# On Adaptable Aspect-Oriented Operating Systems

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

Friedrich-Alexander-University Erlangen-Nuremberg, Germany

## Abstract

*Operating systems for small embedded devices have to cope with a broad variety of requirements as well as strict resource constraints. Family-based operating system development, based on aspect-oriented techniques, is a promising approach to implement operating system product lines that are highly configurable and tailorable. However, static application-specific tailoring of operating systems is not sufficient in the domain of "smart devices", which are needed in typical "pervasive" and "ubiquitous" computing scenarios. To cope with the dynamically changing environments in this domain we advocate for adaptable operating systems as a solution. An adaptable operating system is still tailored down for specific requirements, but can be reconfigured dynamically if the set of required features changes. Furthermore, the combination of adaptability with aspect-orientation potentially allows changes in global operating system policies at runtime. However, the dynamic replacement of aspects requires a dynamic aspect weaving technology, which is naturally quite expensive. As this means a conflict with the resource constraints of small embedded hardware platforms, there is a need for novel ideas. Thus, a family-based approach to dynamic aspect weaving is presented, which allows the resource consumption to scale with the actual requirements on dynamism. At the same time it offers programmers a convenient high-level language to implement both, dynamic and static aspects of the operating system.*

## 1. Introduction

Operating systems for embedded and deeply embedded devices have to scale with a very broad variety of requirements, coming both from the hardware and application level. Different hardware architectures and configurations have to be supported, while resources (in terms of memory, power consumption and CPU speed) are often strictly constrained. Applications typically have very different requirements to the services and strategies implemented by the underlying OS.

### Operating System Product Lines

It is simply impossible to build a "one-fits-all" system that fulfills the requirements of all potential applications, while still being thrifty and economical with system resources. The solution is therefore to tailor down the operating system so it provides exactly the functionality required by the intended application, but nothing more. This leads to a *family-based* or *product-line* approach, where the variability and commonality among OS family members is expressed by *feature models* [13]. Special tools are used to extract and statically configure the concrete operating system based on an application-specific feature selection [1].

The overall quality of an OS product-line depends mostly on the offered levels of variability and granularity. A crucial point is the mapping of all selectable and configurable features to their corresponding, well encapsulated implementation components. Especially the encapsulation of non-functional properties is often limited, due to their crosscutting character. Fundamental system policies, like synchronization or activation points for the scheduler, have typically to be reflected in many points of the OS component code. This makes it almost impossible to implement them as independent encapsulated entities and thereby restricts variability and granularity. *Aspect-oriented programming (AOP)* has proven to be a promising way to deal with crosscutting concerns [2]. It allows encapsulating the implementations of crosscutting concerns in entities called *aspects*, which are then *woven* into the OS component code (e.g. classes) at build time. A well-directed application of AOP principles in the development of OS product lines can therefore lead to a higher variability and granularity of the selectable OS features, as their implementations can not only be encapsulated by classes, but also by aspects. This potentially results in very flexible systems that offer configurability of even fundamental architectural properties [3].

As an example, Figure 1 shows a part of an operating system product line feature model, where features are mapped to those classes and aspects that provide their implementation. By application-specific *feature selection* (Figure 2) it is now possible to create tailored systems that contain only those modules which implement the selected features.

### Additional Requirements of Smart Devices

Static tailoring of an operating system for a specific application works well in many domains. However, in the emerging markets of "smart devices" (like mobile phones, personal digital assistants or "wearables"), the set of executed applications as well as the non-functional requirements to the operating systems do vary. Manufacturers are responding to this challenge by enlarging their devices by more and more system resources and "big" operating systems that implement many features, but are less reusable and scalable. This is unsatisfactory, as it noticeably increases production costs, weight and power consumption of mobile devices. We therefore advocate for *adaptable operating systems*, which provide a well-balanced way of adaptability, while still being based on application-specific tai-

