© Springer-Verlag

Tailor-made Operating Systems for Embedded Parallel Applications*

Antônio Augusto Fröhlich[1] and Wolfgang Schröder-Preikschat[2]

[1] GMD FIRST
Rudower Chaussee 5
D-12489 Berlin, Germany
guto@first.gmd.de
http://www.first.gmd.de/~guto

[2] University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
wosch@cs.uni-magdeburg.de
http://ivs.cs.uni-magdeburg.de/~wosch

Abstract. This paper presents the PURE/EPOS approach to deal with the high complexity of adaptable operating systems and also to diminish the distance between application and operating system. A system designed according to the proposed methodology may be automatically tailored to satisfy an specific application. In order to enable this, the application must be written referring to the *inflated interfaces* that export the system object repository and then be submitted to an analyzer that will proceed syntactical and data flow analysis to extract a blueprint for the operating system to be generated. This blueprint is then refined by dependency analysis against information about the execution scenario acquired from the user via visual tools. The outcome of this process is a configuration file consisting of *selective realize* keys that will support the compilation of the tailored operating system.

1 Introduction

The boom of embedded systems in the recent years projects a near future replete of complex embedded applications, including navigation, computer vision and particularly automotive systems. Some currently available limousines benefit from over 60 networked processors (μ-controllers) and can be considered parallel systems on wheels. Moreover, many of these new embedded applications will demand performance levels that can only be achieved by parallelization and thus new operating systems and tools are to be conceived.

* Work partially supported by Federal University of Santa Catarina, by Fundação Coordenação de Aperfeicoamento de Pessoal de Nível Superior and by Deutsche Forschungsgemeinschaft grant no. SCHR 603/1-1.

Our experiences developing run time support systems for ordinary, i.e., non-embedded, parallel applications [9, 1] convinced us that adjectives such as "all purpose", "global" and "generic" do not fit together with "high performance" and "deeply embedded", whereas different parallel applications have quite different requirements regarding operating systems. Even apparently flexible designs like μ-kernel based operating systems may imply in waste of resources that, otherwise, could be used by applications. The reality for embedded parallel applications, that usually have to operate under extreme resource constrains, not only in terms of processor cycles, but also of memory consumption, can not be different and therefore we should give each application its own operating system.

This paper presents a proposal to automate the process of generating a tailored operating system for a given (embedded) application. This proposal is presented in the realm of Epos, an operating system designed to support (high performance) parallel computing on distributed memory architectures. Epos is actually an extension of Pure [2], that aims to provide a portable, universal runtime executive for resource-sparing (real-time) applications. The following sections describe the Epos/Pure design approach, a case study to demonstrate the necessity for automated tailored operating systems and a new strategy to achieve them.

2 The Design of PURE

Epos basic approach to give each application an operating system that closely fulfils its requirements is to analyze the application looking for hints about the nature of the operating system services requested. After collecting information, a custom Epos will be generated using the building blocks supplied by Pure. In this way, Epos becomes an extension of Pure, to whom it owes most of its design. Thus, describing Epos design implies in describing Pure.

The approach followed by Pure is to understand an operating system as a *program family* [8] and to use *object orientation* [10] as the fundamental implementation discipline. The former concept, program families, helps to prevent the design of a monolithic system organization, while object orientation enables the efficient implementation of a highly modular system architecture.

2.1 Incremental System Design

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 are, then, made on the basis of an *incremental system design* [5], with each new level being a new minimal basis, i.e., an *abstract machine*, for additional higher-level system extensions. A true application-oriented system evolves, since extensions are only made on demand, when the implementation of a new system feature that supports a specific application is required. 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 and the traditional boundary between application and operating system disappears. In other words, the operating system extends into the application and vice versa.

Inheritance is the appropriate technique to either introduce new system extensions or replace existing ones by alternate implementations. Either case, the system extensions are customized with respect to specific user demands and will be present at runtime only in coexistence with the corresponding application. Thus, applications are not forced to pay for (operating system) resources that will never be used.

2.2 Revival of Program Families

Applying the family concept in the software design process leads to a highly modular structure. New system features are added to a given subset of system functions. 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 [6]. Both approaches are, in a certain sense, reciprocal 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. In the realm of embedded distributed/parallel systems, inheritance must be applicable even in the case of crossing address space, node, and network boundaries [7]. With this design strategy, reusability is significantly enhanced, increasing the commonalties of different family members.

