Using AOP to Develop Architectural-Neutral Operating System Components

Olaf Spinczyk and Daniel Lohmann
{os,dl}@cs.fau.de

Friedrich-Alexander-University Erlangen-Nuremberg, Germany
Department of Computer Science 4

Abstract

The architecture of an operating system, e.g. micro kernel or monolithic kernel, is usually seen as something static. Even during the long lasting evolution of operating system code it is extremely hard and, thus, expensive to change the architecture. However, our experience is that architectural evolution is often required and an architecture-neutral way to develop operating system components should be found. After analyzing why architectural flexibility is so difficult to achieve, we propose Aspect-Oriented Programming (AOP) as a solution. An example from the PURE OS family, which is implemented in an aspect-oriented programming language called AspectC++, will demonstrate the usefulness of the approach, which allows to separate the code that implements architectural properties from the core functionality.

1. Introduction

Several different operating system architectures have been proposed in the past and will probably be proposed in the future. All of them have their advantages and disadvantages. The selection of an operating system architecture is made very early in the design process. It has a significant impact on the resulting implementation as it influences synchronization, protection boundaries, inter-component communication, and many other important strategies and aspects. This impact on large fractions of the code makes it difficult to modify architectural design decisions when the requirements on the system change.

As an example consider the integration of multiprocessor (SMP) support in Linux. The first kernel release that supported SMP hardware was version 2.0. However, it still used the old system-wide locking scheme of earlier versions and performed badly in SMP environments. The unavoidable switch towards a fine-grained locking strategy was the trigger for costly and long-lasting architectural evolution. Hundreds of device drivers, file systems, and other components of the system had to be adopted. Now the 2.6 kernel has fine-grained locking in almost all parts of the system, but the process took several years to complete.

Our conclusion from this example is that architectural properties must be better encapsulated to reach architectural flexibility and thereby reduce the cost of architectural evolution. New programming language concepts can help to achieve this goal. Namely Aspect-Oriented Programming is a promising approach for that purpose.

For several years it was not possible to implement an efficient operating system in an aspect-oriented manner due to the lack of appropriate compiler support. With AspectC++[1] [8], an aspect-oriented language extension to C++, our group has developed an important tool that opens a large area of research on aspects in system software. In the remaining sections of this paper we want to demonstrate that with such a language an architecture-neutral development of operating system components is realistic, which would significantly reduce the costs of architectural evolution.

The outline of the paper is as follows. In section 2 we discuss what 'makes' the architecture of an operating system and why it is so difficult to change it. In section 3 we will develop a concept to solve this problem based on aspect-oriented programming language features. Section 4 then gives an example, which shows how the concept can be applied in real world operating system code. At the end of the paper we present related and future work and our conclusions.

2. Problem Analysis

Operating systems that are based on different architectures do not necessarily differ in their provided functionality. In fact it is *how* they provide this functionality what differentiates them, for example:

[1] http://www.aspectc.org/

- How many processes or threads execute operating system code at the same time?
- How do the system components interact with each other, e.g. message-based or procedure-based?
- How are the components associated with protection boundaries, e.g. address spaces?
- How does the system deal with interrupts?

These question are answered by architectural design decisions that are based on (non-functional) requirements like security, dynamic reconfigurability, robustness, etc. As an example consider the requirement that the system should react on interrupts as fast as possible. This requirement will probably lead to the architectural design decision to use fine-grained instead of coarse-grained interrupt synchronization, because that reduces the interrupts latencies.

This interrupt synchronization example shows that the question *how* some requirement is fulfilled is often dominated by the aspect *where* something is done. Here the selection of code locations for calls to synchronization primitives is responsible for the granularity. A lot of these locations, which are typically scattered across several parts of the system, are affected if fine-granular synchronization is implemented.

Operating system implementations were in many cases used for several decades. During this time the requirements on the system evolve. Often the reason is that there is an enormously fast evolution of the supported hardware platforms. This was the case in the Linux example, which was mentioned in the introduction. Also the application profile might change over the years. For instance, an ongoing trend in the Linux area is to support real-time applications.