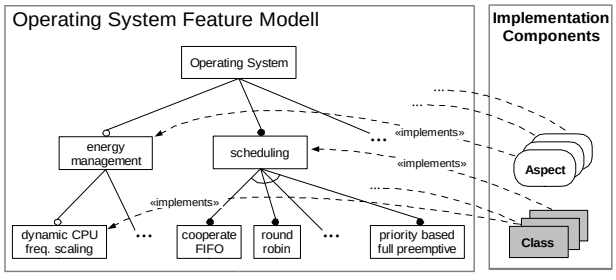


**Figure 1** Available features and their implementation artifacts

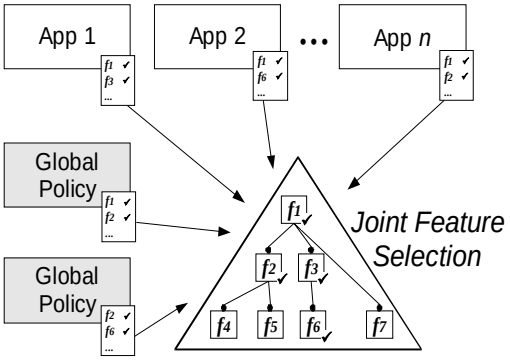


**Figure 2** Feature selection

loring.

## 2. Adaptable Operating Systems

The set of requirements (services and non-functional properties) that has to be fulfilled by a tailored OS depends on the requirements defined by the (potentially) executed applications as well as on the requirements defined by global user policies, like the energy or security mode. The requirements set leads to a feature selection, which corresponds to a specific member of the operating system family (Figure 2).

An *adaptable operating system* is reconfigurable if the set of requirements changes. Such a reconfiguration is necessary, if a new application is about to be executed with some requirements that are currently not offered by the system. It is also necessary, if there are changes in the global (user-defined) requirements, e.g. the system is switched to a low-power mode. And finally, to safe system resources, an adaptable operating system *may* be reconfigured if some application is removed and the set of requirements thereby shrinks.

To our understanding, all these configurations are still members of one family of operating systems, where each configuration (feature selection) leads to one distinct family member. The process of adaptation can therefore be understood as morphing from one feature selection into another. By adaptation to the closest set of the demanded features, the system is always tailored with respect to the actually executed applications.

### Feature Binding Times

A facility for on-demand adaptation needs to be provided by the OS itself in some way. The main question is, how to rebind features at runtime (in particular their implementation classes and/or aspects), if the set of required features changes?

For classes and libraries this task can be done by a dynamic loader/linker, which loads the component and performs all necessary steps to bind it, like relocation and component registration. Such a dynamic library loader is present in many current operating systems.

However, as our product line follows an AOP-based approach, features may also be implemented by aspects. For dynamic loading/unloading of aspects, the system has to provide facilities for *dynamic weaving*. Dynamic weaving means that aspects can be applied or removed during runtime without recompiling and re-deployment of an application [4]. We call an engine for dynamic aspect weaving a *dynamic weaver*. The dynamic weaver is therefore a vital part of any operating system that uses AOP and supports dynamic binding of features.

## 3. Implementation Ideas

The implementation of a dynamically adaptable operating system family has to deal with a wide variety of concerns like the management of requirements, provided features, dynamically loadable modules, the dynamic binding, verification of loaded modules and many other things. As this paper concentrates on the aspect-oriented design used to structure a system family, this section focuses on the technology needed to enable aspect-oriented programming in a dynamically changing operating system context.

### General Assumptions

Our ideas presented in this section are based on the following assumptions:

1. *Minimal Overhead* – The motivation for our work on dynamic adaptation is to provide a better, i.e. more resource efficient, OS support for applications in a dynamically changing environment than it could be provided by a one-fits-all solution. If the necessary infrastructure for dynamic weaving costs more than we can save, the approach fails.
2. *Infrequent Changes* – Configuration changes are triggered by users who start new applications or change some global policies like the power management mode. The rate of these changes is quite low compared to the usual “heart beat” of operating systems. Therefore, the system performance and resource consumption during the normal execution is much more important than an efficient switch between the system configurations.
3. *Static and Dynamic Aspects* – It should be configurable at compile time whether certain features may be selected or deselected at runtime. The general idea behind this is that static features can be implemented more efficiently than dynamically changeable features and, thus, should always be preferred. If a feature has a crosscutting nature it will be implemented by an aspect. Depending on the system configuration it could either become a static or a dynamic aspect. Furthermore, static and dynamic aspects have to coexist in the system. Solutions that only support dynamic weaving are not acceptable due to their low efficiency.

