The Design of Application-Tailorable Operating System Product Lines*

Daniel Lohmann, Wolfgang Schröder-Preikschat, and Olaf Spinczyk

Friedrich-Alexander University of Erlangen-Nuremberg,
Department of Computer Sciences,
Martensstr. 1, D-91058 Erlangen, Germany
http://www4.cs.fau.de

Abstract. System software for deeply embedded devices has to cope with a broad variety of requirements and platforms, but especially with strict resource constraints. To compete against proprietary systems (and thereby to facilitate reuse), an operating system product line for deeply embedded systems has to be highly configurable and tailorable. It is therefore crucial that all selectable and configurable features can be encapsulated into fine-grained, exchangeable and reusable implementation components. However, the encapsulation of non-functional properties is often limited, due to their cross-cutting character. Fundamental system policies, like synchronization or activation points for the scheduler, have typically to be reflected in many points of the operating system component code. The presented approach is based on feature modeling, C++ class composition and overcomes the above mentioned problems by means of aspect-oriented programming (AOP). It facilitates a fine-grained encapsulation and configuration of even non-functional properties in system software.

1 Introduction

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. This is especially true for the broad field of deeply embedded systems. Here, the phrase "deeply embedded" refers to systems forced to operate under extreme constraints in terms of e.g. memory and/or CPU resources, power consumption, and heat dissipation. The market of such systems is huge and subject to an enormous cost pressure. In year 2000 about eight billion microprocessors have been manufactured [32]. Only about two percent of them went into the PC, laptop, workstation or server market, while 98 % were dedicated to embedded systems. About five billions of all were 8-bit microprocessors. From the point of view of procurement, this "old-fashioned" technology is the best

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

G. Barthe et al. (Eds.): CASSIS 2005, LNCS 3956, pp. 99–117, 2006.

compromise with respect to functionality and cost. The situation is not that different today, as a look at automotive industry, chipcard technology, or the consumer product market shows. Moreover, it can not be expected to change soon, given that the envisioned scenarios of *smart dust* [20], *ubiquitous computing* [35] and *proactive computing* [32] crucially depend on the bulk availability of very cheap, self-organizing "intelligent" devices. Because of cost pressure—and in many cases also because of misunderstandings about what the notion of "operating system" stands for—one is faced with a situation in which the wheel is getting to be reinvented fairly often. There is a zoo of commercial operating systems available at the embedded systems market. Nevertheless, about 50 % of the embedded systems products come with proprietary solutions [34]. OS-functionality such as threading and interrupt handling is developed from scratch—again and again. The reason is, that is simply impossible to build a "one-fits-all" system that fulfills the requirements of all potential applications, while still being thrifty and economical with system resources. The solution is therefore to tailor down the operating system so it provides exactly the functionality required by the intended application, but nothing more. Understanding an (embedded) operating system as a *software product line* [36] seems to be a promising way to go. 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* [12], with the features representing the functional and non-functional system properties. This leads to a family-based [27] design approach. Examples for family-based, configurable operating systems in the domain are e.g. eCos by RedHat Inc. [1], the OSEK standard which is widely used in automotive industry [2], or our PURE operating system product line [5] for the domain of deeply embedded devices. Although the results achieved with these systems motivate the reuse of system software components for a number of reasons, operating system product line development is not yet exercised very well in this market. Another example that underpins the increasing demand of software product line engineering is the automotive domain. Automobile electronics makes up about 80 % of all the innovations in a car. Furthermore, 90 % of these innovations come 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. Resolving this contradiction is challenging and calls for highly careful system software designs and implementations. Most crucial in this setting are non-functional properties that are ingredient parts of single components or cross-cut 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. For operating systems, this is of particular concern because of their qualified placement between "a rock and a hard place", namely application

software at the top and computer hardware at the bottom. Software variability was and is an important issue in operating systems, and it will ever be. 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 [26, 18]. 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 [14]. Alternative as well as supplementing approaches are required in order to benefit from object orientation if one wants to develop system software that is reusable and tailorable at the same time. *Aspect-oriented programming* (AOP) [21] appears to be a proper paradigm in order to maintain implementations of non-functional properties separate from software components and, thus, improve reusability of the latter. The paper describes principles of the design and development of operating systems aiming at a very high degree of customization not only with respect to lower-level hardware but also higher-level user programs. Discussed are design rules, techniques, and issues of tool support which are applicable not only in the course of developing embedded operating systems from scratch, but also in the process of re-engineering existing system software. Moreover, the approach presented may also be successfully applied in order to develop and maintain extensible as well as contractible application software. Thus, the paper is about a fairly general approach that is not only limited to the design and development of operating systems. Application domain of the described principles is the field of deeply embedded systems. Fundamental concepts and techniques to produce highly reusable operating system components are presented in section 2. Section 3 is about a case study, the *thread abstraction layer* (TAL) of the PURE family of embedded operating systems [5]. In section 4, we will briefly discuss the approach as followed by the PURE successor CiAO [30] to encapsulate non-functional properties and to isolate cross-cutting concerns. Conclusions are drawn in section 5.

2 Operating System Engineering

Most important in the development of operating systems for the embedded systems domain is the postponement of all those design and implementation decisions that will potentially restrict applicability of system functions or components. This includes that, perhaps, certain decisions are never be made in the OS itself, but are rather postponed to the application programmer. References to implementations of some 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 and implementations.

