On the Design and Development of a Customizable Embedded Operating System[*]

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

Abstract

The design and development of operating systems has to reflect numerous constraints predefined by an application domain. This domain consists, among others, of application software at the top and the computer hardware at the bottom, thus with the operating system in between "a rock and a hard place". There are many application domains with no single operating-system solution for all or even a subset of them. 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 limit component reusability and impair software maintenance. The paper deals with this issue in the scope of operating systems for the embedded-systems domain.

1. Introduction

Operating-system development for deeply embedded parallel/distributed systems sometimes may become a fairly challenging undertaking. Here, the phrase "deeply embedded" refers to systems forced to operate under extreme resource constraints in terms of memory, CPU, and power consumption. The notion "parallel/distributed" relates to the fact that complex embedded systems are becoming more and more networked systems.

Typical cases where appliances are to be controlled by systems in the before mentioned sense come from the automotive field. From the perspective of a computer-science engineer, today's cars are distributed systems on wheels. Their operation is made feasible by a fairly large number of networked electronic control units (ECU), or microcontrollers, with each ECU being equipped with a thimble full of memory only. Depending on the requested furnishings, cars with over 100 ECUs are no rarity any longer.

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

The market of such systems is huge and subject to an enormous cost pressure. In year 2000 about eight billions microprocessors have been manufactured [18]. Only about two percent of them went into the PC, laptop, workstation or server market, while 98 percent 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 compromise with respect to functionality and cost. The situation is not that different today, especially for the automotive industry.

Although the cost of some networked ECUs are in many cases far below that of a single "standard PC", their operating-system demands are often much more challenging. When looking into some more detail at the functions a deeply embedded operating system has to provide, one will identify many commonalities with contemporary, e.g. UNIX-like, operating systems. In many cases some sort of process model is to be supported, interrupt handling and device-driver services are required, synchronization becomes a demanding issue, memory and resource management needs to be provided, and network communication can not be sacrificed [13]. However, the given resource limits prevent the use even of compact micro-kernels such as QNX and L4 [10].

One often hears arguments saying that systems like these have been developed for other (namely more general) purposes, thus bringing them into discussion here would mean comparing apples with oranges. There is a word of truth in it, but the salient point regards the (internal) design decisions of these systems that limit to some extent the applicability to a broader range. Some decisions have been met too early during the system design phase. A typical example is an assumption that a context switch always means exchanging the contents of CPU registers and the address-space mapping, because a process always has to execute within its own address space. In this case, a specific architecture or outward manifestation of the operating system draws throughout many components. This limits component reusability and/or lets streamlining become somewhat difficult if not impossible.

Which operating-system architecture is the best, e.g. monolithic or based on a micro-kernel, promptly becomes a question of philosophy. To express that micro-kernels could be even more compact than they appear, nano-kernels have been introduced. The limitations of nano-kernels, in turn, motivated the invention of pico-kernels. Since even pico-kernels can not be the end of the flagpole, should one therefore better try with femto-kernels? And what comes next?

The dispute on this is not new—and questionable, since all these operating-system architectures are compatible to each other [9]. The choice of architecture should better be a question of the actually to be created system configuration and depends on the application field of the resulting operating system. Architecture is considered an all-embracing *non-functional property* of an operating system. It defines many different aspects under which the system components may have to operate. To make the reusable for various application scenarios, design and implementation of each of the components should be architecture-transparent.

There are a number of non-functional properties hidden inside an operating system. Most problematic are those properties which crosscut the system software. In section 2, we will discuss non-funtional properties other than architecture. Special focus is on embedded systems. Concepts and techniques for the design and implementation of highly reusable operating-system components are presented in section 3. Section 4 in brief shows results from a case study, the PURE family of embedded operating systems. Conclusions are drawn in section 5.

