Configuration Management of Embedded Operating Systems using Application-Oriented System Design

Gustavo Fortes Tondello and Antônio Augusto Fröhlich
Laboratory for Software/Hardware Integration (LISHA)
Federal University of Santa Catarina (UFSC)
PO Box 476 - 88049-900 Florianópolis - SC - Brazil
E-mail: {tondello | guto}@lisha.ufsc.br
Homepage: http://epos.lisha.ufsc.br/

Abstract

This paper presents an alternative to achieve automatic run-time system generation based on the Application Oriented Systems Design method. Our approach relies on a static configuration mechanism that allows the generation of optimized versions of the operating system for each of the applications that are going to use it. This strategy is of great value in the domain of high performance computing since it results in performance gains and resource usage optimization.

Keywords: application-oriented system design, embedded operating systems, configuration management.

1. Introduction

Previous studies have demonstrated that embedded and high performance applications do not find adequate run-time support on ordinary all-purpose operating systems, since these systems usually incur in unnecessary overhead that directly impact application's performance [1, 15]. Each class of applications has its own requirements regarding the operating system, and they must be fulfilled accordingly.

The *Application-Oriented System Design* (AOSD) method [6] is targeted at the creation of run-time support systems for dedicated high performance computing applications, in particular embedded, mobile and parallel ones. An *application-oriented operating system* arise from the proper composition of selected software components that are adapted to finely fulfill the requirements of a target application. In this way, we avoid the traditional "got what you didn't ask for, yet didn't get what you needed" effect of generic operating systems. This is particularly critical for high performance embedded applications, for they must often be executed on platforms with severe resource restrictions (e.g. simple microcontrollers, limited amount of memory, etc).

Application-Oriented System Design has been corroborated by several experiments conducted in the scope of project EPOS [7], including a communication system for clusters of workstations interconnected in a MYRINET network that delivered parallel applications unprecedented communication performance—lowest latency for short messages and maximum bandwidth for large ones [9].

Nonetheless, delivering each application a tailored run-time support system, besides requiring a comprehensive set of well-designed software components, also calls for sophisticated tools to select, configure, adapt and compose those components accordingly. That is, *configuration management* becomes a crucial to achieve the announced customizability.

This paper approaches configuration management in application-oriented operating systems, taking the strategies and tools currently deployed in EPOS as a case-study of automatic operating system configuration for embedded and parallel applications. The following sections describe the basics of the Application-Oriented System Design method, a strategy to automatically configure component-based systems and a strategy to describe the components for that purpose. Subsequently, the current prototypes are discussed along with a real example of the configuration process, followed by an outline of the next steps planed for the project along with author's conclusions.

2. Application-Oriented System Design

The idea of building run-time support systems through the aggregation of independent software components is being used, with claimed success, in a series of projects

[3, 5, 14, 2]. However, software component engineering brings about several new issues, for instance: how to partition the problem domain so as to model really reusable software components? how to select the components from the repository that should be included on an application-specific system instance? how to configure each selected component and the system as a whole so as to approach an optimal system?

Application-Oriented System Design proposes some alternatives to proceed the engineering of a domain towards software components. In principle, an application-oriented decomposition of the problem domain can be obtained following the guidelines of *Object-Oriented Decomposition* [4]. However, some subtle yet important differences must be considered. First, object-oriented decomposition gathers objects with similar behavior in class hierarchies by applying variability analysis to identify how one entity specializes the other. Besides leading to the famous "fragile base class" problem [12], this policy assumes that specializations of an abstraction (i.e. *subclasses*) are only deployed in presence of their more generic versions (i.e. *superclasses*).

Applying variability analysis in the sense of *Family-Based Design* [13] to produce independently deployable abstractions, modeled as members of a family, can avoid this restriction and improve on application-orientation. Certainly, some family members will still be modeled as specializations of others, as in *Incremental System Design* [10], but this is no longer an imperative rule. For example, instead of modeling connection-oriented as a specialization of connectionless communication (or vice-versa), what would misuse a network that natively operates in the opposite mode, one could model both as autonomous members of a family.

A second important difference between application-oriented and object-oriented decomposition concerns environmental dependencies. Variability analysis, as carried out in object-oriented decomposition, does not emphasizes the differentiation of variations that belong to the essence of an abstraction from those that emanate from the execution scenarios being considered for it. Abstractions that incorporate environmental dependencies have a smaller chance of being reused in new scenarios, and, given that an application-oriented operating system will be confronted with a new scenario virtually every time a new application is defined, allowing such dependencies could severely hamper the system.