2.1 Incremental System Design

Predominant issue in the development process of deeply embedded operating systems must be understanding the system software as a *program family* [27]

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 [17]. 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 [11]. Starting point must be a minimal subset of system functions which undergoes a stepwise *functional enrichment* by minimal system extensions. 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 subject 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. Interface inheritance is the right choice only when the family-based design requires multiple implementations of the same interface to coexist. In certain cases it is sensible for such kind of requirement to be considered a non-functional property of object-oriented (operating) system software. In order not to limit reusability of a class implementing that kind of interface, the non-functional property of interface inheritance needs to be separated properly.

2.2 Variabilty Management

By consequently following the family-based approach of software development, highly customizable operating systems are feasable. Variant building, however, is only a first step in the development process. Without being able to organize and manage the many possible variants of an operating system family in an adequate and user-friendly manner, this approach will be doomed to failure. Feature modeling appears to be a promising way to tackle the variability management problem. This technique 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*." [12] 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

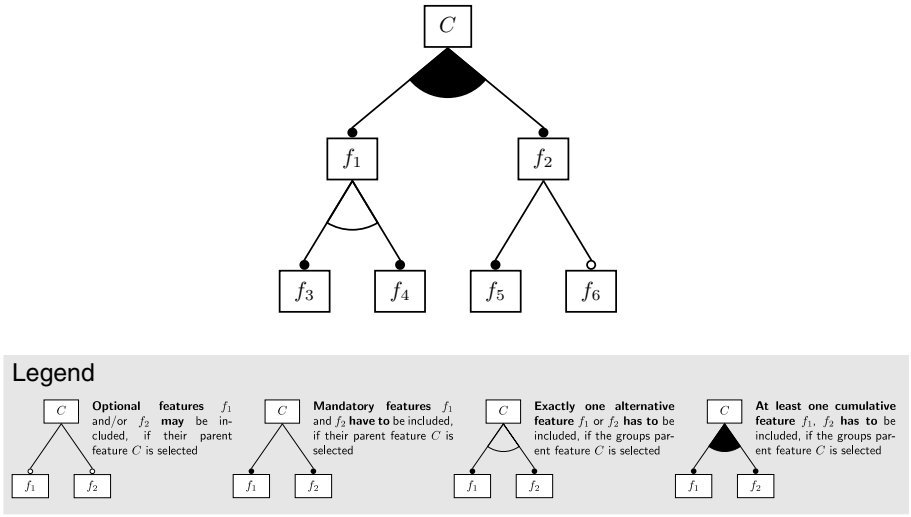

Fig. 1. Example of a Feature Diagram

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 operating 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 either $\{f_1\}$, $\{f_2\}$, or $\{f_1, f_2\}$. 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. For f_2, this leads to the feature set $\{f_5\}$ or $\{f_5, f_6\}$. This technique allows for a compact and precise specification of interdependencies of functional as well as non-functional properties of fairly complex systems [12]. 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 [6].

2.3 Modularization of Non-functional Properties with AOP

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 cross-cutting 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. For software maintenance reasons, a cross-cutting concern needs to be separated from its points of action and implemented as a single module. When a specific family member is going to be instantiated, all missing cross-cutting concerns will be

applied to the relevant software components. Referring to non-functional properties then may become a configuration matter. Automated configuration may take place by having a software transformation tool in charge of interweaving the program module representing a specific cross-cutting concern with all the programs that refer to the corresponding non-funtional property. This kind of final customization of selected software components from a program family can be best achieved using AOP [21]. In this setting, an aspect program implements a specific cross-cutting 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. Several publication show that AOP provides benefits for the development of configurable *infrastructure software* in the broad sense, namely middleware [10, 9, 37, 19, 28] and databases [28, 33] product lines, as well as dynamically configurable web proxies by means of runtime weaving [13]. Regarding operating systems, Coady et al. retroactively evaluated the evolution of four partly non-functional OS concerns in the FreeBSD kernel using the general-purpose AspectC language [8, 7]. It was shown that an aspect-oriented implementation would have led to significantly better evolvability. Due to missing tool support (namely a weaver), her study did cover only a relatively small part of the kernel code base and no heavily crosscutting concerns such as tracing or kernel diagnostics. Not a general-purpose AOP language, but an AOP-inspired language of temporal logic was used by Åberg et al. to integrate the Bossa scheduler framework into the Linux kernel [3]. C4 uses AOP concepts to implement a "semantic patch system" for the application of kernel patches [15].

3 Case Study of a Thread Abstraction Layer

PURE [5] is a family of operating systems targeted at the highly resource-constrained domain of deeply embedded devices and available for a large number of 8 and 16 bit processor platforms. A branch of the PURE family that provides elementary process management functions is the *thread abstraction layer*(TAL). This layer is a refinement of the original PURE threads package and serves for various experimental purposes related to fine-grain (operating system) software product line development. The following two subsections give a brief overview about the concepts and techniques that were used to make PURE software extensible as well as contractible. First, excerpts from the TAL feature model are discussed to exemplify the concept of variability management having been applied to PURE. Second, the functional hierarchy of TAL is presented to illustrate some of the internals of the design and to give also an idea on what fine-grain operating system software product line development means in PURE.