2. Non-Functional Properties

An operating-system architecture is defined by a number of non-functional properties. The most important ones are:

synchronization Certain data structures as well as algorithms are susceptible to race conditions, but these conditions arise only in case of concurrent/parallel execution. Whether or not corresponding functions contain critical sections depends on the way the functions are being used by higher levels. Preemptive scheduling, e.g., is provoked by some event-driven mechanism that forces the currently executing thread to invoke a rescheduling function. The same function will be invoked when a thread voluntarily decides to relinquish processor control. So the need for synchronization is non-functional with respect to scheduling. Furthermore, this need is also non-functional in regard to any other function being part of the execution path of a thread.

protection Some use cases require access control on resources. Such a feature may be provided in various ways by exploiting concepts such as capability-based addressing, access control lists, ring-protected segmentation, sandboxing in general, or type-safe programming. At a more detailed level of implementation and depending on the actual concept, supplementing checks crosscut the software more or less extensively. A typical example is the verification of the user and/or group identfication of a process at many places inside a UNIX kernel. This kind of verification, as well as the protection concept itself, is non-functional with respect to the thus protected function or resource.

isolation In order to secure functions from unauthorized access or fault propagation, hardware-enforced protection domains may be the proper choice. Other approaches use "soft isolation" by relying, e.g., on compiler-enforced protection domains. The different techniques imply different software constraints. In the case of "hard isolation" on MMU basis, alignment restrictions for code and data may or may not arise, need or need not to be considered during design and implementation. Using compiler-enforced protection, only a specific "safe subset" of the programming-language constructs may be allowed. The way of how software isolation is achieved is non-functional with respect to the function to be isolated—as is isolation at all.

sharing If address space protection is being used to secure program entities from each other, lower-level functions may be detained from direct data access. A typical case is DMA, other cases relate to parameter passing in the course of protection-domain crossing function calls (i.e., conventional system calls). The need for intermediate data buffering may arise only in order to let protection boundaries disappear and to improve reusability of lower-level software functions. An alternative is to let the data in question become instances of abstract data types and to invoke specialized access functions whenever necessary. Both techniques are non-functional with respect to the data processing function.

interaction The way system services are encapsulated has impact on the way how the corresponding system functions can be invoked. Logically, a simple procedure call takes place. From the physical perspective, however, this call may have a different "weigth class" depending on the encapsulation concept. Whether or not, e.g., a local, remote, or ”lightweight remote” procedure call needs to be carried out is non-functional with respect to the called function.

These non-functional properties are highly independent from the actual application domain, they are *domain unspecific*. A similar controverse discussion on architecture could take place in the realm of data base or communiations systems. Depending on the actual concepts used to implement these properties, different operating-system architectures will come into being. For example, both monolithic and micro-kernel based operating systems may rely on the same synchronization and protection concept. However, in contrast to the monolithic system, the micro-kernel based system exploits a different isolation concept by encapsulating operating-system services using MMU controlled address spaces.

There are many other cases of domain unspecific non-functional properties of operating systems. Most crucial are those the implementation of which crosscut large fractions of the system software. When being intermixed with the intrinsic functional implementation, these *crosscutting concerns* impair reusability to a vast extent. They link implementations to applications, although the pure functional code may be highly independent therefrom.

The non-functional properties discussed above are also given with many embedded systems. These systems, however, define a very specific domain of highly specific requirements. In this setting, *domain specific* non-functional properties also mean the following:

energy For mobile or autonomous embedded systems energy is a valuable ressource—but also for servers, mainframes, or supercomputers. Energy consumption is a non-functional property of scheduling functions such as process, memory, or I/O scheduling [19]. As a further example, the placement strategy of memory management may have indirect control over energy consumption of a running process if the hardware characteristic of (main) memory in terms of energy requirement and heat differs with the memory banks or subsystems. In order to provide some kind of energy awareness, energy accounting at various places of the system software becomes necessary. Similar to software instrumentation for monitoring purposes, minimal invasive energy-accounting "hooks" need to be set accordingly. These hooks are non-functional code with respect to the context where they are located.

