

# Functional and Non-Functional Properties in a Family of Embedded Operating Systems\*

Daniel Lohmann, Wolfgang Schröder-Preikschat, Olaf Spinczyk  
Friedrich-Alexander University of Erlangen-Nuremberg  
Department of Computer Sciences  
Martensstr. 1, D-91058 Erlangen, Germany  
{lohmann, wosch, os}@cs.fau.de

## Abstract

*The increasing complexity of embedded systems calls for software design and implementation techniques that support specialization without abandonment of reusability. This brings up a conflict as software specialized in supporting a very dedicated case usually cannot be (easily) reused for a different environment. Especially the “standard software” packets such as operating systems, e.g., are concerned with this problem. Particularly critical in this setting are non-functional properties that are ingredient parts of single components or crosscut in the extreme case the entire system software. These properties not only limit component reusability but also impair software maintenance in general. The paper deals with this issue in the scope of operating systems for the embedded-systems domain. It motivates using concepts such as family-based software design, feature modeling and aspect-oriented programming in order to come up with highly customizable and yet reusable system software solutions.*

## 1. Introduction

The design and development of operating systems has to reflect numerous constraints predefined by an application domain. This domain consists, among others, of application software at the top and the computer hardware at the bottom, thus with the operating system in between “a rock and a hard place”. There are many application domains with no single operating-system solution being suited to all of them, or even a subset only.

Due to the need for customized solutions, particularly the embedded systems domain calls for a large assortment of specialized operating-system components. Depending on the application case, not only are number and kind (in

functional terms) of the components varying, but also the same single component may appear in highly different versions. Most crucial in this setting are non-functional properties that are ingredient parts of single components or crosscut in the extreme case the entire system software. These properties not only limit component reusability but also impair software maintenance in general. Being able to deal with software variability—not only in the realm of operating systems—becomes more and more eminent for embedded systems.

The automotive domain gives an example for the increasing demand of software variability management. A modern car can be considered a “distributed system on wheels”: 40 up to over 100 of (8-, 16-, 32-bit) microcontrollers interconnected by a complex network (e.g. LIN, CAN, MOST, Flexray) is the normal case—as is a 1 l/100 km additional fuel consumption due to the weight of all the network cables. At the time being about 35 % of the total costs of a car is in the electronics. Automobile electronics, in turn, makes up about 80 % of all the innovations in a car. Furthermore, 90 % of these innovations comes up with software and not hardware. Thus, software is not only a functional issue of the mechatronics product “automobile”, but also an economical one of high strategic importance.

On the one hand, there is a strong need to reuse software solutions across the different variants and models of a car. On the other hand, in a large number of cases, highly specialized software solutions need to be built depending on the actual car variant or model. Alone relying on object-oriented approaches to cope with the diversity of problems coming up when developing embedded-systems software is not enough. Specialization by means of inheritance, e.g., soon may result in unmaintainable class hierarchies if the combinational complexity increases. Not to mention the risk of performance loss and large memory footprints in the case of an excessive exploitation of interface inheritance and, thus, late binding. Alternative as well as supplement-

\*This work was partly supported by the DFG, grant no. SCHR 603/4.

ing approaches are required in order to benefit from object orientation if one wants to develop system software that is reusable and specializable at the same time.

Understanding an (embedded) operating system as a *software product line* [12] seems to be a promising way to go. Above all, this includes a *family-based design* [7] of the system software. Commonalities of and differences between individual members of the operating-system family, as well as their interdependencies and conflicting combinations, can be adequately expressed on the basis of *feature models* [3], with the features representing the functional and non-functional system properties. Last but not least, *aspect-oriented programming* (AOP) [5] appears to be the proper paradigm in order to maintain non-functional code separate from software components and, thus, improve reusability of the latter.

The paper explains how these software techniques were used in the design and development of the PURE family of embedded operating system [1]. A focus is on non-functional properties, besides others also those ones specifically related to embedded systems (section 2). Concepts and techniques for the design and implementation of highly reusable operating-system components are presented in section 3. Section 4 in brief discusses the PURE development process and section 5 draws the conclusion.