3 The Design of EPOS

Although a tailored Epos will be comprised of Pure building blocks[1], Epos extends Pure, towards applications, in order to cope with two growing problems: the system configuration complexity and the distance between application and system software.

3.1 System Object Adapters

The growth in configuration complexity arises from the fact that, as an adaptable operating system, Pure is designed to yield a large number of building blocks, or system objects, which are, in turn, to be put together to compose the tailor-made operating system. To achieve high performance, these system objects must be fine tuned to each of the execution scenarios aimed and therefore a reasonable

[1] New building blocks to support parallel computing are being conceived according to the Pure design approach.

building block repository will comprise a very large number of elements. This turns system configuration into a nightmare. EPOS approach to the matter is to make system objects adaptable to execution scenarios.

By carefully analyzing the system objects repository, one will promptly realize that those abstractions designed to present the same functionality in different execution scenarios are indeed quite similar. Besides, abstractions conceived to support the same scenario often differ from each other following a pattern. In this way, we propose system abstractions to be implemented as independent from the execution scenario as possible. They would then be put together with the aid of some sort of "glue" specific to each scenario. We named these "glue" *scenario adapters*, since they will adapt an existing system abstraction to a certain execution scenario.

In order to exemplify the use of scenario adapters and also to demonstrate the basis for our assertion that system abstractions implementation follow a pattern along scenarios, we could consider the "semaphore" abstraction for three multi-threaded scenarios: single-task, multi-task-single-processor and multi-task-multi-processor (SMP). For all three execution scenarios, a semaphore abstraction (i.e., the semaphore system object) will involve a counter and a thread queue. Aspects such as crossing the application/operating system boundary in the second and third scenarios or such as synchronizing concurrent invocation of semaphore methods in the third scenario are not intrinsic to the semaphore abstraction, but to the execution scenario. Therefore, such kind of aspects would be better implemented as scenario adapters than as different types of semaphore. Similarly, a "thread" abstraction conceived for the same three scenarios, would show a repetition of code when compared to the semaphore abstraction: crossing system boundary and supporting concurrent execution are intrinsic to the scenarios and not to the abstraction. It is also worth to say that scenario adapters are implemented in a way not to insert overhead on the path from applications to system objects, i. e., when an existing system object fulfils the requirements of a scenario, the scenario adapter simply vanishes.

The approach of writing pieces of software that are independent from certain aspects and later adapting them to a given scenario has been referred to as "Aspect Oriented Programming" [4]. However, there are significant differences in our proposal to make system abstractions adaptable. We are not proposing a language to describe aspects neither tools (*weavers*) to automatically combine aspects and aspect-independent components. The problem we want to tackle is the complexity inherent to adaptable operating systems and we are doing this by reducing the number of system abstractions in detriment to the growth of the number of scenario adapters. We believe this will reduce system complexity because dealing with aspects isolated in scenario adapters is much simpler than dealing with them spread along the system abstraction implementation; and also because new scenario adapters can be derived, automatically or not, from other existing adapters.

3.2 Inflated Class Interfaces

The second goal on Epos extension of Pure is to diminish the gap between operating system and applications. This gap originates mainly from the fact that the operating system building blocks offered to applications are usually conceived considering the optimization of resource utilization in a bottom-up fashion and not the applications expectances. Epos adopt *abstract data type declarations* in the form of *inflated class interfaces* as a mechanism to advertise the system abstractions repository to applications and to automate the selection of the proper building blocks when tailoring an operating system.

These inflated class interfaces shall enable the application programmer to express his expectations regarding the operating system simply by writing down well-known system object invocations (system calls in non object oriented systems) while coding the application. By well-know system object invocations we mean that the operating system services will be made available to applications via abstractions commonly accepted by the parallel systems community, such as "threads", "tasks", "address spaces", "channels", "ports", etc. These interfaces are to be defined according to the fundamental law of object orientation that says [3]: look at the real world while looking for objects. The question of where in the real world one can find a "thread" can be easily answered when we realize that threads, just like numbers, are human conventions and that a couple of classical computer science books should comprise most widely accepted conventions. Nevertheless, our users, i.e., embedded parallel application programmers, are welcome to suggest modification or extensions for these interfaces at any time.