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

dependability This property refers to "the trustworthiness of a computing system which allows reliance to be justifiably placed on the service it delivers" [6]. It encompasses aspects of *reliability*, *availability*, *safety*, and *security*. Adding redundancy to a system is one measure in order to provide highly available services. As a consequence, the requirement may arise to multiply a single system request and to cope with many replies. This particular feature, e.g., is non-functional with respect to the service, or individual function, whose availability shall be improved. In some degree, the former discussion on protection is also true when security and safety of a system is concerned. Reliability is a further non-functional aspect, e.g. in terms of exception handling applied to certain system functions.

As the discussion showed, the term "non-functional" sometimes implies fairly complex functions that need to be implemented in order to provide and enforce a certain property. Dependability is an example of highly elaborated designs and implementations, while synchronization may result in very simple solutions (e.g., in case of interrupt locks). The problem of non-functional properties is not their functional implementation, but their references spread across the implementation 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.

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

3. Reusability and Operating Systems

In order to develop operating-system software for a broad application spectrum, design decision that restrict applicability must be postponed as far as possible. Perhaps certain decisions will never be made inside the system, but rather considered a case for the application programs to be supported. References to (implementations of) non-functional properties are examples of such design decisions.

A first step towards highly customizable embedded operating systems is a careful *feature-oriented domain anal-*

ysis [7]. A significant part of this is *feature modeling*: “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*.” [4] 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, with the nodes refering to specific feature categories. If a feature represented by a parent node of the feature tree is concerned, depending on the category of the feature(s) represented by its child node(s) different constraints can be specified. Assuming a feature (parent) is going to be selected to take some configuration decision, then with respect to a subfeature (child) and its associated implementation modules or abstractions the meaning of its feature category is as follows:

mandatory the subfeature must be included

optional the subfeature may be included

alternative one feature from a set of subfeatures is included

or any non-empty subset of features from a set of subfeatures is included

This technique allows for a compact and precise specification of interdependencies of functional as well as non-functional properties of fairly complex systems. Basing on a tool which aids the construction process of a feature model and supports the mapping of features to implementations, automated generation of highly specialized operating systems becomes possible [1]. Precondition, however, is a design methodology that leads to fine-grained and customizable system software components.

Most important for the succeeding design therefore is to understand the system software as a *program family* [14] and to follow a classical bottom-up approach in the development process. 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 [5].

Realizing a program family by an object-oriented implementation may result in highly flexible and yet efficient system structures. But this will be true only if both design and implementation follow an incremental approach. Starting point must be a minimal subset of system functions which undergoes a stepwise *functional enrichment* by minimal system extensions (see also figure 1). These enrichments can be turned into efficient programs by means of *implementation inheritance*. Note that this does not necessarily hold with interface inheritance. The point of problem is late binding of those methods which are subjected to subsequent specialization in derived classes. This concept may result in overhead-prone implementations and entail very large memory footprints, especially in the case of deep class hierarchies. The decision for late binding must be postponed as far as possible in the design and implementation of object-oriented program families. As a consequence, functional enrichment for creating new object-oriented abstractions of a program family favors implementation inheritance over interface inheritance. Interface inheritance is the right choice only when the family-based design requires multiple implementations of the same interface to coexist.

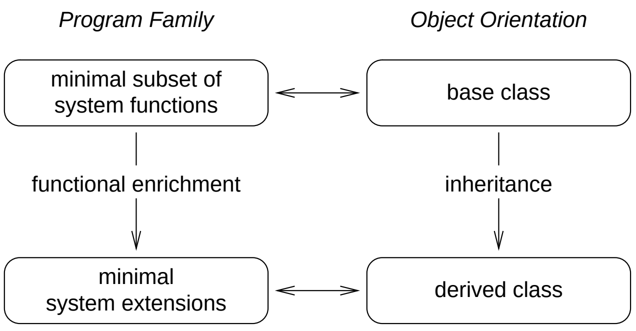

Figure 1. Exploitation of inheritance in the sense of functional enrichment paves the way for an object-oriented program family

