On the Configuration of Non-Functional Properties in Operating System Product Lines*

Daniel Lohmann, Olaf Spinczyk, Wolfgang Schröder-Preikschat
{dl,os,wosch}@cs.fau.de
Friedrich-Alexander-University Erlangen-Nuremberg, Germany
Department of Computer Science 4

ABSTRACT

Reaching a good separation, maintainability and configurability of non-functional concerns like performance, timeliness or dependability is a frequently expressed but still unrealisable hope of using AOP technology. Non-functional properties have the tendency to be emergent, that is, they have no concrete representation in the code, but appear through the complex interactions between software components in the whole. This makes it very hard, if not impossible at all, to express them in a configurable manner by objects or even aspects. The architecture of a software system, however, is known to have a high impact on some non-functional properties. Thus, it may be possible to reach configurability of non-functional software properties by the means of reconfigurable software architectures. This paper discusses the connection between non-functional and architectural properties in the domain of operating system product lines.

1. INTRODUCTION

Reaching better separation of concerns in software systems was and is the driving factor for the development of AOP technologies. One of the big hopes associated with the application of AOP is to get a clearly modularized implementation of even so called *non-functional concerns*. The non-functional properties of a software system are those properties that do not describe or influence the principal task / functionality of the software, but can be observed by end users in its runtime behaviour. *Performance* or *resource utilization* are the most common examples for non-functional properties, but also less observable properties like *robustness* or *dependability* are important members of the class.[1] Even if non-functional properties have no impact on the primary functionality of the software system, they have a big impact on its applicability in the real world. A system that provides perfect functionality, but works terribly slow, is just unusable. Non-functional properties are therefore important concerns, their controllability may be crucial for the success of a software project.

This is especially true in the domain of embedded systems, where hardware cost pressure leads to strictly limited resources in terms of CPU and memory. Under such circumstances, non-functional concerns like memory usage or timeliness may even dominate the functional properties of the final product. As a consequence, there is an ongoing tendency to develop operating systems for embedded devices as tailorable *product lines* that provide a more or less fine-grained selectability of functional features. This facilitates, by leaving out optional functions, some optimization of the system for specific memory constraints. However, existing operating system product lines provide only very limited configurability with respect to other non-functional properties like timeliness, protection or robustness.

1.1 Problem Analysis

The problem with most non-functional properties is that they are *emergent properties*. They are neither visible in the code nor structure of single components, but "suddenly" emerge from the orchestration of many components into a complete system. Properties that manifest in the integrated system only are indeed crosscutting, as they *result* from certain (unknown) characteristics of every single component. Due to their inherent emergence it is, however, not possible to tackle them by decomposition techniques like AOP. They need to be understood holistically, that is, on the global scope of software development. One could say they need to be addressed by *"holistic aspects"*, meaning that the realization of non-functional concerns does not crosscut (just) the code, but the whole *process* of software development.

1.2 The Role of Architecture

The architecture of a software system is known to have a high impact on many non-functional properties. Architecture can well be understood as a *"holistic aspect"*, as it encompasses the set of fundamental design decisions made at the early stages of the software development process. Many architectural decisions are actually driven by non-functional requirements, based on *experience* regarding their effect on such properties. In the operating systems domain, for instance, it is *known* that the isolation of every system components into an own address space (as in μ-kernel OS) has positive effects on safety and fault protection, while a monolithic kernel structure is *known* to lead to lower demands on system resources. From the functional viewpoint (of an application) there is, however, no real difference between a μ-kernel OS and a monolithic kernel OS[7]. Hence, the architecture of the operating system is itself an all-embracing non-functional property[10].

Architecture is usually seen as being something fixed. Most architectural decisions cover large parts of the code, which makes it very expensive to change them later. However, as they crosscut the code they are potentially addressable by aspects. Thus, it should be possible to implement architecture in a *configurable* way and thereby leverage towards an *indirect* configuration of emergent properties.

*This work was partly supported by the German Research Council (DFG) under grant no. SCHR 603/4

[1]This definition of "non-functional" is intentionally from the perspective of an end user. Other stakeholders (e.g. developers or salesmen) would probably define very different properties like *portability* or *unique selling points* as "non-functional".

