Engineering Reliable Service Oriented Architecture:

Managing Complexity and Service Level Agreements

Nikola Milanovic
Model Labs – Berlin, Germany

Information Science
REFERENCE

Published in the United States of America by
Information Science Reference (an imprint of IGI Global)
701 E. Chocolate Avenue
Hershey PA 17033
Tel: 717-533-8845
Fax: 717-533-8661
E-mail: cust@igi-global.com
Web site: http://www.igi-global.com

Library of Congress Cataloging-in-Publication Data

Engineering reliable service oriented architecture : managing complexity and
service level agreements / Nikola Milanovic, editor.
p. cm.
Includes bibliographical references and index.
Summary: "This book presents a guide to engineering reliable SOA systems and
enhances current understanding of service reliability"--Provided by publisher.
ISBN 978-1-60960-493-6 (hardcover) -- ISBN 9781609604943(ebook) 1.
Service-oriented architecture (Computer science) 2. Computer networks--
Reliability. I. Milanovic, Nikola.
TK5105.5828.E54 2011
004.6--dc22
2010033596

British Cataloguing in Publication Data
A Cataloguing in Publication record for this book is available from the British Library.

Chapter 5
Configuration of Non–Functional Properties in Embedded Operating Systems:
The CiAO Approach

Wanja Hofer
Friedrich–Alexander University Erlangen–Nuremberg, Germany

Julio Sincero
Friedrich–Alexander University Erlangen–Nuremberg, Germany

Wolfgang Schröder-Preikschat
Friedrich–Alexander University Erlangen–Nuremberg, Germany

Daniel Lohmann
Friedrich–Alexander University Erlangen–Nuremberg, Germany

ABSTRACT

In embedded operating systems (OSes), non-functional properties like reliability, performance, or memory footprint are of special importance. State-of-the-art OS product lines focus on the configurability of functional characteristics of the system. This chapter proposes an approach that aims at also making non-functional properties indirectly configurable and maintainable by the system configurator. In order to reach this goal, the CiAO OS product line used here has configurable architectural properties, which have no functional influence on the target system, but instead bear an impact on its non-functional properties. Additionally, the chapter develops a feedback approach that gains information about the non-functional properties of an already configured system to assist further configuration decisions, and presents and details the CiAO approach and evaluates it using two case studies from the CiAO operating system.

DOI: 10.4018/978-1-60960-493-6.ch005

MOTIVATION

In the domain of system software, non-functional properties (NFPs) are of fundamental importance to the end user. This is because system software never has a purpose and business value of its own, but it is rather a means to aid the *application* making use of it in fulfilling its (business-value-bringing) purpose. Hence, performance, for example, is an important non-functional criterion to select a suitable operating system (OS) to deploy an application on. In the sub domain of *embedded* operating systems, NFPs can even be mission-critical since some embedded systems applications depend on the fault tolerance or a given upper bound on the latency of the underlying embedded OS. Since the desired NFPs of a piece of system software are different from application scenario to application scenario (and sometimes reflect a trade-off decision), it is the task of the OS engineer to keep the NFPs *configurable*.

In our experience, though, the consideration of NFPs in system software causes problems because NFPs can never be made *directly* configurable. That is because most NFPs are *emergent* in their nature; that is, they have no direct representation in the system's implementation entities. Instead, they result from the orchestration of the properties available in the selected configuration. Hence, NFPs can only be made configurable via *indirect* configuration of other properties.

However, the set of *functional* properties to be selected is fixed, dependent on the application. Other properties, which we call *architectural* properties (APs), are transparent to the application, though, but they still have an enormous effect on the NFPs of the resulting end system. Examples of such APs of OSes include the chosen method of interrupt synchronization in the kernel, the available protection facilities (including memory protection, for instance), or the type of interaction between kernel modules. The latter has been under heavy discussion for decades now, arguing in favor of procedural interaction in monolithic systems versus message passing techniques in microkernels (Lauer and Needham, 1979, Liedtke, 1995). The early decision to adopt one of those alternatives has a significant impact on the NFPs of performance, latency, and memory footprint, among others.

The CiAO family of embedded OSes developed by our research group was designed with *architectural configurability* in mind; that is, even fundamental architectural properties are kept configurable in CiAO's design. Hence, the decision in favor of one or the other shape of an AP is postponed until the configuration stage and therefore left to the system configurator. He can then choose the one configuration option that has the best desired impact on the NFPs of the target system, effectively tailoring the OS (and its architecture) to the needs of the application scenario.

However, if the variability in a software product line exceeds a certain threshold (e.g., by offering architectural variability like in CiAO), the number of transparent configuration options left open to the configurator quickly becomes overwhelming. We therefore also propose a new kind of development process with a feedback approach, which gathers additional knowledge by analyzing product variants regarding their NFPs. It is thereby possible to assist the configurator in making his configuration decisions when aiming at optimizing a specific NFP.

The domain of embedded system software is one that has been concerned with non-functional properties for a long time being. Embedded systems engineers have gathered a lot of knowledge on how to deal with those properties, knowledge that the domain of service-oriented architecture can benefit from.

Structure of the Rest of the Chapter

The remainder of this chapter is structured in a top-down manner. First we give background information and definitions necessary for the understanding of the text. Then, an overview of our CiAO approach and, after that, its details are presented. The following section details the evaluation studies regarding the influence of APs on NFPs. We then present work that is related to our approach, and the chapter is concluded in the last section.

BACKGROUND

Our Understanding of Non-Functional Properties

To define what exactly non-functional properties (NFPs) are is a delicate task. There is no standard definition in the software engineering community available, and different groups inside the community have contradicting definitions. Moreover, even the nomenclature used is not uniform; for example, in different contexts the same kinds of software properties are termed NFPs, quality attributes (Bass, 2006), or soft goals (Cysneiros and do Prado Leite, 2004), among others. Amongst a myriad of examples of such properties, they include security, reliability, safety, performance, maintainability, usability, and code size, to just name a few.

Moreover, many of these terms are broad and generic; different stakeholders may have a varying understanding (and, therefore, expectations) of such properties. We believe that when dealing with such properties, it is necessary to define what is considered to be an NFP in that specific context. For this reason, we narrow NFPs down with the following definition.

Non-functional properties of a software system are those properties that do not describe the principal task or functionality of the software, but can be observed by end users in its run-time behavior (Lohmann et al., 2005).

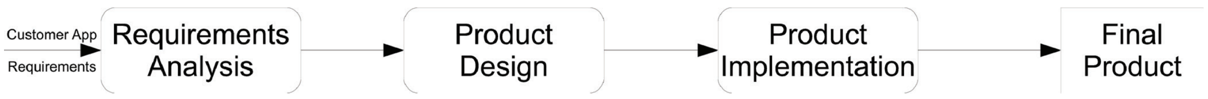
This definition gives good insights into the type of properties that we see as NFPs. It is very well applicable to our domain of families of operating systems; however, it can also be applied to a number of other domains, especially those of infrastructure software (e.g., middleware, database systems, etc.).

