Author's version. The original publication is available at www.springerlink.com. **Please cite as:** Wolfgang Schröder-Preikschat, Daniel Lohmann, Fabian Scheler, Olaf Spinczyk: *Dimensions of Variability in Embedded Operating Systems*. Informatik - Forschung und Entwicklung. Volume 22. Number 1 / December 2007: pages 5–25. DOI http://dx.doi.org/10.1007/s00450-007-0037-x

Wolfgang Schröder-Preikschat · **Daniel Lohmann** · **Fabian Scheler** · **Olaf Spinczyk**

Dimensions of Variability in Embedded Operating Systems

Eingegangen: 2007-03-02 / Angenommen: 2007-11-02

Abstract Design, implementation, and re-engineering of operating systems are still an ambitious undertaking. Despite, or even because, of the long history of theory and practice in this field, adapting existing systems to environments of different conditions and requirements as originally specified or assumed, in terms of functional and/or non-functional respects, is anything but simple. Especially this is true for the embedded systems domain which, on the one hand, calls for highly specialized and application-aware system abstractions and, on the other hand, cares a great deal for easily reusable implementations of these abstractions. The latter aspect becomes more and more important as embedded systems technology is faced with an innovation cycle decreasing in length. Software for embedded systems needs to be designed for variability, and this is in particular true for the operating systems of this domain. The paper discusses dimensions of variability that need to be considered in the development of embedded operating systems and presents approaches that aid construction and maintenance of evolutionary operating systems.

Keywords embedded systems · operating systems · specialization · customization · portability · non-functional properties · cross-cutting concerns

Zusammenfassung Entwurf, Implementierung und innerer Strukturwandel von Betriebssystemen ist nach wie vor ein anspruchsvolles Unterfangen. Trotz, oder gerade auch wegen, der langen Geschichte von Theorie und Praxis in diesem Bereich, ist eine Anpassung bestehender Systeme an Umgebungen mit anderen Bedingungen und Anforderungen als ursprünglich festgelegt oder angenommen alles andere als einfach in funktionaler wie auch nichtfunktionaler Hinsicht. Dies gilt insbesondere für die Domäne eingebetteter Systeme, die einerseits nach hoch spezialisierten und anwendungsgewahren Systemabstraktionen verlangt und sich andererseits aber auch leicht wiederverwendbare Implementierungen dieser Abstraktionen wünscht. Der zuletzt genannte Aspekt gewinnt mehr und mehr an Bedeutung, da eingebettete Systeme immer kürzer werdenden Innovationszyklen unterworfen sind. Software eingebetteter Systeme muss daher im Hinblick auf Veränderlichkeit entworfen werden, was vor allem für die Betriebssyteme dieser Domäne gilt. Der Artikel diskutiert Dimensionen von Veränderlichkeit, die bei der Entwicklung eingebetteter Betriebssysteme zu berücksichtigen sind und stellt Ansätze vor, die der Konstruktion und Wartung evolutionsfähiger Betriebssysteme behilflich sind.

Schlüsselwörter Eingebettete Systeme · Betriebssysteme · Erweiterbarkeit · Spezialisierung · Portabilität · nichtfunktionale Eigenschaften · querschneidende Belange

CR Subject Classification C.3 · D.2.11 · D.2.13 · D.4.7

This work was partly supported by the DFG, grants SCHR 603/4 and SP 968/2.

Wolfgang Schröder-Preikschat
Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4
Martensstr. 1
D-91058 Erlangen
Tel.: +49 (0)9131 8527278
Fax: +49 (0)9131 8528732
E-Mail: wosch@informatik.uni-erlangen.de

Daniel Lohmann
Tel.: +49 (0)9131 8527904
E-Mail: lohmann@informatik.uni-erlangen.de

Fabian Scheler
Tel.: +49 (0)9131 8527909
E-Mail: scheler@informatik.uni-erlangen.de

Olaf Spinczyk
Technische Universität Dortmund
Informatik 12, AG Eingebettete Systemsoftware
Otto-Hahn-Str. 16
D-44221 Dortmund
Tel.: +49 (0)231 7556322
Fax: +49 (0)231 7556116
E-Mail: olaf.spinczyk@udo.edu

1 Introduction

The design and development of an operating system has to reflect numerous constraints predefined by its application

domain. This domain consists, among others, of application software at the top and computer hardware at the bottom, thus with the operating system in between "a rock and a hard place". There are many different sorts of application programs and a manifold of hardware devices (for storage, execution, and input/output) the operating system has to take care about. The decision on a proper function set to support all (or even only a subset of) these programs already in light of a single given hardware platform equals a tightrope walk and, in most cases, results in compromise solutions:

> Clearly, the operating system design must be strongly influenced by the type of use for which the machine is intended. Unfortunately it is often the case with 'general purpose machines' that the type of use cannot easily be identified; a common criticism of many systems is that, in attempting to be all things to all individuals, they end up being totally satisfactory to no-one. [26]

The problem becomes even more serious with so called non-functional properties which, in addition to ordinary functional properties such as thread, memory, and address-space management or file handling, have to be provided by any sort of operating system, although in different flavours. Examples of those non-functional properties are the mode of operation of a computing system (such as single/multi user, real-time, time sharing, etc.) or a certain quality of service to be ensured for the application. General purpose machines are highly vulnerable to malpractice or even malfunction in this regard. In addition, it is fairly difficult if not impossible to make them behave as needed, either due to absence of dedicated system functions, by reason of inability to eliminate unused function implementations or, most notably, because of a poorly organized system software structure. All general purpose machines being in daily use of today largely share the same heritage: their system software has been designed not in a way that eases extension and contraction—as it actually should have been the case since the seventies of last millennium [35].

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 idea on 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 $1l/100km$ additional fuel consumption due to the weight of all the network cables [42; 24]. 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 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. Alone relying on, for example, 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 supplementing approaches are required in order to benefit from object orientation if one wants to develop system software that is reusable and customizable at the same time.

In the following, experiences made with the exploitation of well-known software-engineering approaches in the design and development of embedded operating systems are discussed: the program family concept [34; 14], feature modelling [9], and aspect-oriented programming (AOP) [22]. The three approaches are in strong coherence, not only with respect to their history of development. A program family combines the two properties "reusability" and "specialization". The former relates to common functions shared by some family members, while the latter refers to the different functions that distinguish family members from each other. Embedded operating systems need to be designed and implemented as a program family, primarily specializing in a given application domain while it is highly desirable to assemble them from as many reusable building blocks as possible. Feature modelling appears to be the suitable technique to circumstantiate the common and variable properties of and, thus, to organize a program family. Finally, AOP is a technique that allows one to rework a reusable software asset for the purpose of customization/specialization.

It is argued that operating systems must be a software product line [44] in order to be specifically prepared for present and future challenges in the embedded systems domain. Motivation of this view is drawn from own experiences in the development of various operating systems for the desktop, parallel, and embedded systems domain [37; 38; 4]. Most of the ideas presented get realized in the scope of the ongoing CiAO [28; 27] project.

2 Causes of Variability

Variability of operating systems comes in different flavours: it may originate from horizontal and/or vertical changes in order to add, remove, port, or specialize functions. These changes can be further classified as static or dynamic, with the former being carried out before and the latter during operating-system runtime. In the matter in hand, before runtime means at configuration, compilation, binding, or loading time.

In the following, dynamic changes will not be considered mainly because of two reasons. Firstly, prerequisite for a dynamically alterable system is a software structure that aids static changes. It is known for a quite long time that only "a well structured system can easily be understood and modified" [12]. Above all, this implies a kind of holistic software design methodology. If one is unable to identify modularized ("loosely coupled") sections in a program, attempts to restructure this particular program dynamically can hardly be put into practice, if at all: design for static changes comes before the dynamic case. Secondly, it is out of character to dynamically change software structures of embedded systems. For quite a large number of embedded systems, not only is scarceness of resources (in terms of memory and energy, e. g.) a handicap for carrying out dynamical changes but also the need for a sustained guarantee of a certain quality of service or adherence to (soft, firm, hard) time limits or safety rules.

The ability to changes is motivated in many respects. Examples are debugging [45], but also optimization, maintenance, evolution, or customization. The following subsections discuss different dimensions of variability which are important to be reflected in a design and implementation structure of an embedded operating system. Most of the arguments presented also hold for other kinds of system software, and for application software as well. In the discussion, a view is taken on a software system that is hierarchically organized in layers or levels [32].

2.1 Horizontal Changes

With functions being added to or removed from a given layer the software system gets changed in a horizontal means. This is, for example, the case when the operating-system interface is extended by new functions or contracted by functions no longer used.[1] Behind that interface, at lower levels, the horizontal changes made above may continue downward, depending on the call and/or *"uses" relation* [33] defined by the original hierarchically structured system. (The difference between these two relations is briefly explained next, at the end of this subsection.) This logical continuation of restructuring, however, is not what is understood in the next subsection as "vertical change"

[1] Characteristic functions of this sort are system or supervisor calls such as those ones described in "manual section 2" (`man(2)`) of a Unix-like operating system.