## 2. Non-Functional Properties

Traditionally, operating-system development is a field in which non-functional properties are of fundamental relevance and imply a number of design decisions. The most important non-functional properties in this context are *synchronization*, *protection*, *isolation*, *sharing*, and *interaction*. In general, these properties are fairly independent from the actual application domain. They are *domain unspecific* and typical, e.g., for general-purpose operating systems. Specifically for special-purpose, embedded operating systems additional *domain specific* non-functional properties are to be taken into account.

### 2.1. Embedded Systems Domain

In the realm of embedded systems, the most important non-functional properties are *energy*, *timeliness*, and *dependability*. The subsequent paragraphs discuss these domain-specific properties in some more detail.

**Energy** Especially for mobile or autonomous embedded systems energy is a valuable resource (but also for servers, mainframes, or supercomputers). Energy consumption is a non-functional property of scheduling functions such as process, memory, or I/O scheduling [13]. As a further example, the placement strategy of memory management may

have indirect control over energy consumption of a running process if the hardware characteristic of (main) memory in terms of energy requirement and heat differs with the memory banks or subsystems. In order to provide some kind of energy awareness, energy accounting at various places of the system software becomes necessary. Similar to software instrumentation for monitoring purposes, minimal invasive energy-accounting “hooks” need to be set accordingly. These hooks are non-functional code with respect to the context where they are located.

**Timeliness** Deadlines are qualified as soft, firm, or hard. Depending on the kind of deadline, the methods to guarantee that a certain deadline is met are different. Occasionally, *preemption points* need to be inserted in critical execution paths in order to reduce scheduling latency. Existence, locality, as well as frequency of such a point is a non-functional issue of the respective execution path. Other non-functionalities may concern coordination, e.g. whether blocking synchronization need to be better replaced by non-blocking or even wait-free synchronization in order to relax validation that no deadline is violated.

**Dependability** This property refers to “the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers” (IFIP WG 10.4). It encompasses aspects of *reliability*, *availability*, *safety*, and *security*. Adding redundancy to a system is one measure in order to provide highly available services. As a consequence, the requirement may arise to multiply a single system request and to cope with many replies. This particular feature, e.g., is non-functional with respect to the service, or individual function, whose availability shall be improved. In some degree, security and safety relies on hardware-supported protection and isolation (i.e. domain-unspecific non-functional properties) in order to limit e.g. fault propagation. Reliability is a further non-functional aspect, e.g. in terms of exception handling applied to certain system functions.

### 2.2. Crosscutting Concerns

As the discussion showed, the term “non-functional” sometimes implies fairly complex functions that need to be implemented in order to provide and enforce a certain property. Dependability is an example of highly elaborated designs and implementations, while synchronization may result in very simple solutions (e.g., in case of interrupt locks). The problem of a non-functional property is not its possibly complex functional implementation, but the (explicit/implicit) references to this implementation spread across the implementation of the intrinsic functions of a specific (sub-) system. It is a problem of program fragments re-

peatedly being closely related to functional code for reflecting certain configuration decisions. When being intermixed with the intrinsic functional implementation, these *crosscutting concerns* impair reusability to a vast extent. They link implementations to applications, although the pure functional code may be highly independent therefrom.

In a number of cases, program fragments representing the non-functional properties are as simple as conditional expressions or they solely wrap around the respective function. In other cases, tons of such software prevents one from realizing the gist of the matter. A first step in order to lessen the problems is to cleanly separate non-functional properties by design: *separation of concerns* need to be a must. Ideally, as a following step the code implementing or referencing these concerns should be automatically generated and inserted at the respective places of the system software. Thus, at a fairly late point in time the implementation of an intrinsic function gets adjusted for a specific configuration.