With these interfaces in hand, it should no longer be a problem for a skilled parallel application programmer to select how he wants to express process communication, thread synchronization or any other operating system service. It is important to notice that these inflated interfaces will never be implemented as a single class, but as a (possibly huge) set of classes specific to certain scenarios. They are only a mechanism to help programmers to design and implement their applications. An automatic tool shall bind (reduce) the inflated interfaces to specific implementations.

To support system design based on inflated interfaces, we propose two new object-oriented design notations: *partial realize* and *selective realize*. Both relationships take place between an inflated interface and a class that realizes this interface. However, as the name suggest, a class participating in a partial realization implements only a specific subset of the corresponding inflated interface. In this scope, selective realization means that only one of several possible realizations will be connected to the inflated interface at a time. To support selective realization, each class joining the relation is tagged with a key. By changing the value of this key one can select a specific, usually partial, realization for an interface. These two design elements are depicted in figure 1.

As important as design elements, partial and selective realization have their counterparts for system implementation so that tailoring an operating system can be done simply by defining values for selective realize keys. These keys are

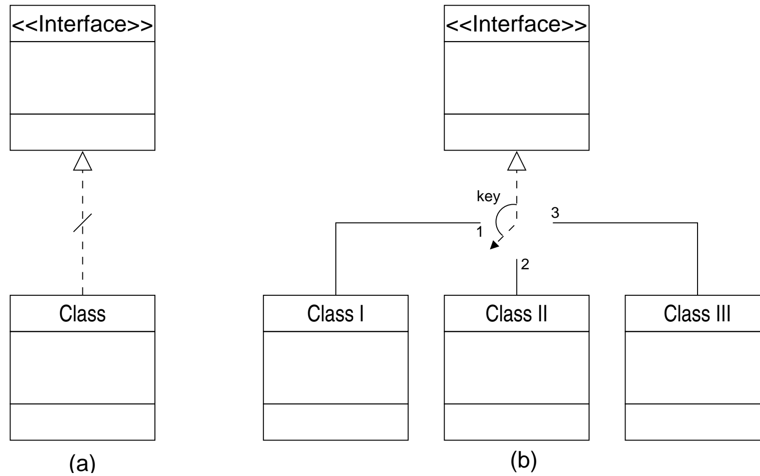

Fig. 1. Partial realize (a) and selective realize (b) relationships.

defined in a single configuration file and dispense the use of both, conditional compilation and "makefile" customization. Furthermore, the implementation of these relationships may be used to bind non object oriented inflated interfaces to object oriented implementations. This is useful, for instance, to bind an application written in Fortran or C to Epos.

4 Automatically Tailoring an Operating System

With the design and implementation mechanism described so far, we can now consider the automatic generation of an operating system to closely fulfil an embedded parallel application. Our strategy begins top-down at the application, with the programmer specifying the application requirements regarding the operating system by designing/coding the application while referring to the set of inflated interfaces that export the system abstractions repository. An application designed and implemented in this fashion can now be submitted to an analyzer (figure 2) that will conduct syntactical and data flow investigations to determine which system abstractions are really necessary to support the application and how they are invoked. This tool shall generate an operating system blueprint that will, for instance, define the use of multi-tasking instead of single-tasking, of multi-threading instead of single-threading, of insolated protected address spaces instead of a single unprotected address space and so on.

Our primary operating system blueprint is, unfortunately, not complete. We got good hints about how the ideal operating system for a given application should look like, but there are aspects that can not be deduced by analyzing the application. As an example, we could consider the decision of generating an operating system that supports multiple processes with protected address spaces