The goal of our work is to show how we address the configuration of properties that fall under these definitions. Approaching such properties is a challenging task mainly due to two reasons. On one hand, a primary goal is to improve the understanding of NFPs already at the stage of software configuration. In a perfect scenario, this would be to provide the system configurator with means to express the non-functional requirements on the product. On the other hand, after having developed several families of operating systems, we have learned that many NFPs are *emergent*. That means that they are the result of the *interaction* of many components, which effectively hinders the possibility of *direct* configuration of such properties. Therefore, our techniques presented in this chapter aim at closing this gap. The complex interactions that will influence the system's NFPs cannot be appropriately predicted at design stage, which makes attempts of preparing configuration mechanisms during design not very reliable. As a result, we have decided to extend our approach to tackle NFPs not only during the design stage, but also in other development stages, and even *post* implementation. The idea is to learn from configured and running systems, and to gather information how the system's components are interacting, and how this interaction influences the investigated NFPs. Subsequently, this information should be used to improve the configuration of future systems.

The Classical and the SPL Software Development Processes

In order to be able to address NFPs in a thorough and holistic way, we have developed an own software development process, the CiAO development process, which is detailed in the main section of this chapter. It is based on the classical software development process and the process proposed for

Figure 1. The classical software development process



software product lines, both of which are briefly introduced here.

Figure 1 shows the main stages of the traditional software development process. The customer expresses his requirements on the desired product in an appropriate way; those requirements are then analyzed by the software engineer, enabling him to develop a software architecture. This design is then implemented using the languages and platforms that are appropriate for the target environment. If another customer asks for a similar, but different product, these stages are basically repeated, and the new product is developed from scratch.

This problem is tackled by the canonical software *product-line* (SPL) development process (see Figure 2). Here, not a single product is considered, but a whole family of products targeting a specific domain.

In the *domain engineering* process (upper half in Figure 2), the product line itself is developed, while in application engineering, a specific product is built from the outcome of the domain engineering without much effort. First, domain experts having comprehensive domain knowledge scope the domain and specify the desired variability and configuration options (domain analysis). This variability is often expressed in a feature model containing a feature diagram representing the configuration space. After this, a reference architecture is built during the domain design step, followed by the domain implementation. The assets that constitute the product-line implementation are stored and described by a family model.

In *application engineering* (lower half in Figure 2), the customer's demands on the product are first investigated in a requirements analysis step, which results in a feature selection in the feature model previously developed in the domain engineering process. This feature selection can then be used to automatically derive a final product variant using the family model and the product-line assets. Hence, those assets are then re-usable across multiple products in the product line, enabling an advance in important factors like time to market and product quality, for instance.

THE CiAO APPROACH

CiAO (CiAO is Aspect-Oriented) is a family of OSes targeting the embedded systems domain,

Figure 2. The software development process as proposed by the software product-line community. (adapted from (Czarnecki and Eisenecker, 2000))

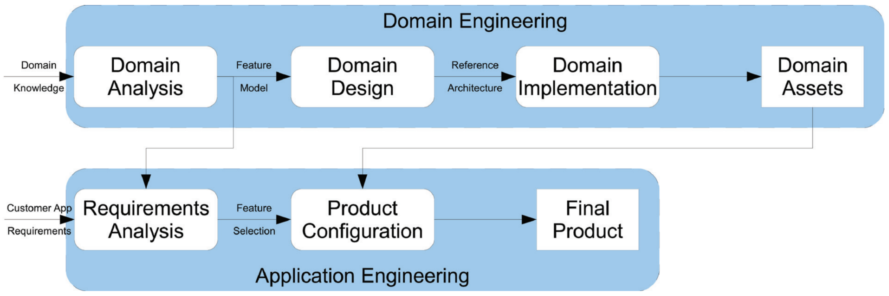
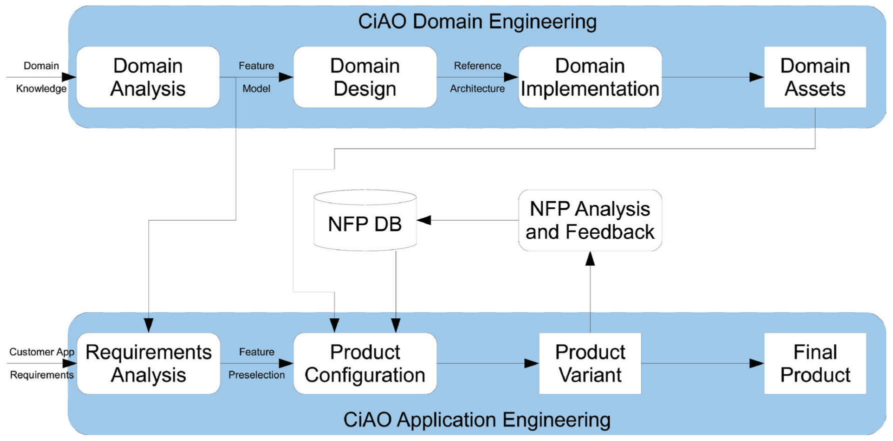


Figure 3. The CiAO software development process



especially those systems deployed on microcontrollers in the automotive industry. The main goal of the project is to show that even fundamental architectural properties of an OS can be kept configurable, thereby effectively enabling the *indirect* configuration of non-functional properties. This goal is reached by using techniques from the field of software product-line engineering in the analysis stage, and by developing aspect-aware design patterns in order to be able to deploy aspect-oriented artifacts in the implementation. This way, CiAO reaches a very good separation of concerns while providing very fine-grained and deep configuration possibilities at the same time. By selecting a specific combination of architectural property configurations, it is possible to get an architecture that is optimized according to the non-functional requirements of a particular scenario. However, since those NFPs do not emerge until the production configuration stage, a specialized development process is needed to treat NFPs as "first-class citizens" in *all* development stages.

Overview of the CiAO Software Development Process

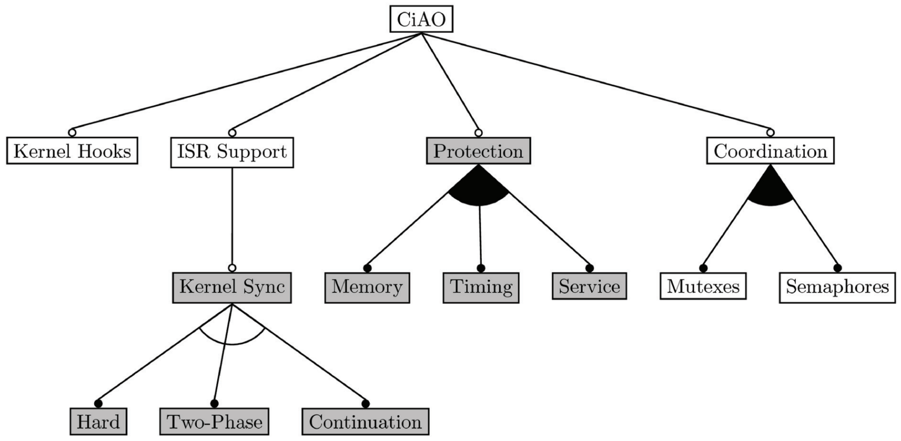
We have adapted the canonical software product-line development process (see also the previous section) to better address non-functional properties. The resulting CiAO software development process is depicted in Figure 3.

