Streamlining Object-Oriented Software for Deeply Embedded Applications*

Danilo Beuche, Wolfgang Schröder-Preikschat, Olaf Spinczyk, Ute Spinczyk
University of Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
{danilo, wosch, olaf, ute}@ivs.cs.uni-magdeburg.de

Abstract

Most of today's computer systems are embedded and a substantial amount of them are deeply embedded systems with very limited resources. Object-oriented software is rarely to be found in those systems. In this paper we discuss reasons why currently object orientation is seldom used in this area. We also propose a set of design and implementation techniques based on modern object-oriented methods, which reconcile the reusability and flexibility of object-oriented software with the required efficiency of deeply embedded systems.

1 Introduction

Most of today's computer systems are more or less hidden in embedded applications like mobile phones or cars. Even some kitchen aids have their own processors. More and more application areas are "computerized". In many if not most of them only very limited computational power is available. These so called "deeply" embedded systems are a great challenge for today's software producers.

In contrast to PCs the various constraints like cost reduction, power consumption and used memory space lead to the usage of very small processors. They are small in computational power, usually only a few MIPS, or even less. Sometimes RAM-less systems are used, but typical RAM sizes range from a few bytes to some MBytes. The software is usually hold in ROM which can take from 1 kByte to a dozen MBytes.

Many of these products are hard or soft real-time systems which must respective should provide deterministic behavior regarding time. Some of the systems are even safety-critical systems e.g. avionic instruments or steering control systems in cars.

Developing software which meets these constraints is not an easy task. A software architecture for this domain should provide

Portability: Today available hardware changes and improves very fast. Existing software must be easily usable on new platforms.

Scalability: The software should be scalable so that it is possible to provide as much or little functionality as needed for a specific application. Application software must not include "dead code", which is never used.

*This work has been partly supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1 and the Bundesministerium für Wirtschaft (BMWi), grant no. 01 MS 801/7

Extensibility: To minimize the probability of introducing bugs and to facilitate software maintenance in general, extensions should require as little modification of existing source code as possible.

Composability: It must be easy to select the right configuration for a given set of requirements.

Efficiency: None of the previous items are of any relevance if the resulting software artifact is not small and fast enough to run on the desired hardware.

Today's software for deeply embedded systems is seldom developed using object-oriented design and implementation methods. To meet the strong constraints concerning run-time and memory usage, those programs are still very often written in assembly and/or C and optimized for the dedicated platform. Porting software to even a slightly different systems is quite difficult then.

In section 2 of this paper we discuss common reasons why object-oriented methods fail to be usable in deeply embedded context. The discussion will be based on the items given above.

In section 3 we present an (old) concept, the program family [13], which provides the basis to develop an efficient object-oriented software structure. Software built based on this concept delivers programs which are small enough to run on almost all available platforms. In addition, the program family concept is in theory scalable to large systems too. As an example, we present our operating system family PURE, which mainly aims at the area of deeply embedded systems.

Our experience showed that both developing and using larger systems based on the family concept calls for additional support. The main part of the paper contains our ideas, namely how to use feature modeling and aspect-oriented programming to achieve usability of the program family concept in real-world scenarios.

2 Why Object Orientation fails so often

For all general requirements mentioned in the previous section separate solutions in the object-oriented world exist. But if the solutions are used in combination, the efficiency requirement is violated in most cases.

2.1 Portability

Portability is not really an issue where object-oriented methods are needed, however object-oriented design and implementation allows strict encapsulation of non-portable parts into separate methods and/or classes. Problems arise because the non-portable parts are often very small, e.g. a single line of code, but are called many times and from many places. If no inlining of such methods is used, a performance impact may arise. Many object-oriented languages support explicit or implicit inlining of such small methods. But then the efficiency relies mostly on the optimization capabilities of the compiler used.

2.2 Scalability

Scalability is one of the most important aspects in the domain of deeply embedded systems. The architecture of the software systems must allow down-scaling of the functionalities. It is not acceptable to have any unused code in the final system. Many object-oriented systems fail at this point. The main reason is quite simple: the use of abstract classes to provide a general interface. If abstract classes are used as base classes and the derived classes implement specializations, many cases occur where it is not possible to detect whether an overridden method is used or not. In the example shown in figure 1 it is not possible to decide if pointer `aPtr` will ever refer to an instance of `B` so `B::foo` must be always included if an instance of a class, which was derived from `B`, is used.