### 3. Operating-Systems Engineering

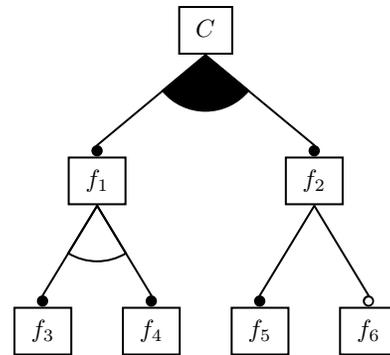
Unbroken thread in the development of operating systems especially for the embedded domain must be the postponement of all those design and implementation decisions that will restrict applicability of system functions or components. This includes that, perhaps, certain decisions will never be made inside the system, but rather considered a case for the application programs to be supported. References to (implementations of) non-functional properties are examples of such design decisions. The following subsections discuss the cornerstones of an operating-system development process that supports highly scalable and customizable designs as well as implementations.

#### 3.1. Feature Modeling

Feature modeling is understood as “the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a *feature model*.” [3] Goal is to come up with directives for and a first structure of a design of a system that meets the requirements and constraints specified by the features.

Common is a graphical representation of the feature model in terms of a *feature diagram*. The diagram is of tree-like structure (fig. 1), with the nodes referring to specific feature categories. Four feature categories are defined: *mandatory*, *optional*, *alternative*, and *or*. A feature diagram describes the options and constraints that shall exist within a system. It models the variable and fixed properties of a family of programs which implement that system.

The diagram shown in figure 1 describes a specific concept  $C$ , e.g. the process management subsystem of an op-



**Figure 1. Feature diagram:**  $f_1$  and  $f_2$  are or-features of concept  $C$ ,  $f_3$  and  $f_4$  are alternative features of  $f_1$ , and  $f_2$  implies a mandatory feature  $f_5$  and an optional feature  $f_6$ .

erating system. If concept  $C$  gets to be included in the final system configuration, then any non-empty subset of features from the set  $\{f_1, f_2\}$  of or-features is also included. The *feature set* with respect to  $C$  at this level of abstraction is  $\{f_1, f_2, \{f_1, f_2\}\}$ . If feature  $f_1$  is present, one feature from the set  $\{f_3, f_4\}$  of alternative features must be included. Thus, the feature set of  $f_1$  consists of either  $f_3$  or  $f_4$ . If feature  $f_2$  is selected, mandatory feature  $f_5$  must and optional feature  $f_6$  may be included in the final configuration.

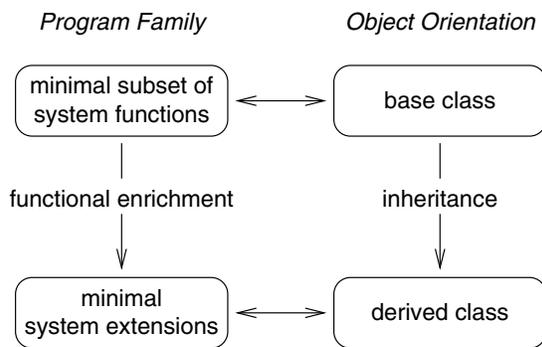
This technique allows for a compact and precise specification of interdependencies of functional as well as non-functional properties of fairly complex systems. Basing on a tool which aids the construction process of a feature model and supports the mapping of features to implementations, automated generation of highly specialized operating systems becomes possible [2].

#### 3.2. Program Families and Object Orientation

The next important issue in the development process is to understand the system software as a *program family* [7] and to follow a classical bottom-up approach. Strictly speaking, design decisions are to be met bottom-up, but the design process is to be controlled in a top-down manner. The idea is to design family members that are particularly tailored to support specific application scenarios by sharing as many as possible system abstractions, i.e. reusable components. A highly distinct *functional hierarchy* of “fine-grain sized” components is the outcome. The entire system structure is a logical one in the sense that the design is hierarchical, and not its implementation [4].

Realizing a program family by an object-oriented implementation may result in highly flexible and yet efficient

system structures. But this will be true only if both design and implementation follow an incremental approach. Starting point must be a minimal subset of system functions which undergoes a stepwise *functional enrichment* by minimal system extensions (fig. 2). These enrichments can be turned into efficient programs by means of *implementation inheritance*. Note that this does not necessarily hold with interface inheritance. The point of problem is late binding of those methods which are subjected to subsequent specialization in derived classes. This concept may result in overhead-prone implementations and entail very large memory footprints, especially in the case of deep class hierarchies. The decision for late binding must be postponed as far as possible in the design and implementation of object-oriented program families. As a consequence, functional enrichment for creating new object-oriented abstractions of a program family favors implementation inheritance over interface inheritance.