Extension Major goal by adding a new function is to reuse existing functions or implementations as much as possible. In some cases this may be straightforward, namely when the existing assets are ready for being reused. In other cases (some of) the existing functions need to be restructured in order to make parts of them reusable in the course of implementation of the new function to be added. Whether or not the one or other way can be taken depends on the functional *and* non-functional properties of the existing functions. So there may also be cases in which reuse is not practicable at all and implementation from scratch as undesirable consequence arises.

Contraction At first sight it appears as if the removal of an unused function from the system is a trivial task—but this is a sophism. If at design time the option to remove functions later on has not been considered, or even has been simply forgotten, contraction of existing (system) software may be a cumbersome undertaking. A system whose software design eases extension (by adding new functions) also eases contraction (by removing the added functions later on).

Removing a function being part of a call relation usually results in a binding error, at latest, when the complete operating system is going to be assembled and generated. If the respective function, however, is part of a "uses" relation, its non-existence must not necessarily be reported by an error message before runtime. Note that "uses" does not mean "call" but rather "existence" of a correct implementation of a specific function or set of functions [33]. For example, an interrupt handler is "used" but never called by any program in the system. As a consequence, removing a function being part of a "uses" relation may result in uncapable system operation and, at best, becomes directly apparent through some kind of runtime error.[2]

2.2 Vertical Changes

Operating systems are software products of typically long lifetime. There are a number of species in the general as well as special purpose sector today who can look retrospectively at a product history of two to almost three decades, i. e., dated back to the mid-seventies of last century. UNIX [43] and Unix-like operating systems fall into this category, or Multics ([8; 31], 1963/69–2000), but also special-purpose systems such as QNX [18], first released at the beginning of the eighties. Since their first appearance at the horizon, the systems were subject to a number of adaptations because of new hardware platforms they should run atop and different application environments they should support. The changes to the system software mostly were of vertical nature: existing

[2] In the context of the development of Mach [1], rumours were afloat saying that the fairly large massiness of the "microkernel" was because removal of low-level file handling functions resulted in an inoperable system and, thus, was not a choice. Provided that it was more than a rumour, this Mach experience is a sign for a badly organized "uses" relation between the various kernel building blocks.

functions were adapted to a different use case while maintaining the same (functional) interface specification.

Porting Adaptation of some existing implementation to changes coming up with an alteration in the (real/virtual) machine having been used so far for information processing is generally known as porting. The ideal picture is that, in order to move the software system to a new (different) machine, only minor changes will be necessary if all the machine-dependent parts are encapsulated by a small set of dedicated abstractions, sometimes termed "port package" or hardware abstraction layer (HAL). In functional respect this ideal picture became reality and is documented by a number of research and commercial systems being in service today. Nevertheless, porting an operating system by rewriting the port package cannot be considered a trivial task. The real challenge in porting, however, is to make sure that the transported software system behaves as expected and, e. g., has been known from experiences made with its former installation—and this has a lot to do with the non-functional properties the system showed before and one is aware of.

The real-time issues of embedded systems give examples in which changes with respect to non-functional properties may entail serious problems (not only) at application level. Processor speed, memory access latency, cache behaviour, pipelining effect, or processor architecture (CISC vs. RISC) in general are hardware attributes which have direct or indirect impact on the non-functional properties of a software system. Other critical attributes are the signalling of interrupts by the hardware (edge- vs. level-triggered), especially when critical sections are protected by using some kind of "disable/enable interrupts" pattern. As a consequence, porting system software may raise the need for vertical changes in the context of some higher-level software in order to further ensure correct system functioning. Not rarely do exist situations in which these changes are cross-cutting concerns and have to take effect at several locations of system software far above the layer made up by the port package.

Changes of non-functional system properties may also discover design or implementation defects or shortcomings, which make porting even more complicated. Particularly problematic are race conditions that had no implication on the original system but bobbed up suddenly and unexpectedly in the ported system.[3] This is the good case, because a software flaw will be realized and corrected. The bad case happens the other way round, namely when changes in non-functional system properties introduce (those or other kinds of) undetected software flaws.

Specialization Vertical changes in the course of porting system software have their origin at the hardware interface and propagate upwards. Specialization goes the other way round and refers to vertical changes that have their origin at the application (i. e., system call) interface and propagate downward. It starts with re-implementation of an existing function and may cause further vertical or horizontal changes at lower layers. Motivation behind the change may be the correction of some software error or the improvement of a function in regard of higher execution speed, smaller memory footprint, or less energy consumption. Thus the change concerns a certain non-functional property of an existing system function.

Usually, specialization of a function is not directly cross-cut in that it propagates in horizontal means to an unrelated function.[4] But indirectly there may be cases where the changes made have strange effects in this regard. Assume function f_x gets specialized, which then propagates downward because the new implementation of f_x may perform even better if function f_z, called by f_x, is specialized too (but, maybe, in a different fashion than f_x). Further assume, function f_z is also used (i. e, called) by function f_y: f_z is shared by f_x and f_y. The changed non-functional property of f_z may impact correct operation of f_y much in the same way as has been discussed before with porting. Thus, side-effect of changes being propagated downward may be changes that propagate upward to some other function and indirectly influence the runtime behaviour of that very function. The changes made to f_z in order to improve performance of f_x must be conform not only to the "uses" relation between f_x and f_z but also to the one between f_y and f_z. In other words: the implementation of f_z must be correct in respect of the specifications of both f_x and f_y.

2.3 Summary

Purpose of the discussion was to provide an insight into the complexity of undertaking changes to software systems in general and operating systems in particular. The necessity for changes is unquestioned, but they must take place in a controlled manner and should be free of side-effects. The latter cannot be always guaranteed, why it is essential trying to organize system functions in a "uses" hierarchy [33]. It turns out that finding the "uses" hierarchy is anything but simple, and the right one more than ever.

Based on a case study from the operating-systems domain, the following section takes up and deepens some of the issues discussed here. Goal is to provide a further motivation for the software-engineering approaches put forward in the sections thereafter.

3 Case Study: Flow Control

The closer one approaches to implementation, variability issues become more and more obvious and largely dictate the

[3] Own experiences from having ported the kernel of the PEACE [38] parallel operating system from a MC68020- to an i860-based machine: it took several weeks to identify an undetected critical section of no longer than 10 lines of C++ code, and less than one hour to come up with the portable solution.

[4] Apart from magic effects a different memory layout may have or which may come in because of branches to differently aligned addresses. Meant is slower or faster program execution due to changes in the physical representation of code and data sections, depending on the underlying processor in use.

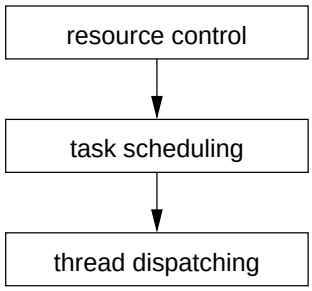


Fig. 1 Functional hierarchy of (operating system) abstractions used to implement flow control for multi-threaded computing systems.

anyway demanding act of systems programming. Note that these issues do already exist at design time, they belong to what is sometimes paraphrased as "operating-systems expertise". Adoption of this specific domain knowledge is an iterating process of design and implementation. In the following some technical difficulties in the design and development of embedded operating systems are exemplified. Goal is to put over the problem that comes across with certain technical facts that originate various cross-cutting concerns of non-functional properties. The case study is on process management in general and flow control in specific.

Flow control in operating systems can roughly be separated in three major building blocks, the implementation of each of which exhibiting various non-functional properties of partly cross-cutting character. These blocks are resource control, task scheduling, and thread dispatching. Figure 1 shows the functional hierarchy which is typically defined between them. More precisely, in the given case this hierarchy describes both the call and "uses" relation.

3.1 Resource Control

The classical mechanism for controlling access to shared resources is the semaphore [10]. This well-known concept has been implemented in sheer countless versions. For ease of discussion, we will focus only on the classical primitives `P()` and `V()` as defined for a counting (or general) and binary semaphore. Nevertheless, all the different versions have one in common, which is the point of interest in the following: semaphore primitives `P()` and `V()` are to be regarded as *indivisible operations*.

In consideration of the different dimensions of variability discussed in the previous section, a semaphore is potential subject to both horizontal and vertical changes. Note that a semaphore (or a similar mechanism) is required only if resource control is an issue of the system. This means:

1. the presence of concurrently executing threads sharing at least one *reusable indivisible resource* or interacting on the basis of *consumable resources* and
2. the need for coordination of the concurrently executing threads and their resource accesses at runtime.