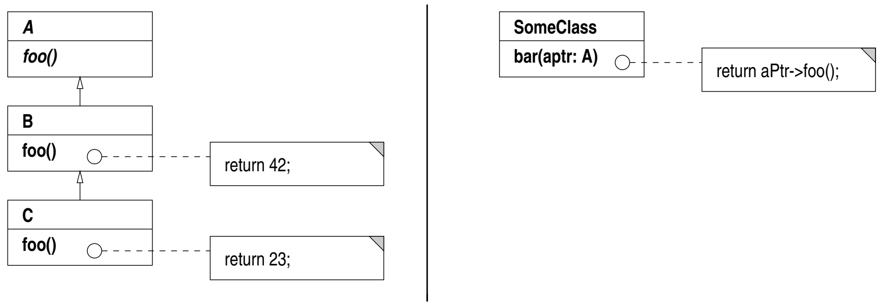

Figure 1. A problematic class hierarchy

In fully static scenarios where every instantiated class is known at compile time it would be possible to leave out unused code, but in most cases such knowledge cannot be gathered easily. Especially, if any kind of dynamic loading of code is available every fragment which could possibly be loaded has to be analyzed a priori. Dynamism on method level is required if such an analysis cannot be done, but for deeply embedded systems the overhead due to a fully dynamic architecture is not tolerable. An object-oriented operating system with a dynamic architecture hardly fits into 4 kBytes. Examples like JavaCard [12] show how small such systems are, but unfortunately they are not small (and fast) enough for deeply embedded applications. On the other hand, for embedded systems with less strong resource constraints a dynamic architecture has many benefits like loading and unloading of components on demand.

2.3 Extensibility

Extensibility is one of the most important issues in an architecture for embedded systems. To be able to keep pace with the quickly growing demands of the market, the designer of an embedded system always has to consider future changes. So, a competitive software system should be open. Reuse through extension is one of the strongest points for object-oriented systems. But if reuse means changing existing abstractions, an error-prone software will be the outcome. This happens because the previously existing behavior may change and break other parts of the system which relied on it. A way out of this are contracts between the abstractions. Beugnard et al. [3] discusses different kinds of contracts and their usage. Another approach is to have immutable abstractions, which do not need to be touched if the system is extended. So existing applications do not need to bother with a changing behavior of the system.

Extensions should be made easy and flexible. A good and flexible way to design extensible systems is the use of patterns [9]. For instance the use of the strategy pattern for selecting the scheduling algorithm in an operating system lets one introduce a new scheduling algorithm without changing any other part of the system. Unfortunately the use of patterns conflicts with the efficiency requirement. This is due to the inherent dynamic structure of patterns. Even if only a single strategy is used, the pattern code requires dynamic invocation of an implementation of that strategy. This leads both to consumption of more processor cycles for resolving the dynamic invocation and also to unnecessary code and data (dynamic dispatch code and virtual function tables) compared to a solution where the strategy would be called directly.

2.4 Composability

Composability has two main aspects. The first aspect is to have an architecture which allows integration of components from different sources, e.g. drivers for specific hardware or communication protocols. The current available solutions like CORBA, COM, ComponentPascal [16] or JavaBeans use binary components. All these component models are too heavy-weight to be applicable in the deeply embedded context, the necessary software infrastructure is just too expensive. A second point against the use of pure black-box binary components is the way they are customized to the need of applications. The code of the component is fixed, different behavior is achieved by modifying the component attributes.

Source code component models provide easier generation of lean systems, because unnecessary code can be left out if the configuration of the execution context is known at compile time. But a component model which is based only on static compile time component composition is not suitable if dynamically changing systems are to be supported. While for small deeply embedded systems a fully static approach fits best, for more powerful systems dynamic composition of varying degree can be useful.

Thus, what is favored is an architecture which supports both, static and dynamic composition of components. This allows source code components to be customized according to the actual system context and a dynamic combination of them.

The second aspect of composability is the question which of the available components in a system meet the requirements of the application. For instance in real-time embedded systems, the behavior of a component regarding time plays an important role. The chosen combination of components must ensure e.g timeliness and also deterministic behavior. Again, as with the extensibility, contracts and also other formal methods play an important role here.