**Figure 2. Analogy between the program family concept and object orientation: Exploitation of inheritance in the sense of functional enrichment paves the way for an object-oriented program family.**

Interface inheritance is the right choice only when the family-based design requires multiple implementations of the same interface to coexist. Such a requirement is also a case of a non-functional property of a software system. If there is a choice to have multiple implementations of the same interface included in final configurations, then the feature diagram of the system describes this fact using an or-feature or an optional feature. Whether or not a component implementation needs to be bound to an abstract interface then may become a matter of feature-based configuration.

### 3.3. Aspect Orientation

Not in every case is it sensible to follow a development process that solely relies on a universal family-based design

and object-oriented implementation as described above. Eminent problematic issues are the crosscutting concerns given with many non-functional properties. Trying to reflect these concerns in a hierarchical design may lead to an explosion of the resulting functional and/or class hierarchy. As a rule of thumb: the more crosscutting a specific concern is, the more complex the resulting hierarchical system structure will be.

For software maintenance reasons, a crosscutting concern need to be separated from their points of action and implemented as a single module. When a specific family member is going to be instantiated, all missing crosscutting concerns will be applied to the relevant software components. Referring to non-functional properties then may become a configuration matter. Automated (feature-oriented) configuration may take place by having a software transformation tool in charge of interweaving the program module representing a specific crosscutting concern with all the programs that refer to the corresponding non-functional property.

This kind of final customization of selected software components from a program family can be best achieved using *aspect-oriented programming* (AOP) [5]. In this setting, an aspect program implements a specific crosscutting concern. These programs take care of the manifestation of a particular non-functional property by describing code transformations that need to be applied to selected components. The transformation process is performed by an aspect weaver.

AOP turns out to become a powerful paradigm in the design and development of system software in general. For example, stubs can be generated that hide the style of system-service invocation (e.g. local, remote, crossing address-space boundaries, performing mode changes, etc.) from the system components: the stubs encapsulate the non-functional property “interaction”. Furthermore, synchronization primitives can be inserted automatically to make e.g. thread-unaware components thread safe [11] as is shown in the following section. Component instrumentation, e.g. for monitoring purposes, is made feasible as well [6]. Last but not least, to give pattern-based object-oriented designs a final polishing, AOP appears to be a promising technique for streamlining system code [2].

## 4. PURE Engineering

Central theme in the development of PURE was to postpone design and implementation decisions as far as possible. PURE is a family of operating systems. A couple of members of the PURE family were designed with respect to the very specific demands of deeply embedded systems. At the time being, the PURE family is made of about 350 C++ classes implemented in over 990 compilation units. PURE

runs on nine different processor types ranging from 8- to 64-bit technology.

Particular emphasis in the development process was the factorization of non-functional properties appearing as crosscutting concerns and their modular implementation. PURE instances are created in an automated way by using a feature-oriented configuration tool [8] and exploiting AOP on the basis of AspectC++ [10].

Feature modeling of PURE resulted in a feature diagram of about 250 features allowing for about  $2^{105}$  different valid feature combinations. The smallest possible PURE feature set comes up with just three features (CPU, target platform, and compiler), leading to the selection of 20 classes in the configuration process. A feature set for a typical PURE configuration (with preemptive multitasking) has about 20 features. This set describes all the (functional/non-functional) properties of a given member from the PURE nucleus sub-family.

An excerpt of the feature diagram modeling the nucleus family is shown in figure 3. The PURE *nucleus concept* allows for three fundamental configurations as defined by the two or-features *thread concept* and *interrupt concept*:

1. If only the thread-concept feature is required, the nucleus configuration excludes any means of interrupt handling. In this configuration, interrupt handling is entirely up to the application program if needed.
2. If only the interrupt-concept feature is required, the nucleus configuration excludes any means of thread handling. In this configuration, thread handling is entirely up to the application program if needed.
3. If both features are required, the nucleus configuration comes with thread handling as well as interrupt handling.