Neither of the two aspects must be given in all use cases, in particular not for special-purpose operating systems. First, there are embedded applications which do not require threads in order to model concurrency, but rather rely on events such as (hardware) interrupts to enable non-sequential execution of the programs and otherwise operate in a strictly cooperative manner. Due to its blocking nature, semaphores are not applicable for resource control within event/interrupt handlers and, in this particular use case, would therefore be meritless. Second, in case of time-triggered systems coordination of threads has been done before runtime. The system is given a well-organized but static thread schedule free of any resource access conflicts. Again, there is no need for a semaphore mechanism to be implemented by the underlying operating system. Third, even for event-triggered systems semaphores are not a must at runtime: by relying on stack-based scheduling [3] of tasks/threads flows of control can be created and maintained so that semaphores consume nothing but memory resources and are neither called nor "used". Thus, a semaphore mechanism is definitely subject to horizontal changes in an embedded operating system.

The issue of vertical changes is a bit more complicated, as it is general a subject of a larger and "obscure" class of problems. This issue relates, on the one hand, to the various techniques one can employ to ensure indivisible semaphore operation and, on the other hand, to the pattern how a semaphore can be used to propagate signals (i. e., consumable resources) from device driver level to threads. Furthermore, this issue also relates to the question of whether or not the implementation should be keeping track about blocked threads in a waiting list private to a semaphore instance and the dependency of the thread scheduling discipline on such an implementation. That is to say, this issue relates to the "uses" hierarchy, namely whether task scheduling is above or below resource control. The following two paragraphs elaborate on this issue.

Indivisible Operation A semaphore implementation is a typical case for a *conditional critical region* [19]. Fundamental techniques for protection of such sort of critical region are, e. g., *mutual exclusion* by using lock variables, *inhibit preemption* by making it a kernelized monitor [30], or *non-blocking synchronization* by relying on dedicated elementary operations of the underlying processor [17; 29]. The latter has strong consequences with respect to vertical changes downward in that a tightly coupled implementation of scheduler and semaphore is demanded. Creating a "uses" relation between the two building blocks is hard, if not impossible, because of the mutual dependencies in their implementations.

The aforementioned technique, mutual exclusion, makes the implementation unsuitable for use at device driver level. Normally, a `V()` could be used to produce a consumable resource (i. e., a signal) a thread at some higher level wants to consume (using `P()`) in order get aware of some device-related event. The functional property of `V()`, which generally is of non-blocking nature, in order to indicate availability of such a resource would indeed enable this use pattern, but not its non-functional property as implied through mutu-

al exclusion. Thus, `V()` may block and force an interrupted thread into a deadlock (e. g. when in overlaps execution of the `P()` of the thread who wants to await the signal: the interrupted thread may deadlock itself).

Last but not least there is the technique (inhibit preemption) which, when being employed, makes the semaphore implementation a *non-preemptive critical section*. There are basically two approaches to achieve this. Common to both is that the causal event that may lead to preemption is disabled for the length of the critical section. The difference is in the type of event, namely whether it relates to *interruption* or *resumption* of program execution. In the former case are interrupts to be disabled and enabled, in the latter case is thread dispatching to be delayed. Strictly speaking, one has to discriminate between disabling either hardware or software interrupts. Which way to go depends on non-functional properties such as edge- or level-triggered hardware interrupts and the execution time of the critical sections inside `P()` and `V()`, in particular whether or not this time is bounded. The latter aspect is largely determined by task scheduling and, if the semaphore keeps track of blocked threads, the queuing discipline of the waiting list when `P()` blocks the calling and `V()` unblocks a waiting thread.

Scheduler Dependency If the semaphore consists of a waiting list of threads blocked in `P()`, the queuing discipline employed must be consistent with the one implemented by task scheduling. A simple first-in, first-out (FIFO) method is prone to malfunction if task scheduling happens in a priority-oriented manner. In such a situation namely `V()` may cause *priority violation* when, according to FIFO, the next thread going to be removed from the head of the waiting list is not the one of highest priority. The effect is that task scheduling will no longer be able to perform its operations correctly and, thus, tasks (or threads) may miss their deadlines.

The design decision to use a semaphore waiting list implies that the correct operation of task scheduling depends on the existence of a correct implementation of resource control with respect to the specification of task scheduling. In other words: this decision lets task scheduling "use" resource control—and vice versa. Assuming this design decision, figure 1 only shows the call relation but no correct "uses" relation.

Problem is the mutual "use" of resource control and task scheduling. A correct "uses" hierarchy is acyclic [33]. If two programs depend on, or may take benefit from, each other the thus created cyclic "uses" relation must be broken. This is done by *sandwiching* [33] one of the two programs. In our example a "double sandwich" needs to be established. This is because `P()` depends on task scheduling which depends on `V()` which, in turn, depends on task scheduling. The correct "uses" relation shows figure 2, given the assumption that a semaphore implements its own waiting list of blocked threads.

Figure 2 implies a subsystem implementation of higher (structural) complexity than a subsystem that follows a "uses" relation as described by figure 1. Nevertheless, maintaining a semaphore waiting list may have advantages in regard to better performance when threads need to be unblocked. The "uses" hierarchy helps indicate dependencies of the sort just mentioned and that must be taken into account for potential vertical changes.

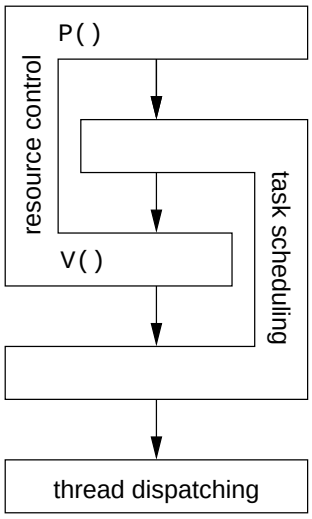


Fig. 2 "Uses" hierarchy of (operating system) abstractions used to implement flow control for multi-threaded computing systems, supposed resource control implements a waiting list of blocked threads.

3.2 Task Scheduling

Like resource control so is also task scheduling potential subject to horizontal change: in a time-triggered system did not only resource control took place off-line, namely before runtime of that very system, but also task scheduling. For these kinds of applications an embedded operating system is not equipped with a scheduler, as no dynamic (on-line) scheduling of tasks/threads happens to take place. This is different to an event-triggered (priority-oriented) system which, as a matter of fact, "uses" on-line task scheduling.

Vertical changes to the task scheduling building block typically come with the actual scheduling discipline needed to support a particular application. There are numerous disciplines one may choose from. In the following, only the classification profiles are discussed in conjunction with the impact they have on the interaction of the three flow control building blocks (fig. 1 or fig. 2).

Cooperative scheduling assumes that threads voluntarily call the scheduler to relinquish processor control and allocate the processor to some other thread. The non-functional property given to all scheduling disciplines of this class is the absence of any critical section in the system that otherwise comes into existence because of thread preemption.

Interrupt-driven scheduling is an extension to cooperative scheduling. The scheduler gets activated by some device interrupt, performs housekeeping according to its scheduling

discipline, but does not preempt the interrupted thread. Logically, the scheduler makes up a critical section and thus needs to be synchronized proper. Several synchronization options do exist for this particular case. The various sorts my work non-blocking [29; 13], wait-free [17], or *constructive* and, thus, tolerate overlapped execution of scheduler functions, or they disable the causal event responsible for overlapping. More precisely, the latter case means to either disable interruption of thread execution or delay continuation of a specific scheduling action as long as the critical section is active [36]. Note that these techniques (disable interrupt/-delay continuation) are transitive and implicitly synchronize functions that are part of a call relation out of the critical section. That is to say, thread dispatching would automatically run in synchronized mode. Constructive synchronization means that, by design, the choice of data structures and algorithms ensures the absence of any race condition in the system. All scheduling disciplines of this class have the same non-functional properties as cooperative scheduling and they introduce new non-functional properties to the system due to and depending on the kind of synchronization.

Preemptive scheduling is an extension to interrupt-driven scheduling. In difference to interrupt-driven scheduling, the scheduler may decide to preempt the interrupted thread. However, this preemption takes place only at specified *preemption points* of the abstract processor "operating system". These points may be spread in fairly large and, maybe, irregular offsets across system software. For example, entry and/or exit of any scheduler function call could be such a (small-grained) preemption point just as entry and/or exit of any system call (coarse-grained). All scheduling disciplines of this class have the same non-functional properties as interrupt-driven scheduling.

Full preemptive scheduling is an extension to preemptive scheduling. In difference to preemptive scheduling, thread preemption may take place at any time whatsoever. Strictly speaking, any instruction of the underlying (physical) processor and being accessed by a thread during execution denotes a preemption point. All scheduling disciplines of this class have the same non-functional properties as preemptive scheduling, except that for synchronization one now has the choice only between non-blocking, wait-free, or constructive.