We enriched the steps in the layers of domain engineering and application engineering, which are described in the following two sub sections. Furthermore, we introduced additional steps in a feedback loop that has access to a database of non-functional properties of the product line. These additions are described in the remaining sub sections.

CiAO Domain Engineering

Domain engineering in the CiAO development process is similar to the steps proposed in software product-line engineering. However, those steps now encompass the consideration of non-functional properties from the very beginning. This is done by extending the domain knowledge by features that are *transparent* to the application developer—because it is mostly *those* features that have a significant *indirect* impact on the perceived non-functional properties of the target system. Thus, additional configuration options are considered in the domain analysis and offered in the feature diagram that would not be examined in the classical approach.

Figure 4. An excerpt of the CiAO feature diagram. Features that are functionally transparent to the application are depicted in gray color.



In the CiAO family of operating systems, most of those additional options refer to the configurability of CiAO's architecture. By keeping those architectural properties configurable in the domain design, the system configurator can still decide in favor of one flavor or the other, depending on the desired focus on the system's non-functional properties. Two of those properties—interrupt synchronization and memory protection—and their configurable influence on some non-functional properties is presented in the next section.

Another focus of the CiAO research project is to show that architectural configurability can still be maintainable in the implementation. In order to reach this goal and because many architectural properties are highly cross-cutting in the design and implementation, CiAO is implemented using aspect-oriented programming (AOP) techniques. In particular, the implementation language and aspect weaver AspectC++ (Spinczyk and Lohmann, 2007) is used, which is a superset of the C++ programming language. With this programming paradigm, we showed that is possible to have maintainable, concern-separated system software code that has configurable architectural properties in its domain implementation (Lohmann et al., 2007b, Lohmann et al., 2007a).

A small excerpt of the CiAO feature diagram, which resulted from the domain analysis step, is shown in Figure 4. It is referenced throughout the rest of the chapter to give examples of some of the steps performed in the development process.

CiAO Application Engineering

Application engineering in the CiAO development cycle also differs from the one in SPL engineering. The system configurator, who translates the customer requests into selections in the feature model provided by the domain analysis step, will not be able to make *all* configuration decisions. That is because CiAO aims at providing options that are transparent to the application, like those configuration options pertaining to the system architecture. Hence, the output of the requirements analysis step can only be a *pre*selection of features, and it is only a sample variant that is configured and generated first.

After that, our feedback approach comes into play. The generated sample variant is analyzed for its NFPs and checked against the NFP database, which results in an altered feature selection and therefore product variant. This iteration is repeated until the final product conforms to the desired non-functional requirements, or until the best solution is reached within given time constraints. The exact procedure of the CiAO configuration process, the feedback approach, and its application is discussed in the following sections.

The CiAO Configuration Process

As CiAO is a software product line, a member of this family of systems (i.e., a concrete solution) is derived according to a given specification (i.e., the feature selection) that conforms to the formal feature model. This transformation process is driven by a family model, which plays a central role. It is responsible for mapping the selected features to concrete implementations units. That is, according to the set of selected features, the corresponding components (e.g., classes, aspects, etc.) are customized (e.g., by conditional compilation, preprocessing, etc.) and then copied to another source-code tree, where it can be compiled to generate the required family member.

Feature models mostly correspond to a set of *functional* features (disclosed during domain engineering) that can be turned on and off, whereas the set of valid configurations is determined by the feature tree hierarchy and its extra interdependencies. However, it is often the case that several features define the same *interface* for a specific service, but derive different *implementations* for the service provided by the interface. This means that, from the user's perspective, the difference between the several features implementing the same service can only be distinguished by its effect on the system's NFPs. In traditional approaches, during the software configuration the system configurator either has to have strong knowledge about the internals of the system, or he has to have assistance from the developers in order to decide among features that implement the same interface. The root of the problem here is the missing information about the effect of otherwise identical features on the system NFPs.

In order to address this issue, we augment the feature model with non-functional information, which can be obtained from two different sources. First, from the system designers, who made the architectural decisions and are aware of the possible impact of features on NFPs. (Architectural decisions are often a trade-off between two or more NFPs). Second, from tests performed on generated family members. Even though the information provided by the system designers may be, to some extent, helpful, we believe that performing tests is the appropriate approach. Information from tests is not only able to assist the configuration process, but also assures that the implementation conforms to the expected behavior that motivated the design decisions. In short, it is real data, and it helps software *evaluation* and *evolution*, as it is able to reveal flaws in the design and the implementation.

THE FEEDBACK APPROACH

As we have detailed in the previous sections, many NFPs cannot be appropriately predicted at design time and, hence, not be directly configured at configuration time. Therefore, we believe that in order to get useful information about the system's NFPs, the use of tests on generated products is a promising alternative. The real behavior of many types of NFPs can not be detected until after the family member is configured and generated. That is, information regarding the interaction among the components that comprise the entire system can only be observed after the system is prepared to be deployed. For example, insights about the RAM footprint or code size can be gained from *static* tests performed after generation. Additionally, information about latency or performance can be captured by performing *dynamic* tests on the running system variants.

The feedback approach extends the traditional SPL development techniques in order to provide information regarding NFPs during product configuration. We introduced new structures and mechanisms so that the SPL infrastructure can be used to generate products that will be tested against the NFP that is to be investigated. This information is saved, organized, and re-inserted in the SPL process (see also Figure 3). It also enables the user to benefit from it in the configuration of *further* products. This feedback process is organized in three layers:

1. The SPL Repository comprises the software components that can be assembled together to generate products (see Figure 3, Domain Assets). Additionally, components that are used merely to *capture* non-functional information from generated products (like performance measurement aspects that instrument the product) are also available. We have shown that *aspects* are very adequate for this task (Gilani et al., 2007).
2. The User Configuration is responsible for providing the mechanisms for product configuration. Besides the traditional configuration process (selecting features from a feature model), we provide the user with non-functional information (see Figure 3, Product Configuration). As NFPs are very specific to each product, or even to each feature, this information can be displayed in different graphical ways, for example, sliders, graphs, charts, etc. Moreover, during configuration the user can select the aforementioned components that are responsible for measuring some of the NFPs of the product.
3. The Concrete Solution Domain encompasses the generated product, the compiling environment, and the run-time environment used to generate and test the product (see Figure 3, Product Variant).