Nevertheless, one can reduce such dependencies by applying the key concept of *Aspect-Oriented Programming* [11], i.e. aspect separation, to the decomposition process. By doing so, one can tell variations that will shape new family members from those that will yield scenario aspects. For example, instead of modeling a new member for a family of communication mechanisms that is able to operate in the presence of multiple threads, one could model multithreading as a scenario aspect that, when activated, would lock the communication mechanism (or some of its operations) in a critical section.

Based on these premises, Application-Oriented Systems Design guides a domain engineering procedure (see Figure 1) that models software components with the aid of three major constructs: families of scenario-independent abstractions, scenario adapters and inflated interfaces.

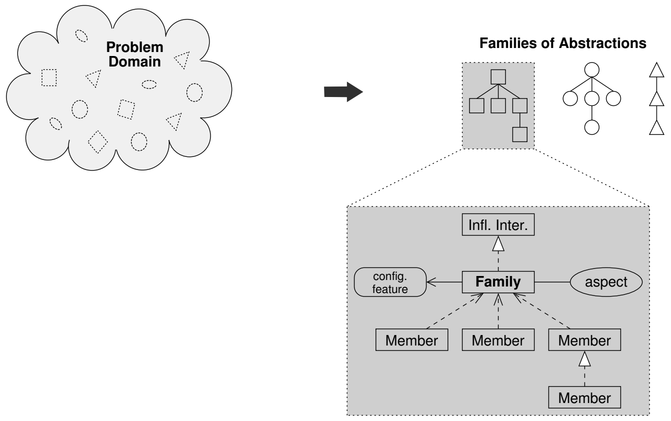

Figure 1. Overview of application-oriented domain decomposition as regards abstractions.

Families of scenario independent abstractions

During domain decomposition, abstractions are identified from domain entities and grouped in families according to their commonalities. Yet during this phase, aspect separation is used to shape scenario-independent abstractions, thus enabling them to be reused in a variety of scenarios. These abstractions are subsequently implemented to give rise to the actual software components.

The implementation of the members of a family of abstractions is not restricted to the use of specialization as we would do in object-orientation, although it can occur, when convenient. For example, members could be implemented as classes conjunctly distributed as a package through aggregation or composition. Afterwards, some families may contain mutually exclusive members, that is, only one of the members can be present in the system configuration at a time.

Scenario adapters

As explained earlier in this article, Application-Oriented System Design dictates that scenario dependencies must be factored out as *aspects*, thus keeping abstractions scenario-independent. However, for this strategy to work, means must be provided to apply factored aspects to abstractions in a transparent way. The traditional approach to do this would be deploying an *aspect weaver*, though the *scenario adapter* construct [8] has the same potentialities without requiring an external tool. A scenario adapter wraps an abstraction, intermediating its communication with scenario-dependent clients to perform the necessary scenario adaptations.

Inflated interfaces

Inflated interfaces summarize the features of all members of a family, creating a unique view of the family as a "super component". It allows application programmers to write their applications based on well-know, comprehensive interfaces, postponing the decision about which member of the family shall be used until enough configuration knowledge is acquired. The binding of an inflated interface to one of the members of a family can thus be made by automatic configuration tools that identify which features of the family were used in order to choose the simplest realization that implements the requested interface subset at compile-time.

3. Software Component Configuration

An operating system designed according to the premises of Application-Oriented System Design, besides all the benefits claimed by software component engineering, has the additional advantage of being suitable for automatic generation. The concept of inflated interface enables an application-oriented operating system to be automatically generated of out of a set of software components, since inflated interfaces serve as a kind of requirement specification for the system that must be generated.

An application written based on inflated interfaces can be submitted to a tool that scans it searching for references to the interfaces, thus rendering the features of each family that are necessary to support the application at run-time. This task is accomplished by a tool, the `analyzer`, that output an specification of requirements in the form of partial component interface declarations, including methods, types and constants that were used by the application.

The primary specification produced by the `analyzer` is subsequently fed into a second tool, the `configurator`, that consults a build-up database to create the description of the system's configuration. This database holds information about each component in the repository, as well as dependencies and composition rules that are used by the `configurator` to build a dependency three. Additionally, each component in the repository is tagged with a "cost" estimation, so that the `configurator` will chose the "cheapest" option whenever two or more components satisfy a dependency. The output of the `configurator` consists of a set of keys that define the binding of inflated interfaces to abstractions and activate the scenario aspects eventually identified as necessary to satisfy the constraints dictated by the target application or by the configured execution scenario.