In the CiAO project[11] we are currently working on the development of an operating system family that provides configurability of certain architectural properties.

1.3 Structure of the Paper

The rest of the paper is organized as follows: In the next section, we discuss the influence of architectural concerns to non-functional properties in the domain of operating systems. Section 3 describes our approach towards the design of architecture-neutral OS components, which is also explained by an example in section 4. Finally, conclusions are drawn and the paper is briefly summarized.

2. CONCERNS OF OS ARCHITECTURE

To reach configurability of architectural properties in operating system product lines, it is necessary to have *architecture-transparent* OS components, that is, components which are designed and implemented to be independent from the actual architecture to use. It is essential to clearly separate the functional component code from those elements that reflect architectural decisions. The following lists some of the more important properties that "make" the architecture of an operating system kernel[10], together with the emergent properties they are known to influence. The focus is on embedded systems:

synchronization If the kernel supports concurrent/parallel execution of control flows, concurrent data access must not lead to race conditions. Synchronized access to data may be implemented by waiting-free algorithms, special hardware support (e.g. atomic CPU operations), interrupt locks or higher-order synchronization protocols. Locks may be allocated on a coarse-grained or fine-grained base. The chosen kernel synchronization strategy has a noticeable impact on **latency**, **timeliness** and **performance**.

isolation The different components of an operating system may have access to the whole system state or to well-isolated subsets only. Components may be isolated by design through type-safe programming languages, by hardware support (segmentation or address spaces via memory management units (MMUs) or translation lookaside buffers (TLBs)) or even by distributing them across hardware boundaries. Isolation may cause additional requirements on data alignment, sharing and interaction. The chosen isolation strategy has a noticeable impact on **memory usage**, **safety**, and **performance**.

interaction System services may be invoked and interact with each other by plain procedure calls, local message passing, inter-process calls (IPCs) or remote procedure calls (RPCs). Interaction may imply implicit synchronization, data duplication or (in the case of RPCs) even fail on occasion. The chosen interaction strategy often goes in line with isolation. It has a noticeable impact on **latency**, **memory usage** and **performance**.

The above listed properties are fundamental building blocks of any operating system architecture[9]. In our research activities on applying AOP principles to the PURE operating system family[17, 13], we had the experience that it is not possible to implement an *ex post* configurability of such fundamental properties. The reason is that most architectural properties do not only lead to characteristic code patterns in the component code (which are addressable by aspects), but also to a number of *implicit constraints* that are not visible in the code. The developer of an implementation without isolation, for instance, implicitly relies on the possibility to pass complex data structures by simple untyped references. An implementation that uses message-based interaction implicitly relies at some places on the serialization of inter-component invocations. However, it is nearly impossible to detect which parts of the code implicitly rely on which constraints. An automatic transformation of such component code to another architecture, e.g. by aspects, is not feasible.

The integration of symetric multiprocessing (SMP) support into Linux is an impressive example for the enormous impact of an architectural property (kernel synchronization) to a non-functional property (performance). It is also a good example for the high costs of architectural transformations in legacy code: The first kernel release that supported SMP hardware was version 2.0. As most components still relied on the coarse-grained kernel synchronization scheme of earlier versions, it performed badly in SMP environments. To improve the performance property, a switch towards a fine-grained synchronization strategy was unavoidable. Hundreds of device drivers, file systems, and other components of the system had to be adapted[2]. Now the 2.6 kernel has fine-grained locking in almost all parts of the system and performs quite well, but the process took several years to complete.

3. THE CIAO APPROACH

Our conclusion from the experiences with PURE is that an operating system has to be designed *specifically* for architectural transparency. In the CiAO project we are now developing a new family of operating systems that fulfills the requirements for architectural configurability. This is a challenging task, as one needs to become aware of all the explicit and implicit elements that are induced by an architectural property. As it is not possible to build software without *any* architecture, a set of abstractions is needed that generalizes over the concrete property implementation. These abstractions are then used by components and later transformed, by aspects, into their architecture-specific representation.