3.1 Feature Modeling

The TAL feature model aims at describing commonalities of as well as differences between the various possible variants of a system software component commonly

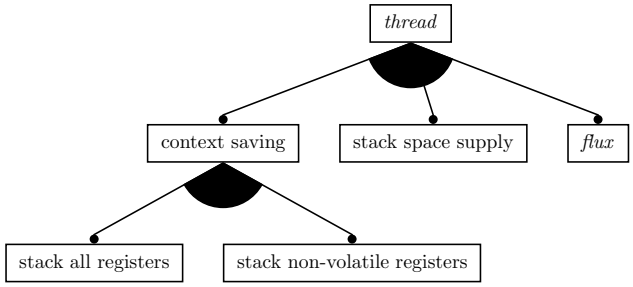

Fig. 2. Thread concept. This TAL feature is made of a hierarchy of or-features covering functions that save/restore a thread context (*context saving*), take care of expansion directions and alignment restrictions of a stack (*stack space supply*), and manage a thread of control of program execution (*flux*).

known as a threads package. Focus was on the deeply embedded systems domain. Above all this means that the system design resulting from the feature model must be minimal in any respect: each level of abstraction introduced need to be a minimal extension to the minimal subset of system functions existing so far. Figure 2 shows the three main subfeatures of the *thread concept*, which are defined as follows:

context saving. Spans functions needed to save and restore a thread context. A stack-based approach is assumed. The feature is constituted by two or-features that differentiate between three combinations of context saving functions. A TAL configuration may encompass functions to stack all and/or only non-volatile CPU registers. The latter are a subset of the former and make thread switching more lightweight (in execution time and memory space).

stack space supply. Provides fundamental stack management functions concerned with allocation, alignment restrictions, and expansion direction (top down or bottom up) of a stack.

flux. covers the functions needed to implement the flow of control represented by a thread and its binding to program text. Figure 3 shows a refinement of this subfeature.

As shown in figure 2, the TAL thread concept consists of three or-features. Thus, an application is provided with seven configuration options at this level, depending on the number of thread subfeatures selected. This is in line with the idea of program families: PURE applications are not forced to go with all TAL functions, but rather is given choices from which they may or may not make their decisions. Heart of TAL is *flux* (fig. 3), which describes a hierarchy of abstractions modeling a thread of control including its binding to program text. The decisive idea is to postpone decisions on how to represent and manage the context of a thread as far as possible. Figure 3 shows a feature hierarchy which corresponds to an implementation that implies functional enrichment of a minimal subset of threading functions. The *flux* subfeatures model the evolution steps from flyweight to lightweight threads. Their meaning is as follows:

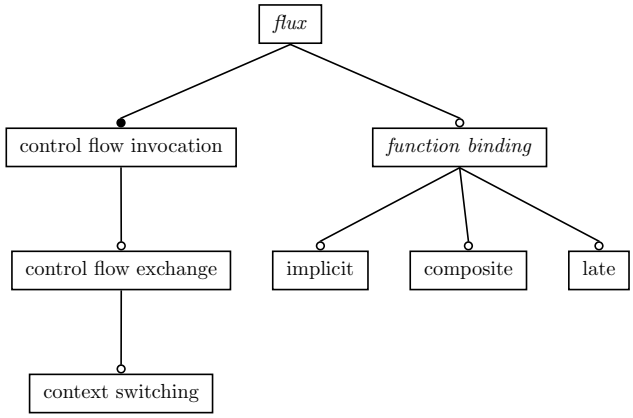

Fig. 3. Flux concept. The feature diagram models functional enrichment of thread abstractions, starting from a simple run-to-completion mode of operation (*control flow invocation*) and an optional binding of user-defined code to a thread (*function binding*).

control flow invocation. Describes the minimal subset of system functions needed to instantiate and terminate a thread. The principle of operation of a thread at this level of abstraction is *run to completion.* The spawning thread inherits the processor state (stored by the working registers) to the spawned thread and implicitly releases CPU control. Upon termination, the spawning thread takes over the processor state again and receives back CPU control.

control flow exchange. A minimal system extension that allows for thread switching in a coroutine-like fashion. Thus, run to completion is no longer the only principle of operation provided at this level of abstraction. Both threads, i.e. spawner and spawnee, may resume each other by sharing the processor state except the contents of the stack pointer register.

context switching. Another minimal system extension which adds functions to save and restore the processor state of a thread. This abstraction requires the *context saving* feature shown in figure 2. The key idea is that every thread is responsible to manage its processor state on its own: the state needs to be saved before resuming execution of another thread and will have to be restored after having been resumed execution by some other thread. Thus, no thread needs to know about the size and organization of the processor state of another thread.

function binding. This *flux* subfeature models different ways of how to bind user-defined functions to a thread. By default, the code executed by a thread always is in-line with the basic block or scope that instantiated the thread. However, if *function binding* is selected, the code to be executed by a thread may be subject to (1) *implicit* binding using a default function, (2) *composite* binding using a template function, or (3) *late* binding using a virtual function.