The last step in the generation process is accomplished by the `generator`. This tool translates the keys produced by the `configurator` into parameters for a statically metaprogramed component framework and causes the compilation of a tailored system instance. An overview of the whole procedure is depicted in Figure 2

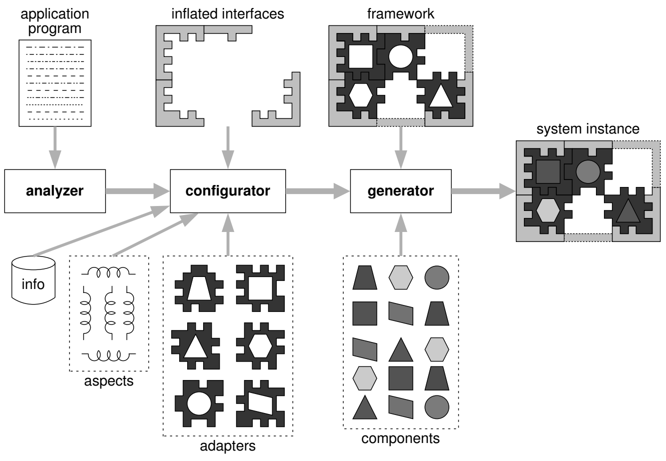

Figure 2. An overview of the tools involved in automatic system generation.

4. Software Component Description

The strategy used to describe components in a repository and their dependencies plays a key role in making the just described configuration process possible. The description of components must be complete enough so that the `configurator` will be able to automatically identify which abstractions better satisfy the requirements of the application, and this without generating conflicts or invalid configurations and compositions.

The strategy to describe components proposed here, could indeed be taken further as to specify components, for it encompasses much of the information needed to implement components, including their interfaces and relationships to other components. It is based on a description declarative language implemented around the *Extensible Markup Language* (XML) [16] and target at the description of individual families of abstractions[1]. The most significant elements in the language will be explained next, taking as basis the corresponding *Document Type Definition*) (DTD) fragments.

4.1. Families of abstractions

The declaration of a family of abstractions in our language consists of the family's inflated interface, an optional set of dependencies, and optional set of traits, its common package and a set of family members (software components), like this:

```
<!ELEMENT family (interface, dependency*,
   trait*, common, member+) >
```

The inflated interface of a family, as explained earlier, summarizes the features of the whole family and is specified as follows:

```
<!ELEMENT interface (type, constant,
   constructor, method) *>
```

The common package of a family holds type and constant declarations that are common to all family members. It is specified as:

```
<!ELEMENT common (type, constant) *>
```

The `member` element shown bellow is used to describe each of the members in a family. It is at the heart of the automatic configuration process, enabling tools to make the proper selection while looking for inflated interface realizations. A family member is declared as:

```
<!ELEMENT member (super, interface, trait,
   cost, feature, dependency) *>
```

The `super` element enables a member to inherit declarations of other members in the family, allowing for the creation of incremental families much as in *Incremental System Design* [10]. A member's interface designates a total or partial realization of the family's inflated interface in terms of members `type`, `constant`, `constructor` and `method`. Element `trait`, which can also be specified for the family as whole, designates a configurable information that can be set by users, via configuration tools, in order to influence the instantiation of a component[2]. A trait of a component can also be used to specify configuration parameters that cannot be automatically figured out, such as the number of processors in a target machine and the amount of memory available.

Additionally, each member of a family is tagged with a relative `cost` estimation that is used by the configuration tools in case multiple members satisfy the constraints to realize the family's inflated interface in a given execution scenario. This cost estimation is currently rather simplistic, consisting basically of an overhead estimation made by the component developer. More sophisticate cost models, including feed-back from the configuration tools, are planed for the future.

4.2. Functional dependencies

Although we can use the `analyser` tool to discover the dependencies of the application regarding the families' interfaces, this tool cannot be used to discover the dependencies that a family's implementation has on another one. Thus, this kind of depency must be explicited by the programmer through the `dependency` member. This dependency may happen to the entire family or just on individual members.

The `dependency` element has only one attribute that describes the name of the family whose dependency consists. We should also include in this dependency description the partial interface with the constructors and methods of one family called by the other one, allowing the `configurator` to be able to resolve any kind of dependency. However, in the actual development stage of Project EPOS we have never identifyied a family of abstractions whose members had dependencies on another family differentiated by the partial interface description. These differences usually occur regarding non-functional dependencies (described next). As so, we have decided to keep the simpler description of the family name only. In the future, if the situation where the partial interface is necessary to decide which abstraction needs to be selected to satisfy the dependency may occur, our model will have to be extended to support this detail level.