The mechanisms described so far are appropriate for generating and testing single family members. That is, after the configuration process, the system configurator is able to confirm if the generated product meets his expectations regarding NFPs. Nevertheless, this is not enough if we want to use this information to guide the configuration process, because we would need information about the entire family, and not only about single configurations.

To solve this problem, the naive approach would be to generate all possible family members; this is not a feasible solution, though. Normally, system families have several hundreds of features, and they can generate several thousands of different family members. Therefore, an exhaustive study of all members is not computationally possible. However, important for our approach is how groups of features interact with each other to influence NFPs; in many cases, several features will not have any interaction at all. This means that many tests on different configurations will produce the same measurement results for a specific NFP that is investigated. In order to avoid redundant tests, and consequently to reduce the test space, a *metric* regarding *feature interaction* is to be defined so that only tests that reveal important information regarding how the feature interaction will influence the required NFP are performed. For examples of uses of these metrics, see the following section about the CiAO studies.

Ultimately, the organization of the SPL in this fashion aims at improving the *configuration experience* in the following ways:

1. Providing exact non-functional information about products that have been previously configured;
2. Using heuristics and regression techniques to provide approximated information about products that have only been partially configured;
3. If several software components implement the same functional behavior, the user should be able to select the most appropriate one depending on their non-functional characteristics;

RESTRICTING THE FEATURE MODEL FOR TESTING

An important characteristic of families of systems is that the code base does not only represent one system, but all valid family members. However, to perform tests on this code, specific members must first be configured and then generated. Moreover, aiming at understanding how features interact in different configurations requires whole sets of family members to be generated and tested. The strategy of testing the set of *all* family members is not practicable.

Therefore, our approach provides means for the application engineer to specify sets of products to be tested. The objective of specifying tests is to avoid generating members where interactions will not influence the desired NFPs. Of course, for some NFPs this definition may not be trivial. However, for others it may be pretty straightforward. For example, consider the scenario where the NFP represents the time required for the computation of a specific math algorithm. This algorithm may be configured in different ways by means of several features, and the math algorithm itself is a feature of a complex system comprised of other algorithms that can be configured independently. If those other algorithms do not share code with the investigated math algorithm, testing each of the algorithms separately (and not all of the combinations) would reduce the testing space, and the performance measurement results of each algorithm that is tested individually would not be altered.

In order to accomplish such kinds of restrictions on feature models, we came up with the idea of *partial configurations*. In traditional feature models, a feature selection is simply a list of features that are required to be present in the member to be generated. (The generation can only be performed when the selection is valid, though.) We have extended this concept, now enabling the user to set the features that must be present (*selected features*), the set of features that must not be present (*blocked features*), and other features that may be let open (*open features*). A list of such selections of features specifies a *partial configuration*. We developed a tool that is able to generate all valid configurations that conform to the specification of a partial configuration. Our tool transforms the feature model into a binary decision diagram (BDD) and by using the BuDDy library (BuDDy Developers, 2009) we are able to generate the set of valid configurations that conforms to the restrictions described by the partial configuration.

Using this tool, the engineer is able to set the features he wants to be present in all tests (of the test set) by defining them as *selected*. The ones that are known not to have an influence on the NFP to be tested are set to *blocked*. The features that actually represent the variability among the members of the test set are marked as *open*.

Applying the Feedback Approach

In order to explain how partial configurations can be used to generate a reduced set of tests and still produce meaningful results, we will illustrate an example using the feature model depicted in Figure 4. This feature model is only a small excerpt of the complete CiAO feature model, which has around 150 different features to be configured and represents about 10,000 different valid configurations. However, even this reduced subset of features (14 features) is able to represent 256 valid configurations. This fact shows that one should be very careful when defining the set of configurations to be tested; even a small number of features may represent a high number of configurations—and, consequently, can require a prohibitive testing time.

Nevertheless, one should also keep in mind that the testing process aims at capturing feature interactions that will influence a specific NFP. Interestingly, for many NFPs it is possible (with internal implementation knowledge) to determine the set of features that will influence the required NFP. For example, in the case of performance, or also latency, one could measure the time required for a certain computation by starting a timer at a specific point in the code, and stopping it at the point where the computation is finished. If this scenario represents our NFP *performance*, it is possible to determine which features insert code between these points (the *metric* that specifies *feature interaction*), or the ones that have their code called from within the code block having its *performance* measured. That is, when testing a specific NFP, it is often possible to describe the set of features that influence the desired NFP by using the concept of partial configurations. Moreover, in our experiments we have identified that normally the *critical* parts of the code, where, for example, *latency* is important, will not be composed of too many features, and therefore a testing process can be performed in a feasible amount of time. Likewise, when a feature specifies an *interface* and its sub features represents different *implementations*, testing can also be optimized in this manner, and only the variability represented by this sub tree must be tested.

The feature model depicted in Figure 4 can generate 256 different configurations; in this case, testing all possible configurations may be feasible. However, for some NFPs it would simply be a waste of time, since for many of those tests the output would be exactly the same and would not contribute to the understanding of how features interact to influence a specific NFP. For example, if one is interested in the performance of the different configurations of the *protection* mechanism, a test set can be defined by a partial configuration where the set of features to represent the minimal infrastructure are set as *selected*, the features under *protection* are marked as *open* (establishing the variability of the test set), and the rest is marked as *blocked*. By doing so, we can reduce the test set to 8 different combinations. These are basically all possible combinations of the features used for the configuration of the *protection* mechanism, namely *memory protection*, *timing protection*, and *service protection*.

In the evaluation of this work (see the following section), we deal with the NFPs of *latency* and *performance*; both of them are *measurable* NFPs. If one is interested in using the Feedback Approach for NFPs like *reliability* or *security*, a way to quantify these properties must be found. For example, for *security* one could take a set of different testing attacks and assign the number of failed and succeeded tests to the variant of interest. Regarding *reliability*, stress testing could be employed, and the time required for the crash (or success after a timeout) can be assigned to the variants under test.

CiAO STUDIES: THE EFFECT OF CONFIGURABLE APs ON NFPs

As outlined in the approach description, CiAO aims at making the system's NFPs configurable by indirectly keeping fundamental architectural properties configurable. In the domain of embedded operating systems, this includes the following concerns, amongst others.

- **Interrupt synchronization:** If control flows can be executed asynchronously *and* they can access critical kernel state, the consistency of that state has to be ensured by some synchronization mechanism. There are several ways to ensure synchronization, all of them bearing distinct advantages and disadvantages.
- **Component isolation:** In simple embedded systems, application components can influence each other; for instance, by accessing each other's memory areas. To increase robustness, isolation means can be deployed, including constructive isolation by using type-safe programming languages or memory protection using hardware support like memory protection units (MPUs). Other isolation techniques refer to *temporal* isolation so that applications in real-

time systems will not miss their deadlines because of other (potentially misbehaving) applications.