The possible different implementations of architectural properties highly influence each other. Method calls are, for example, a suitable abstraction for the interaction concern. However, to be able to transform them by aspects into a message-based communication scheme, it is necessary to have a clear distinction between inter- and intra-component invocations. This can be realized by naming conventions. Moreover, if isolation enforces message-passing to other address spaces, untyped references must not be used as arguments, as they are not resolvable for transportation into another address space. Message-based interaction is, however, synchronized by design, while interaction by method calls is not. As a consequence, critical sections in the component code always have to be marked explicitly.

This implementation interdependence advises a bottom up design process. *Domain analysis* is the first step in finding suitable abstractions for a specific architectural property. Domain analysis encompasses a detailed analysis of the property implementations in all architectures to support, e.g. by taking a close look at existing operating systems. The result of domain analysis is a set of commonalities and differences between the architecture-specific implementations, represented as *feature diagrams*[4]. The commonalities are the anchor for developing the abstractions of the architecture-neutral model during the design phase. Differences

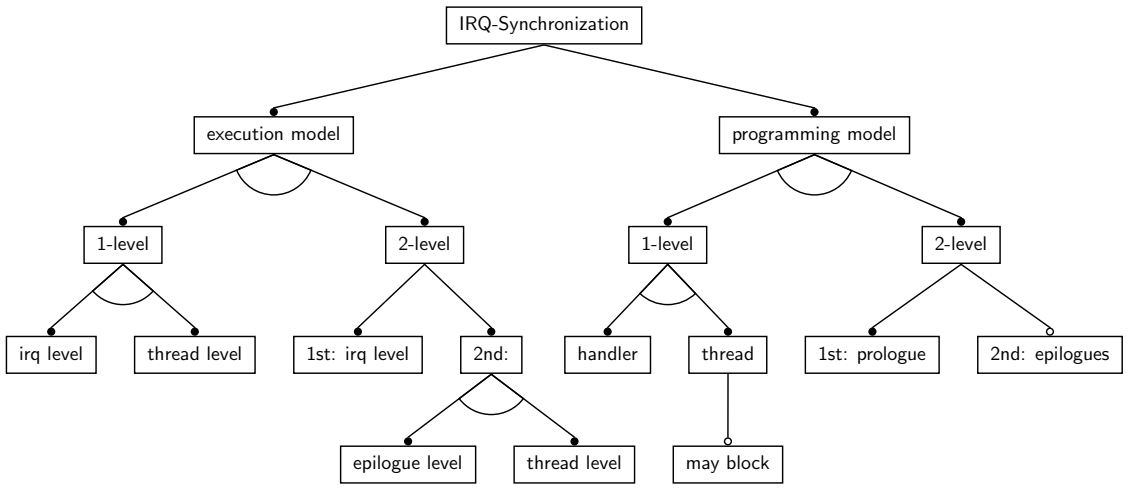

Figure 1: Feature diagram of the *IRQ Synchronization* domain

are integrated into the model step-by-step by generalization, if possible, or are separated out into architecture-specific models. The architecture-neutral model finally can be used as a reference architecture for the architecture-independent implementation of functional components.

4. EXAMPLE: INTERRUPT SYNCHRONIZATION

Most operationg system kernels support two different notions of *control flows*. Continuing, long-running control flows are typically supported by a *thread abstraction*. Control flows to perform short-term reactions on (non-deterministic) external hardware events are implemented by *interrupt handlers*. From the perspective of the operating system, a thread can be understood as walking top down through the kernel, while the control flow of interrupt handlers goes bottom up through the kernel. If, by any chance, interrupts and threads can meet on their execution paths (e.g. by accessing some common state), the kernel needs to ensure synchronized access to this state. The strategy provided for this purpose is usually referred to as *interrupt synchronization.* Interrupt synchronization is an important part of the kernel synchronization concern discussed above.

4.1 Domain Analysis

Analyzed Systems

The following systems were analyzed: Linux, Windows (NT/2000/XP), Solaris, PURE[1] and L4Ka[8]. For systems with SMP support the single-CPU case was analyzed.

General Observations

The execution of an interrupt control flow is initiated by hardware. In case of an IRQ signal, the CPU interrupts the current executing control flow and branches into an IRQ handler function. The IRQ handler function must not block, as this might freeze the system. If an IRQ handler needs to access some resource which is currently in use by some thread (or some other IRQ handler), it cannot wait for the resource to be released. Therefore, every OS needs some mechanism to *delay* the execution of the interrupt code, or at least of those parts accessing the ressource, until the resource is available.