Not in every case is it sensible to follow a development process that solely relies on a universal family-based design and object-oriented implementation as described above. Eminent problematic issues are the crosscutting concerns given with many non-functional properties. Trying to reflect these concerns in a hierarchical design may lead to an explosion of the resulting functional and/or class hierarchy. As a rule of thumb: the more crosscutting a specific concern is, the more complex the resulting hierarchical system structure will be. For software maintenance reasons, a crosscutting concern need to be separated from their points of action and implemented as a single module. When a specific family member is going to be instantiated, all missing crosscutting concerns will be applied to the relevant software components. Referring to non-functional properties then may become a configuration matter. Automated (feature-oriented) configuration may take place by having a software transformation tool in charge of interweaving the program module representing a specific crosscutting concern with all the programs that refer to the corresponding non-funtional property.

This kind of final customization of selected software components from a program family can be best achieved using *aspect-oriented programming* (AOP) [8]. In this setting, an aspect program implements a specific crosscutting concern. These programs take care of the manifestation of a particular non-functional property by describing code transformations that need to be applied to selected components. The transformation process is performed by an aspect weaver. This way e.g. stubs can be generated that hide the style of system-service invocation (e.g. local, remote, crossing address-space boundaries, performing mode changes, etc.) from the system components: the stubs encapsulate the non-functional property "interaction". Furthermore, synchronization primitives can be inserted automatically to make e.g. thread-unaware components thread safe [11, 17]. Component instrumentation, e.g. for monitoring purposes, is made feasible as well [12]. Last but not least, to give pattern-based object-oriented designs a final polishing, AOP appears to be a promising technique for streamlining system code [3].

4. A PURE Case Study

The PURE family of embedded operating systems [2] has been developed in the before mentioned sense. Central theme in the development of PURE abstractions is to postpone design and implementation decisions as far as possible. Particular emphasis was the factorization of non-functional properties appearing as crosscutting concerns and their modular implementation. PURE instances are created in an automated way by using a feature-oriented configuration tool (pure::variants [15]) and exploiting AOP on the basis of AspectC++ [16]. At the time being, the PURE family is made of about 350 C++ classes implemented in over 990 compilation units. PURE runs on nine different processor types ranging from 8- to 64-bit technology.

Prior to the automated feature-oriented configuration, PURE variants were created manually by enabling/disabling of about 64 preprocessor switches. This process proved to be highly error-prone for the exponential increasing large number of possible but not necessarily sensible or correct configurations. The result of feature modeling of the PURE family was a feature diagram of about 250 features allowing for about 2^{105} different valid feature combinations. The smallest possible PURE feature set comes up with just three features (CPU, target platform, and compiler), leading to the selection of 20 classes in the configuration process. A feature set for a typical PURE configuration (with preemptive multitasking) has about 20 features. This set describes all the (functional/non-functional) properties of a given member from the PURE nucleus subfamily.

A fundamental building block of PURE is the nucleus, a collection of abstractions together providing a thread execution and cooperation environment. In order to support different applications profiles, the PURE nucleus comes in different variants which likewise implementing different operating modes:

interruptedly Reactive execution of tasks purely on interrupt handling basis.

serialize Interrupt transparently synchronized reactive execution of tasks. Minimal extension to *interruptedly*.

exclusive Single threaded application program controlling the system.

cooperative Cooperative execution of tasks of a multi-threaded application. Minimal extension to *exclusive*.

non-preemptive Cooperative execution of tasks of a multi-threaded event-processing application. Minimal Extension to *cooperative* and *serialize*.

preemptive Event-driven execution of tasks of a multi-threaded application. Minimal extension to *non-preemptive*.