If *flux* is going to be selected, TAL comes at least with *control flow invocation.* All other *flux* subfeatures are optional so that no application program of TAL

```
slot = label();           // remember current thread of control
split(flux);              // spawn additional thread of control
if (slot != label()) {    // did a control flow switch occur?
  ...                     // yes, spawnee takes on execution
  latch(slot);            // spawnee finishes and resumes spawner
}                         // spawnee never returns to here
...                       // no, spawner continues execution
```

Fig. 4. Flyweight thread instantiation (C-like). A new thread is spawned using `split()`, which returns twice. In order to determine whether the spawner or the spawnee returns, `label()` is used: the spawnee returns when `label()` after `split()` delivers a value different from `label()` before `split()`. The spawnee returns first and passes back CPU control to its spawner using `latch()`.

will be forced to pay for functions that it does not need. In addition, the features are organized in such a manner that the resulting implementations will follow the incremental system design approach and, thus, appear as minimal system extensions. To get an idea of how the minimal subset of TAL functions can be used to instantiate threads that will operate according to run to completion, see figure 4. Functions `label()`, `split()`, and `latch()` basically implement the *control flow invocation* feature. The resulting assembly-level code generated from this C fragment is shown in figure 5. TAL functions are implemented as `inline` functions, mostly. The code sequence shown in figure 5 is semantically equivalent to the code sequence of figure 4: it is the result of the compilation process using the GNU C/C++ compiler. The two examples demonstrate what family-based design of PURE actually implied, namely coming up with a large number of tiny system functions. The motivation to start out with a minimal subset of threading functions (as shown in figures 4 and 5) that only save/restore a very minimal

```
    leal  -4(%esp),%edx    # slot = label()
    pushl $1f              # split(flux)
    movl  flux,%esp        #      "      activate spawnee
1:                         # spawner resumes execution
    leal  -4(%esp),%eax    # <aux> = label()
    cmpl  %eax,%edx        # if (slot == <aux>)
    je    2f               # goto 2
    ...                    # spawnee takes on execution
    movl  %edx,%esp        # latch(slot)
    ret                    #      "      goto 1
2: ...                     # spawner continues execution
```

Fig. 5. Flyweight thread instantiation (x86-like). The example shows how run to completion is actually realized for the spawned thread: the spawner transforms into the spawnee by assigning `flux` to the stack pointer register. The spawnee terminates by (1) assigning the spawners stack pointer (`slot`) to the stack pointer register and (2) restoring the spawners program counter (`ret`).

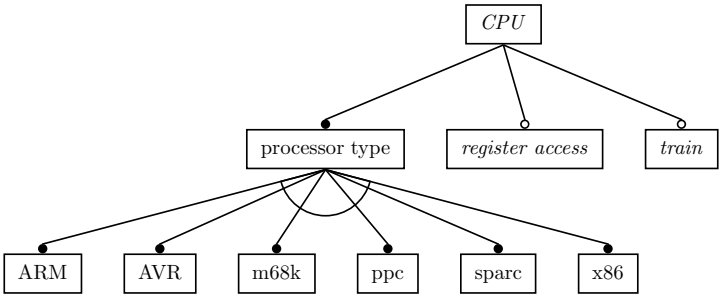

Fig. 6. CPU concept (excerpt). Mandatory feature *processor type* specifies the CPU architecture that can be supported by a PURE family member. Optional feature *register access* is root of a bunch of subfeatures related to processor state management. Support for trap/interrupt handling is modeled by the optional feature *train*.

processor state consisting of program counter and stack pointer registers was to have a compiler in charge of context switching. A compiler exactly knows about the non-volatile processor state of a thread and that state may differ from thread to thread. The idea was to be able to take advantage of compiler pragmas that specify the size of the processor state to be saved/restored upon thread switches in dependence on the actual scope where the thread switch takes place.

Another important issue of TAL (and the encompassing operating system kernel) is CPU management. Figure 6 shows an excerpt of the feature model describing the CPU concept. Mandatory feature is the *processor type*, which

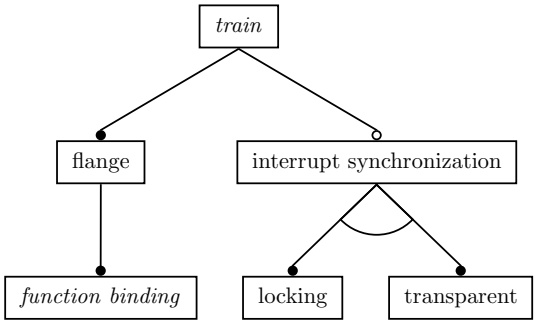

Fig. 7. Train concept (excerpt). Mandatory feature *flange* models the binding technique used to make trap/interrupt handlers physically known to the CPU. Basically, this feature directly maps to the *function binding* feature of *flux* (fig. 3). Optional feature *interrupt synchronization* describes the alternatives for the coordination of event-triggered activities in PURE. Either "hard synchronization" using interrupt locking or interrupt transparent non-blocking "soft synchronization" (without relying on dedicated CPU instructions) is supported.