3 Program Families — the First Step

A very promising concept is the program family concept of Parnas. The motivation of Parnas was that *"some users may require only a subset of services or features that other users need. These 'less demanding' user may demand that they are not to be forced to pay for the resources consumed by the unneeded features"* [13]. This sentence matches perfectly to the domain of deeply embedded system software. Parnas solution was to use so-called *program families*. A program family consists of a minimal subset of functions, called the *minimal base*, and a set of extensions. Each extension should also be minimal in functionality. The combinations of different sets of minimal extensions are called the *family members*.

Although there are a number of object-oriented operating systems like Choices [5] or PANDA [1] which represent families according to Cahill [4], only few of them like PEACE [15] are program families in the narrow sense of the citation above.

The interesting thing about program families and object-orientation is that the hierarchy of the minimal base and minimal extensions can be easily expressed by using inheritance. As shown in figure 2, the minimal base is a base class and the minimal extensions are derived classes. However, note that this is contrary to the use of the abstract base class discussed in section 2.2. The advantage of applying inheritance in an incremental system design is, that even with standard compilation systems, no unnecessary code will be included in the resulting family members. Remember the class hierarchy given in section 2.2. If instead of using class `A` as abstract base class a construct like C++'s `typedef` would be used when appropriate, it is possible for a standard compilation

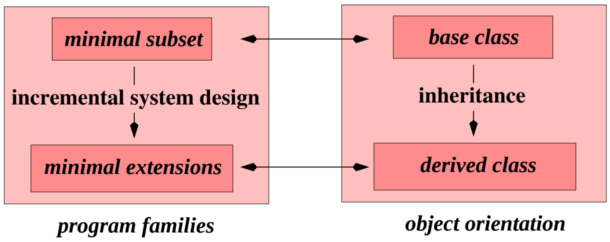

Figure 2. Program family meets object-orientation

systems to leave out `B::foo` if it is never referred to. Only if different classes with the same derived interface have to be used in one family member, abstract classes at implementation level are useful.

The PURE system developed by our group exploits the family concept to its full extent and uses object-oriented principles to implement the design in C++. It focuses on systems ranging from deeply embedded systems to medium sized embedded systems. In the remainder of this section we discuss our experiences with the current version of PURE and see how it fulfills the requirements discussed previously.

3.1 Lessons Learned

Developing a highly modular operating system family like PURE is one story, maintaining the resulting software is a different one. The resulting figures of PURE prove that the consequent application of the program family concept results in a very efficient implementation for the deeply embedded systems area. Table 1[1] shows system sizes and context switch times for different family members ranging from a single active object over simple cooperative scheduling to full preemptive members. These numbers are discussed in more detail in [2] and [14].

Configuration	Code+Data Size (bytes)	Context Switch (cycles)
single active object	434	n/a
cooperative scheduling	1648	61
preemptive scheduling	4062	368

Table 1. Comparison of different PURE configurations.

Our experience showed also that without additional support the program family concept becomes hard to use when the system family grows. Right now there are some 250 classes in the PURE family and about 50 configuration points with one or more configuration option, but it still must grow substantially to fully support the desired application domain. The static PURE configuration is heavily based on the usage of C++ preprocessor mechanisms and Makefiles. As configuration options are not independent from each other a means to express relations of configuration options is needed. This prevents choosing of illegal (resulting in faulty systems) or superfluous configuration options. Our current implementation has no explicit support for such checks. Closely related is the question which set of extensions a system needs to provide a defined set of functionalities.

These problems influence the aspects of composability and extensibility. If it is too complicated for application programmers to decide which operating system configuration suits their needs, a system is unusable.

[1]Figures were taken on a Pentium II, egcs-1.0.2

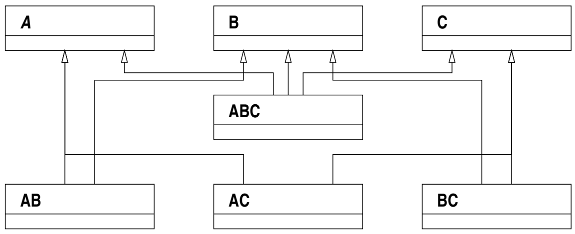

Figure 3. Class explosion

A critical issue with our program family is the usage of inheritance to combine extensions. Although it is the only way to achieve maximal performance under certain conditions, it causes problems. One problem related to this is the class number explosion. In figure 3 a simple example is presented. There are three independent extensions implemented in class A, B and C. To represent the possible combinations of them with derived classes, four classes are needed. The problem increases dramatically if the system consists of many classes. To solve this problem flexible connections of extensions are required. Object aggregation and composition could be used instead of inheritance in many cases. But as discussed in section 2.2, it may be more costly to do it this way. On the other hand, usage of dynamic combination techniques would have the additional benefit, that the number of extensions could be reduced. With the current implementation structure a different or additional extension for dynamic cases is needed. If the scheduler e.g. requires dynamically exchangeable scheduling strategies additional classes need to be introduced to provide this functionality. This introduces more complexity into the system design.