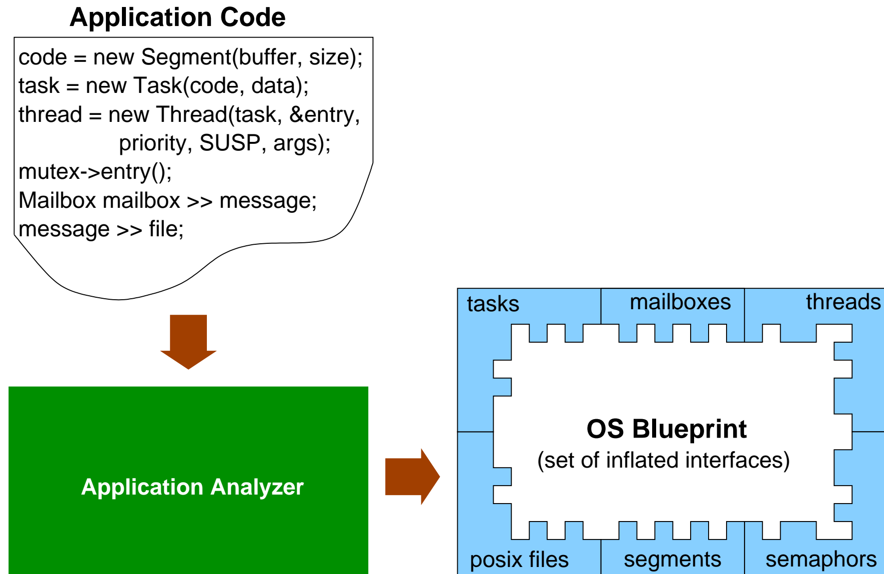

Fig. 2. Extracting an operating system blueprint from the application.

based on a micro-kernel or an operating system for a single, possibly multi-threaded, process to be embedded into (linked to) the application. The fact that the application does not show any evidence that multiple processes may need to run concurrently in a single processor, does not necessarily mean that this situation will not happen. The multi-task support may be required because the application may need to span more processes than the available number of processors, and this will not be perceivable until the user tell us something about the intended execution scenario. Other factors such as target architecture, number of processors available, network architecture and topology are fundamental to tailor a good operating system, but are usually not expressed inside the application. Therefore, we still need user intervention to describe the application's execution scenario, however, the description of the available resources will be due to the operating system developers and the interaction with the user will be done via visual tools.

Refining the blueprint obtained when analyzing the application with the context information acquired via this visual tool will render a much more precise description of how the ideal operating system for a given application should look like. This refined blueprint is the result of a dependency analysis and is expressed via a configuration file consisting of selective realization key values. With the definition of these keys, the inflated interfaces referred by the application programmer are bound to scenario specific implementations. For example, the inflated thread interface from the first step may have included remote invocation and migration, but reached the final step as a simple single-task, priority scheduled thread for a certain μ-controller. A representation of an application

tailored operating system generated according to this model is depicted in the figure 3.

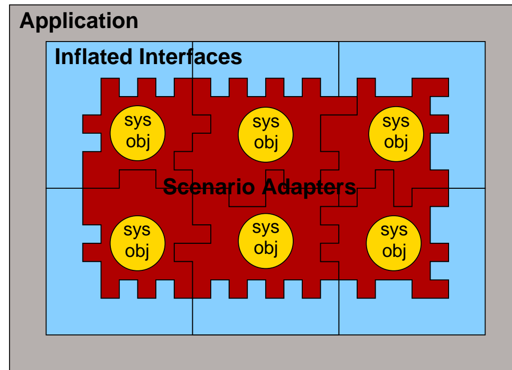

Fig. 3. An operating system tailored to an application.

It is important to understand that, at the early stages of the operating system development, very often a required building block will not yet be available. Even then, the proposed strategy is of great value, since the operating system developers get a precise description for the missing building blocks. In many cases, a missing building block will be quickly (automatically) adapted from another scenario using the scenario adapters described earlier.

Only if the operating system developers are not able to deliver the requested building blocks in a time considered acceptable by the user, either because a building block with that functionality have not yet been implemented for any scenario or because the requested scenario is radically different from the currently supported scenarios, we will shock the user asking him to select the best option from the available set of system abstractions (scenario adapters) and to adapt his program. In this way, our strategy ends where most tailorable operating systems start. Moreover, after some development effort, the combination of scenario adapters and system abstractions shall satisfy the big majority of parallel embedded applications.

5 A Case Study on Configuration

A good example of how tailoring an operating system to a specific application may improve performance can be observed when selecting the proper member of PURE nucleus family. PURE is made of a nucleus and nucleus extensions. The nucleus implements a *concurrent runtime executive* (CORE) for passive and active objects. By means of minimal *nucleus extensions* (NEXT) features such as application-oriented process and address space models, blocking (thread) synchronization, and problem-oriented (remote) message passing are added to the

system. These extensions are present only if demanded by the application. They transform the nucleus into a distributed abstract thread processor. The coarse structure of PURE is depicted in Figure 4.