Delay Mechanisms

The most simple way to enforce delayed execution is using *hard synchronization*, which, however may result in high latency and lost interrupts. For this reason, most operating systems follow a more sophisticated approach and implement the delayed execution by some *software mechanism*. The following describes the approaches used by the analyzed systems:

hard synchronization This approach is, because of its simplicity, often used on small μ-controller OS that execute only very few tasks. The idea is to delay the propagation of the interrupt signal by disabling the IRQ line. Most interrupt controllers are able to hold a signaled but disabled interrupt until the IRQ line is reenabled again. However, if interrupts are disabled too long or too often, latency goes up and IRQ signals might be lost.

prologue/epilogues This approach is used by Linux [2, 14], Windows[15], PURE[13] and many other operating systems. The general idea is to explicitly divide the code to be executed in case of an interrupt into a critical and an uncritical part. The critical part, called *prologue*, is executed with low latency at interrupt level. It should perform only the most time-critical tasks and may only access resources that are protected at interrupt level. Before termination, the prologue may request the delayed execution of the part which is not time-critical by registering one or more *epilogues*[2]. Epilogues are queued until the kernel propagates them for execution, which is (typically) the case after all nested interrupt handlers have terminated and before the scheduler is activated. Epilogues thereby have priority over threads, but are interruptable by prologues if new IRQ signals come in. Threads inside the kernel can temporary disable the propagation of epilogues to access shared resources. In this case, epilogue propagation is delayed until the thread finishes its access.

driver threads This approach is common for μ-kernel OS like L4Ka[8]. The general idea is to lift all code to be executed in case of an interrupt up to the thread level. The kernel itself contains only a generic interrupt handler, which sends

[2]Epilogues correspond to *bottom halves* or *tasklets* on Linux and to *deferred procedure calls (DPCs)* on Windows.

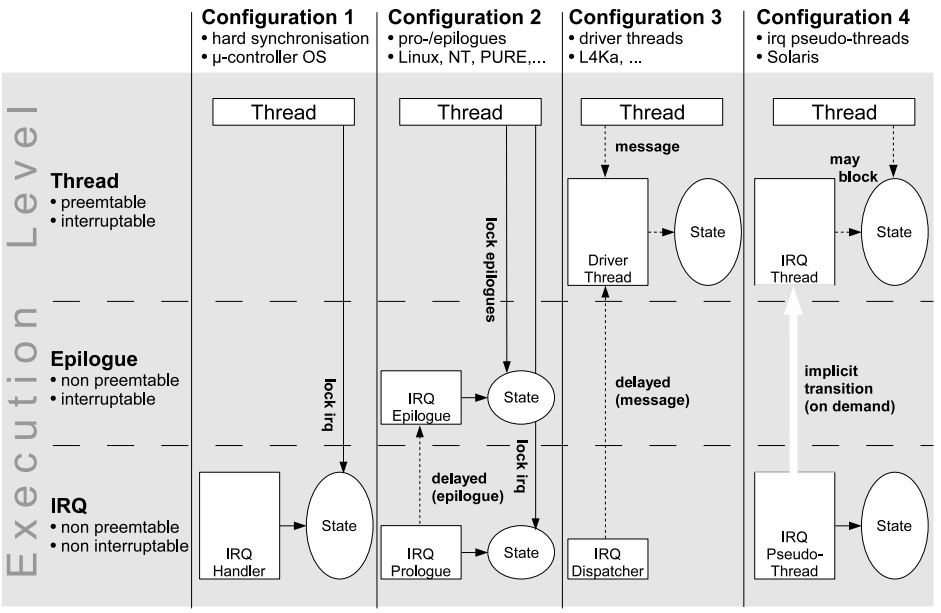

Figure 2: Different configurations of interrupt synchronization in a device driver

a message to the thread registered for an interrupt signal and activates the scheduler[3]. If the thereby activated driver thread has the highest priority, it is then selected for execution and starts the real processing of the interrupt request. Because the code is executed inside a thread, it may even block on other threads (e.g. use synchronous IPCs). Interrupt synchronization is thereby mapped to ordinary thread synchronization, no special mechanism is required.