Another benefit from using more dynamic forms of combination like object aggregation or object composition instead of inheritance is that many design decisions become easier. If inheritance is used, we have to be very careful when and where an extension is introduced into the hierarchy. If the extension is introduced too early, it will be unnecessary in some cases. If the extension is introduced too late, it might result in systems, where due to the incremental structure of the PURE implementation other extensions with unnecessary functionalities also have to be included. With dynamic composition of extensions it should be easier to leave out unnecessary functionality.

4 Feature-Modeling, AOP and Patterns — Putting Pieces Together

In the context of PURE the search for solutions to the problems mentioned before happens under the following premises:

- The System configuration must become simpler.
- The Configuration must deal with both, functional and non-functional requirements.
- Non-functional requirements like optimization or code instrumentation must be expressible.
- New extensions should be created easily.
- It must be possible to generate systems which are as fast and as small as “handmade” systems.

Our approach is to use a combination of different techniques to achieve these goals. In figure 4 the main techniques are presented. The configuration is based on features and associated feature models. The software is organized in scalable components, which use pattern based designs. The final system is created by aspect weavers, which optimize the software structure according to the configuration requirements.

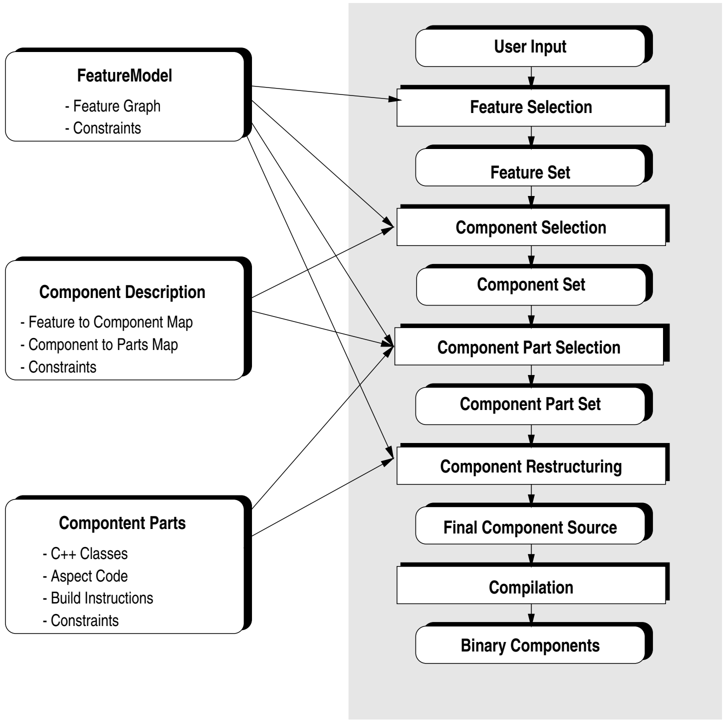

Figure 4. Combination of methods

The key issues *feature modeling* and *pattern restructuring with AOP* are discussed in the remainder of this section.

4.1 Features

To express the capabilities of a software system like PURE which covers a broad range of possible applications, domain engineering methods should be used. Domain engineering deals with whole application domains rather than a single, specific application and, in fact, PURE tries to cover the domain of operating systems for deeply embedded systems.

An important criteria for selecting a method was that it provides us with information for both, developers and users of the program family. We took parts of the Feature-Oriented Domain-Analysis described in [10]. It exploits features and dependency rules between features to describe an application domain. A feature may be a functionality like the support for TCP or a non-functional requirement like speed optimization. This combination of functional and non-functional aspects of a software system in one model was very attractive in our case. When application developers have to select the building blocks of the operating system, they can use its uniform representation based on feature models.

