

# On the Development of Object-Oriented Operating Systems for Deeply Embedded Systems—The PURE Project<sup>\*</sup>

Danilo Beuche, Abdelaziz Guerrouat, Holger Papajewski,  
Wolfgang Schröder-Preikschat, Olaf Spinczyk, Ute Spinczyk

University of Magdeburg  
Department of Computer Science  
Universitätsplatz 2  
D-39106 Magdeburg, Germany  
{danilo,guer,papajews,wosch,olaf,ute}@ivs.cs.uni-magdeburg.de

## 1 Introduction

Embedded systems are becoming more and more important — and they are becoming more and more complex. Getting through daily life without being faced with electronically controlled devices is almost unthinkable. This holds not only for the general consumer market regarding cameras, HIFI, kitchen aids, washing machines etc., but also for other markets such as aircraft or automotive industries. Today’s limousines, for example, can be considered (large scale) distributed systems on wheels. There are cars in daily operation consisting of over 60 networked processors (i. e.  $\mu$ -controllers). Although these systems are quite large with respect to the number of  $\mu$ -controllers, they are still small with respect to memory size. In the above mentioned case, 1–2 MB of global memory (for the entire embedded distributed system) is not uncommon — and this would already be more than luxurious. Typically, a single  $\mu$ -controller-based *electronic control unit* (ECU) will be equipped with less than 4 KB of memory.

The complexity of these “decentralized computer architectures” can be managed no longer by the application alone. Dedicated embedded operating systems are required to ensure manageability, adaptability, portability, and yet efficiency of the software. At the same time operating system and application together may not exceed the strongly limited available code size of a few KB.

PURE [6] aims at offering an operating system not only trimmed to a dedicated application area, but to the application itself. In this sense it competes with the special purpose solutions which can hardly be afforded any longer. The idea is to build a highly configurable system and let the application designer choose the necessary functionality. While PURE is not limited to any application area, its main focus lies on deeply embedded systems where a resource-sparing *portable, universal runtime executive* for (real-time) applications is of major concern.

---

<sup>\*</sup> This work has been supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1.

## 2 Design

The approach followed by PURE is to understand an operating system as a *program family* [5] and to use *object orientation* [7] as the fundamental implementation discipline. The former concept (program families) helps prevent the design of a monolithic system organization, while object orientation enables the efficient implementation of a highly modular system structure.

The program family concept does not dictate any particular implementation technique. A so called “minimal subset of system functions” defines a platform of fundamental abstractions serving to implement “minimal system extensions”. These extensions, then, are made on the basis of an *incremental system design* [3], with each new level being a new minimal basis (i. e., *abstract machine*) for additional higher-level system extensions. A true application-oriented system evolves, since extensions are only made on demand, namely, when needed to implement a specific system feature that supports a specific application. Design decisions are postponed as long as possible. In this process, system construction takes place bottom-up but is controlled in a top-down (application-driven) fashion. In its last consequence, applications become the final system extensions. The traditional boundary between application and operating system disappears. The operating system extends into the application, and vice versa.

Because of the strong analogy between the notions of “program family” and “object orientation” it is almost natural to construct program families by using an object-oriented framework [2]. Both approaches are, in a certain sense, dual to each other. The minimal subset of system functions in the program family concept has its counterpart in the superclass of the object-oriented approach. Minimal system extensions are thus introduced by means of subclassing. Inheritance and polymorphism are the proper mechanisms to allow different implementations of the same interface to coexist. Code reuse is significantly enhanced, increasing the commonalities of different family members.

PURE is an *operating-system construction set*. 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 architecture—and this is achieved by a family-based design and an object-oriented implementation.

PURE does not prescribe an operating-system architecture. Rather, a construction set for the development of operating systems is established. Whether an operating system is monolithic or based, e. g., on micro-kernel 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.

### 3 Results

PURE is implemented in C++ and runs (as guest level and in native mode) on i80x86-, i860-, alpha-, sparc-, and ppc60x-based platforms and on C167-based, “CANned”  $\mu$ -controllers. At the time being, the nucleus consists of over 100 classes exporting over 600 methods. Thus, PURE is of a really high modular structure. As Table 1 shows, the high modularity of PURE yet results in a small and compact implementation. The numbers were produced using the C++ compiler `egcs-1.0.2` release for i80x86 processors. Shown is the memory consumption of

**Table 1.** PURE memory consumption.

family member	size (in bytes)			
	text	data	bss	total
<b>prototype</b>				
interruptedly	3256	8	392	3656
reconcile	6406	8	416	6830
exclusive	1289	0	0	1289
cooperative	23988	28	428	24444
non-preemptive	24016	28	428	24472
preemptive	27096	36	428	27560
<b>product</b>				
interruptedly	812	64	392	1268
reconcile	1882	8	416	2306
exclusive	434	0	0	434
cooperative	1620	0	28	1648
non-preemptive	1671	0	28	1699
preemptive	3642	8	428	4062

members of the nucleus family. The members implement different strategies for interrupt propagation (*interruptedly*, *reconcile*) and thread scheduling (*exclusive*, *cooperative*, *non-preemptive*, *preemptive*).