Changes in the requirements make it hard to find *the right* operating system architecture at the beginning of the design process, thus architectural evolution is often unavoidable. However, the modification of architectural properties is hard and costly to implement.

3. Language Support

Single properties of a software that affect large fractions of the program code, like the architectural design decisions discussed in the last section, are often called *crosscutting concerns*. Their existence has a negative impact on the maintainability and reusability of code. AOP [5] is a concept that tries to reduce these problems by providing programming language elements that allow the *modular* implementation of crosscutting concerns. The approach followed in this paper is to apply these language elements in the development of an operating system to reduce the costs of architectural evolution.

AspectC++ [8] is an aspect-oriented derivate of C++. By presenting the most important language features of AspectC++ the following paragraphs will introduce the language and the AOP concept at the same time.

The most important language feature to modularize a crosscutting concern is the *pointcut* concept. A pointcut is a set of points in the code (so called *join points*), which are affected by the same crosscutting concern. In AspectC++ these sets can be defined in a very flexible way by using a declarative language consisting of predefined pointcut functions, wildcards for matching names, and algebraic operations to combine pointcuts. For example, if the concurrent execution of certain functions has to be avoided, some synchronization object must be acquired before and released after the execution of these functions. The relevant points in the code can be identified by the following pointcut definition:

```
pointcut funcs() =
  execution("void enqueue(Node*)") ||
  execution("Node *dequeue()");
```

Here the pointcut `funcs()` is the union of all executions of `enqueue()` and `dequeue()` in the whole system.

It is then possible to define some action that should be executed when any of the join points in the pointcut is reached at run time. This is achieved with a so called *advice* definition as shown here:

```
aspect MutexSync {
  ...
  Mutex _m;
  advice funcs(): before() {
    _m.lock();
  }
  advice funcs(): after() {
    _m.unlock();
  }
};
```

The first advice definition means that *before* the body of any function described by `funcs()` is executed the mutex object `_m` should be locked. Consequently the second advice definition ensures that the mutex object is unlocked again *after* the critical section is left. Both advice definitions are encapsulated in a named modular unit, which is an *aspect*. Besides the advice definitions aspects can, similar to classes, store and manage state information (the mutex variable `_m` in the example), which is also accessible by the advice code bodies. The compiler is responsible for the invocation of the advice code from the specified join points. It is not necessary to call the synchronization primitives from anywhere outside this aspect.

The AspectC++ language provides many more language elements. We have omitted explanations of all these features here to focus on the goal of our approach: The presented `MutexSync` aspect is a minimal example that shows

how aspects can encapsulate knowledge about *where* something has to be done in the system (pointcut definitions) and *what* exactly should be done (advice definitions).

Our approach is to use these mechanisms to implement the code that results from architectural design decisions.As these decisions also consist of a *where* and *what*, we assume that AOP can help to modularize the implementation of architectural OS properties. This could be the key to architectural flexibility, which would allow static configuration and easy evolution of the OS architecture.

4. Example: PURE Device Drivers

This section provides an example that demonstrates how AOP enables system developers to write their device driver code in an architecture-neutral way. The example is taken from the PURE operating system family[2] [2], which aims to support applications in the area of deeply embedded systems by providing fine-grained configuration capabilities. These configuration capabilities even include the configurability of certain architectural properties at compile time [1]. Here we will concentrate on the synchronization and concurrency aspect of device drivers.

PURE device drivers are designed as fine-grained hierarchies of C++ classes. In our new AOP-based implementation, a driver itself does not contain any code for synchronization or creation of new threads. All this is done by external aspect definitions, which may be statically configured. The external synchronization aspects are based on a library of reusable *abstract* aspects, which implement different synchronization strategies. These abstract aspects define *pure virtual pointcuts* for the synchronization points. This means that the decision *where* the mechanism is applied is postponed until a derived concrete aspect redefines the pure virtual pointcuts. The aspect library itself only defines *what* should happen if certain synchronization points are reached.