The interrupt concept defines the mandatory feature *interruptive*. This nucleus family member enables the reactive execution of tasks purely on interrupt handling basis. This means that there is no necessity for threading in order to bring in concurrency into the system. At this level of abstraction, interrupt synchronization is entirely up to the application program if needed. The optional feature *coordinative* stands for a member of the nucleus family that supports interrupt transparent synchronization [9] of the reactive execution of the tasks. Thus, interrupt synchronization is considered a “minimal system extension” and introduced as functional enrichment of the “minimal subset of system functions” defined by the interruptive PURE nucleus.

As figure 3 indicates, the thread concept comes in two different flavors. The alternative features *uni-threaded* and *multi-threaded* allow for a nucleus configuration that supports either a single-threaded or a multi-threaded mode of execution of the application program. In the former case,

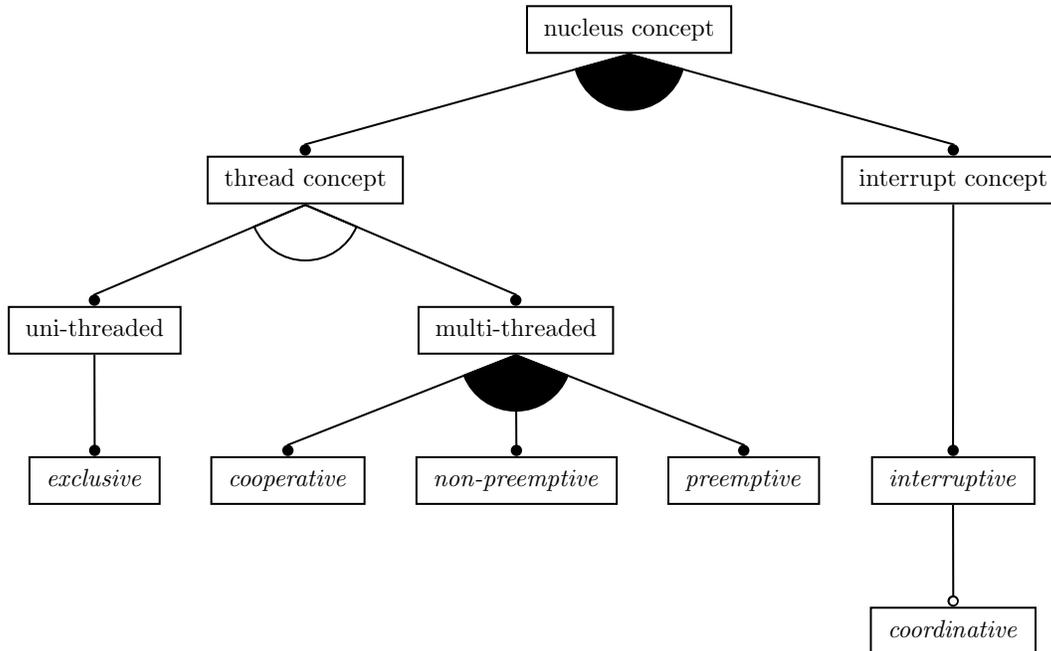
the mandatory feature *exclusive* results in a nucleus configuration that leaves processor control entirely up to the application program. In the latter case, the or-features *cooperative*, *non-preemptive*, and *preemptive* describe the properties of the thread-scheduling subsystem of the nucleus. Depending on the application needs, the PURE nucleus may be run in cooperative, non-preemptive, or preemptive mode of operation, or in any other combination except the empty set.

The feature diagram of the PURE nucleus specifies the logical interdependencies that exist between the various family members. In detail, there are many more features given with the present design. The next step in the PURE development process is to design a functional hierarchy of building blocks that implement the features described by the feature diagram. An example of such a design is shown in figure 4. This design describes that family member *cooperative* is a “minimal system extension” to the “minimal subset of system functions” provided by family member *exclusive*. Similar holds with the family members *non-preemptive* in relation to *cooperative* and *preemptive* in relation to *non-preemptive*.