A PURE system appears in two different shapes, depending on the actual development phase. During prototyping, a conventional, procedure-based system representation founds the basis. In this case, every class serves as compilation/binding unit. As a consequence, every generated object module exports a number of references (i.e. symbols). In order to create a PURE product, a macro-based system representation is used. In this case, class methods are the compilation/binding units and every object module exports only a single reference. Furthermore, the C++ feature of inline-functions is exploited extensively. Even deep class hierarchies then result in flat data abstractions and the number of function calls is reduced to a minimum. Compared to a PURE prototype, memory consumption of a PURE product is about 3–7 times smaller, but system generation time can be the 23-fold.

## 4 Problems

The program family concept helps to develop scalable (system) software. Concerning operating systems, an open design can be achieved that lets applications pay only for the resources they really need. The decision on which abstractions are necessary to bring the hardware close to the application, or vice versa, is made statically at the “office table” and not dynamically during runtime.

A disadvantage of the family concept is that it gets increasingly difficult to select the appropriate components as the family grows. To find the best suited operating system for a given application both functional and non-functional requirements have to be considered. The former corresponds to the demanded capabilities (for example whether or not preemption should be provided), while the latter describes issues concerning the implementation.

The approach to cope with the problem of the manifold dependencies between the different family members is to attribute the various PURE abstractions. These attributes describe the capabilities a class/component provides as well as its interface and the requirements it poses on the rest of the system, for example the dependency on other classes. As only some of these attributes may be deduced automatically, others have to be specified manually by the operating system designer [1].

Sometimes a configuration decision has global, or at least subsystem-wide, effects. Those decisions mostly deal with non-functional requirements like the selection of an error handling strategy or whether code size or speed optimization of the system is required. The program code that is influenced by that kind of configuration decision might be the complete system. The problem here is that the implementation of specific aspects of a program does not fit into the concept of functional decomposition. Code that is related to a specific (configuration) aspect is tangled with the “normal” code.

This is exactly the problem that aspect-oriented programming (AOP) [4] is about. Following the principle of separation of concerns, the idea of AOP is to separate the so called component code from the aspect code. The aspect code can consist of several aspect programs, each of which implements a specific aspect in a problem-oriented language. Afterwards, an *aspect weaver* takes the component and the aspect code, interprets both, finds join points and weaves all together to form a single entity.

Aspect-oriented programming delivers a direct relationship between the design and implementation entities, i. e. aspects and aspect programs. For a highly configurable system such as PURE, this feature is extremely useful to let the configuration process be the simple selection of an aspect program. Without AOP a configuration of this kind of aspects would lead to a code polluted with directives for conditional compilation.

## 5 Conclusion

Deeply embedded operating systems are designed to specifically support the execution of (control) applications under extreme resource constraints. In con-

sideration of the specific demands of these applications, it becomes extremely difficult, if not impossible, to successfully adopt existing operating systems.

In addition to providing a highly modular operating system for deeply embedded systems, the PURE project can be seen also as an experiment in the exploitation of object orientation for the implementation of resource-sparing system software. The “old-fashioned” program-family concept has been applied consequently to create featherweight system abstractions. In addition, “standard” development tools have been used for system generation. The results are quite promising—but not yet totally satisfactory.

Future work concentrates on the development of an operating-system workbench. The idea is to offer to system and even application programmers a set of tools that enable the construction, static/dynamic configuration, and (automatic) generation of application-oriented operating systems. An important issue thereby plays aspect-oriented programming. We are currently working on an aspect-weaver environment that mainly consists of a C++ parser, manipulator, and unparser to enable source to source transformations of C++ code. The idea is to have different aspect weaver plug-ins that shall be able to modify the C++ syntax tree driven by their corresponding aspect program. A first prototype for the optimization, i.e. flatening, of C++ implementations of pattern-based object-oriented designs has been developed. An aspect program, interpreted by an aspect-weaver plug-in, transforms reference relationships between objects into inheritance relationships and, thus, not only removes virtual function calls but also definitions of virtual function tables from the original C++ program.

## References

1. D. Beuche. An Approach for Highly Configurable Operating Systems. In J. Bosch and S. Mitchell, editors, *Object-Oriented Technology. ECOOP'97 Workshop Reader*, volume 1357 of *LNCS*, pages 531–536. Springer-Verlag Berlin/Heidelberg, 1998.
2. J. Cordsen and W. Schröder-Preikschat. Object-Oriented Operating System Design and the Revival of Program Families. In *Proceedings of the Second International Workshop on Object Orientation in Operating Systems (I-WOOOS '91)*, pages 24–28, Palo Alto, CA, October 17–18, 1991.
3. A. N. Habermann, L. Flon, and L. Coopriider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.
4. 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.
5. D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.
6. F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. 1998. Accepted for the *International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*.
7. P. Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, 1986.