### Language Support

While building the operating system, aspects are separated from the early design phase and differentiated into static and dynamic aspects at the configuration phase. For describing the static aspects AspectC++ is used. AspectC++[5] is a general purpose aspect-oriented extension of C++ developed by the authors, and follows an AspectJ-like approach of AOP. It is implemented as a C++ preprocessor, based on a source code transformation system that transforms AspectC++ code into C++ code. Afterwards a conventional C++ compiler is used to compile the executable code.

We are working on an extension to AspectC++ that allows the developer to write both static and dynamic aspects in the same AspectC++ language. Thus, it would be transparent for the developer whether s/he is describing a static or a dynamic aspect. Following this *single language approach*, the decision whether some aspect is static or dynamic can be postponed to the configuration stage and has no impact on the implementation stage. This approach supports deriving systems that offer as much dynamicity as necessary while it still allows resolving as much statically as possible.

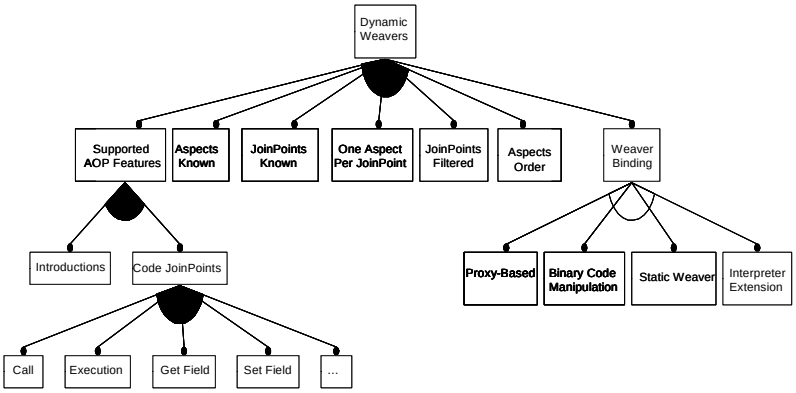


**Figure 3** Feature model for dynamic aspect weavers [11]

## Approaches for Dynamic Weaving

To support dynamic weaving of aspects, several approaches were proposed by the AOSD community. Most of them are intended for use in Java environments and based on Java-specific APIs, runtime byte code manipulation or virtual machine extensions [4, 6, 7, 8].

Only few approaches have been proposed for the C/C++ domain [9, 10]. They basically follow two general implementation techniques:

- *Machine code manipulation* – Aspects are woven at runtime by on-demand insertion of jump statements to the aspect code into the machine code at all affected joinpoint positions. This technique is followed by the MicroDyner project [9].
- *Runtime aspect registration* – Aspects are woven by registering them against a runtime registration system. The runtime registration system manages lists of all registered aspects and all available joinpoints and thereby performs the binding of aspects to joinpoints. The original C++ code is instrumented, either by hand or with the help of tools, to call the runtime system at each potential joinpoint. The runtime system then calls all aspects registered for this joinpoint. This technique is followed by the DAO C++ project [10].

These approaches proposed for the C/C++ domain are not acceptable from our point of view, as they are quite expensive at runtime. This is especially true for the DAO C++ approach, where the runtime system has to be called at each potential joinpoint, regardless if there is an aspect registered for this joinpoint or not. The MicroDyner approach avoids these costs by on demand weaving in the machine code, which should perform much better at runtime. However, the machine code itself has to be prepared to support dynamic weaving. At each joinpoint position, some NOP bytes have to be reserved for the potentially inserted jump to the aspect code. For hundreds or thousands of potential joinpoints this sums up to a remarkable amount of memory. Furthermore, for the joinpoints to be visible at machine code level, the compiler must not optimize or inline any part of the code. And finally, this is a machine- and compiler-specific technique and therefore not practicable for the broad variety of hardware platforms in the domain of (deeply) embedded systems.