in turn consists of a number of alternative features. Each of these alternatives stands for the processor platform that is supported by TAL. Usually, for a given system configuration, only one target platform will be supported. The optional feature *register access* describes abstractions provided to read and write the registers of the CPU indicated by *processor type*. Register access functions are implemented by means of operator overloading using a C++ class instance for each of the registers provided by a particular CPU. An overloaded assignment operator performs write access, while the overloaded type cast operator performs read access. The operators are implemented as inline assembly functions. They are used, e.g., to implement thread context management already in a high-level and problem-oriented programming language such as C++. The third subfeature of concept *CPU* models the art of trap/interrupt (*train*) handling for a selected *processor type*. A refined feature model of *train* is shown in figure 7. In that subtree, mandatory feature *flange* describes the kind of *function binding* in order to make problem-oriented trap/interrupt handlers known to the CPU. This is realized by letting *train* logically share the same binding techniques with concept *flux* (see also fig. 3). A major part of *train* is made of *interrupt synchronization*, which is an optional feature: not in every use case will interrupts raise race conditions and, thus, need to be synchronized for coordination purposes. Two alternatives are given:

1. Interrupt locking, i.e., interrupts are disabled and (re-) enabled to secure critical code sections. This is the traditional case of coping with concurrency issues due to hardware interrupts and is fairly easy to implement. However, blocking of interrupts comes with the risk of loosing hardware events and, thus, turns out not to be a good choice especially for embedded real-time systems with high interrupt frequency.
2. Interrupt transparent synchronization [29], i.e., interrupts are never disabled by an operating system kernel. This feature corresponds to a set of synchronization abstractions that allow for interrupts at any time. Coordination is achieved using a variant of non-blocking synchronization.

Interrupt transparent synchronization can be done with and without specific (e.g. CAS-like) CPU instructions. As a consequence, the alternative feature *transparent* consists of an ensemble of or-features, with each of these subfeatures describing a specific synchronization technique.

Developing feature models to aid the design process of a family of operating systems and for documentation purposes is one aspect. Using these models to support the configuration and generation process of operating systems is another aspect. With `pure::variants` [4] a feature-based configuration tool has been developed that supports the workflow from the creation of a feature model up to the automatic generation of user-customized operating systems for very specific problem domains. The tool not only allows for creation but also verification of feature models such that logically consistent system configurations will be the outcome of the generation process.

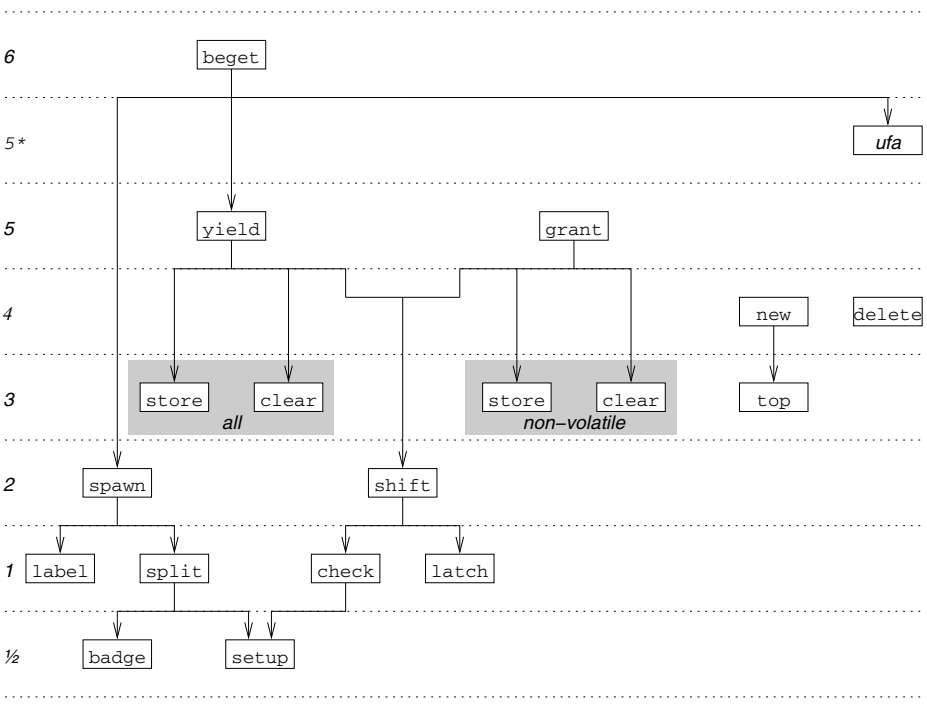

Fig. 8. Functional hierarchy of TAL. The levels serve the following purposes: 1 *control flow invocation*, 2 *control flow exchange*, 3 *context saving* and *stack space supply*, 4 *stack space supply*, 5 *context switching*, 5∗ *function binding*, 6 thread instantiation. Level $\frac{1}{2}$ supports level 1 only: its functions came into existence with the design of `check()`, which showed commonalities with the already existing `split()`. These commonalities then became subject to factorization which led to `badge()` and `setup()`.