Figure 1 on the following page shows the PURE implementation of the MutexSync strategy and depicts how the abstract (*what*) and concrete (*where*) aspect definitions affect on the component code, a floppy driver class.

The abstract aspect `MutexSync` defines two pure virtual pointcuts, `classes()` and `funcs()`. The pointcut `classes()` is aimed to define the set of classes that should be synchronized by this strategy. It is used in the following advice definition to insert a new data member `_m` of type `Mutex` in each of these classes. As in the example above, the pointcut `funcs()` is used to define the functions that need to be synchronized. The AspectC++ function `tjp->that()` returns a pointer to the object on which the function is executed. It is used in the advice bodies to access the mutex data member that was introduced into these classes.

The concrete aspect `FloppySync` derives from the abstract library aspect and redefines the pointcuts, thereby establishing the link between the abstract synchronization strategy and the join points in the component code, where the aspect code has to be applied. The pointcut expression `execution("% FloppyDriver::% (...)")` uses wildcards (%, ...) to match the *executions* of all functions of class `FloppyDriver`.

Besides the `MutexSync` aspect, the PURE synchronization library provides mechanisms according to the first and second readers/writers scheme, the producer/consumer model, etc. There are also similar aspects, which rely on the blocking semantics of `send()` and `receive()` primitives for inter-process communication as synchronization model. For instance, the abstract `ServerSync` aspect shown in figure 2 on page 5 illustrates how mutual exclusion on the same component code can be achieved with a single server thread that sequentially processes requests. The server thread is implemented by the class `ActionServer`. All `ActionServer` instances are active objects with their own thread of control, which executes the function `run()`. This function implements the job processing loop.

`ServerSync` defines so called *around* advice. Around advice is not executed before or after, but *instead* of the original call. In the advice code the AspectC++ function `tjp->action()` is used to obtain a so called *action* object. This object can be used to continue the intercepted action (the call) later or in a different environment. Here the address of the action object is given to the server thread in the driver instance that was the target of the call (`tjp->target()`). The thread on the server side then uses the passed action object to continue the call operation by invoking `action->trigger()`.

Again, this synchronization strategy is applied to our floppy driver by deriving a concrete aspect `FloppySync`. The pointcut expression `call("% FloppyDriver::% (...)") && !within("FloppyDriver")` matches all *calls* to functions of class `FloppyDriver` that are not located inside of `FloppyDriver` itself. In other words: all points where the driver component is invoked from other components. Note that the synchronization advice is now given to the caller side (*call* pointcuts), while it was given to the callee side (*execution* pointcuts) in the `MutexSync` example.

With aspects like this we have reached a very high level of abstraction. Simply changing the `FloppySync` aspect is sufficient to associate a server thread with each floppy driver and use message-based instead of procedure-based communication. By choosing other base aspects, several drivers could share the same mutex object, or the aspect could be

[2] PURE systems can be configured and downloaded from `http://www.pure-systems.com/`.