3.3 Thread Dispatching

Coming to the decision which of the many threads ready to run will be allocated the processor for execution is generally known as scheduling and will be done by the task scheduling building block. Conversion of this planning order, thus assigning the processor to the selected thread, is also known as dispatching. The latter is what thread dispatching is about. Note that, in contrast to task scheduling and resource control, thread dispatching never will be subject to horizontal change as long as a multi-threaded mode of operation is requested by the application.

Logically, thread dispatching represents a critical section. The steps which are to be passed through are (single processor case):

1. Saving of the processor state of the releasing thread into a state buffer local to that very thread.
2. Refreshing of the pointer to the descriptor of the currently executing thread.
3. Restoring the processor state of the acquiring thread from a state buffer local to that very thread.

Just as a physical processor, e. g. of IA-32 type, needs an instruction pointer (or program counter) in order to identify the current machine instruction in the program to be executed, an abstract processor like an operating system needs a thread pointer in order to identify the current activity in the system to which resource management must actually be effective. The crux is that these two pointers must be updated as an *elementary operation* of the abstract processor in order to correctly switch between two threads. That is to say, the update must be instantaneous i. e. indivisible. Strictly speaking, step 2 and step 3 together must be indivisible, otherwise it may happen that, in case of the currently executing thread is getting preempted, step 1 overwrites the not yet restored processor state of the currently executing thread.

This example shows that thread switching depends on non-functional properties defined by the task scheduling building block of the flow control subsystem. In fact, thread switching bears a race condition in case of (full) preemptive scheduling, as necessary condition, and non-transitive (i. e. non-blocking, wait-free, or constructive) synchronization of task scheduling, as sufficient condition.[5] If task scheduling employs some kind of transitive synchronization, thread switching implicitly is indivisible and, thus, synchronized too. Otherwise one is spoilt for choice of the synchronization technique proper to make thread switching atomic. These techniques maybe the same as discussed with task scheduling in the previous subsection.

There is a way to make thread switching independent of the synchronization technique used by task scheduling. The idea is to have the two pointers to be updated only in a logical sense, i. e. to map them to a single physical pointer and provide a function that "computes" both from that very pointer. Necessary condition is that the physical processor is capable of an operation that allows for an indivisible write to a memory location storing a pointer of the abstract processor defined by the programming language (e. g. C/C++) used to implement the operating system. Most physical processors come with appropriate machine instructions, or in other words: in most cases matches the pointer size as defined by the abstract processor the one of the physical processor. Given these assumptions, constructive synchronization

[5] This requires to correct the so far developed "uses" hierarchy: task scheduling again needs to be sliced into another two parts with the upper part "using" thread dispatching which, in turn, "uses" the lower part of task scheduling.

of thread switching becomes possible, i. e., explicit use of synchronization primitives is unnecessary as the race condition of thread switching disappeared.

An implementation of this idea typically uses the stack pointer of the physical processor as actual parameter to the mapping function. The descriptor of a thread is stored at the bottom of the runtime stack of that very thread. In addition, all the stacks are of same maximum size, which must be a power of two, and are aligned according to that size. Computation of the pointer to the descriptor of the currently active thread may be achieved as sketched by the following C-like function: $(sp() \mid (2^N - 1)) - sizeof(TD) + 1$, with TD representing the descriptor of the currently executing thread identified through the actual value ($sp()$) of the stack pointer. Note that TD may be either the thread descriptor thereat or a pointer thereon, depending on the descriptor size and the leeway of permanent local storage on the runtime stacks.

This way, thread switching gets constructively synchronized by calling some kind of `resume()` procedure to implicitly get the instruction pointer saved as return address on the runtime stack. Within `resume()`, the stack pointer is changed to the top of stack of the thread whose execution is going to be resumed. This step implicitly gets the thread pointer changed to the descriptor of that very thread. The critical operation is writing a memory address into the stack pointer, which must be atomic.

3.4 Summary

The discussion revealed numerous points of variability that come into play the closer one approaches to implementation. Knowingly, the description was on a fairly detailed technical level in order to give an idea of the difficulties in designing adaptable system software (not only) for embedded systems. By far not all options have been handled. The discussion showed how certain non-functional properties crosscut different software functions in a way that seriously complicates system configuration.

In the following three sections approaches are briefly presented that aid the construction, configuration, and adaptation of system software. These approaches are program families, feature modelling, and aspect-oriented programming.

4 Family-Based Design

> We consider a set of programs to be a program family if they have so much in common that it pays to study their common aspects before looking at the aspects that differentiates them. [35]

Generally, in the design and development of any kind of software this guideline must always be at the back of one's mind. Family-based design means a design that aids reuse of software assets for a very broad range of application domains. Today, the notion of a software product line [44] resounds throughout the land in conjunction with reusable and yet application-aware software. Software product lines extend program families by a strong methodology that supports the complete process of software development and maintenance.

In the context of operating-system development, the key issues of family-based design is to start with a minimal subset of system functions that is, by definition, reusable for any kind of application domain. That is to say, no design decisions have been met that would prevent reuse of that minimal subset. Step by step this subset gets extended by minimal system extension by means of functional enrichment. Again, the extensions are minimal and made with high degree of reusability in mind. The closer one gets to the application, the more specialized and, thus, the less reusable an extension will be. Art of this bottom-up design process is to find ways that allows one to postpone design decisions related to specific application requirements as far as possible. Outcome is a distinctive functional hierarchy consisting of numerous and sometimes utmost slim functions or abstractions.

Large misbelief is that those kinds of multi-layered software systems cannot be turned into highly efficient (object-oriented) implementations:

> It is the system design that is hierarchical, not its implementation. [14]

Macro programming, function inlining, implementation inheritance, multiple inheritance based on multiple inclusion, template-based meta-programming, and aspect-oriented programming give examples on how to be able to come up with a mostly flat system representation of a fairly small memory footprint [4]. The following subsections sketch this design philosophy by means of excerpts from a threads package implementation.

4.1 Minimal Subset of System Functions

Development of the minimal subset of system functions starts with sketching the idea on the intended use case of threading. This idea becomes manifest in the program shown in figure 3 and is based on two fundamental design decisions:

1. a thread is implemented as a coroutine [6] and
2. creation of which follows a fork-style of programming [7].

In that program, function `assume()` corresponds to a fork and `resume()` serves thread switching by reactivating the coroutine associated with some thread. Destruction of a coroutine (i. e. thread) takes place automatically when leaving the scope within which the coroutine has been declared (in the given case, `main()`). Actual declaration of the coroutine means to allocate a stack instance (`pool[]`) and coroutine pointer (`son`) used for identification. The **`new`**-Operator serves generation of the coroutine pointer with respect to alignment restrictions as dictated/recommended by the physical processor. This pointer becomes the stack pointer of the declared coroutine. Depending on whether the stack of the physical processor expands downward or upward, **`new`** computes a well-aligned address to the top of a "virgin" stack.

```
#include "luxe/Act.h"

#define STACKSIZE 64
#define LEEWAY 16

int main (int argc, char *argv[]) {
    static Act *son, *dad;

    char pool[STACKSIZE];
    son = new(pool, STACKSIZE - LEEWAY) Act;

    if ((dad = son->assume())) {
        for (;;) {
            // son working...
            dad = dad->resume();
        }
    }
    son = son->resume();

    // dad working...
}
```

Fig. 3 Strawweight thread: use case.

```
#include "luxe/type/size_t.h"
#include "luxe/machine/pc_t.h"

class Act {
protected:
    pc_t tbc;          // where to be continued...
public:
    void* operator new (size_t, char*, size_t);

    Act* assume ();    // create act
    Act* resume ();    // switch act
};
```

Fig. 4 Strawweight thread: abstraction.

```
#include "luxe/Act.h"

Act* Act::assume () {
    asm ("movl 4(%esp), %eax");
    asm ("movl (%esp), %edx");
    asm ("movl %edx, (%eax)");
    return 0;
}

Act* Act::resume () {
    register Act* aux;
    asm ("movl %%esp, %0" : "=r" (aux));
    asm ("movl 4(%esp), %esp");
    return aux;
}
```

Fig. 5 Strawweight thread: implementation (IA-32).

At bottom layer, the context of a coroutine is made up of nothing else but a *resumption address*. This address, an instance of type `Act`, always is stored on top of the stack of a suspended coroutine. Thus, whenever a coroutine suspends execution it saves its resumption address on its stack (initially, the first object on that stack). This is implicitly done by having a procedure call in charge of coroutine switching. This procedure is implemented by function `resume()`, which is called by one coroutine and returns to another coroutine. Return value of this function is the pointer to the coroutine which recently called `resume()`, thus, suspended execution.

