling of domain transitions – has been moved to the software level.

5. OTHER RELATED WORK

We are not aware of any existing work that addresses the issue of configurable MP by means of AOP. Several projects have applied AOP techniques to improve the implementation of other concerns in Linux [9, 2], FreeBSD [6], and NetBSD [8], as well as in the embedded OS kernels eCos [12] and PURE [14]. In these projects, AOP was applied as an *ex post* mechanism to *existing* kernels. The CiAO idea, in contrast, is to use aspects as a fundamental concept for kernel design from the very beginning.

6. SUMMARY AND CONCLUSIONS

Having been a crucial feature in general-purpose operating systems for 30 years, fault protection and isolation concepts now become more and more important in the domain of embedded systems. ECU consolidation in the automotive industry is a major driver for this trend; the new AUTOSAR OS software standard explicitly requires memory protection.

However, in the domain of embedded systems, memory protection has to be understood and provided as a statically configurable and tailorable policy. Aspect-oriented programming provides the necessary means to reach this configurability in the implementation. By a combination of hardware and software techniques, light-weight application-tailored protection concepts can be achieved.

7. REFERENCES

[1] AUTOSAR homepage. http://www.autosar.org/.

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

[3] AIKEN, M., FÄHNDRICH, M., HAWBLITZEL, C., HUNT, G., AND LARUS, J. Deconstructing process isolation. In *MSPC '06: Workshop on memory system performance and correctness* (2006), ACM, pp. 1–10.

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

[5] BROY, M. Challenges in automotive software engineering. In *ICSE '06* (2006), ACM, pp. 33–42.

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

[7] DEVILLECHAISE, M., MENAUD, J., MULLER, G., AND LAWALL, J. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In *AOSD '03* (Mar. 2003), ACM, pp. 110–119.

[8] ENGEL, M., AND FREISLEBEN, B. TOSKANA: a toolkit for operating system kernel aspects. In *Transactions on AOSD II* (2006), no. 4242, Springer, pp. 182–226.

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

[10] GOLM, M., FELSER, M., WAWERSICH, C., AND KLEINÖDER, J. The JX operating system. In *USENIX '02* (Berkeley, CA, USA, June 2002), pp. 45–58.

[11] JESTY, P. H., HOBLEY, M., EVANS, R., AND KENDALL, I. Safety analysis of vehicle-based systems. In *8th Safety-Critical Systems Symposium (SCSS '00)* (2000), Springer, pp. 90–110.

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

[13] LOHMANN, D., STREICHER, J., SPINCZYK, O., AND SCHRÖDER-PREIKSCHAT, W. Interrupt synchronization in the CiAO operating system. In *AOSD-ACP4IS '07* (2007), ACM, p. 6.

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

[15] SPINCZYK, O., AND LOHMANN, D. The design and implementation of AspectC++. In *Knowledge-Based Systems, Special Issue on Creative Software Design* (2007), Elsevier. (to appear).

[16] STILKERICH, M., WAWERSICH, C., SCHRÖDER-PREIKSCHAT, W., GAL, A., AND FRANZ, M. OSEK/VDX API for Java. In *PLOS '06* (San Jose, California, USA, Oct. 2006), ACM, pp. 13–17.