As table 1 shows, the memory footprint of the respective nucleus instances are quite small despite of their extreme modular structure. The numbers refer to the x86 port of PURE.

nucleus instance	size (in bytes)			
	text	*data*	*bss*	total
exclusive	434	0	0	434
interruptedly	812	64	392	1268
cooperative	1620	0	28	1648
non-preemptive	1671	0	28	1699
serialize	1882	8	416	2306
preemptive	3642	8	428	4062

Table 1. PURE memory footprints

PURE demonstrates that highly modular, extensible, portable, and maintainable object-oriented system software (in C++) not at all must be a contradiction in terms. The memory footprint numbers shown in table 1 confirm the scalability of the PURE nucleus, although there is still some potential for optimization. A PURE system provides only the functions as required by the given application, no more and no less.

5. Conclusion

A PURE operating system is meant to be an "open operating system". All its abstractions are revealed to a system designer or even application programmer. The entire system is represented as a library, or a set of libraries, of small and "handy" object modules. These modules are small with respect to the number of exported references to functions or variables. This helps, e.g., state-of-the-art binders creating slim-line operating systems that contain only those components used (i.e. referenced) by a given application. Prerequisite however is a highly modular system structure—and this is achieved by a family-based design and an object-oriented implementation.

Instead of inventing a new system architecture, PURE provides abstractions that allows one to construct many of those architectures. An operating-system architecture is not prescribed by PURE. Rather, a construction set for the development of operating systems is established. Whether an operating system is monolithic or based, e.g., on microkernel technology, is up to the actual "mechanic" who uses PURE elements to create a product according to some blueprint. In order to create a tailor-made operating system, the blueprint comes from the application itself.

In PURE, architecture is considered a non-functional property of an operating system, as is synchronization, protection, isolation, and sharing. In addition to these domain unspecific non-functional properties, PURE also tries to cope with embedded-systems domain specific ones, such as energy, timeliness, and dependability. Feature modeling is used to express the commonalities of and differences amongst the various members of the PURE family. Crosscutting concerns of non-functional properties are dealt with by means of aspect-oriented programming and automated aspect weaving.

References

[1] D. Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2004.

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

[3] D. Beuche, O. Spinczyk, and W. Schröder-Preikschat. Fine-grain Application Specific Customization for Embedded Systems. In *Proceedings of the IFIP WCC 2002 Stream 7 on Distributed and Parallel Embedded Systems (DIPES 2002)*, Montreal, Canada, August 25–30, 2002.

[4] K. Czarnecki and U. W. Eisenecker. *Generative Programming—Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[5] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[6] IFIP. Working Group 10.4 on Dependable Computing and Fault Tolerance. `http://www.dependability.org/wg10.4`, 2003.

[7] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, Nov. 1990.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. Technical Report SPL97-008 P9710042, Xerox PARC, February 1997.

[9] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *ACM Operating Systems Review*, 13(2):3–19, Apr. 1979.

[10] J. Liedtke. On μ-Kernel Construction. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 237–250, Copper Mountain Resort, Colorado, 1995.

[11] D. Lohmann and O. Spinczyk. Architecture-Neutral Operating System Components. *19th ACM Symposium on Operating System Principles (SOSP'03)*, Oct. 2003. WiP session.

[12] D. Mahrenholz, O. Spinczyk, and W. Schröder-Preikschat. Program Instrumentation for Debugging and Monitoring with AspectC++. In *The Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2002)*, Washington DC, USA, April 29–May 1, 2002. IEEE Computer Society.

[13] OSEK/VDX Steering Committe. OSEK/VDX Operating System, Oct. 1997. Version 2.0 revision 1.

[14] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[15] pure-systems GmbH. Variant Management with pure::variants. `http://www.pure-systems.com`, 2004. Technical White Paper.

[16] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In *The 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 18–21, 2002.

[17] O. Spinczyk and D. Lohmann. Using AOP to Develop Architecture-Neutral Operating System Components. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 188–192, Leuven, Belgium, Sept. 2004.

[18] D. Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, May 2000.

[19] A. Weißel and F. Bellosa. Process Cruise Control: Event-Driven Clock Scaling for Dynamic Power Management. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002)*, Grenoble, France, Oct. 2002.