Thread creation is always concerned with the issue of giving a thread an initial runtime context from which it can start execution. In the case discussed here, this context is defined by the program itself, more precisely, by the control flow which creates the coroutine of a thread. Figure 3 shows what that means. Coroutine creation is accomplished by a call to function `assume()`. The effect of this call is to leave an initial resumption address, i. e., an instance of type `Act`, on the stack of the established coroutine (`son`). This address is the return address of `assume()`. In such a setting `assume()` returns twice and the return value indicates which control flow returns. The first return, `assume()` $\rightsquigarrow 0$, goes to the creator of the coroutine. In contrast, the second return, `assume()` $\rightsquigarrow$ `toc`, goes to the created coroutine, with return value `toc` $\neq 0$ and identifying the coroutine which did the initial `resume()`. The **`for`**-loop shown in figure 3 implements the body of the new coroutine, which, in a complete system, will consist of additional statements that "bootstrap" a thread in an application-aware manner.

The interface to the abstraction (`Act`) that implements the simple coroutine concept just sketched and which lays the basis for more complex thread concepts is shown in figure 4, its implementation is shown in figure 5. At this level of abstraction, the state private to a thread consists only of a stack pointer, which actually is a pointer to an instance of type `Act`, and a resumption address, which is indeed that `Act` instance being pointed to by the stack pointer (`Act*`). All other state of a thread is shared with all other threads (of this type) of the same program, which in particular also holds for the residual processor registers.

Threads of type `Act` are "strawweight" because thread switching only means to (1) save the resumption address of the running coroutine (as side-effect of the call to `resume()`), (2) remember the stack pointer as return value, (3) switch the runtime stack by overwriting the stack pointer, and (4) continue execution of the specified coroutine (as side-effect of the return from `resume()`). In addition, the creation of threads of this type means to (1) setup the initial resumption address of a new coroutine (as side-effect of the call to `assume()`), (2) copy that address to a memory location specified by the pointer to the coroutine of the thread going to be created, and (3) return 0. In fact, `assume()` constructs an `Act` instance at a memory location specified through a parameter—in terms of C++, this function plays the role of an `Act` constructor.

4.2 Minimal System Extensions

Providing thread concepts of higher "weight class", i. e., bigger processor state, is a typical case of minimal system extensions to the level of abstraction implemented by `Act`. For this purpose, `Act` gets functionally enriched through implementation inheritance by inheriting its properties to template class `Flux` (fig. 6) and providing extensions that lead to implementations of different non-functional properties in terms of execution speed and memory consumption (see table 1). The parameter (fig. 7) to `Flux` specifies the weight class of

```
#include "luxe/Act.h"
#include "luxe/machine/FluxVariety.h"

template<FluxVariety T>
class Flux : public Act {
public:
    Act* induce (Flux<T>*&);
    Act* resume (Act&);    // switch flux
    Act* unwind (Act&);    // switch flux, inlined
};
```

Fig. 6 Abstraction of different weightily threads. Function `induce()` extends `Act::assume()` by inheriting the processor state to the created thread. Function `unwind()` is the **`inline`**-version of, and reused by, function `resume()` and performs the actual thread switch.

```
#include "luxe/machine/ActMode.h"

enum FluxVariety {
//  Strawweight   = Act,
    Flyweight     = GPR|OFP|OVR,
    Bantamweight  = GPR|OVR,
    Featherweight = GPR|BMR,
    Lightweight   = GPR
};
```

Fig. 7 Thread weight classes and their mapping to `Act` state saving modes: general purpose registers (GPR), omit frame pointer (OFP), omit volatile registers (OVR), block-move registers (BMR).

```
#include "luxe/Flux.h"
#include "luxe/machine/ActState.h"

template<FluxVariety T>
inline Act* Flux<T>::unwind (Act& next) {
    Act* peer;
    if (T & SOS) {
        ActState<T|BMR> *apr;
        apr = ActState<T|BMR>::stack();
        peer = next.resume();
        apr->clear();
    } else {
        ActState<T|BMR> apr;
        apr.cache();
        peer = next.resume();
        apr.apply();
    }
    return peer;
}
```

Fig. 8 Generic thread switching. Template parameter `T` specifies the thread weight class and whether the state is saved on/restored from the stack (SOS) using push/pop instructions if applicable or saved into/restored from a state buffer variable.

a thread instance created from that very parameterized data type.

Figure 8 shows a generic implementation of thread switching. The weight class (specified as template parameter) refers to the thread being switched away (i. e., the caller), not to the one being switched to. Another notably property of this implementation is that a thread not only saves but also restores its processor state self-contained. Strictly speaking, the thread who is switching away is not involved in restoring the processor state of some other, maybe different weightily thread. This approach allows switching between threads of different weight class, i. e., type. Common to all threads is the `Act` concept only and every thread of type `Flux` is required to "bootstrap" itself after having resumed execution. Note that this method of (low-level) thread switching is different from the conventional one in which the thread switching away is required to be of the same type as the thread to be switched to.

Table 1 Memory footprint of fundamental thread switching functions: `Flux<T>::resume()` ↦ `Flux<T|SOS>::unwind()`, IA-32. Listed are static (text, no data in this case) and dynamic (stack) memory requirements. Distinguished are needs for the call (left term) and the body (right term) of the respective function, with the sums giving the subtotal needs. The total need of a weight class computes from its subtotal plus the needs of strawweight thread switching.

weight class	static	dynamic	subtotal	total
straw	8+7	4+4		23
fly	11+11	8+12	42	65
bantam	11+13	8+16	48	71
feather	11+7	8+32	58	81
light	11+19	8+28	66	89

Table 1 shows the memory footprint of individual members of the `Flux` family (including `Act`) for IA-32 type of processors. The numbers give the non-functional properties in terms of memory consumption for each of the members. They demonstrate how the resource requirements of various members do vary with functionality. The figures are much more dramatic in case of PowerPC (G4) type of processors, for example, which range from 40 bytes (strawweight) up to 1152 bytes (lightweight) for the total amount of memory occupied. A byte saved, is a byte got: this is still of importance for embedded systems—and the presented design meets exactly these needs.

5 Feature Modelling

Feature modelling is understood as "the activity of modelling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a *feature model*" [9]. 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. Feature modelling is particularly suitable for capturing the common and variable properties of program families.

5.1 Feature Diagrams

Common is a graphical representation of the feature model in terms of a *feature diagram*. The diagram is of tree-like structure (see figure 9), with the nodes referring to specific feature categories. Four fundamental feature categories are defined: *mandatory*, *optional*, *alternative*, and *cumulative*[6]. A feature diagram describes the options and constraints that

[6] Feature category "cumulative" corresponds to category "or" as known, e. g., from [9]. This notion is used for better understanding in a general sense and is in the same diction as the other three concepts.

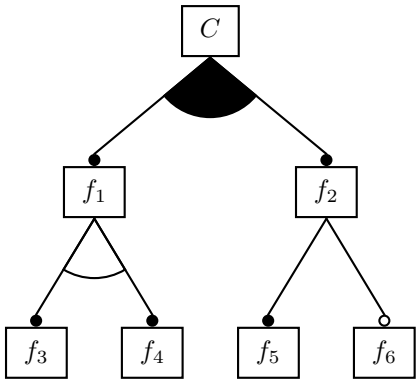


Fig. 9 Feature diagram: f_1 and f_2 are cumulative features of concept *C*, f_3 and f_4 are alternative features of f_1, and f_2 implies a mandatory feature f_5 and an optional feature f_6.

shall exist within a system. It models the variable and fixed properties of a family of software and hardware assets which implement that system.

The diagram shown in figure 9 describes a specific concept *C*, e.g. the flow control 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 cumulative 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 software systems becomes possible [5].

5.2 Flow Control Features

An example on how this technique can be used to describe interdependencies of the different variants of the flow control subsystem discussed in the previous section gives the feature diagram shown in figure 10. For the ease of understanding, the figure sketches an excerpt, only, and focuses on properties related to thread processing, thread synchronization, and event synchronization.

According to the semantics of the individual feature nodes (see figure 9), flow control may be either cooperative or preemptive. Only in the case of preemptive thread processing, the need for coordination of concurrently executing threads of control arises. Note that for a number of use cases, especially those ones of the deeply embedded systems domain, it suffices when the operating system provides cooperative (task) scheduling. This makes sense for applications programs whose tasks are not run in an event-driven but, maybe, time-triggered mode or who implement event handling on its own for whatever reason.

Preemptive scheduling of threads calls for synchronization measures for which a number of options do exist in the literature. Whether a single option is suited to help solve all sorts of synchronization problems depends on the actual use case. Sometimes a single option suffices. At some other time several options are required or beneficial, each of which optimized with respect to a certain problem or class of synchronization. This is reflected by modelling preemptive scheduling as a cumulative feature and, thus, allowing for system configurations that support any combination of synchronization techniques.