IRQ pseudo-threads This very sophisticated approach is used by Solaris[6]. The general idea is, again, to map interrupt synchronization to thread synchronization and thereby avoid the need for an extra interrupt synchronization mechanism. However, instead of sending a message to a waiting thread and activating the scheduler in case of an interrupt, Solaris directly switches to a special pre-allocated pseudo-thread. The pseudo-thread owns a complete thread context (instruction pointer, stack), but is not a deschedulable entity as it is still running on interrupt level while executing the handler code. Only if the control flow is required to block (e.g. because of waiting for a locked mutex), the kernel *transparently* lifts up the pseudo-thread to become a real schedulable (but blocked) thread entity, ends the interrupt, and activates the scheduler. Henceforth, the interrupt handler code is executed on thread level until it terminates.

Commonalities and Differences

The various approaches used for interrupt synchronization result in different *execution models* for interrupt code. They also lead to different *programming models* for the developer of device drivers, in which interrupt handling typically takes place. The feature diagram in Figure 1 depicts these two models as main dimensions of commonality and difference in the domain of IRQ synchronization.

[3]The message is actually an IPC sent to a usermode thread in another address space (e.g. of a device driver process). However, this is not relevant here, as it is part of the isolation and interaction properties.

In the *execution model dimension*, one can distinguish approaches that execute the complete IRQ handler on one synchronization level from those, where the interrupt code is spread over two different synchronization levels. The simple *hard synchronization*, as well as the L4Ka *Driver threads* belong to the first category, whereas Solaris' *IRQ pseudo-threads* and *prolog/epilogues* belong to the second. In the 2-level approaches, an interrupt control flow always begins execution on interrupt level. It continues delayed execution on either thread level (Solaris) or epilogue level (Windows, Linux, PURE).

In the *programming model dimension*, one can again distinguish between 1-level and 2-level approaches. However, only for *prologue/epilogues* the developer has to split the code explicitly between boths levels, by optionally requesting the execution of epilogues. *Driver threads* and *IRQ pseudo-threads* offer an identical programming model, as the latter performs an automatic transition from interrupt level to thread level on demand. Optionally, the thread-based approaches permit the interrupt control flow to block on other threads.

4.2 Generalization

The variability in the execution model is desired, as it corresponds to the different implementation of the architecural property, which in turn lead to the variability regarding non-functional properties. The variability in the programming model, however, has to be generalized for the development of architecture-neutral components. The goal is to be able to configure the execution model *without* having to buy another programming model. Figure 2 shows the possible configurations, as well as the resulting structure, of a device driver which is accessed from thread and interrupt level.[4]

Finding a common set of abstractions for the architecure-neutral progamming model requires some reduction to the common de-

[4]The possible configurations also depend on the configuration of other system components, like multi-threading support, which is required by the thread-based configurations.

nominator. If the specific architectures depends on certain assumptions, the most restrictive ones have to make it into the architecture-neutral model. For instance, only the *pro-/epilogues* model enforces an explicit splitting of the driver into two different levels of execution (Figure 2). Nevertheless, it has to become a part of the architecture-neutral model. This is possible, as the enforced explicit splitting is just an additional requirement which does not conflict with other requirements. In other cases, however, it might be necessary to seperate out an abstraction into the architecture-specific model. The optional *may block* feature (Figure 2) is an example for an architecture-specific abstraction that is only available in the thread-based models. A device driver implementation which depends on this feature can not be transformed into the *hard synchronization* or *pro-/epilogues* model.

The aim of the architecture-neutral driver model is to provide enough context information to enable a transformation of the driver code into the architecture-specific model by aspects. This basically means to insert the "right" synchronization primitives at the "right" places. Logically, it is access to state which has to be synchronized. However, this can be mapped to method synchronization, as all state information is considered to be accessible by a restricted set of methods only. In our model, each method of a device driver is placed in one of three different synchronization classes:

synchronized Methods of this class are, depending on the actual configuration, synchronized by some higher-order protocol. If invoked by an interrupt, the actual execution is typically delayed. If invoked by a thread, parallel invocation from interrupt has to be prevented. This is the default class for driver methods. It corresponds to the following execution levels: *IRQ* (Configuration 1), *Epilogue* (Configuration 2), *Thread* (Configuration 3), *IRQ + Thread* (Configuration 4)

blocked For Configuration 2 (*pro-/epilogues*), methods of this class correspond to execution level *IRQ*. For all other configurations they are simply merged into the class *synchronized*.

transparent Methods of this class do not need any synchronization at all, as they perform atomic operations only or use interruption-transparent algorithms. Hence, they can be invoked from any control flow at any time.