Our feature model is based on the collected features of the domain which is organized in directed acyclic graphs. Each graph represents a main feature, called concept and its dependent features. The nodes are features, the directed edges represent the kind of dependency between nodes. A small example of such a graph is shown in figure 5, the notation is taken from Czarnecki [6]. The selection of a feature somewhere in a graph leads to the inclusion of all nodes along the path to the concept node. For instance, selection of `Thread` requires also the selection of nodes `Scheduling`, `Coroutine`, `Multiple` and `ActiveObject`. This behavior is very similar to the construction principle of a program family: A minimal extension representing a functionality, requires other extensions and a minimal base to work. When we started to express existing parts of PURE with features, it turned out to be relatively easy because of this analogy. If the analysis of a domain is done carefully, we believe the reverse direction works as well.

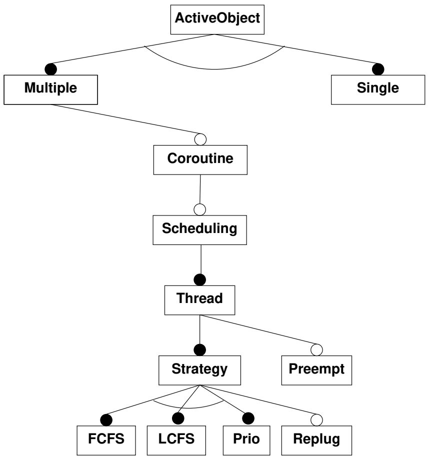

Figure 5. A simple feature model graph

For main parts of PURE we already have working models, which are evaluated by a configuration tool. The selected features are mapped to components. Each component uses the feature set to determine which parts (e.g. classes, flags or methods) of the component are needed to provide the required features. In our current implementation of component descriptions, such rules are expressed in PROLOG rules and facts. Each component part may use the same information to determine which files are needed to implement itself. This is described in PROLOG rules and facts as well. The advantage of our configuration process is that it is open. Other methods can be integrated, for instance formal specification models of components to verify that the chosen configuration of components meets a given specification of the system. The goal of the basic framework is to provide an abstraction from implementation details when choosing system features. Figure 6 shows the configuration process applied to a small example system family. The family is an implementation of the feature model in figure 5. At two places (Classes `StrategyDat` and `Activity`) a variable base class is used to provide an efficient implementation. After the configuration process the correct base class relations are made.

Our results are very positive: instead of making decisions for each of the 50 possible configuration options in PURE individually, a system can be described by only a couple of features. In the simplest case (single active object) one feature is sufficient to describe the desired requirements. A more complicated selection (preemptive system, VGA support, x86 PC platform) needs seven features. In most cases the component descriptions and mappings to implementation files are merely a collection of class names and files. We needed 22 additional rules describing component configurations for a total of 67 classes with 15 configuration flags.

4.2 Aspects

Sometimes configuration decisions have 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

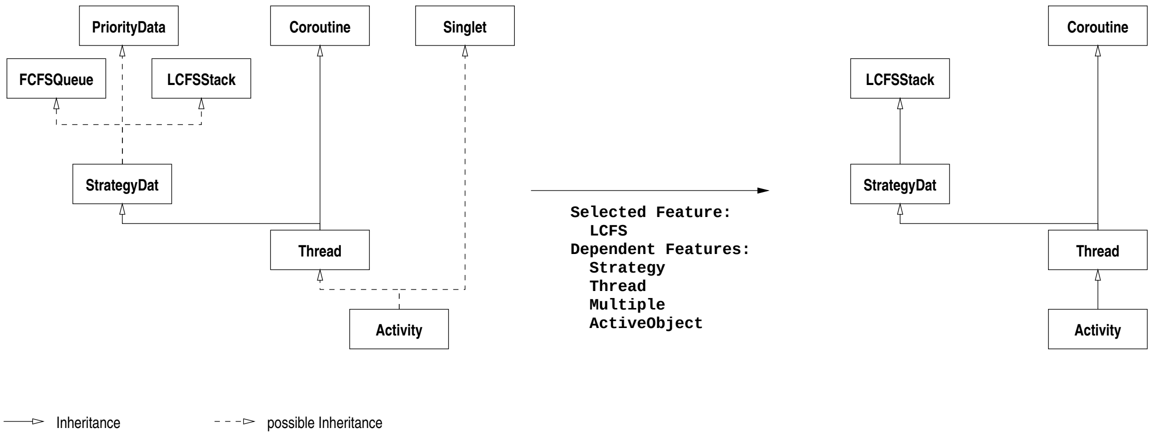

Figure 6. Applied feature model

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.