It is important to understand, that dynamic weaving is *always* expensive compared to dynamic loading of classes or libraries. Dynamically loaded aspects can potentially affect joinpoints in the *whole system*, while the (relatively few) junction points of a dynamically loaded library are usually well known in advance.

To overcome these disadvantages, we propose a family-based approach of constructing application specific dynamic weavers from a family of weavers [11]. These weavers are based on the technique of runtime aspect registration, but tailored down for the requirements of a specific application's profile. The following sections will explain what this approach means in the context of dynamically adaptable operating systems.

### Low Cost Dynamic Weaving

All dynamic aspect weaver implementations we have examined provide a fixed set of AOP features that can be applied at any potential joinpoint. For example, DAO C++ [10] supports the AOP feature to intercept the ordinary control flow before and after the execution of functions (before/after execution advice). There are no restrictions concerning the set of functions (joinpoints) potentially affected by dynamically loaded aspects, it might be *any* function. On the one hand, this is a desirable feature. On the other hand, the generated calls from every function to the runtime system are quite expensive. Even functions that will never be affected by aspects are slowed down and require more memory space.

The operating system family we are envisioning here should be scalable in its resource consumption depending on the required system features. This idea is now extended to the dynamic weaving support provided by the system, which leads to the feature diagram shown in Figure 3. Following this approach the weaver construction is parameterized with specific environment constraints, which are defined by a feature selection.

For example, in the domain of embedded devices the set of classes, and thereby the set of available joinpoints, is usually known in advance ("JoinPoints Known"). Hence, it is possible to match aspects already at their compile-time to the set of joinpoints they later need to be registered for. Furthermore, it is often possible to explicitly filter the huge set of available joinpoints to a quite small subset that "makes sense", like the potential points of interest for system strategies and other crosscutting concerns ("JoinPoints Filtered"). If even the set of potential aspects is known in advance ("Aspects Known"), it is possible to generate such a filter automatically from their pointcut descriptions. Furthermore, in this case the maximum number of registered aspects for each joinpoint can be pre-

calculated, so it is possible to fix the size of the runtime advice lists associated with each joinpoint, and thereby omit the necessity for using costly dynamic data structures. One more benefit is, that the order of aspect execution can be defined and resolved statically, if all aspects are known in advance.

The general idea is, to incorporate *a priori* knowledge about the system and its execution environment to tailor down the dynamic weaver infrastructure. This can drastically reduce the costs (in terms of performance and memory consumption) of dynamic aspect loading. If the set of effective joinpoints is small, it should even be feasible to implement dynamic aspect loading as efficient as dynamic class loading.

### Server-Side Weaving

We believe that dynamic weavers can be tailored down *without* giving up configurability. However, there will always be a trade-off between flexibility and cost reduction. Major or unanticipated changes to the system may not be covered by the tailored dynamic weaver. This is especially true for strictly resource constrained systems, where it might be necessary to limit dynamic weaving to very few aspects only, or to omit it at all. In these cases, server-side weaving might be an option.

The idea of *server-side weaving* is, to replace the image of the whole system (or a major system component) by another one that was built according to the new feature set. The new image is downloaded from a server which provides images for all possible configurations in a database, or just compiles the image for a requested configuration on demand. While the idea of replacing the whole image seems to be a kind of "brute-force approach", it can be quite efficient for small devices like cell phones. As the operating systems are tailored down and mostly bound statically, their images should not be larger than a few kilobytes. This can be transmitted and flashed in seconds. Under the assumption of infrequent changes, this reconfiguration has to be done only rarely, e.g. a few times per day.

### Configuration Transition

An important issue for on-demand reconfiguration is the transition of state. The runtime replacement of classes or aspects might involve significant changes of the system data structures. This typically leads to serious problems on the object instance level, as instances of different class versions may coexist in the system. These problems are even harder in multithreaded environments. Overall, it is nearly impossible to find a general solution for doing structural modifications on a living non-stateless system without giving up correctness.