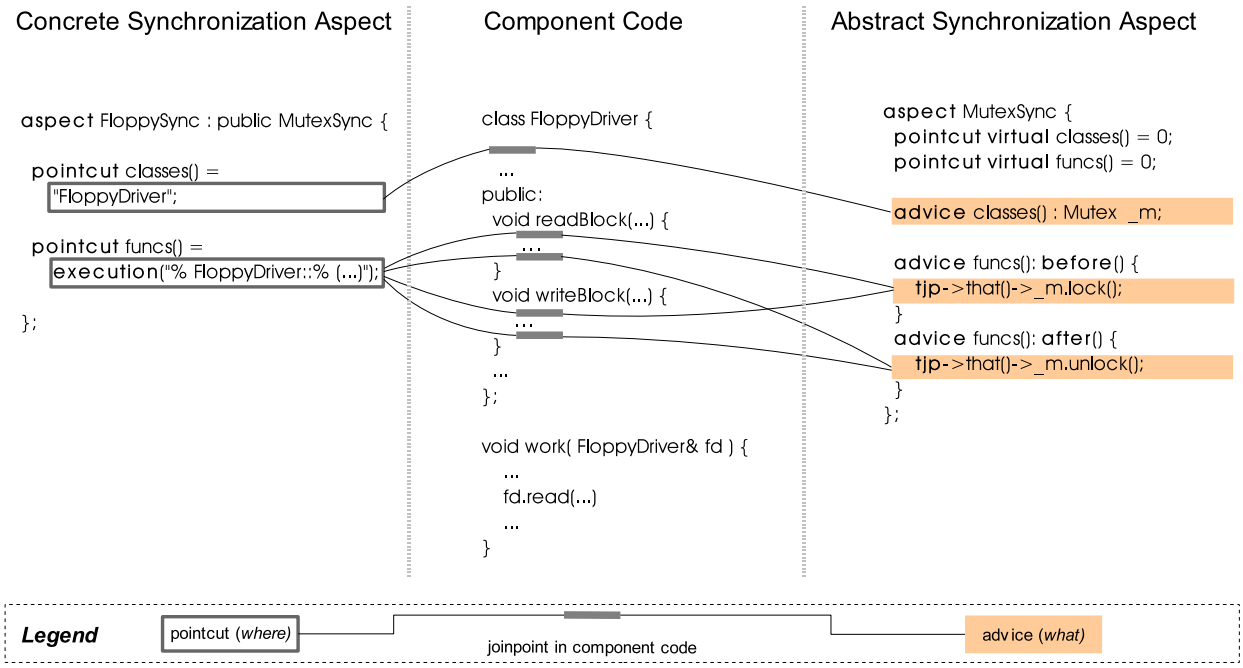

Figure 1. Applying the MutexSync mutual exclusion strategy to an OS component

omitted at all if the driver is reused in a single-threaded environment. Even though the presented example was simplified and many important AspectC++ features were not mentioned, we are optimistic that it nevertheless showed the high flexibility and value of the approach.

5. Conclusions and Future Work

This paper presented our approach to encapsulate code that implements architectural properties of operating systems with aspects. This helps to cope with architectural evolution, architectural design decisions loose their strategic character, and even the configuration of architectural properties becomes feasible in the context of operating system families or construction kits.

Our experience from the PURE project is that many architectural aspects can be implemented with our AspectC++ language and compiler. However, a successful extensive encapsulation of system properties by aspects rises and falls with the design quality of the functional components. To reveal the full power of our approach, OS-components need to be designed "aspect-aware" from beginning. That is, follow strict naming rules, provide a fine-grained functional separation of concerns and clearly specify pre-/ and post-conditions around potential join points. Because of its fine-grained organization, PURE was a very good test system to examine our approach. However, PURE was never designed with aspects in mind. This led to limitations in the overall applicability of aspects. In the CiAO[3] project[6] we are now developing a new operating system family, based on AOP techniques "from scratch" . The ambitious end goal of CiAO is to provide customization of *all* fundamental architectural properties. For instance, a monolithic kernel, micro kernel, or library OS version should be generated from the same OS component sources.

.

6. Related Work

Only a few other papers have been published about aspect-oriented programming in the operating systems domain. For instance, in the a-kernel project a prototyped AOP extension for the C programming language[4] was used to improve the modularity of prefetching code in the FreeBSD kernel [3]. Netinant et al. proposed a framework for the aspect-oriented construction of operating systems [7]. While both contributions show the big potential of AOP for operating systems, their focus was not on the evolution or configuration of architectural properties.

A remarkable paper about the THINK framework describes how operating systems with different architectures can be constructed from architecture-neutral components

[3] CiAO is Aspect-Oriented

[4] This language, AspectC, is not to be confused with AspectC++, particularly as at the time of this writing there is still no compiler available for it.