- **Kernel interaction:** The way components inside the kernel interact with each other has a significant impact on the perceived robustness and performance. Historically, suggested approaches range from monolith-like systems to microkernel systems (Liedtke, 1995).

Interestingly, the focus on configurability of those architectural concerns is increasing in the recent time. Proof for that is the specification of AUTOSAR OS (AUTOSAR, 2006a, AUTOSAR, 2006b), which aims to standardize the system software present on automotive microcontrollers. This specification includes several so-called *scalability classes*, which provide distinct isolation properties like memory protection and timing protection. Hence, those architectural properties are already prescribed to be implemented in a configurable way in this standard.

We have implemented two configurable architectural properties in CiAO—interrupt synchronization and memory protection—, which are presented in the following, together with an evaluation of their impact on different non-functional properties.

CASE STUDY 1: INTERRUPT SYNCHRONIZATION

Short Domain Analysis

Our first case study is based on the configurable architectural property of interrupt synchronization; that is, on how to keep the kernel state synchronous when possibly being accessed from within asynchronous events (e.g., interrupt handlers). After having analyzed the domain of operating systems, we found that there are three basic models to achieve interrupt synchronization, all of which are implemented (and therefore configurable) in CiAO.

1. **Hard synchronization:** This model lowers every critical section accessing kernel state to interrupt level, implemented by enabling and disabling interrupts or by setting the interrupt level accordingly. Hard synchronization has a very low performance overhead, but risks higher event-handling latency, depending on the length of the critical sections. This model is widely deployed in relatively simple embedded operating systems.
2. **Two-phase synchronization:** This model divides each interrupt handler into two different parts, named prologue and epilogue in CiAO. Prologues can be executed in a timely fashion with low latency, but they are not allowed to access critical kernel state. This is done in the corresponding epilogue, which can be delayed by the kernel when accessing critical state. Epilogues have priority over user threads, however, and user threads are only executed when no epilogues are pending. Two-phase synchronization is used in many desktop operating systems like Linux or Windows, but also in embedded OSes like OSEK OS (OSEK/VDX Group, 2005) or AUTOSAR OS (AUTOSAR, 2006b).
3. **Continuation synchronization:** This model enhances epilogues to full continuations (the thread abstractions in CiAO) with a context of their own. This way, user threads can be activated while interrupt handlers are waiting for a shared resource. This facilitates fine-grained locking of kernel components (having a positive impact on the perceived latency), but bears the highest performance and memory overhead of the three models. The Solaris OS uses a similar model to synchronize its kernel (Kleiman and Eykholt, 1995).

CiAO Design

We designed a generic driver model that leaves the driver developer unaware of the finally deployed interrupt synchronization method, which is not determined until the system configuration time. This way, the driver obeys common handler interfaces, and is adapted by aspects to fit the configured synchronization model. This configurability is designed and implemented with full separation of concerns, enabled by considering aspects as a design means from the beginning of the CiAO engineering process (*aspect awareness*). The design achieves a complete separation of concerns, separating the three dimensions of *what*, *where*, and *how* to apply interrupt synchronization.

For details concerning the aspect-aware design of configurable interrupt synchronization in CiAO and how the driver and OS components are integrated, please refer to (Lohmann et al., 2007b).

Influence on Non-Functional Properties

After having implemented the architectural property of interrupt synchronization to be configurable in CiAO, we investigated the impact of this variability dimension on the non-functional property of latency. Here, latency is understood as the elapsed time between the beginning of the hardware interrupt handler and interrupt termination (*iret* instruction). We also measured the time from the beginning of the handler until the first *prologue* instruction, and until the first *epilogue* instruction, respectively. The results are depicted in Table 1. We have tested only the features that can have its implementation code accessed (function call, data structure changes, etc.) from the *block of code* defined by the boundaries of our time measurement routines; in this case, those features are the different interrupt synchronization variants. This is the *metric* that represents the feature interaction for this NFP in order reduce the size of the test space (see also the previous section about the feedback approach).

Table 1. Measurements of the non-functional property of latency for several configurable interrupt synchronization methods in CiAO. Measurements were performed on a TriCore TC1796b running at 50 MHz with a hardware trace analyzer (Lauterbach). The results were measured (and turned out to be stable) over 10 iterations.

ns	$t_{prologue}$	$t_{epilogue}$	t_{iret}
Hard Sync.	160	160	320
Two-Phase Sync.	160	800	1200
Continuation Sync.	320	1200	2160

One can clearly see that the chosen IRQ synchronization method has a significant impact on the latency of interrupt handlers in the system. This refers both to the time until completion and to the time until the first part (prologue) and the second part (epilogue) of the interrupt handler are executed. However, by choosing the continuation synchronization model, for example, it is possible to reach more fine-grained synchronization domains inside the kernel, leading to less contention and, therefore, to better performance in certain situations. Likewise, choosing two-phase synchronization over hard synchronization leads to lower interrupt locking times, which makes it less likely to lose interrupt signals, depending on the microcontroller architecture.

Hence, the architectural property of IRQ synchronization is transparent to the application developer *functionally*, but it has a significant impact on the emerging *non-functional* properties of the resulting system, latency being among them.

Case Study 2: Memory Protection

Short Domain Analysis

The second study we performed is concerned with an architectural property that is prescribed by the AUTOSAR embedded software standard: memory protection. This means of spatial isolation between applications is supposed to increase robustness by limiting the memory access of application components to legal ranges, disallowing access to the memory areas of other applications or the underlying system software. Depending on the chosen AUTOSAR scalability class, memory protection is to be provided by the operating system or not.

On the hardware side, memory protection is enforced by an MPU (memory protection unit), which bears reprogrammable range registers specifying the access rights to distinct memory areas. An MPU is a simplified variant of a full-featured memory management unit (MMU) known from PC systems, which can additionally perform paging. The MPU is only reprogrammable in the supervisor mode of the CPU so that only the OS and not the applications are able to alter memory protection properties.

During the domain analysis step, we found the following protection models to be suitable to be designed and implemented in CiAO.

1. **No protection:** This trivial variant does not feature any protection mechanism at all. However, there is no performance overhead either.
2. **Kernel protection:** This version separates the kernel from all applications by providing two separate protection domains.
3. **Application protection:** This model additionally separates the applications from each other, comprising *n* protection domains for *n* deployed applications, plus one domain for the kernel. This means that invocations of functions that are exported by another application involve some overhead for adjusting the access privileges.
4. **Task protection:** This most fine-grained model even protects task-local data like the task stack from the modification by other tasks or interrupt handlers in the same application.