4.3. Non-functional properties and dependencies

The description of the interfaces in a family of abstractions is the main source of information for the proposed configuration tools, but correctly assembling a component-based system goes far beyond the verification of syntactic interface conformance: non-functional and behavioral properties must also be conveyed. For this purpose, our component description language includes two special elements: `feature` and `dependency`. These elements can be applied to virtually any other element in the language to specify features provided by components and dependence among

1 A complete description of the software component repository is obtained simply by merging individual families' descriptions.

2 Traits are made available at compile-time to the static metaprograms that build up the component framework.

components that cannot be directly deduced from their interfaces. Enriching the description of components with features and dependencies can significantly improve the correctness of the assembly process, helping to avoid inconsistent component arrangements.

For instance, consider a family of wireless network abstractions. Some members could declare a “reliable” feature, making them eligible to support an application whose execution scenario demands for reliable communication. Similarly, members of a family of communication protocols could specify the dependency on a “reliable” wireless network infrastructure, while other could implement the feature themselves.

A feature has a name and a value. The name should be regarded as a meaningful feature in the application domain. Considering the example above, we could specify the reliable feature of a wireless network as follows:

```
<family name="Wireless_Network">
  <interface>...</interface>
  <common>...</common>
  <member name="Wi-Fi">
    <interface>...</interface>
    <feature name="reliable" value="false"/>
  </member>
</family >
```

and the dependency in the protocol family, extendend to allow the inclusion of inner `feature` elements, as:

```
<family name="Wireless_Protocol">
  <interface>...</interface>
  <dependency family="Wireless_Network"/>
  <common>...</common>
  <member name="Active_Message">
    <interface>...</interface>
    <dependency family="Wireless_Network">
      <feature name="reliable" value="true"/>
    </dependency>
  </member>
</family >
```

It is important to mention that the fact of the `Active_Message` member of the `Wireless_Protocol` family requiring a reliable `Wireless_Network` does not summarily excludes the `Wi-Fi` member: the `configurator` would first check whether a scenario aspect is available that could be applied to a non-reliable network in order to make it behave as a reliable one. In the particular case of EPOS, such a scenario aspect exists and would enable the correct integration of both components.

5. Example

To validate the ideas proposed here, we have developed a very simple example to demonstrate the aplication of the concepts.

Our example consists on a C++ implementation of the Philosofer’s Dinner using two families of EPOS abstractions: `Thread` and `Synchronizer`. The program was written using the inflated interfaces of each family, without specifying the member implementation to use.

The output of the `analyser` for our application was:

```
<interface name="Synchronizer">
  <constructor>
    <parameter type="int"/>
  </constructor>
  <method name="p" return="void"/>
  <method name="v" return="void"/>
</interface>

<interface name="Thread">
  <constructor>
    <parameter type="System::Int::
       Thread_Common::Self const&">
  </constructor>
  <constructor>
    <parameter type="int (*)(int)"/>
    <parameter type="int"/>
    <parameter type="short const&"/>
    <parameter type="short const&"/>
  </constructor>
  <method name="suspend" return="void"/>
  <method name="wait" return="void">
    <parameter type="int *"/>
  </method>
  <method name="yield" return="void"/>
</interface>
```

Now we must enter this data into the `configurator`. For the selection of the member of the `Synchronizer` family, it would consult the XML file of the family, partly listed bellow:

```
<family name="Synchronizer" type="
    abstraction" class="dissociated">
  <interface>
    ...
    <method name="id" return="const Id &"/>
    <method name="valid" return="bool"/>
    <method name="lock" return="void"/>
    <method name="unlock" return="void"/>
    <method name="p" return="void"/>
    <method name="v" return="void"/>
    <method name="wait" return="void"/>
    <method name="signal" return="void"/>
    <method name="broadcast" return="void"/>
  </interface>
  <common>...</common>
  <member name="Mutex" type="inclusive" cost
      ="1">
    <interface>
      ...
      <method name="id" return="const Id &"/>
    <method name="valid" return="bool"/>
    <method name="lock" return="void"/>
    <method name="unlock" return="void"/>
   </interface>
   ...
  </member>
  <member name="Semaphore" type="inclusive"
      cost="5">
   <interface>
    ...
    <method name="id" return="const Id &"/>
    <method name="valid" return="bool"/>
    <method name="p" return="void"/>
    <method name="v" return="void"/>
   </interface>
   ...
  </member>
  <member name="Condition" type="inclusive"
      cost="10">
   <interface>
    ...
    <method name="id" return="const Id &"/>
    <method name="valid" return="bool"/>
    <method name="wait" return="void"/>
    <method name="signal" return="void"/>
    <method name="broadcast" return="void
        "/>
   </interface>
   ...
  </member>
</family>
```

One could easily see that the `Semaphore` member was the one choosen, since it is the only one that implement the `p()` and `v()` methods that were used by the application[3].