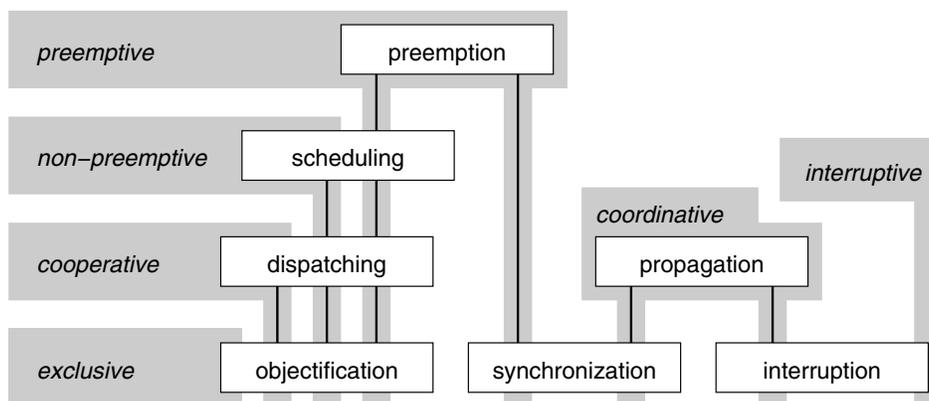
Of particular interest in this functional hierarchy is family member *preemptive* because of its non-functional property. Preemptive scheduling is realized in PURE by an asynchronous invocation of the fundamental scheduling functions provided by the scheduling building block, i.e. the family member *non-preemptive*. There is absolutely no difference between the two levels, except that the preemption building block (in contrast to scheduling) executes in a synchronized mode.

The non-functional property of *preemptive* is synchronization. This property represents a crosscutting concern. Its implementation is separated from the functional code by means of AOP using AspectC++, more specifically, by applying the *pointcut* concept to the critical scheduling functions. A pointcut is a set of points in the code (so called *join points*), which are affected by the same crosscutting concern. In AspectC++ these sets can be defined in a very flexible way by using a declarative language consisting of predefined pointcut functions, wildcards for matching names, and algebraic operations to combine pointcuts. The pointcut definition shown in figure 5 enumerates the (non-preemptive) scheduling functions `block()`, `ready(Thread*)`, and `yield()`, each of which representing a critical section when being reused in order to support preemptive mode of operation. Calls to these functions need to be synchronized.

In addition to the pointcut definitions, actions need to be defined that are to be executed when any of the join points in the pointcut is reached at run time. Any of these actions is called an *advice*. Figure 5 shows the definition of the two actions needed to take care about synchronization of



**Figure 3. Feature diagram of the PURE nucleus family. The PURE nucleus concept defines a thread concept and/or an interrupt concept, with the former one specifying either a uni- or a multi-threaded mode of operation. The emphasized feature names indicate the different operating modes provided by the family.**



**Figure 4. Functional hierarchy of the PURE nucleus family. The figure shows how the more fundamental family members are reused to form more complex family members. In PURE, preemptive scheduling is provided by reusing non-preemptive scheduling and coordinating event-driven activities. Coordination is achieved using aspect-oriented programming. Invocation of synchronization functions is considered a non-functional property of preemptive scheduling. Insertion of the respective function calls is automated using AspectC++.**

```

pointcut critical() =
    execution("void block()") ||
    execution("void ready(Thread*)") ||
    execution("void yield()");

aspect Synchronization {
    advice critical(): before() {
        enter();
    }

    advice critical(): after() {
        leave();
    }
};

```

**Figure 5. Modularization of a non-functional property “synchronization” in the AOP language AspectC++.**

the critical scheduling functions. The first advice definition means that *before* the body of any function described by `critical()` is executed, entrance to the critical section is requested by calling `enter()`. Similarly the second advice causes the call on `leave()` *after* the critical section is left.