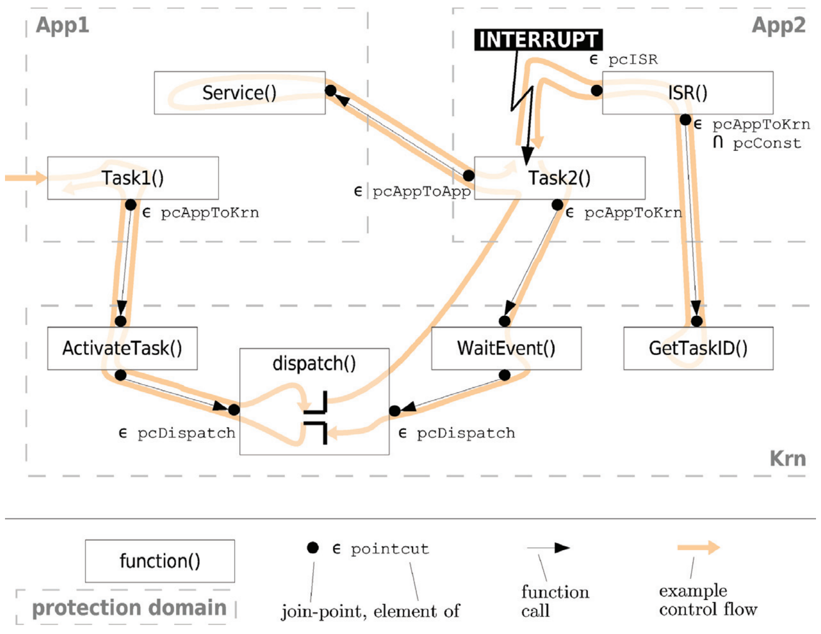
CiAO Design

As with the configurable property of interrupt synchronization, the application developer is oblivious of the finally deployed memory protection method (if any) as chosen by the system configurator. Depending on the applied protection model, different points in the control flow in the CiAO system must be affected by an appropriate reprogramming of the MPU. Since CiAO's design is aspect-aware, those points are exposed as *join points* in the AOP sense, being advisable by configurably deployed aspects. This way, generic pointcuts representing the critical points like inter-application function calls, application—kernel transitions, interrupt handler invocations, or task dispatches can be supplied. An example control flow in a CiAO system is depicted in Figure 5.

The aspects implementing the protection method to be enforced are then implemented by advising the exposed join points in the form of the supplied pointcut expressions. The kernel protection aspect, for instance, reprograms the MPU to allow access to kernel state whenever entering the kernel and disallowing it upon exit from the kernel, including the first-time dispatch of tasks. Application protection, however, additionally needs to reprogram the MPU upon inter-application calls and upon inter-application task dispatches.

The CiAO reference hardware platform—the Infineon TriCore TC1796b—allowed us to provide an additional configuration variant called the *semi-trusted mode*. This variant exploits the peculiarity that on the TriCore platform memory protection is not implicitly disabled in supervisor mode. This allows applications to be run in supervisor mode with memory protection enforced without the need for kernel traps. An offline analysis can ensure that the applications do not reprogram the

Figure 5. Control flows and memory protection domains in an example CiAO system



MPU in their code, thereby ensuring the safety of the system.

For a more detailed explanation of CiAO's memory protection design and the way applications interface with the OS components, and for implementation details of the generic pointcuts and the configurable aspects in AspectC++, please refer to (Lohmann et al., 2007a).

Influence on Non-Functional Properties

We evaluated the influence of the different memory protection models on the number of clock cycles needed for the execution of several characteristic functions; that is, the non-functional property of performance. All of those investigated functions either cross an inter-application boundary or the one between the kernel and an application (see Figure 5). The results are listed in Table 2 and shortly discussed here. The *metric* used to define feature interaction, and the corresponding test space, is analogous to the one presented in the previous section; that is, we tested the features that have its code accessed from the *blocks of code* that were measured. These features were the different kinds of memory protection in this case.

- Since GetTaskID() is a non-modifying kernel function, the memory protection domain is not switched in any of the protec-

Table 2. Measurements of the non-functional property of performance of selected representative function calls for several configurable memory protection variants in CiAO.

ns	GetTaskID()	ActivateTask()	dispatch()	Service()
No Protection	3	24	88	0
Semi-Trusted Mode	3	43	148	89
Full Protection	3	86	148	174

tion models. Hence, the clock cycles remain constant.

- The ActivateTask() system call, however, modifies kernel structures. No memory protection does not involve any overhead, whereas the semi-trusted variant needs 43–24=19 extra clock cycles to reprogram the MPU to allow write access to kernel space and disallow it after the call. A full trap with switch to supervisor mode and back to user mode costs 86–24=62 additional cycles.
- The kernel-internal dispatch() function switches the protection domain from one application to another in both protected modes, leading to a cost of 148–88=60 cycles. Since this function is invoked inside the kernel, there is no difference between the semi-trusted and the standard protected mode.
- An inter-application call like Service() does not cost anything without protection; the call can even be inlined. In the semi-trusted mode and the application protection configuration, the MPU is reprogrammed to the new application domain and back after the call, which is worth 89 cycles. The version that requires a full trap costs 174 cycles in total.

The architectural property of memory protection has a big influence on the performance of a system, as can be seen in the configurable implementation of the property in CiAO. The decision in favor of one of the protection methods and variants to be deployed is effectively a trade-off between the two non-functional properties of safety and performance.

RELATED WORK

As this work aims at tackling non-functional properties in *all* stages of development, there is a broad range of related work, which is discussed in the following.

Non-Functional Properties and Software Product Lines

Siegmund et al. propose the use of a *semi-automated derivation* (SAD) to assist developers in selecting product features in SPLs with a large number of features (Siegmund et al., 2008, Rosenmüller et al., 2008). The basic idea is to hide variation points that are irrelevant due to non-functional requirements that should be met. They claim that traditional approaches do not consider non-functional properties or alternatives for a feature implementation. To solve this problem, they present an integrated software product line model (ISPLM), which integrates code units and their non-functional properties into the feature model.

(Benavides et al., 2005) propose an extension to feature models (as proposed by Czarnecki and Eisenecker, 2000) to accommodate information about *extra-functional* features (NFPs, in our view). In this approach, attributes like price or development time can be assigned to features. The features and their attributes are transformed to a constraint satisfaction problem (CSP) so that an automated reasoning can be applied to it. Hence, optimal products can be generated according to a determined criterion (an extra-functional feature).

These methods take advantage of information about NFPs to improve product configuration. However, we think that both are more related to the field of variability *management* (Loesch and Ploedereder, 2007, Schirmeier and Spinczyk, 2007). They ease the configuration process in large SPLs but do not offer the ability to explicitly configure NFPs or to inform the user about the *real* influence of a feature on the required NFP.

(Etxeberria and Sagardui, 2008) present an approach for the quality evaluation of software product lines. In this approach, an extended feature model is used to identify the variability that has an impact on quality in order to reduce evaluation efforts. This technique is comparable to our idea of partial configuration. However, in contrast to our work, they state that the design stage is a good point to assure that the quality attributes are met.