4.2.1 Aspect-Oriented Programming

This is exactly the problem that aspect-oriented programming (AOP) [11] 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 because the process of configuration can be reduced to 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. AOP even provides means to express the code that is needed to fulfill non-functional requirements in separated modules, thus it provides a direct correspondence of features and source code units. This simplifies the software development and especially the task of component description (see figure 4).

The great disadvantage of AOP and, in our opinion, the reason why AOP can still be seen as an "academic" concept is the fact that nothing works without tool support. The aspect specific weaver tool interprets the aspect code that is written in a problem-adequate language. Weaver tools or frameworks (for C++ component code) are not available until today. So we decided to develop an aspect weaver environment called "PUMA for C++". It is a parser and manipulator for C++ code with a plug-in interface for multiple aspect weavers.

4.2.2 Scaling System Structures

The first PUMA-based aspect weaver is used for the restructuring of pattern-based PURE subsystems. It is a very important example because it reconciles the flexibility of pattern-based designs and implementations in the sense of scalability, extensibility, and composability with the efficiency

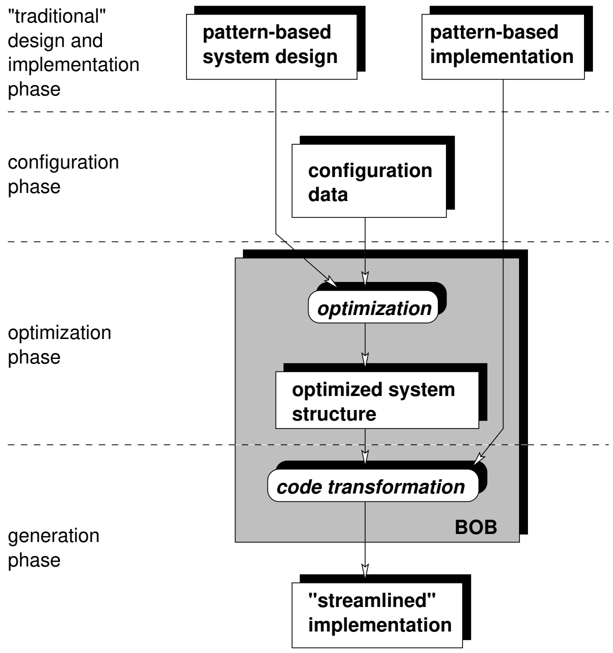

Figure 7. Process view of the pattern restructuring approach

needed in the area of deeply embedded systems. The following paragraphs illustrate with a small sample subsystem how this tool can be used. A detailed discussion of the approach can be found in [8].

The reason why pattern-based subsystems are prone to inefficiency if they were used as part of a program family is that the excessive use of abstract base classes and dynamic binding, that is favored in most design patterns, leads to an overhead in configurations where this flexibility is not required by the environment of the subsystem. The idea of our approach is to omit this overhead by streamlining the subsystem structure automatically according to the actual system configuration.

Figure 7 gives an overview about the development phases, the required input files, the internal structure of the aspect weaver, and the result of the restructuring process. The "pattern-based implementation" box represents the component code, the "pattern-based system design" and the "configuration data" are the aspect programs in the sense of AOP. The component code is the source code of the subsystem that is manipulated in the "code transformation" step. To do this the PUMA-based transformation system needs a representation of the "optimized system structure" that is determined by an optimizer that uses the aspect code as its input.

The software developer has to provide a description of the subsystem structure. This is done in a problem-oriented way by associating the class names to the role names of the applied design pattern. Applied patterns should be documented anyway, so this additional effort is acceptable for us. The structure of the pattern is known from a pattern library, so the subsystem's structure can be derived.

In the configuration phase "configuration data" has to be provided. It contains a requirement definition that describes which classes and especially which class relations are required in the current system configuration. This is another configuration complexity, but it can be reduced with a feature model, because the requirement definition can be derived from the selected features automatically.