If methods from the same class invoke each other, no synchronization is necessary. Synchronization primitives have to be inserted for transitions from thread or interrupt level to *synchronized* or *blocked*. Configuration 2 additionaly requires synchronization of transitions between the classes *synchronized* and *blocked*. The following section describes with a brief example how this can be implemented in AspectC++[16].

4.3 Implemention Sketch

Consider a simple device driver for the system timer, as in the following listing.

```
class Timer {
  ... // state
public:
  void init( long time );
  long get() const;
  void add_event(const EventCallback* cb);

private:
  void tick();
  void process_events();
  friend class irq_dispatcher;

  void handler() {
    tick();
    process_events();
  }

  // what belongs to which synchronization class
  pointcut int_handler()  = "% Timer::handler()";
  pointcut blocking()     = "% Timer::init(...)"
                          || "% Timer::tick()";
  pointcut transparent()  = "% Timer::get(...)const";
  pointcut synchronized() = "% Timer::%(...)"
   && !int_handler() && !blocking() && !transparent();
};
```

The driver offers a public interface for threads to set and get the system time (`init()`, `get()`) and to be notified at a certain time (`add_event()`). The private `handler()` method is invoked by the low-level interrupt dispatcher in case of an interrupt signal. It advances the system time and notifies all registered events that have expired. The `get()` method performs an atomic read operation and is therefore considered to be *transparent*, while `init()` is assigned to *blocked*, as it performs a non-atomic write operation on the internal timer value, which is also modified by `tick()`. All other methods (except `handler()`) are assigned to *synchronized*.

The `Timer` driver code is architecture-neutral regarding the interrupt synchronization property. The following aspect is used to transform it to use *hard synchronization*:

```
aspect Configuration1 {
  pointcut block() = Timer::synchronized()
           || Timer::blocking();
  advice call( block() && !within( block()
         || Timer::int_handler() ) : around() {
    disable_int();
    tjp->proceed();
    enable_int();
  }
};
```

The aspect for the *prologue/epilogues* model has to give some extra advice for the delayed execution of epilogues and for the potential transitions between the synchronization classes *blocked* and *synchronized*:

```
aspect Configuration2 {
  pointcut block() = Timer::blocking();
  pointcut delay() = Timer::synchronized();
  advice call( delay() )
         && !within( "% Timer::%(...)" ) : around() {
    lock_epilogues();
    tjp->proceed();
    leave_epilogues();
  }
  advice call( block() ) && !within( block()
         || Timer::int_handler() ) : around() {
    disable_int();
    tjp->proceed();
    enable_int();
  }
  advice call( Timer::synchronized() ) && !within(
         Timer::synchronized() ) && cflow(
         execution( Timer::int_handler())): around(){
     add_epilog (tjp->action ());
  }
};
```

For the *driver threads* model no advice has to be given, as all necessary synchronization is implicitly done by the message-based interaction used to invoke methods. As discussed in section 2, *inter-*

action is another architectural property which is not in the scope of this paper.

```
aspect Configuration3 {
  // nothing to do!
};
```

Finally, the necessary synchronization primitives for the *IRQ pseudo-threads* model are applied by this aspect:

```
aspect Configuration4 {
  pointcut exclude() = Timer::synchronized()
           || Timer::blocking();
  advice call( exclude() )
         && !within( exclude() ) : around() {
    lock_mutex();
    tjp->proceed ();
    unlock_mutex();
  }
};
```

5. SUMMARY AND CONCLUSIONS

Many non-functional properties of software systems are emergent and, thus, need to be addressed on a global scope by some sort of "holistic aspects". Architecture can be understood as such a "holistic aspect", as it has a noticeable impact on many non-functional properties. Architectural decisions do crosscut significant parts of the actual implementation of every component. This gives the opportunity to address them by aspects and thereby configure non-functional properties *indirectly* by the means of configurable architectures. On the other hand, software components have to be specifically designed with architectural configurability in mind, which can be a quite complicated task. The work on CiAO is clearly at an too early stage to evaluate the benefits of using aspects for this purpose on the large scale. From what we did so far we can, however, draw some preliminary conclusions:

The devil is in the details While it is broadly accepted that aspects are feasible for encapsulating crosscutting concerns, their applicability for the non-trivial case always seems to be a question on its own. For the (relatively complex) interaction patterns found in operating systems this is specifically true, as subtle implementation details can have an enormous impact on correctness or performance. Hence, an in-depth analysis of the technical details is unavoidable for a reliable evaluation if and how AOP is beneficial for the encapsulation of certain architectural properties.

Applicability to other OS concerns Interrupt synchronization is just one of the properties that "make" the architecture of an operating system. It is a natural starting point for a bottom-up process, which is required to tackle the inherent interdependencies between architectural properties. Architecture-neutral models for other fundamental properties, including isolation and interaction, have to be developed as well. This will probably be again a matter of very specific details. However, we are optimistically that the general approach as described in section 3 works for these other properties as well.

Additional Requirements to AspectC++ The interrupt synchronization example demonstrates that AspectC++ already provides the ability to perform quite complex context-dependent transformations. Nevertheless it is rather likely that we have to carefully extend AspectC++ to address other architectural properties. To implement, for instance, component interaction via IPCs as an aspect, one has to be able to give advice that forwards the whole calling context to a thread running in another address space.[5]

6. RELATED WORK

There is some related work in the domain of applying AOP techniques to operating systems. Coady et all demonstrated the encapsulation of an architectural OS property (prefetching) by an aspect in the FreeBSD kernel[3]. However, the focus of this work was not on configuration of architectural properties. The THINK framework demonstrates, how operating systems with different interaction schemes can be constructed from architecture-neutral components by using special "binding components"[5]. THINK does not use AOP, it is based on COM interfaces and does not support the configuration of other architectural properties. Related work that suggest to exploit aspects for specifying synchronization constraints is to numerous to list, however, the work of Lopes[12] probably had the most noticeable impact to this topic.

7. REFERENCES

[1] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.

[2] D. P. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly, 2001.

[3] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE '01*, 2001.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. AW, May 2000.

[5] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *2002 USENIX TC*, pages 73–86. USENIX, June 2002.

[6] S. Kleiman and J. Eykholt. Interrupts as threads. *ACM OSR*, 29(2):21–26, Apr. 1995.

[7] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *ACM OSR*, 13(2):3–19, Apr. 1979.

[8] J. Liedtke. On μ-kernel construction. In *15th ACM Symp. on OS Principles (SOSP '95)*. ACM, Dec. 1995.

[9] A. Lister and R. Eager. *Fundamentals of Operating Systems*. Macmillian, 4 edition, 1988.

[10] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. On the design and development of a customizable embedded operating system. In *SRDS Dependable Embedded Systems (SRDS-DES '04)*, Oct. 2004.

[11] D. Lohmann and O. Spinczyk. Architecture-Neutral Operating System Components. *19th ACM Symp. on OS Principles (SOSP'03)*, Oct. 2003. WiP session.

[12] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[13] D. Mahrenholz, O. Spinczyk, A. Gal, and W. Schröder-Preikschat. An aspect-orientied implementation of interrupt synchronization in the PURE operating system family. In *5th ECOOP W'shop on Object Orientation and Operating Systems*, pages 49–54, Malaga, Spain, June 2002.

[14] A. Rubini and J. Corbet. *Linux Device Drivers*. O'Reilly, 2001.

[15] D. A. Solomon and M. Russinovich. *Inside Microsoft Windows 2000*. MS Press, 3 edition, 2000.

[16] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *40th Int. Conf. on Technology of OO Languages and Systems (TOOLS Pacific '02)*, pages 53–60, Sydney, Australia, Feb. 2002.

[17] O. Spinczyk and D. Lohmann. Using AOP to develop architecture-neutral operating system components. In *11th SIGOPS Eur. W'shop*, pages 188–192, Leuven, Belgium, Sept. 2004. ACM.

[5] Within the same address space this is already possible in AspectC++ by using so-called *action objects*[17, 16].