For this reason, we follow a more pragmatic approach. The idea is to suspend the system during the reconfiguration process. Because the systems are relatively small, the whole cycle of suspending, reconfiguration, and restarting the device should take only a few seconds, which is clearly acceptable in the domain of smart devices. It requires restartable applications and a persistency mechanism for all affected kernel objects. Furthermore, it might be necessary to track entering and leaving of threads at module boundaries, to find checkpoints where a particular module can safely be suspended and exchanged.

## Dynamic Weaving with AspectC++

The operating system code has to be prepared to support "low cost" dynamic weaving or the replacement of coarse-grained

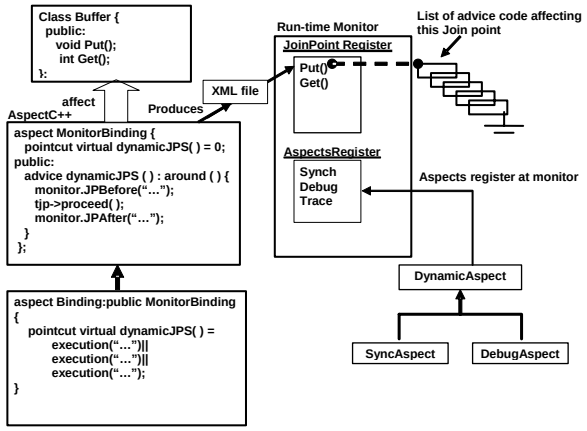


**Figure 4** AspectC++ dynamic weaving framework

modules at runtime, which is necessary for server-side weaving. For instance, server-side weaving requires a loose coupling between replaceable modules. An indirection between the modules is needed wherever the module boundaries could be crossed. This and other similar preparations can be seen as a crosscutting concern. Therefore, it is no surprise that the static AspectC++ weaver is a helpful tool for implementing the dynamic weaving infrastructure.

Another good example for this idea of "dynamic weaving by static weaving" is the runtime weaver binding by a static aspect as already shown in Figure 3. Here the idea is that all *potential* dynamic joinpoints can be described by a static aspect implementation, which is responsible for the invocation of a central dynamic aspect manager component, whenever a dynamic joinpoint is reached. This technique is illustrated in Figure 4. Here the aspect "Binding" transparently implements the connection between the component code "class Buffer" and the "Runtime Monitor" where all dynamic aspects are registered. By changing the "Binding" aspect it is easy to control the set of affected potential dynamic joinpoints. This set can be tuned according to advance knowledge and features selected in our feature diagram (Figure 3) to reduce the overhead, which is related to our weaving infrastructure.

The static AspectC++ weaver generates an XML-based report about the joinpoints, which were affected by the compiled aspects. In Figure 4 this information is used by the "Run-Time Monitor" to create data structures for each potential dynamic joinpoint. In the presented system configuration this data structure is a list used to maintain aspect code registered dynamically for the joinpoint. Currently the dynamic aspects are simple classes written manually in C++. As soon as some dynamic aspect is registered at the runtime monitor, the list of joinpoint data structures are traversed to find out which joinpoints are affected by this aspect. A pointer to the aspect code (implemented as C++ methods) is added to the list of each affected joinpoint. Once a certain joinpoint is reached by a thread of control the runtime monitor is invoked and the registered aspect code is executed.

Currently, we are extending the compiler for AspectC++ to transform aspect implementation into classes compatible with

our new infrastructure for dynamic weaving. This allows system developers to use the same expressive aspect language for both static and dynamic aspects. By following this *single language approach* the decision whether an aspect is static or dynamic can be postponed until configuration time. It can then be driven by the trade-off between the application requirements and performance.

## 4. Related Work

There is some work going in the direction of applying aspect-oriented approaches to re-engineer parts of the existing operating systems [2]. However, this work is purely based on static aspect weaving techniques. There is not much work in the direction of using AOP for the *dynamic* adaptation of operating systems. Netinant [12] proposed a proxy based approach for adapting operating systems at runtime. Although this work is also based on run-time aspect registration, our approach is significantly different, because static aspect weaving is used for binding the joinpoints to the runtime system. This provides transparency for the component code programmers, which is an important property of true AOP.