TOOL SUPPORT

pure::variants (Beuche, 2003) is a tool that supports variant management of SPLs. It is independent of programming languages and it enables the definition of the problem domain by means of feature models, and the solution domain by family models. It also automates the process of product generation. Regarding the configuration of NFPs, the current version is able to assign bugs (from a bug-tracking system) to specific features. Therefore, during product configuration, the application engineer is informed about the *known* bugs that will be present in the final product.

Gears (Krueger, 2007b) is a tool and framework that enables the development and evolution of SPLs; it applies the three-tiered SPL methodology (Krueger, 2007a). In Gears, SPLs are comprised of three elements, software assets (source code, documentation, etc.), product feature profiles (to model each product in the portfolio), and the Gears configurator, which automatically assembles products based on their specifications.

FeaturePlugin (Antkiewicz and Czarnecki, 2004) is an open-source Eclipse plugin for designing and configuring feature models. It supports the concepts of staged configuration (Czarnecki et al., 2005b) and feature cardinalities (Czarnecki et al., 2005a). The tool focuses on providing advanced techniques of feature modeling and not on supporting the whole process of SPL development.

FAMA (FeAture Model Analyser) (Benavides et al., 2007) is an extensible framework for the automated analysis of feature models. It is able to denote feature models in several logic representations; therefore, different solvers can be used in the analysis process. Currently, it supports CSP, SAT (boolean satisfiability problem), and BDD (binary decision diagrams), but it is flexible enough to have other solvers added to it. Cardinality-based feature models are allowed and the following operations are supported: finding out if a feature model is valid (there exists a valid selection that satisfies all constraints), finding the total number of valid products, listing all valid configurations, and calculating the commonality of features (the number of valid products they appear in).

There are commercial, free, and open-source alternatives for the design of feature models. However, only pure::variants is able to provide non-functional information during product configuration; at the moment only at a very basic level, though.

REASONING IN FEATURE MODELS

Important to our work is also the process of *reasoning* in feature models. A seminal work by Benavides et al., (2006) presents the mapping from feature model components (e.g., optional features, mandatory features, and group features) to diverse logical representations, namely SAT, BDD, and CSP. Transforming feature models in these representations enables the use of off-the-shelf solvers that can perform several analyses that are relevant for feature models (e.g., validity, number of solutions, etc.).

The relation of feature models and grammars has also been studied (Batory, 2005). Batory, also motivated by the ability to use off-the-shelf satisfiability solvers, presented the mapping from feature models first to iterative tree grammars, and then to propositional formulas. However, his main goal was to simplify the laborious task of debugging feature models.

Recently, the relation of feature models and logic representation has been further explored. (Czarnecki and Wasowski, 2007) propose a method for the inverse transformation from propositional formulas to a feature model representation. (Janota and Kiniry, 2007) study the representation of feature models in higher-order logic. A formalized meta-model is presented; however, no tool support is provided.

Aspect-Oriented Programming and Operating Systems

There is some work describing the synthesis of operating system kernels and aspect-oriented programming already published; none of the projects target the configurability of architectural properties and, indirectly, of non-functional properties, however.

PURE is an operating-system family that aims to support even deeply embedded systems (Beuche et al., 1999). Its abstractions are designed in minimal extensions, providing for fine-grained configuration possibilities. However, it was originally designed in an object-oriented way, oblivious to AOP. Only after exploring the ability of AOP to modularize the cross-cutting concerns present in a PURE system, aspects were considered (Spinczyk and Lohmann, 2004), and only implemented for selected concerns like interrupt synchronization (Mahrenholz et al., 2002). Therefore, PURE was not designed to be aspect-aware from the beginning as is CiAO, and PURE does not have the (indirect) configurability of non-functional properties as an explicit goal.

The TOSKANA toolkit (Engel and Freisleben, 2005, 2006) enables the deployment of aspects into an OS kernel. The prototype is demonstrated using the NetBSD kernel; in general, the targeted OS domain is the PC domain and not embedded systems, where the consideration of non-functional properties is especially important (see also the motivation section). Furthermore, the toolkit *instruments* the kernel in order to be able to weave and unweave aspects *dynamically* at run time. The induced run-time and memory overhead is not tolerable in embedded computing; in this domain, static configuration and tailored implementations with distinct non-functional properties are needed.

Several studies were conducted on how to apply AOP ex post to existing kernels. Particularly, Coady et al. showed how to modularize prefetching in the FreeBSD operating system kernel (Coady et al., 2001), and sketched the design of an aspect-oriented page daemon and quota manager (Coady and Kiczales, 2003). However, the aspect weaver proposed for this task, AspectC, does not have a functioning implementation; the examples were hand-woven and therefore do not provide a real cost evaluation in terms of non-functional properties like performance or latency.

CONCLUSION

Non-functional properties are inherently complex to be dealt with, but nevertheless mission-critical in many systems. The main challenge in handling NFPs is that most of them are untraceable; that is, there is no line of code or implementation module that an NFP can be solely attributed to. Furthermore, NFPs are highly domain-specific in their nature.

Therefore, NFPs have traditionally often been assessed by the system architects, who have the expertise and experience to do that. When using a product-line approach, though, the requirements on the products are often very different between the desired variants, and the system configurator does not have that kind of knowledge about the product line that is to be configured.

That is why, in our opinion, NFPs have to be dealt with in a *holistic* way, approaching them on multiple levels both in domain engineering and application engineering, from analysis to implementation. Furthermore, the configurator has to be assisted in making his decisions regarding the NFPs of the final product by semi-automated means. The CiAO development process with its feedback approach, which was presented in this chapter, is a big step into that direction.

ACKNOWLEDGMENT

We would like to thank Christoph Elsner for giving valuable comments on a draft of this text.

This work was partly supported by the German Research Council (DFG) under grants no. SCHR 603/4 and SCHR 603/7.

REFERENCES

Antkiewicz, M., & Czarnecki, K. (2004). FeaturePlugin: Feature modeling plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA workshop on Eclipse technology eXchange (Eclipse '04 at OOPSLA '04)*, (pp. 67–72). Vancouver, Canada.

AUTOSAR. (2006a). *Requirements on operating system (version 2.0.1). Technical report*. Automotive Open System Architecture GbR.

AUTOSAR. (2006b). *Specification of operating system (version 2.0.1). Technical report*. Automotive Open System Architecture GbR.

Bass, L. (2006). Principles for designing software architecture to achieve quality attribute requirements. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, (p. 2), Washington, DC, USA. IEEE Computer Society.

Batory, D. S. (2005). Feature models, grammars, and propositional formulas. In *Proceedings of the 9th Software Product Line Conference (SPLC '05)*, (pp. 7–20).