Similarly, the selection of the member of the `Thread` family member would be made based on the XML description, whose partly listing is:

```
<family name="Thread" type="abstraction"
    class="incremental">
  <interface>
   ...
   <method name="id" return="const Id &"/>
   <method name="valid" return="bool"/>
   <method name="state" return="volatile
       const State &"/>
   <method name="priority" return="const
       Priority &"/>
   <method name="priority" return="void">
    <parameter type="const Priority &" name
        ="priority"/>
   </method>
   <method name="join" return="int"/>
   <method name="pass" return="void"/>
   <method name="suspend" return="void"/>
   <method name="resume" return="void"/>
   <method name="yield" return="int"
       qualifiers="class"/>
   <method name="exit" return="void"
       qualifiers="class">
    <parameter type="int" name="status"
        default="0"/>
   </method>
  </interface>
  <common>...</common>
  <member name="Exclusive_Thread" type="
      exclusive" cost="1">
   <interface>
    ...
    <method name="id" return="const Id &"/>
    <method name="valid" return="bool"/>
    <method name="exit" return="void"
        qualifiers="class">
     <parameter type="int" name="status"
         default="0"/>
    </method>
   </interface>
   ...
  </member>
  <member name="Cooperative_Thread" type="
      exclusive" cost="10">
   <super name="Exclusive_Thread"/>
   <interface>
    ...
    <method name="pass" return="void"/>
   </interface>
   ...
  </member>
  <member name="Concurrent_Thread" type="
      exclusive" cost="15">
   <super name="Cooperative_Thread"/>
   <interface>
    ...
    <method name="state" return="volatile
        const State &"/>
    <method name="join" return="int"/>
    <method name="suspend" return="void"/>
    <method name="resume" return="void"/>
    <method name="yield" return="int"
        qualifiers="class"/>
   </interface>
   ...
  </member>
  <member name="Priority_Thread" type="
      exclusive" cost="20">
   <super name="Concurrent_Thread"/>
   <interface>
    ...
    <method name="priority" return="const
        Priority &"/>
    <method name="priority" return="void">
     <parameter type="const Priority &"
```

3 The constructors required by the application would also be analysed by the `configurator`, but since in this case they would be irrelevant for the selection, we have ommited them.

```
          name="priority"/>
     </method>
    </interface>
    ...
  </member>
</family>
```

Note that this is an incremental family, and, as so, every member is a specialization of the previous. Hence, any one of the members supports the elements declared in its own interface, as well as the elements inherited from its parents. Because of this, there are two members of the family that would satisfy the requisits of the application: `Concurrent_Thread` and `Priority_Thread`. In this case, the `configurator` would select the one with the lowest cost estimation, that is, the `Concurrent_Thread`[4].

After this first iteration, the `configurator` would check the dependency information for the selected families and members and, if necessary, include any other required family. In this case, both families members' implementations depends on the `CPU` family, and so it would be also included in the configuration and the member specific to the target machine processor (informed by the user) would be selected (promoving portability). A new iteration would be started to check the dependencies for the newly included families, an so on, until no new family need to be included.

6. Limitations

The mechanism (configuration keys) used to bind the inflated interfaces to the real implementation of one of their members only allows the connection of each inflated interface with just one member at a time.

If the above example was extended to use another kind of synchronization in conjunction with the semaphores already used, the `configurator` would correctly identify that two or more members would have to be selected to satisfy the requirements. However, that would be impossible to accomplish, since it would only be posible to bind one of them with the inflated interface of the `Synchronizer` family at a time.

The only posible solution for this situation at this moment is to ask the application programmer to write the calls to the synchronizers directly over the members' interfaces, instead of using the generic family's interface, thus forcing the programmer to explicit show his decisions about what kind of synchronizer he is using. If he does so, the `configurator` would understand that both members are necessary and their implementations would be included in the system's configuration without the necessity for any binding to the inflated interface.

4 The constructors required by the application would also be analysed by the `configurator`, but since in this case they would be irrelevant for the selection, we have ommited them.