There are other approaches for dynamic adaptation by means of dynamic weaving which are mostly Java based and only a few in the C++ domain. These approaches have already been briefly discussed in section 3.

## 5. Conclusion

In this position paper, we presented our ideas about adaptable aspect-oriented operating systems by making use of the program family concept and dynamic aspect weaving. We motivated the need for dynamic weaving from the structure of our family implementation, where each feature is implemented as a module. If the feature represents a crosscutting concern, the module might be an aspect. Consequently, if we allow feature selections to be changed at runtime, we have to support dynamic aspect weaving.

To reconcile our demand on minimal resource usage with (inherently expensive) dynamic weaving, we presented the idea of a configurable weaver family that exploits *a priori* knowledge about possible system changes. Parts of this family have already been implemented. Additionally, we sketched another variant called server-side weaving, which we consider to be applicable in many cases.

Moreover, a single language approach has been suggested by which all aspects, static or dynamic, could be expressed in the same high-level aspect language AspectC++. The AspectC++ language has already been used in several research projects and a compiler is available from www.aspectc.org.

Future work will mainly be a proper evaluation of our weaver family with measurements that prove the assumption that we can scale the resource consumption with the actual requirements on dynamism and that even in resource constrained environments at least some level of dynamic weaving can be supported. In parallel we will step-by-step extend our CiAO operating system family [3], which should become the demonstrator for the technology present in this paper.

## 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.* Proceedings of ISORC'99, May 1999, St Malo, France

[2] **Y. Coady, G. Kiczales, M. Feeley, N. Hutchinson, and J. S. Ong:** *Structuring Operating System Aspects*. CACM, pp. 79–82, October 2001.

[3] **D. Lohmann and O. Spinczyk:** *Architecture-Neutral Operating System Components.* WiP Session on SOSP'03, October 19th-22nd, 2003, Bolton Landing NY, USA.

[4] **A. Popovici, T. Gross, and G. Alonso:** *Dynamic Weaving for Aspect Oriented Programming*. Proceedings of AOSD'02, April 2002, Enschede, The Netherlands.

[5] **O. Spinczyk, A. Gal, and W. Schröder-Preikschat:** *AspectC++: An Aspect-Oriented Extension to C++*. In Proceedings of TOOLS Pacific'02, February, 2002, Sydney, Australia.

[6] **R. Pawlak, L. Seinturier, L. Duchien, and G. Florin:** *JAC: A flexible framework for AOP in Java***.** Reflection 2001

[7] **S. Aussmann and M. Haupt:** *Axon – Dynamic AOP through Runtime Inspection and Monitoring*. First workshop on advancing the state-of-the-art in Runtime inspection (ASARTI'03), 2003

[8] **Y. Sato, S. Chiba, and M. Tatsubori:** *A Selective, Just-In-Time Aspect Weaver*. Proceedings of GPCE'03, 2003, Erfurt, Germany.

[9] **Y. Chen,** *Aspect-Oriented Programming (AOP): Dynamic Weaving for C++,* Master thesis, August 2003, Vrije Universiteit Brussel and École des Mines de Nantes

[10] **S. Almajali and T. Elrad:** *A Dynamic Aspect Oriented C++ Using MOP with Minimal Hook.* Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS Technical Report 04.01, March, 2004, Lancaster, UK.

[11] **W. Gilani and O. Spinczyk:** *A Family of Dynamic Weavers.* Proceedings of the 2003 Dynamic Aspect Workshop (DAW04 2003), RIACS Technical Report 04.01, March, 2004, Lancaster, UK.

[12] **P. Netinant, C. A. Constantinides, T. Elrad, and M. E. Fayad:** *Supporting the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks*. Proceedings of the International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA'00), pp. 271-278, June 2000, Las Vegas, NV, USA.

[13] **K. Czarnecki and U. W. Eisenecker:** *Generative Programming – Methods, Tools, and Applications.* Addison-Wesley, 2000