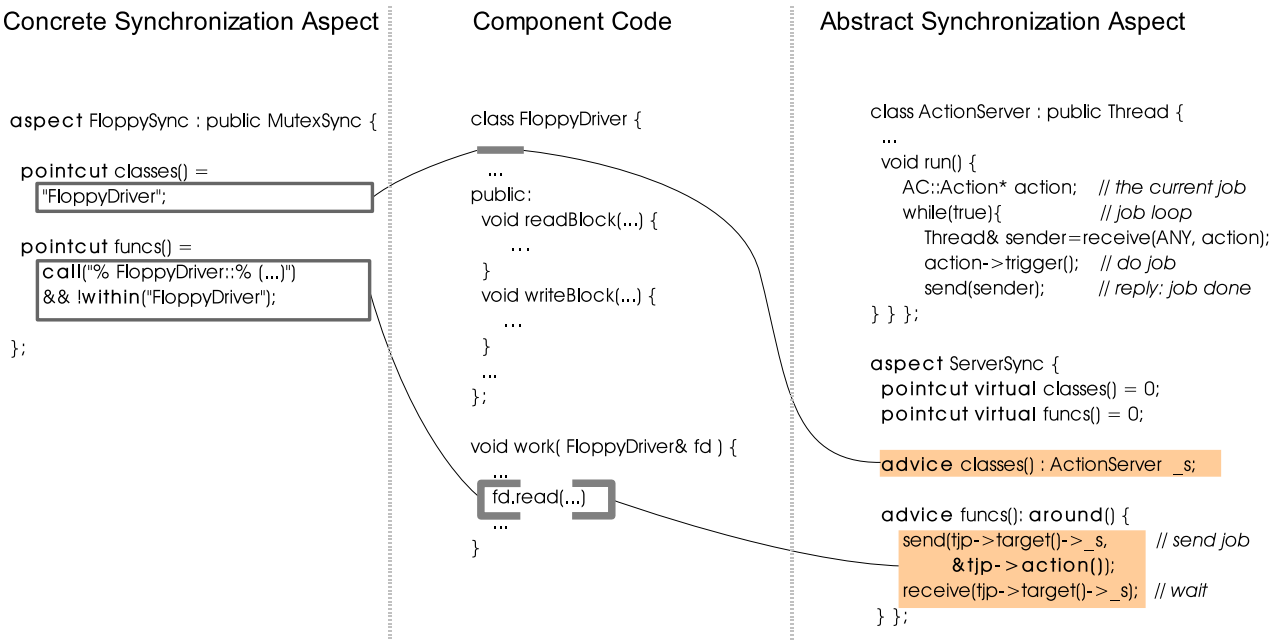

Figure 2. Applying ServerSync mutual exclusion strategy to an OS component

[4]. THINK uses special "binding components" to manipulate the component interaction. However, THINK was developed on the base of a binary component model (COM), which we considered too heavy weight for the deeply embedded target domain of PURE and CiAO. Furthermore, aspects provide many more kinds of join points than just component boundaries and can therefore lead to a much higher level of flexibility. This flexibility is a chance and a risk at the same time, which we are going to explore.

References

[1] D. Beuche, A. A. Fröhlich, R. Meyer, H. Papajewski, F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. On architecture transparency in operating systems. In *Proceedings of the 9th ACM SIGOPS European Workshop "Beyond the PC: New Challenges for the Operating System"*, pages 147–152, Kolding, Denmark, Sept. 2000.

[2] 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 *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, pages 45–53, St Malo, France, May 1999.

[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 *Proceedings of the Joint European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Internation Symposium on the Foundations of Software Engineering (FSE-9)*, 2001.

[4] J.-P. Fassino, J.-B. Stefani, J. Lawall, and G. Muller. THINK: A software framework for component-based operating system kernels. In *Proceeding of the 2002 USENIX Technical Conference*, pages 73–86. USENIX Association, June 2002.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

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

[7] P. Netinant, C. A. Constantinides, T. Elrad, and M. E. Fayad. Supporting aspectual decomposition in the design of operating systems. In *Proceeding of the 3rd ECOOP Workshop on Object-Orientation and Operating Systems (ECOOP-OOOSWS'2000)*, pages 38–46. Universidad de Oviedo, June 2000. ISBN 84-8317-222-4.

[8] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, pages 53–60, Sydney, Australia, Feb. 2002.