In this setting, constructive synchronization stands for concepts and techniques which, at design time, help to prevent the need for explicit synchronization at runtime. An example gives the technique described in section 3.3, namely the use of a function in the course of thread dispatching in order to map the pointer to the current thread of control to the stack pointer of that very thread and, thus, allow the implicit atomic update of logically two different pointers—provided that the underlying processor supports the atomic update of a single pointer. Note that this approach may fail, e. g., in case of 8-bit processors on the one side and 16- or 32-bit sized pointer types defined by the programming language or compiler on the other side. Similar holds for 16-bit processors versus 32-bit pointer types. Thus, whether or not this approach succeeds is a question of the semantic gap between the physical and the virtual machine in use for the particular application case. This kind of dependency in most cases cannot (easily be) expressed by a hierarchically structured feature diagram, but rather is a case for additional specifications of constraints and conflicts with respect to a specific feature or group of features: often, they are cases of cross-cutting concerns not only of a feature diagram but also software system. It is up to the feature model to include these supplementing specifications, and not necessarily the feature diagram.

The other three subfeatures of the cumulative feature “preemptive” stand for explicit synchronization at runtime using blocking, non-blocking, or arrestive concepts in order to prevent the occurrence of race conditions. A distinction is made between explicit synchronization of threads (blocking, non-blocking) and events (arrestive). Blocking synchronization typically goes back on lock variables, conditional critical sections, semaphores, and monitors to implement temporary mutual exclusion of otherwise concurrently executing threads [10; 15; 16; 19; 20]. Using priority-oriented scheduling and having semaphores maintain their own waiting lists of blocked threads requires to strictly follow the queuing discipline of the scheduler, otherwise priority violation may be the consequence. That is to say, there is a cross-cutting concern with respect to scheduler and semaphore. Furthermore, with such class of scheduling as foundation, blocking synchronization in general may cause priority inversion [25]. This raises a serious problem for (hard) real-

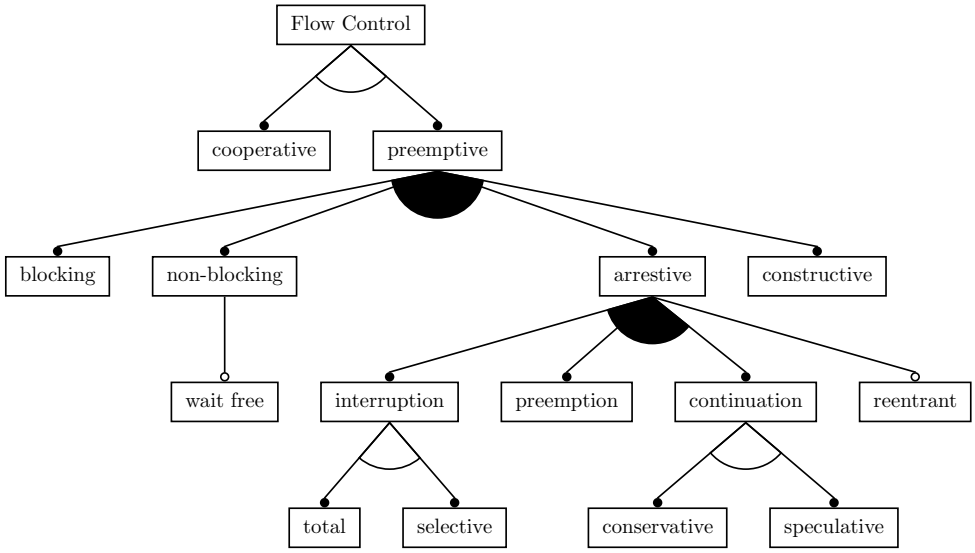


Fig. 10 Feature diagram describing the common and variable properties of functions used to implement flow control in operating systems (excerpt). The features shown likewise map to a specific system behavior with respect to thread processing (cooperative, preemptive), thread synchronization (blocking, non-blocking), and event synchronization (arrestive, constructive).

time systems and, thus, calls for specific solutions such as a kernelized monitor [30] or provisions for priority inheritance/ceiling [40], or a simpler variant like stack-based priority ceiling [2].

As a consequence, subfeature "blocking" implies a couple of constraints on the presence of certain functions in the operating system provided that a specific mode of operation need to be supported. Most notably is that not all of these functions are in a call relation and, therefore, cannot be resolved by conventional linker techniques. Rather, these functions are in a "uses" relation [33]. So whether a kernelized monitor or some sort of priority ceiling protocol shall be exploited must be specified by some meta-level linking information. In the given case, this information comes from other features being mapped to implementations and which need to be selected upon system configuration time. As adumbrated, not only is real-time mode of operation a non-functional property of cross-cutting concern, but also blocking synchronization.

All the problems discussed just now do not exist with non-blocking (and, maybe, wait-free) synchronization [17; 29; 13; 21]. However, solutions in this direction are not always straightforward and often depend on the availability of dedicated machine instructions. The latter case limits portability. In addition, reuse of indivisible resources cannot be easily controlled this way, if at all. Thus, non-blocking synchronization is a preferred option, but not in all situations.

Arrestive synchronization means to temporarily disable events which are the potential reason for thread preemption. This art of synchronization does not block threads, but rather the occurrence of certain events. As indicated by the corresponding cumulative feature shown in figure 10, the events blocked relate either to interruption (by total or selective disabling of interrupts i. e. first-level interrupt handlers), continuation (by disabling second-level interrupt handlers), or preemption (by disabling the dispatcher). The continuation alternatives shown refer to options one has in order to serialize and queue up arriving second-level interrupt handlers while a critical section is active [36]. Temporarily disabling of thread dispatching to take care of non-preemptive critical sections [30] typically works in a similar manner. Note that preemption takes place in form of a continuation of an interrupt handler. Instead of disabling any kind of continuation, arrestive synchronization of preemption means to disable only one kind of continuation. Thus, an optimization takes place here in favour of a higher degree of potential concurrency in the system.

As became clear from the discussion in section 3, in some configurations arrestive synchronization has to work in cascaded fashion. When arrestive synchronization is used, for example, to ensure atomic operation of a semaphore and the scheduler, then synchronization of the latter must be reentrant. This is because of the call relation between semaphore and scheduler, for instance when `P()` calls the scheduler to block the currently executing thread which, in turn, has called that very `P()`. In such a situation, nested non-preemptive critical sections are given. As the called critical section (scheduler) may be called not only from within another critical section (`P()`) but also by normal (non-critical) programs, its synchronization statements must be reentrant. Similar holds for the call relation between scheduler and dispatcher. Reentrant code may have drawbacks with respect to execution speed, why it is reasonable to make it an optional feature in the feature diagram of figure 10. Also note that this feature is a further case of a cross-cutting concern.

The following sections describes how to achieve synchronization of a critical section that stays in at least two different call relations: (1) by some other "surrounding" cri-

tical section and (2) by no critical section at all. Reentrant synchronization statements are needed in the first case, but not in the second case. Of course will reentrant synchronization also be applicable for the second case, but at a higher cost for this particular case—and this will be in contradiction to the program family concept:

> Some users may require only a subset of the services of features that other users need. These "less demanding" users may demand that they are not be forced to pay for the resources consumed by the unneeded feature. [35]

6 Aspect-Oriented Programming

The examination of variability in (embedded) operating systems made in the previous sections showed that not all kinds of potential software changes are supported by going back on conventional concepts such as abstract data types, parameterized data types, or inheritance. These concepts works well if there is (1) a unique hierarchical relation to the assets subject to changes and (2) a rudimentary simple structural complexity defined by that relation. Changes given because of non-functional properties that crosscut the system software are hardly to accomplish and, depending on how extensive these cross-cutting concerns are, may lead to a poorly understandable and manageable software structure in general.

Aspect-oriented programming (AOP [22]) is a technique that attempts to improve separation of concerns. With conventional modularization techniques, achieving separation of concerns if two concerns are "cross-cutting" raises problems and typically leads to the *code tangling* and *code scattering* phenomena. Code tangling means that on the implementation level the code of two (or more) concerns is intermixed rather than separated. Scattering means that the code of one concern is not localized, but can be found in various different modules.

AOP aims at supporting modular high-level concern implementations. For example, the code that implements a specific synchronization policy should be a separate module that represents the human-readable policy description almost directly in a programming language. Hence, the synchronization policy could be evolved independently from the other modules, which also could be reused in other contexts without or with different synchronization schemes.

Today, most AOP languages use the concepts and terminology that was first introduced by AspectJ [23]. Following we will give a brief overview of the most common AOP language elements in general and the AspectC++ [41] notion in particular, as required for understanding the remaining parts of this paper. Even though the introduction is based on AspectC++, it basically holds for any statically woven AOP language.