Both advice definitions are encapsulated in a named modular unit, which is an *aspect*. Besides the advice definitions, aspects can (similar to classes) store and manage state information, which is also accessible by the advice code bodies. The AspectC++ compiler (resp. aspect weaver) expands the advice code at the specified join points, it interweaves functional and non-functional code. That is to say, the preemption building block shown in figure 4 is automatically generated from the aspect program depicted in figure 5. Scheduling functions which are critical in an preemptive environment remain fully reusable. Every single point of invocation of these functions is considered a join point where synchronization code is automatically inserted in order to make preemptive scheduling work.

By turning the design of the PURE nucleus family into an object-oriented implementation using C++ and by enforcing domain-specific configuration decisions with AOP on the basis of AspectC++, a *software product line* was the outcome. The individual products of this product line scale with respect to the functions they provide to an application program. This is shown by the numbers of table 1, which specifies the memory footprints of the respective nucleus instances. The individual products were created from a program family and, thus, share a lot of commonalities depending on their level of abstraction in the functional hierarchy (fig. 4).

nucleus instance	size (in bytes)			
	<i>text</i>	<i>data</i>	<i>bss</i>	total
<i>exclusive</i>	434	0	0	434
<i>interruptive</i>	812	64	392	1268
<i>cooperative</i>	1620	0	28	1648
<i>non-preemptive</i>	1671	0	28	1699
<i>coordinative</i>	1882	8	416	2306
<i>preemptive</i>	3642	8	428	4062

**Table 1. Software product line of the PURE nucleus family (x86 port)**

## 5. Conclusion

In PURE, architecture is considered a non-functional property of an operating system, as is synchronization, protection, isolation, and sharing. In addition to these domain unspecific non-functional properties, PURE also addresses domain specific non-functional properties of embedded systems such as energy, timeliness, and dependability. Feature modeling is used to express the commonalities of and differences amongst the various members of the PURE family. The instantiation of a specific application-aware PURE configuration is controlled by a feature model and supported by a feature-based configuration tool. Crosscutting concerns of non-functional properties are dealt with by means of aspect-oriented programming (using AspectC++) and automated aspect weaving.

The PURE development shows that the design and implementation of highly reusable and yet specialized operating-system abstractions or functions must not be a contradiction in terms. Key to success was understanding an operating system as a software product line. The outcome was a solution that scales with the demands of many embedded systems. PURE demonstrates that feature-based development of an operating-system family is a very promising approach in order to master the increasing functional complexity of embedded systems in spite of utmost resource scarceness.

## References

- [1] D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE Family of Object-Oriented Operating Systems for Deeply Embedded Systems. In *Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'99)*, St Malo, France, May 1999.
- [2] D. Beuche, O. Spinczyk, and W. Schröder-Preikschat. Fine-grain Application Specific Customization for Embedded Systems. In *Proceedings of the IFIP WCC 2002 Stream 7 on*

*Distributed and Parallel Embedded Systems (DIPES 2002)*, Montreal, Canada, August 25–30, 2002.

- [3] K. Czarnecki and U. W. Eisenecker. *Generative Programming—Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.
- [4] A. N. Habermann, L. Flon, and L. Coopriider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.
- [5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox PARC, February 1997.
- [6] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *The Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Washington DC, USA, April 29–May 1, 2002. IEEE Computer Society.
- [7] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.
- [8] pure-systems GmbH. Variant Management with pure::variants. <http://www.pure-systems.com>, 2004. Technical White Paper.
- [9] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. On Interrupt-Transparent Synchronization in an Embedded Object-Oriented Operating System. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 270–277, Newport Beach, California, March 15–17, 2000. IEEE Computer Society. ISBN 0-7695-0607-0.
- [10] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *The 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 18–21, 2002.
- [11] O. Spinczyk and D. Lohmann. Using AOP to Develop Architecture-Neutral Operating System Components. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 188–192, Leuven, Belgium, Sept. 2004.
- [12] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999. ISBN 0201694387.
- [13] A. Weißel and F. Bellosa. Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002)*, Grenoble, France, Oct. 2002.