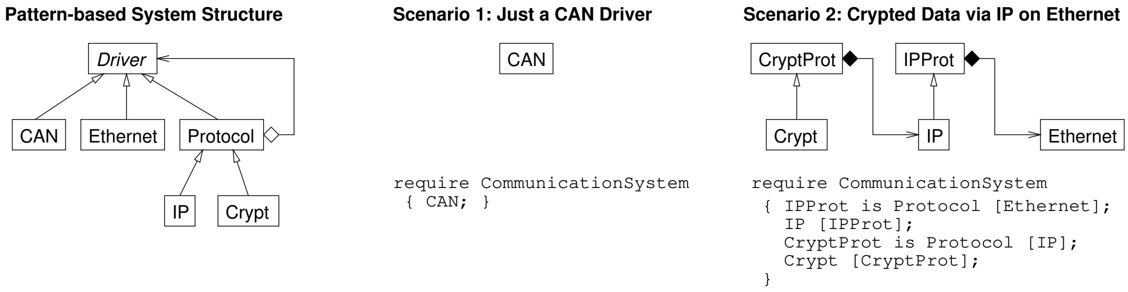

Figure 8. Sample transformation

4.2.3 Example

By selecting different requirement definitions it is possible to generate various subsystem structures. Each one is optimized according to the requirements. This is illustrated in figure 8. The left class diagram contains the structure of the sample communication system that is designed according to the decorator pattern [9]. The structure is very flexible because low-level drivers and protocols can be added without touching existing source code and the using environment is able to create any possible chain of protocols that is terminated by a low-level driver. The disadvantage is that every function call in this chain is a virtual function call leading to a waste of runtime and memory space for virtual function tables and dispatch code.

With the presented approach it is now possible to generate a family of implementations. If, for example, some system configuration only requires a CAN bus driver the structure in the middle can be generated. If you want to send crypted data via IP on an ethernet you can get the structure on the right. The black diamond-shaped boxes mean aggregation, so a single object can be instantiated that contains all the necessary state information and virtual function calls are avoided.

To demonstrate the results of the presented approach we have implemented a communication subsystem similar to the one from figure 8. To simplify the implementation we used a local inter-process communication via a shared mailbox instead of IP or CAN. As protocols we used check-summing and encryption. With this “renaming” we had a memory space reduction of 4.8% in both scenarios. This is not much but the subsystem was embedded into a larger system that was not optimized. So we expect better numbers when we reimplement larger pieces of PURE with patterns in the near future. The runtime reduction of 23.3% was much better. This value is about as big as the time reported by other groups, e.g. [7], that is spent in dynamic dispatch code in normal C++ applications.

5 Conclusions

In this paper we proposed an approach which meets the five discussed requirements for the deeply embedded software domain. The base of the approach is the program family concept. This concept — if applied carefully — allows very efficient, scalable and portable solutions. However our object-oriented operating system family PURE showed deficiencies in the areas of extensibility and especially composability. To deal with these issues we use the domain-engineering technique feature-modeling and aspect orientation to get a system which is easy to extend but remains as efficient as our original software structure. Another benefit of our approach is that through the use of patterns we are able to reuse the same implementation for both static and dynamic systems without a decrease in efficiency.

Future work will involve the complete reengineering of the PURE systems to use this approach. To make it usable in applications with hard and soft real-time requirements additional methods for proving correctness and guaranteed behavior have to be integrated.

References

[1] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. PANDA—Supporting Distributed Programming in C++. In O. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, volume 707 of *Lecture Notes in Computer Science*, pages 361–383. Springer-Verlag, July 1993.

[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] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, pages 38–44, July 1999.

[4] V. Cahill. Flexibility in object-oriented operating systems: A review. Technical Report TCD-CS-96-05, Dept. of Computer Science, Trinity College Dublin., July 1996.

[5] R. Campbell, G. Johnston, and V. Russo. Choices (Class Hierarchical Open Interface for Custom Embedded Systems). *Operating Systems Review*, 21(3):9–17, 1987.

[6] K. Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Technische Universität Ilmenau, Germany, 1998.

[7] K. Driesen and U. Hölzle. The Direct Cost of Virtual Function Calls in C++. In *OOPSLA'96 Proceedings*, October 1996.

[8] M. Friedrich, H. Papajewski, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Efficient Object-Oriented Software with Design Patterns. In *Proceedings of the 1st International Symposium on Generative and Component-Based Software Engineering (GCSE'99)*, Lecture Notes in Computer Science, Erfurt, Germany, 1999. Springer Verlag. To appear.

[9] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

[10] K. Kang, S. Cohen, J.Hess, W. Peterson, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[11] 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.

[12] Sun Microsystems. Javacard technology page. http://www.javasoft.com/products/javacard/.

[13] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[14] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In F. J. Rammig, editor, *Proceedings of the International IFIP WG 9.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems (DIPES '98)*, Paderborn, 1998. Kluwer Academic Press. ISBN 0-7923-8614-0.

[15] W. Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[16] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley, Reading, MA, 1997.