Benavides, D., Ruiz-Cortés, A., & Trinidad, P. (2005). Automated reasoning on feature models. *Proceedings of Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, (LNCS 3520), (491–503).

Benavides, D., Segura, S., Trinidad, P., & Ruiz-Cortés, A. (2006). A first step towards a framework for the automated analysis of feature models. In *Managing Variability for Software Product Lines*. Working With Variability Mechanisms.

Benavides, D., Segura, S., Trinidad, P., & Ruiz-Cortés, A. (2007). FAMA: Tooling a framework for the automated analysis of feature models. In *Proceeding of the First International Workshop on Variability Modeling of Software-Intensive Systems (VAMOS)*.

Beuche, D. (2003). *Variant management with pure variants*. Technical report, pure-systems GmbH. http://www.pure-systems.com/.

Beuche, D., Guerrouat, A., Papajewski, H., Schröder-Preikschat, W., Spinczyk, O., & Spinczyk, U. (1999). On the development of object-oriented operating systems for deeply embedded systems-the PURE project. In *Object-Oriented Technology: ECOOP '99 Workshop Reader*, Lisbon, Portugal. (LNCS 1743), (pp. 27–31). Springer-Verlag.

BuDDy Developers. (2009). *BuDDy project*. Retrieved from http://sourceforge.net/projects/buddy

Coady, Y., & Kiczales, G. (2003). Back to the future: A retroactive study of aspect evolution in operating system code. In M. Akşit (Ed.), *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, (pp. 50–59). Boston:ACM Press.

Coady, Y., Kiczales, G., Feeley, M., & Smolyn, G. (2001). Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 3rd Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering (ESEC/FSE '01)*.

Cysneiros, L. M., & do Prado Leite, J. C. S. (2004). Nonfunctional requirements: From elicitation to conceptual models. *IEEE Transactions on Software Engineering*, *30*(5), 328–350. doi:10.1109/TSE.2004.10

Czarnecki, K., & Eisenecker, U. W. (2000). *Generative programming. Methods, tools and applications*. Addison-Wesley.

Czarnecki, K., Helsen, S., & Eisenecker, U. W. (2005a). Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, *10*(1), 7–29. doi:10.1002/spip.213

Czarnecki, K., Helsen, S., & Eisenecker, U. W. (2005b). Staged configuration through specialization and multilevel configuration of feature models. *Software Process Improvement and Practice*, *10*(2), 143–169. doi:10.1002/spip.225

Czarnecki, K., & Wasowski, A. (2007). Feature diagrams and logics: There and back again. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, (pp. 23–34).

Engel, M., & Freisleben, B. (2005). Supporting autonomic computing functionality via dynamic operating system kernel aspects. In P. Tarr (Ed.), *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, (pp. 51–62). Chicago: ACM Press.

Engel, M., & Freisleben, B. (2006). TOSKANA: A toolkit for operating system kernel aspects. In Rashid, A., & Aksit, M. (Eds.), *Transactions on AOSD II, (LNCS 4242)* (pp. 182–226). Springer-Verlag.

Etxeberria, L., & Sagardui, G. (2008). Variability driven quality evaluation in software product lines. *Proceedings of the Software Product Line Conference, 2008. SPLC '08. 12th International*, (pp 243–252).

Gilani, W., Sincero, J., & Spinczyk, O. (2007). Aspectizing a Web server for adaptation. In *Proceedings of the Twelfth IEEE Symposium on Computers and Communications (ISCC'07)*, Aveiro, Portugal. IEEE Computer Society Press.

Janota, M., & Kiniry, J. (2007). Reasoning about feature models in higher-order logic. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, (pp. 13–22).

Kleiman, S., & Eykholt, J. (1995). Interrupts as threads. *ACM SIGOPS Operating Systems Review*, *29*(2), 21–26. doi:10.1145/202213.202217

Krueger, C. W. (2007a). The 3-tiered methodology: Pragmatic insights from new generation software product lines. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, (pp. 97–106).

Krueger, C. W. (2007b). BigLever software gears and the 3-tiered SPL methodology. In *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on object-oriented programming systems and applications*, (pp. 844–845). New York: ACM.

Lauer, H. C., & Needham, R. M. (1979). On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, *13*(2), 3–19. doi:10.1145/850657.850658

Liedtke, J. (1995). On μ-kernel construction. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, ACM SIGOPS Operating Systems Review. ACM Press.

Loesch, F., & Ploedereder, E. (2007). Optimization of variability in software product lines. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, (pp. 151–162).

Lohmann, D., Spinczyk, O., & Schröder-Preikschat, W. (2005). On the configuration of non-functional properties in operating system product lines. In *Proceedings of the 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, (pp 19–25). Chicago, IL, USA. Northeastern University, Boston (NU-CCIS-05-03).

Lohmann, D., Streicher, J., Hofer, W., Spinczyk, O., & Schröder-Preikschat, W. (2007a). Configurable memory protection by aspects. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS '07)*, (pp. 1–5). New York: ACM Press.

Lohmann, D., Streicher, J., Spinczyk, O., & Schröder-Preikschat, W. (2007b). Interrupt synchronization in the CiAO operating system. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, New York: ACM Press.

Mahrenholz, D., Spinczyk, O., Gal, A., & Schröder-Preikschat, W. (2002). An aspect-oriented implementation of interrupt synchronization in the PURE operating system family. In *Proceedings of the 5th ECOOP Workshop on Object Orientation and Operating Systems (ECOOP-OOOS '02)*, (pp. 49–54). Malaga, Spain.

OSEK/VDX Group. (2005). *Operating system specification 2.2.3*. OSEK/VDX Group. Retrieved from http://www.osek-vdx.org/

Rosenmüller, M., Siegmund, N., Schirmeier, H., Sincero, J., Apel, S., Leich, T., et al. (2008). FAME-DBMS: Tailor-made data management solutions for embedded systems. In *Proceedings of the Workshop on Software Engineering for Tailor-Made Data Management (SETMDM)*.

Schirmeier, H., & Spinczyk, O. (2007). Tailoring infrastructure software product lines by static application analysis. In *Proceedings of the 11th Software Product Line Conference (SPLC '07)*, (pp. 255–260). IEEE Computer Society Press.

Siegmund, N., Kuhlemann, M., Rosenmüller, M., Kästner, C., & Saake, G. (2008). Integrated product line model for semi-automated product derivation using non-functional properties. In *Proceedings of the International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS)*, (pp. 25–23).

Spinczyk, O., & Lohmann, D. (2004). Using AOP to develop architecture-neutral operating system components. In *Proceedings of the 11th ACM SIGOPS European Workshop*, (pp.188–192). New York: ACM Press.

Spinczyk, O., & Lohmann, D. (2007). The design and implementation of AspectC++. *Knowledge-Based Systems*. *Special Issue on Techniques to Produce Intelligent Secure Software*, *20*(7), 636–651.