6.1 Terminology

The most relevant AOP concepts are *join-point* and *advice*. An *advice* definition describes a transformation to be performed at specific positions either in the static program structure (*static cross-cutting*) or in the runtime control flow (*dynamic cross-cutting*) of a target program. A *join-point* denotes such a specific position in the target program. Advice is given by *aspects* to sets of join-points called *pointcuts*. Pointcuts are defined declaratively in a *join-point description language*. The sentences of the join-point description language are called *pointcut expressions*. An aspect encapsulates a cross-cutting concern and is otherwise very similar to a class. Besides advice definitions, it may contain class-like elements such as methods or state variables.

The following example serves to illustrate typical syntactical elements of an aspect language, which is AspectC++ in the given case:

```
aspect ElementCounter {
    int elements;
    advice call("% Queue::enqueue(...)") : after() {
        elements++
    };
};
```

Aspect `ElementCounter` increments its member variable `elements` *after* each call to `Queue::enqueue()`. In AspectC++, pointcut expressions are built from *match expressions* and *pointcut functions*. Match expressions are already primitive pointcut expressions and yield a set of *name join-points*. Name join-points represent elements of the static program structure such as classes or functions. Technically, match expressions are given as quoted strings that are evaluated against the identifiers of a C++ program. The expression `"% Queue::enqueue(...)"`, for instance, returns a name pointcut containing every (member-) function of the class `Queue` that is called `enqueue`. In the case of overloaded functions with different argument types the expression would match all of them. *Code join-points* on the other hand, represent events in the dynamic control flow of a program, such as the execution of a function. Code pointcuts are retrieved by feeding name pointcuts into certain pointcut functions such as `call()` or `execution()`. The pointcut expression `call("% Queue::enqueue(...)")`, for instance, yields all the events in the dynamic control flow where a function named `Queue::enqueue` is about to be called.

As pointcuts are described declaratively, the target code itself has not to be prepared or instrumented to be affected by aspects. Furthermore, the same aspect can affect various and even unforeseen parts of the target code. These principles of *obliviousness* and *quantification* are considered a major advantage of AOP [11].

6.2 Static Cross-cutting

An aspect that encapsulates *static cross-cutting* alters the static structure of the program. In most AOP languages, such

modifications of the static structure are restricted to the extension of classes by new elements like methods, state variables or base classes.

In AspectC++, the encapsulation of static cross-cutting is supported by a specific type of advice called *introduction*. Consider the following aspect, which adds support for thread local storage to a class modelling a thread descriptor:

```
aspect ThreadLocalStorage {
    advice "os::ToC" : slice class {
        int tlsentry;
    public:
        int getTLS() { return tlsentry; }
        void setTLS(int v) { tlsentry = v; }
    };
    ...
};
```

The aspect *introduces* a (private) state variable and some (public) access methods into the thread descriptor class, or, more precisely, into all classes that are matched by the expression `"os::ToC"`.

6.3 Dynamic Cross-cutting

An aspect that encapsulates *dynamic cross-cutting* intercepts certain events in the control flow of a running program. Aspects basically provide means to execute some advice code *before*, *after*, or instead of (*around*) the current statement if the event occurs. In the following, this is demonstrated by three different variants of an aspect which intercepts entries into and exits from the kernel, thus supports implementation of a kernelized monitor. The advice body is identical for all three variants of the `KernelLock_x` aspects, $x = \{1,2,3\}$: it acquires the lock (which is a member of the aspect), proceeds to the intercepted function (**`tjp`**`->proceed()`) and finally releases the kernel lock. Variant 1 is made of an aspect in which the advice is triggered whenever any function or method from the class or namespace `kernel` is about to be *executed*:

```
aspect KernelLock_1 {
    pointcut kernel() = "% kernel::%(...)";
    os::Lock lock;      // aspect member variable

    advice execution(kernel()) : around() {
        lock.enter();
        tjp->proceed(); // execute intercepted method
        lock.leave();
    }
};
```

This, however, works only if kernel functions do not invoke each other, as calls to `lock.enter()`/`lock.leave()` must not be nested. Variant 2 provides a less restrictive solution by intercepting the kernel invocation on the *caller* side:

```
aspect KernelLock_2 {
    ...
    advice call(kernel())
    && !within(kernel()) : around() {
        ...
    }
};
```

The ***`call`***`()` pointcut function yields all events in the control flow, where a given function is about to be *called*. The ***`within`***`()` pointcut function simply returns all join-points in the given classes, functions or namespaces. By *intersecting* (`&&`) all calls to `kernel()` with the negation (`!`) of all join-points inside `kernel()`, the pointcut expression finally evaluates to those calls to a `kernel()` function that are not made from a `kernel()` function itself. This, however, has another potential drawback: as the interception now takes place on the caller side, not only the operating system but also application code has to be woven with the aspect. In many cases, this is not feasible. In variant 3 kernel invocation is therefore again intercepted on the callee side, but further filtered to certain control flows:

```
aspect KernelLock_3 {
    ...
    advice execution(kernel())
    && !cflow(within(kernel())) : around() {
        ...
    }
};
```

The ***`cflow`***`()` pointcut function yields all code join-points that occur while being in a given control flow. Pointcut function ***`execution`***`()` yields all code join-points, where a given function is about to be *executed*. The above pointcut expression therefore evaluates to any non-nested execution of a `kernel()` function. Compared to variant 2, this solution does not require to weave the application code and furthermore reliably detects indirectly nested kernel calls.

6.4 Join-Point Context

In many cases, advice for dynamic cross-cutting needs to read and/or modify the join-point-specific invocation context such as the actual argument values passed to the intercepted function. To fulfil the goal of quantification, join-point specific context information has to be provided through a generic interface, as the same advice implementation should be applicable to many different join-points, such as functions with different signatures. Most AOP languages provide a *join-point API* for this purpose. In AspectC++, the join-point API is implicitly available in advice bodies through the **`JoinPoint*`** **`tjp`** type and instance pointer:

```
aspect Tracing {
    ...
    advice execution("% ...::%(...)"
    && !"void ...::%(...)") : after() {
        JoinPoint::Result res = *tjp->result();
        cout << "leaving " << tjp->signature()
             << " returning" << res;
    }
};
```

The after-advice implementation of the above `Tracing` aspect is generic. It can be applied to any function with a non-void return type, as the join-point API provides the required abstractions from the actual return type.

6.5 Weaving

Aspect weaving is the term used to describe the process of transforming the structure or behaviour of a program in or-

der to let aspects "affect" other modules. The AspectC++ compiler weaves by transforming AspectC++ code into ordinary C++ code [39]. It is a preprocessor that mainly generates transparent wrapper functions. This kind of weaving is called "static weaving" as it is performed at compile-time. "Dynamic weaving" is a different weaving approach that supports to weave aspect code into a running program. In this paper we focus on static weaving only. Note that a static aspect weaver can support aspects that affect static as well as dynamic join points.

7 Conclusion

The paper discussed dimensions of variability in embedded operating systems. Although the focus was on the domain of embedded systems, all the issues considered are also a case for operating systems targeting other domains. However, the embedded systems domain raises very specific demands (not only) on the system software which do not always play the same decisive role in other domains.

Operating Systems need Software Engineering The attribute "embedded" implies tight integration of an operating system with its environment. This requires a software structure that aids integration, thus allows for adaptation of system software to varying demands of the application domain. Family-based software design, feature modelling, and aspect-oriented programming are sound software-engineering approaches that support the development of system software being application-aware, tailor-made, and composed from yet highly reusable assets. The paper discussed how these approaches can be used to develop and maintain embedded operating systems for ease of extension, contraction, porting, and specialization.

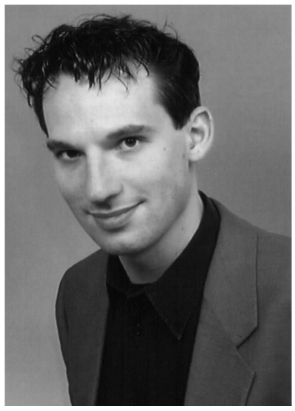
Software Engineering needs Operating Systems At all times have been operating systems challenging case studies for software engineering. In particular, several key concepts of software engineering go back on experiences people made with the design and development of system software. Good examples are modules, layers, the "uses" hierarchy, and program families which all played a central role in the context of the paper presented. Nothing changed today—far from it! The large diversity of operating systems especially in the embedded systems domain on the one hand and their high complexity (not necessarily in terms of number of code lines), when compared to other sorts of software, still raises big challenges to software engineering methods and tools. One of the key issues in this regard is variability management. Embedded operating systems are distinguished candidates for pushing research just as tool development into this direction forward.