3.2 Functional Hierarchy and Component View

The TAL feature model is turned into an implementation using a very fine-grain incremental system design approach. Result of this process is a functional hierarchy (fig. 8). Figure 8 makes explicit the levels of abstraction a TAL function is assigned to. Level 5∗ is not really part of TAL, but rather of the application program using TAL. This level stands for some *user function abstraction* (UFA) that corresponds to the *function binding* feature of *flux* (fig. 3). In addition, level $\frac{1}{2}$ stands for a level of abstraction that resulted from a refinement step in the design process: when level 2 was designed, one figured out commonalities in the implementation of `split()` and `check()`, which then where factorized out and led to the additional level. Level 2 takes care of *control flow exchange*, levels 3 and 5 cover *context saving* and *switching* issues, while level 4 and function `top()` of level 3 turn the feature *stack space supply* into implementation. Level 6 is responsible for thread instantiation. TAL offers a very high degree of customization at the cost of a fairly complex internal structure. The structural complexity becomes manageable for an expert using e.g. feature-modeling and

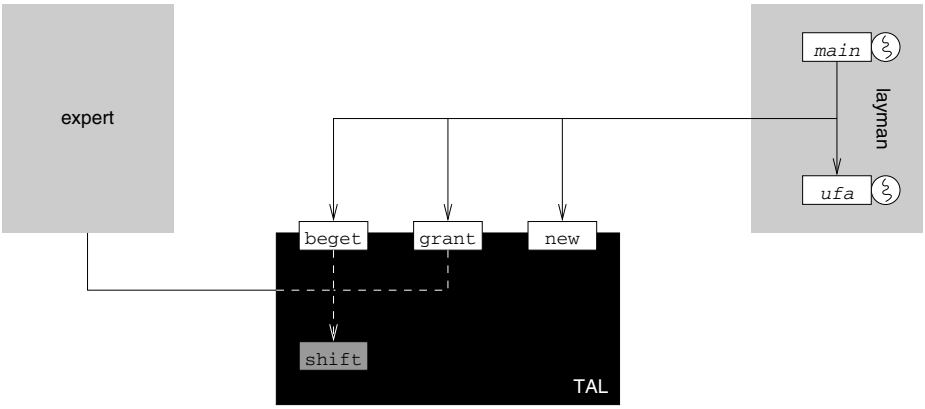

Fig. 9. Component view of TAL

configuration tools such as `pure::variants`. Nevertheless, a layman will be lost for all the many puzzle bricks offered by TAL. For these sorts of customers, TAL appears to be a black box that comes with a minimal export interface. Figure 9 shows this component view in some more detail. Actually, there are only three "fallback functions" making up TAL to an easy to use threads package. These functions are `new()` to allocate stack space for a thread, `beget()` to instantiate a thread, and `grant()` to pass control between threads while maintaing the processor state invariant for inactive threads. The user-defined code to be executed by a thread (on behalf of `beget()`) comes with the UFA instance as provided by the user itself. In fact, TAL is an *open component* [16] that provides a basis for operating system product line development in the small, in particular for a process management subsystem. For the large case, TAL becomes a component whose export interface hides the internal complexity from the user. This way, a high degree of reusability is achieved not only for the expert but also for the layman. Since the export interface is made of customizable system functions, even the layman is given some options for specialization.

4 Aspect Orientation of Operating Systems

The approach discussed so far is not only suited to model functional relationships between abstractions or members of operating system families, but also non-functional ones. Being able to model non-functional interdependencies, however, is only one issue, another issue is to implement them in a modular way to generally improve software maintenance [23, 25]. PURE proved that it must not be a contradiction to come up with a highly modular operating system design and implementation and at the same time keeping the many building blocks manageable. Key to success was family-based design, feature modeling, and aspect-oriented programming, as well as tool support. However, AOP came into play at a fairly late point in time of the PURE development. It mainly served

re-engineering purposes of selected pieces of the entire PURE software base. PURE is not an aspect-oriented operating system, but benefits from AOP in various respects. PURE re-engineering in terms of AOP was considered a first experiment and showed that it would pay to consider aspect orientation as a central design issue being followed from the very beginning. With the PURE successor CiAO[1] [24, 25], we are now developing a new family of operating systems that aims to achieve an even higher level of configurability. CiAO focuses on non-functional properties of operating systems whereby these properties technically appear as cross-cutting concerns which impair maintainability of a reasonable large fraction of the system software. Emphasis is on the configuration of *architectural features*, i.e. to consider the duality of operating system structures [22] as a non-functional system property.

4.1 Non-functional Properties Considered Harmful

Traditionally, operating system development is a field in which non-functional properties are of fundamental relevance and imply a number of design decisions. Examples of such properties 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. Especially for embedded operating systems, additional *domain specific* non-functional properties are of importance such as energy, timeliness, and dependability. The term "non-functional" sometimes implies fairly complex implementations in order to provide and enforce a certain property. But this is not really the problem. 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 problems with non-functional properties are the possibly many (explicit/implicit) references to their implementations spread across the software of the intrinsic functions of a specific (sub-) system. It is a problem of program fragments repeatedly being closely related to functional code for reflecting certain configuration decisions. When being intermixed with the intrinsic functional implementation, these *cross-cutting concerns* impair reusability to a vast extent. They link implementations to applications, although the pure functional code may be highly independent therefrom. Most non-functional properties are *emergent properties*. They are neither visible in the code nor structure of single components, but "suddenly" emerge from the orchestration of many components into a complete system. Properties that manifest in the integrated system only are indeed cross-cutting, as they result from certain (unknown) characteristics of every single component. Due to their inherent emergence it is, however, not possible to tackle them by decomposition techniques. They need to be understood holistically, that is, on the global scope of software development. One could say they need to be addressed by "*holistic aspects*", meaning that the realization of non-functional concerns does not cross-cut (just) the code, but the whole process of software development. In a number of cases, program fragments representing the non-functional

[1] CiAO is Aspect-Oriented

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.