7. Supporting Tools

At the present, we have prototype implementations of the `analyzer` for applications written in C++ and JAVA. These tools are able to parse an input program and produce a list of the system abstraction interfaces (inflated or not) used by the program, identifying which methods have been invoked and, in the case of JAVA, in which scope they have been invoked.

This information serves as input for the `configurator`, which is currently being developed. The `configurator` is indeed implemented by two tools. The first one is responsible for executing the algorithm that will select which members of each family will be included in the customized version of the system. This algorithm consists in reading the requirements found by the `analyzer` and compare them with the interfaces of each member of the family as specified in the repository. Every time a new member is selected, its dependencies are recursively verified, including in the configuration any members from other families that are needed to satisfy them. The second part of the `configurator` is a graphical tool that allows the user to browse an automatically generated configuration, making manual adjustments, if needed. Moreover, the user will have to enter some important information not discovered automatically: the configuration of the target machine (architecture, processor, memory, etc.) and the values of the traits of each component.

At last, the configuration keys outputted by the `configurator` are used by the `generator`, which is implemented as a wrapper for the *GNU Compiler Collection*, to compile the system and generate a boottable image.

8. Further work

We are finishing the implementation of the `configurator` that will be capable of automatically generating the configuration for a customized version of EPOS. Further works could refine the specification and implementation of the system configuration model in two aspects:

- Inclusion of behavioral specification in the component description model. This specification would cover dependencies like: some method of a component can only be invoked if the component is in determinate state. This kind of specification would have to be validated by some sort of formal mechanism, like a Prolog inference engine or a Petri network.
- Evolution of the mechanism used to select members by performance. Today, this task is accomplished using the specification by the programmer of a cost es-

timate of each member on each family if the form of overhead. More elaborated mechanisms would include an automated way to measure real performance of each member during execution time.

9. Conclusion

In this article we have presented an alternative to achieve automatic run-time system generation taking as base a collection of software components developed according with the Application-Oriented System Design methodology. The proposed alternative consists of a novel component description language and a set of configuration tools that are able to automatically select and configure components to assembly an application-oriented run-time support system.

The described configuration tools are in the final phase of development and allow the exposition of the system libraries to application programmers through a repository of reusable components described by their inflated interfaces, which are automatically bound to a specific realization at compile time. This is possible due to the component specification model that contains all the information needed to generate valid and optimized configurations for each application.

We have shown an example of the configuration process for a simple but real application, and the limitations of the model regarding the selection of more than one different realization for a component at the sime time.

This architecture makes possible the creation of optimized versions of the operating system for the target applications, assuring that the performance levels and resource usage optimization for embedded and parallel aplications will be certainly better that those achieved with general-purpose operating systems.

References

[1] Thomas Anderson. The Case for Application-Specific Operating Systems. In *Proceedings of the Third Workshop on Workstation Operating Systems*, pages 92–94, Key Biscayne, U.S.A., April 1992.

[2] Lothar Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conference on Advanced Systems Engineering*, Heidelberg, Germany, June 1999.

[3] Danilo Beuche, A. Guerrouat, H. Papajewski, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute 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*, St Malo, France, May 1999.

[4] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edition, 1994.

[5] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 38–51, St. Malo, France, October 1997.

[6] Antônio Augusto Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.

[7] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. High Performance Application-oriented Operating Systems – the EPOS Aproach. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, pages 3–9, Natal, Brazil, September 1999.

[8] Antônio Augusto Fröhlich and Wolfgang Schröder-Preikschat. Scenario Adapters: Efficiently Adapting Components. In *Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics*, Orlando, U.S.A., July 2000.

[9] Antônio Augusto Fröhlich, Gilles Pokam Tientcheu, and Wolfgang Schröder-Preikschat. EPOS and Myrinet: Effective Communication Support for Parallel Applications Running on Clusters of Commodity Workstations. In *Proceedings of 8th International Conference on High Performance Computing and Networking*, pages 417–426, Amsterdam, The Netherlands, May 2000.

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

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyväskylä, Finland, June 1997. Springer.

[12] Leonid Mikhajlov and Emil Sekerinski. A Study of the Fragile Base Class Problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382, Brussels, Belgium, July 1998. Springer.

[13] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[14] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component Composition for Systems Software. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, U.S.A., October 2000.

[15] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and 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*, Paderborn, Germany, October 1998.

[16] World Wide Web Consortium. *XML 1.0 Recommendation*, online edition, February 1998. [http://www.w3c.org].