References

1. Accetta M, Baron RV, Golub DB, Rashid RF, Tevanian, Jr A, Young MW (1986) MACH: A new kernel foundation for UNIX development. Tech. rep., Carnegie Mellon University, Computer Science Dept., Pittsburgh, PA, USA
2. Baker TP (1990) A stack-based resource allocation policy for realtime processes. In: Proceedings of the 11th IEEE Real-Time Systems Symposium (RTSS '90), IEEE, Lake Buena Vista, FL, USA, pp 191–200
3. Baker TP (1991) Stack-based scheduling of realtime processes. Real-Time Systems 3(1):67–99
4. Beuche D, Guerrouat A, Papajewski H, Schröder-Preikschat W, Spinczyk O, Spinczyk U (1999) 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), IEEE Computer Society Press, St Malo, France, pp 45–53
5. Beuche D, Papajewski H, Schröder-Preikschat W (2004) Variability management with feature models. Science of Computer Programming 53(3):333–352
6. Conway ME (1963) Design of a separable transition-diagram compiler. Communications of the ACM 6(7):396–408
7. Conway ME (1963) A multiprocessing system design. In: Proceedings of the AFIPS Fall Joint Computer Conference (FJCC), Spartan Books, Las Vegas, NV, USA, pp 139–146
8. Corbató FJ, Vyssotsky VA (1965) Introduction and overview of the Multics system. In: Proceedings of the AFIPS Fall Joint Computer Conference (FJCC), Spartan Books, Las Vegas, NV, USA, pp 185–196
9. Czarnecki K, Eisenecker UW (2000) Generative Programming—Methods, Tools, and Applications. Addison-Wesley
10. Dijkstra EW (1965) Cooperating sequential processes. Tech. rep., Technische Universiteit Eindhoven, Eindhoven, The Netherlands, (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996)
11. Elrad T, Aksit M, Kiczales G, Lieberherr K, Ossher H (2001) Discussing aspects of AOP. Communications of the ACM 44(10):33–38
12. Goullon H, Isle R, Löhr KP (1978) Danymic restructing in an experimental operating system. In: The 3rd International Conference on Software Engineering, IEEE Computer Society Press, Atlanta, GA, USA, pp 295–304
13. Greenwald MB, Cheriton DR (1996) The synergy between non-blocking synchronization and operating system structure. In: Proceedings of the 2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI), ACM/USENIX Association, Seattle, WA, USA, pp 123–136
14. Habermann AN, Flon L, Cooprider LW (1976) Modularization and hierarchy in a family of operating systems. Communications of the ACM 19(5):266–272
15. Hansen PB (1972) Structured multprogramming. Communications of the ACM 15(7):574–578
16. Hansen PB (1973) Operating System Principles. Prentice Hall International
17. Herlihy MP (1991) Wait-free synchronization. ACM Transactions on Programming Languages and Systems 13(1):123–149
18. Hildebrand D (1992) An architectural overview of QNX. In: Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures, USENIX Association, Seattle, WA, USA, pp 113–126
19. Hoare CAR (1971) Towards a theory of parallel programming. In: Hoare CAR, Perrot RH (eds) Operating System Techniques, Academic Press, London, New York
20. Hoare CAR (1974) Monitors: An operating system structuring concept. Communications of the ACM 17(10):549–557
21. Hohmuth M, Härtig H (2001) Pragmatic nonblocking synchronization for real-time systems. In: Proceedings of the USENIX Annual Technical Conference, USENIX Association, Boston, MA, USA, pp 217–230

22. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier JM, Irwin J (1997) Aspect-oriented programming. Tech. Rep. SPL97-008 P9710042, Xerox PARC
23. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG (2001) An overview of AspectJ. In: Proceedings of the 15th European Conference an Object-Oriented Programming (ECOOP 2001), Springer-Verlag, LNCS 2072, Budapest, Hungary, pp 327–353
24. Koetz J (2006) Personal communication. Audi AG
25. Lampson BW, Redell DD (1980) Experiences with processes and monitors in mesa. Communications of the ACM 23(2):105–117
26. Lister AM, Eager RD (1993) Fundamentals of Operating Systems, 5th edn. The Macmillan Press Ltd.
27. Lohmann D, Streicher J, Hofer W, Spinczyk O, Schröder-Preikschat W (2007) Configurable memory protection by aspects. In: Proceedings of the 4th Workshop on Programming Languages and Operating Systems (PLOS '07), ACM Press (Digital Library), New York, NY, USA, (to appear)
28. Lohmann D, Streicher J, Spinczyk O, Schröder-Preikschat W (2007) Interrupt synchronization in the CiAO operating system—experiences from implementing low-level system policies by AOP. In: Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patters for Infrastructure Software (AOSD-ACP4IS '07), ACM Press (Digital Library), New York, NY, USA
29. Massalin H, Pu C (1991) A lock-free multiprocessor OS kernel. Tech. Rep. CUCS–005–91, Columbia University
30. Mok AKL (1983) Fundamental design problems of distributed systems for hard real-time environments. PhD thesis, Massachusetts Institute of Technology, MIT, Cambridge, MA, USA, technical Report MIT/LCS/TR-297
31. Multicians (2007) http://www.multicians.org
32. Parnas DL (1974) On a 'buzzword': Hierarchical structure. In: IFIP Congress '74, Information Processing '74, North-Holland Publishing Company, Stockholm, Sweden, pp 336–339
33. Parnas DL (1975) Some hypotheses about the "Uses" hierarchy for operating systems. Tech. Rep. BS I 75/2, TH Darmstadt
34. Parnas DL (1976) On the design and development of program families. IEEE Transactions on Software Engineering SE-5(2):1–9
35. Parnas DL (1979) Designing software for ease of extension and contraction. IEEE Transactions on Software Engineering SE-5(2):128–138
36. Schön F, Schröder-Preikschat W, Spinczyk O, Spinczyk U (2000) 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), IEEE Computer Society, Newport Beach, CA, USA, pp 270–277
37. Schröder W (1987) Eine Familie UNIX-ähnlicher Betriebssysteme — Anwendung von Prozessen und des Nachrichtenübermittlungskonzeptes beim strukturierten Betriebssystementwurf. PhD thesis, Technische Universität Berlin
38. Schröder-Preikschat W (1994) The Logical Design of Parallel Operating Systems. Prentice Hall International
39. Schröder-Preikschat W, Lohmann D, Gilani W, Scheler F, Spinczyk O (2006) Static and dynamic weaving in system software with AspectC++. In: Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS-39), IEEE Computer Society Press, Kauai, HI, USA, vol 9, pp 214–223
40. Sha L, Rajkumar R, Lehoczky JP (1990) Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers 39(9):1175–1185
41. 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, DOI http://dx.doi.org/10.1016/j.knosys.2007.05.004
42. Stümpfle M (2003) Personal communication. DaimlerChrysler AG
43. Thompson K, Ritchie DM (1974) The UNIX timesharing system. Communications of the ACM 17(7):365–375
44. Weiss DM, Lai CTR (1999) Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley
45. Wilner D (1997) Vx-files: What really happened on Mars? Keynote at the 18th IEEE Real-Time Systems Symposium (RTSS '97)

Wolfgang Schröder-Preikschat studied computer science at Technical University of Berlin, Germany, where he also received his Ph.D. (1987) and university lecture qualification (1994). After spending about ten years as a research associate and director of the system software department at the German National Research Center of Computer Science (GMD), Research Institute for Computer Architecture and Software Technique (FIRST), Berlin, Dr. Schröder-Preikschat became a full professor for computer science at the Universities of Potsdam (1995–1997), Magdeburg (1997–2002), and Erlangen-Nuremberg (since 2002), Germany. His main research interests are (real-time) embedded systems, distributed/parallel operating systems, structured computer organization, and software engineering. Dr. Schröder-Preikschat is member of ACM, EuroSys, GI, and IEEE.



Daniel Lohmann studied computer science at University of Koblenz, Germany. He currently holds the position as research associate at University of Erlangen-Nuremberg, Germany, and works towards his Ph.D. in the field of aspect-oriented (embedded) operating systems. His main interests are operating systems, (deeply) embedded systems, software product lines, aspect-oriented software development, and generative programming. Mr. Lohmann is member of ACM, EuroSys, and GI.

Fabian Scheler studied computer science at University of Erlangen-Nuremberg, where he currently holds the position as research associate and works towards his Ph.D. in the field of real-time (embedded) systems. His main interests are real-time systems and operating systems. Mr. Scheler is member of EuroSys and IEEE.

Olaf Spinczyk studied computer science at Technical University of Berlin and received his Ph.D. from University of Magdeburg, Germany (2003). After having held the position of an assistant professor at University of Erlangen-Nuremberg (2003-2007), he recently moved on to University of Dortmund, Germany, to fill the position of a full professor in the area of embedded systems software. His main interests are efficient tailor-made operating systems, embedded systems, software-product lines, and aspect-oriented programming. Dr. Spinczyk is member of ACM, GI, and EuroSys.