4.2 Separation of Cross-Cutting Concerns

Central topic of CiAO is to consequently isolate cross-cutting non-funtional properties both by design as well as by means of language support. CiAO strongly follows an aspect-oriented design and is implemented in AspectC++ [31], an aspect-oriented extension to C++. As an idea sketch of AspectC++, a synchronization aspect is being considered in the following. In CiAO, as was in PURE (and is in almost any other operating system), synchronization is a typical cross-cutting concern. Its implementation is separated from the functional code by means of the AspectC++ *pointcut* concept. A pointcut is a set of points in the code (so called *join points*), which are affected by the same cross-cutting 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 10 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 10 shows the definition of the two actions needed to take care of synchronization of the criticial 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()`. Similarily the second advice causes the call on

```
pointcut critical() =  execution("void block()") ||
                       execution("void ready(Thread*)") ||
                       execution("void yield()");
aspect Synchronization {
    advice critical(): before() { enter(); }
    advice critical(): after()  { leave(); }
};
```

Fig. 10. Modularization of a non-functional property "synchronization" in the AOP language AspectC++. The AspectC++ aspect weaver translates `aspect` into a C++ `class`. In addition, it looks for `critical()` join points in a given source code and, once matched, inserts the `advice` code before/after them, accordingly.

Table 1. Memory footprint (in bytes, x86) of members of the PURE nucleus family

	text	*data*	*bss*	total
exclusive processor usage	434	0	0	434
interruptive mode of operation	812	64	392	1268
cooperative scheduling	1620	0	28	1648
non-preemptive scheduling	1671	0	28	1699
coordinative interrupt propagation	1882	8	416	2306
preemptive scheduling	3642	8	428	4062

`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 component code and code that manifests a certain non-functional property. The two advice functions `enter()` and `leave()` implement the *interrupt synchronization* feature shown in figure 7, either by means of interrupt locking or in an interrupt transparent manner. Moreover, by using `pure::variants`, the synchronization aspect will be implicitly applied when that feature is going to be selected during the configuration process of the system software. That is to say, `pure::variants` automatically calls the AspectC++ compiler with the synchronization aspect as additional input when interrupt synchronization needs to be a feature of the resulting system. The AOP approach was motivated by experiences having been made with PURE. By turning the design of the PURE family into an object-oriented implementation using C++ and by enforcing domain-specific configuration decisions with AOP on the basis of AspectC++, a highly efficient software product line was the outcome. Table 1 shows some of the results, giving the memory footprints of individual products of the *nucleus family* of PURE. In this example, the nucleus member providing preemptive scheduling has been automatically generated by (1) reusing the two branches "non-preemptive scheduling" and "coordinative interrupt propagation" and (2) applying the synchronization aspect to that mixing. Thus, a new family member was generated automatically from an already existing system software product line by using AOP techniques. Non-preemptive scheduling functions which are critical in a 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.

5 Conclusion

Developing and maintaining software product lines of (embedded) operating systems largely benefits from aspect-oriented programming. By relying on an aspect language such as AspectC++, many non-functional properties can be expressed as aspects and, thus, separated from functional code. This significantly improves

reusability of that code. Typical cases of domain unspecific non-functional properties of an operating system are synchronization, protection, isolation, and sharing. For the domain of embedded systems, non-functional properties such as energy, timeliness, and dependability additionally need to be taken into account. An aspect weaver will take care of interweaving aspect code with functional code. Application of such a tool depends on configuration decisions related to the specific problem domain for which a specialized system software solution is going to be created. By considering the aspect weaver an ingredient part of an operating system workbench that supports feature-based configuration (e.g., by means of tools such as `pure::variants`), giving functional code non-functional properties becomes an automated process. The PURE development shows that 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 to understand an operating system as a software product line. The outcome was a solution that scales with the demands of many embedded systems. As indicated by the TAL case study, 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. AspectC++ evolved as a logical consequence from the PURE development and mainly was applied to selected PURE components in the course of re-engineering. With CiAO, the PURE successor, an aspect-oriented operating system is being developed now in which aspect orientation (and in particular AspectC++) plays the central role from the very beginning. Goal is to come up with an aspect-oriented operating system that, on the one hand, fulfills the many very specific requirements of deeply embedded systems and, on the other hand, improves reusability as well as maintainability of the respective system software better than PURE was able to do.

References

1. eCos homepage. http://ecos.sourceware.org/.
2. OSEK/VDX standard. http://www.osek-vdx.org/.
3. R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A.-F. L. Meur. On the automatic evolution of an os kernel using temporal logic and aop. In *18th IEEE Int. Conf. on Automated Software Engineering (ASE '03)*, pages 196–204, Montreal, Canada, Mar. 2003. IEEE.
4. D. Beuche. Variant management with pure::variants. Technical report, pure-systems GmbH, 2003. http://www.pure-systems.com/.
5. D. Beuche, A. Guerrouat, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. The PURE family of object-oriented operating systems for deeply embedded systems. In *2nd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '99)*, pages 45–53, St Malo, France, May 1999.
6. D. Beuche, O. Spinczyk, and W. Schröder-Preikschat. Fine-grained application-specific customization for embedded software. In *Proceedings of the International IFIP TC10 Workshop on Distributed and Parallel Embedded Systems (DIPES 2002)*, pages 141–151, Montreal, Canada, Aug. 2002. Kluwer Academic Publishers, ISBN 0-140207156-6.

7. Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In M. Akşit, editor, *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 50–59, Boston, MA, USA, Mar. 2003. ACM.
8. Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE '01*, 2001.
9. A. Colyer and A. Clement. Large-scale AOSD for middleware. In K. Lieberherr, editor, *3rd Int. Conf. on Aspect-Oriented Software Development (AOSD '04)*, pages 56–65, Lancaster, UK, Mar. 2004. ACM.
10. A. Colyer, A. Clement, R. Bodkin, and J. Hugunin. Using AspectJ for component integration in middleware. In *18th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '03)*, pages 339–344, New York, NY, USA, 2003. ACM.
11. J. Cordsen and W. Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *2nd Int. W'shop on Object Orientation in Operating Systems (I-WOOOS '91)*, pages 24–28, Palo Alto, CA, October 17–18, 1991.
12. K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* AW, May 2000.
13. M. Devillechaise, J. Menaud, G. Muller, and J. Lawall. Web cache prefetching as an aspect: Towards a dynamic-weaving based solution. In M. Akşit, editor, *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 110–119, Boston, MA, USA, Mar. 2003. ACM.
14. K. Driesen and U. Hölzle. The direct cost of virtual function calls in C++. In *11th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '96)*, Oct. 1996.
15. M. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch (1) considered harmful. In *10th W'shop on Hot Topics in Operating Systems (HotOS '05)*. USENIX, 2005.
16. A. Gal, W. Schröder-Preikschat, and O. Spinczyk. Open components. In *Proceedings of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, pages 75–78, Tampa, Florida, Oct. 2001.
17. A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *CACM*, 19(5):266–272, 1976.
18. W. Harrison and H. Ossher. Subject-oriented programming—a critique of pure objects. In *8th ACM Conf. on OOP, Systems, Languages, and Applications (OOPSLA '93)*, pages 411–428, Sept. 1993.
19. F. Hunleth and R. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *2002 Joint LCTES & SCOPES Conferences (LCTES/SCOPES '02)*, pages 38–45, Berlin, Germany, June 2002. ACM.
20. J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: Mobile networking for "smart dust". In *International Conference on Mobile Computing and Networking (MOBICOM '99)*, pages 271–278, 1999.
21. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *11th Eur. Conf. on OOP (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, June 1997.
22. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *ACM OSR*, 13(2):3–19, Apr. 1979.
23. D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *10th IEEE Int. W'shop on Object-oriented Real-time Dependable Systems (WORDS '05)*, pages 413–420, Sedona, AZ, USA, Feb. 2005.

24. D. Lohmann and O. Spinczyk. Architecture-Neutral Operating System Components. *23rd ACM Symp. on OS Principles (SOSP '03)*, Oct. 2003. WiP presentation.
25. D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat. On the configuration of non-functional properties in operating system product lines. In *4th AOSD W'shop on Aspects, Components and Patterns for Infrastructure Software (AOSD-ACP4IS '05)*, pages 19–25, Chicago, IL, USA, Mar. 2005. Northeastern University, Boston (NU-CCIS-05-03).
26. S. Matsuoka and A. Yonezawa. *Analysis of inheritance anomaly in object-oriented concurrent programming languages.* MIT Press, Cambridge, MA, USA, 1993.
27. D. L. Parnas. On the design and development of program families. *IEEE TOSE*, SE-2(1):1–9, Mar. 1976.
28. A. Rashid and N. Leidenfrost. Supporting flexible object database evolution with aspects. In G. Karsai and E. Visser, editors, *3rd Int. Conf. on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *LNCS*, pages 75–94. Springer, Oct. 2004.
29. F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. On interrupt-transparent synchronization in an embedded object-oriented operating system. In *3rd IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '00)*, pages 270–277, Newport Beach, CA, USA, Mar. 2000.
30. O. Spinczyk and D. Lohmann. Using AOP to develop architecture-neutral operating system components. In *11th SIGOPS European W'shop*, pages 188–192, Leuven, Belgium, Sept. 2004. ACM.
31. O. Spinczyk, D. Lohmann, and M. Urban. Advances in AOP with AspectC++. In H. Fujita and M. Mejri, editors, *New Trends in Software Methodologies, Tools and Techniques (SoMeT '05)*, number 129 in Frontiers in Artificial Intelligence and Applications, pages 33–53, Tokyo, Japan, Sept. 2005. IOS Press.
32. D. Tennenhouse. Proactive computing. *CACM*, pages 43–45, May 2000.
33. A. Tešanović, K. Sheng, and J. Hansson. Application-tailored database systems: a case of aspects in an embedded database. In *8th Int. Database Engineering and Applications Symp. (IDEAS '04)*, Coimbra, Portugal, July 2004. IEEE.
34. C. Walls. The Perfect RTOS, 2004. embedded world 2004.
35. M. Weiser. The computer for the 21st centrury. *Scientific American*, 265(3):94–104, 1991.
36. D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process.* Addison-Wesley, 1999.
37. C. Zhang and H.-A. Jacobsen. Quantifying aspects in middleware platforms. In *2nd Int. Conf. on Aspect-Oriented Software Development (AOSD '03)*, pages 130–139, New York, NY, USA, 2003. ACM Press.