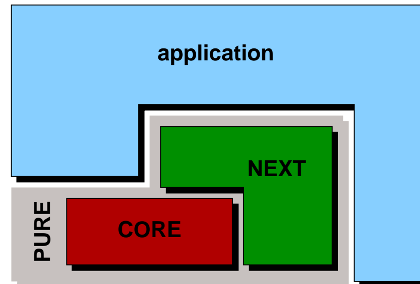

Fig. 4. PURE architecture.

Depending on the actual application requirements, the nucleus components appear in different configurations. Each of these configurations represent a member of the nucleus family. PURE's current implementation includes six family members, each of which implementing a specific operating mode. Each member is built by one or more function blocks and described by a functional hierarchy of these blocks. The function blocks can be reused in various configurations. In order not to restrict the family, design decisions have been postponed as far as possible and encapsulated by higher-level abstractions. As a consequence, PURE can be customized, for example, with respect to the following scenarios:

1. One way of operating the CPU is to let PURE run *interruptedly*. This family member merely supports low-level trap/interrupt handling. The nucleus is free of any thread abstraction. It only provides means for attaching/detaching exception handlers to/from CPU exception vectors (*interruption*).
2. In order to *reconcile* the asynchronously initiated actions of an *interrupt service routine* (ISR) with the synchronous execution of the interrupted program, a minimal extension to 1. was made. The originating family member ensures a synchronous operation of event handlers (*driving*) in an interrupt-transparent manner (*serialization*).
3. The second basic mode of operating the CPU means *exclusive* execution of a single active object. In this situation, the nucleus provides only means for *objectification* of a single thread. The entire system is under application control, whereby the application is assumed to appear as a specialized active object. There is only a single active object run by the system.
4. A minimal extension to 3. leads to *cooperative* thread scheduling. No other design decisions are made except that threads are implemented as active objects and scheduled entirely on behalf of the application (*threading*). There may be many active objects run by the system.

5. Adding support for the serialized execution of thread scheduling functions (*locking*) enables the *non-preemptive* processing of active objects in an interrupt-driven context. Thread scheduling till happens cooperatively, however the nucleus is prepared to schedule threads on behalf of application-level interrupt handlers. Actions of global significance, and enabled by interrupt handlers, are assumed to be synchronized properly (*serialization*).
6. Multiplexing the CPU between threads in an interrupt-driven manner establishes the autonomous, *preemptive* execution of active objects. In this case, the nucleus is extended by a device driver module (*multiplexing*) taking care of timed thread scheduling.

Referring to figure 4, CORE takes care of interruption, serialization, locking, threading and objectification, while NEXT covers (device) driving and (CPU) multiplexing.

5.1 Functionality vs. Complexity vs. Performance

The PURE system is implemented in C++ and runs (as guest level and in native mode) on i80x86-, i860-, sparc-, and ppc60x-based platforms. A port to C 167-based, “CANned” μ-controllers is in progress. At the time being, the nucleus consists of over 100 classes exporting over 600 methods. Every class implements an abstract data type. Inheritance is employed extensively to build complex abstract data types. For example, the thread control block is made of about 45 classes arranged in a 14-level class hierarchy.

As table 1 shows, the highly modular nucleus structure still results in a small and compact implementation. The numbers were produced using GNU `g++` 2.7.2.3 for the i586 running Linux Red Hat 5.0.

Table 1. PURE memory consumption.

family member	**size** (in bytes)			
	text	*data*	*bss*	total
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

Table 2 shows the number of (i586) CPU clock cycles spent during thread scheduling, i.e., the overhead imposed on applications at run-time due to thread scheduling. The clock frequency in this experiment was 166 MHz and the measurement was made reading the i586 (on-chip) counter register. Again, the abovementioned C++ compiler was used.

Table 2. PURE scheduler latency.

family member	CPU cicles
interruptedly	no scheduler
reconcile	no scheduler
exclusive	no scheduler
cooperative	49
non-preemptive	57
preemptive	300

At first sight, these numbers may seem insignificant, but when we realize that some deeply embedded applications (e.g. car engines) require the scheduler to be invoked once every 10 μs, assigning the *preemptive* family member to an application that could run with the *exclusive* family member will imply in a slow down of 15%. We still do not have performance measurements for the family members implemented to support parallel computing, but previous experiments with the PEACE family based operating system [9] showed that applications can improve performance in up to 74% just by being assigned to run with the proper operating system (e.g., a single-task, stand-alone environment supported by a library vs. a multi-task, distributed environment supported by a μ-kernel).

All these performance figures demonstrate the still "featherweight" structure of PURE, although quite a large amount of abstractions (classes, modules, functions) are involved in all the occurrences: "*It is the system design which is hierarchical, not its implementation*" [5]. Besides, these figures demonstrate the necessity to give each application its own operating system.

6 Further Work

The strategy to automatically adapt an operating system to a given (embedded) parallel application proposed by PURE and EPOS can drastically improve application performance, since the application will get only the operating system components it really needs. Besides, these components are fine tuned to the execution scenario aimed. However, our strategy is not able to deliver an *optimal* operating system. Consider, for instance, the decision for a thread scheduling policy: several thread implementations, with different scheduling policies, may fit into the blueprint extracted by our tools, as long as they match the selected interfaces (selectively realize the requested interfaces) and satisfy the dependencies. Nevertheless, it is unnecessary to say that there is an optimal scheduling policy for a given set of threads running in a given scenario.

The decision of which variant of a system abstraction to select when several ones accomplish the application's requirements is, in the current system, arbitrary. Further development of our tools shall include *profiling* primitives to collect run-time statistics for the application. These statistics will then drive operating system reconfigurations towards the optimal. To grant that we can

generate an *optimal* system, however, would imply in formal specification and validation of our system objects, what is not in the scope of Epos, but of the Wabe project. Wabe aims to implement a workbench for the construction of optimal operating systems. It is now being developed under a DFG project at the Universities of Magdeburg and Potsdam.

7 Conclusion

In this paper we presented the Pure/Epos approach to deal with the high complexity of adaptable systems, to diminish the distance between application and operating system and to automate the generation of application tailored operating systems. The complexity problem is tackled with the adoption of *scenario adapters*: software structures that support the adaptation of aspect independent system abstractions to specific execution scenarios; and by *inflated class interfaces*: a mechanism to advertise the system abstraction repository with a (very) reduced number of well-known components. Inflated interfaces are also an effective way to diminish the distance between applications and operating systems, since the application programmer is no longer requested to deal with the complex collection of basic system objects directly.

A system designed according to the methodology proposed in this paper can be automatically tailored to satisfy an specific application. In order to enable this, the application must be written referring to the inflated interfaces that export the system abstractions repository and then be submitted to an analyzer. This analyzer will proceed syntactical and data flow analysis to extract a blueprint for the operating system to be generated. The blueprint is then refined by dependency analysis against information about the execution scenario acquired from the user via visual tools. The outcome of this process is a configuration file comprised of *selective realize* keys that will support the compilation of the tailored operating system. These tools are now under development at GDM-FIRST Institute and at the University of Magdeburg.

References

[1] Antônio A. Fröhlich, Rafael Avila, Luciano Piccoli & Helder Savietto. A Concurrent Programming Environment for the i486. In *Proceedings of the 5th International Conference on Information Systems Analysis and Synthesis - InterSymp'96*, Orlando, USA, July, 1996.

[2] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk & Ute Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems - DIPES'98*, Paderborn, Germany, October 1998.

[3] Grady Booch. *Object Oriented Analysis and Design with Applications.* Benjamin/Cummings, Redwood City, CA, USA, 1994.

[4] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier & John Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP'97, Lecture Notes in Computer Science*, pages 220–242, Springer-Verlag, 1997.

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

[6] 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*, pages 24–28, Palo Alto, CA, USA, October 1991.

[7] J. Nolte and W. Schröder-Preikschat. Dual Objects — An Object Model for Distributed System Programming. In *Proceedings of the Eigth ACM SIGOPS European Workshop, Support f or Composing Distributed Applications*, 1998.

[8] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *Transaction on Software Engineering*, SE-5(2), 1979.

[9] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice-Hall, 1994.

[10] P. Wegner. Classification in Object-Oriented Systems. *SIGPLAN Notices*, 21(10):173–182, 1986.