PURE Embedded Operating Systems—CiAO[*]

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

Abstract

The increasing complexity of embedded systems calls for operating systems that are highly specialized and at the same time are made of a number of reusable building blocks. This brings up a conflict as software specialized in supporting a very dedicated case usually cannot be (easily) reused for a different environment. As described in the paper, supporting specialization without abandonment of reusability was the major goal in the design and development of the experimental PURE operating-system family. The paper motivates the idea of an operating-system product line (CiAO) in order to come up with highly customizable and yet reusable system software solutions.

1. Introduction

Due to the need for highly customized solutions, particularly the embedded systems domain calls for a large assortment of specialized operating-system components. Depending on the application case, not only are number and kind (in functional terms) of the components varying, but also the same single component may appear in highly different versions. Most crucial in this setting are non-functional properties that are ingredient parts of single components or crosscut in the extreme case the entire system software. These properties do not only limit component reusability but also impair software maintenance in general [12].

Being able to deal with software variability in general becomes more and more eminent for embedded systems. PURE [5] took especially this sort of problem up. One of the primary goals of the PURE project (1997 – 2002) was to come up with solutions that help to manage configuration variability. Outcome was a *portable universal real-time executive* for deeply embedded systems in shape of a construction kit of highly adaptive C++, C, and assembly language building blocks. In this regard, PURE and eCos [1] share a lot of commonalities—except for the following major difference.

Key aspect in the PURE development was to understand an (embedded) operating system as a *program family* [17]. Commonalities of and differences between individual members of the operating-system family, as well as their interdependencies and conflicting combinations, were adequately expressed on the basis of *feature models* [7], with the features representing the functional and non-functional system properties. PURE family members were automatically assembled from the construction kit using feature-based and application-aware configuration [6, 3]. The PURE successor CiAO[1] (2002–present) goes a step further and understands an embedded operating system as a *software product line* [22]. In addition, CiAO consequently employs *aspect-oriented programming* [9] in order to maintain non-functional code separate from software components and, thus, improve reusability of the latter.

The rest of the paper is organized as follows. Section 2 briefly introduces the software-engineering concepts and techniques used in PURE to provide application-aware system software particularly suited to the demands of deeply embedded systems. The problem domain of the discussion comes from the automotive-systems industry and considers some constraints in building system software for today's cars. An overview about ongoing work regarding a pure aspect-oriented design and implementation of embedded operating systems following CiAO ideas is presented in section 3. This section also sketches some ideas on how to smoothly transfer between event- and time-triggered CiAO systems. Section 4 draws the conclusion.

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

[1] *CiAO is aspect-oriented*

2. Embedded Operating-Systems Engineering

Unbroken thread in the development of PURE was the postponement of all those design and implementation decisions that would have restrict applicability of system functions or components. This included that, perhaps, certain decisions were never made inside the system, but rather considered a case for the application programs to be supported. The following subsections discuss the cornerstones of an operating-system development process that supports highly scalable and customizable designs as well as implementations.

2.1. Problem Domain: Automotive System

A car, from a computer science point, is a “distributed system on wheels”: 40 up to over 100 of (8-, 16-, 32-bit) microcontrollers interconnected by a complex network (e.g. LIN, CAN, MOST, FlexRay) is the normal case—as is a $1\,l/100\,km$ additional fuel consumption due to the energy requirements and weight of the electronic equipment [21, 10]. These microcontrollers build the heart of the electronic control units (ECUs) that take care about the various monitor and control functions. The “on-board network” of a car typically consists of about 3 – 6 fieldbuses (depending on model and configuration) interconnected by a gateway node. A shift from event- to time-triggered fieldbus-based subnetworks can be currently encountered.

Generally, an ECU is a highly specialized device in terms of hardware and software, with the latter implementing most of the intellectual property. The majority of the ECUs is being crafted by supplier industry. Both variant and vendor of the operating system to be used for all the ECUs may be prescribed by the OEM, depending on its production process. European automotive industries committed on the OSEK^{TM} standard, which comes in two different flavours with respect to operating-system support for time- or event-triggered mode of operation [16, 14]. Using OIL [15], each OSEK operating system gets customized by the ECU supplier to meet the needs of the particular application, i.e., the requirements of the OEM. At all, this results in a fairly large number of different OSEK instances offering per ECU different task concepts and numbers, event numbers, synchronization mechanisms and strategies, for example, in addition to the ECU-specific driver software.

In the automotive-systems domain, configuration variability is not only an issue of the operating but also communication system. FlexRay [8], for example, defines about 200 different parameters, about 100 of which can be configured by software. The complexity of the FlexRay configuration space led to the discussion amongst OEM and suppliers on how to artificially constraint the parameter set and to leave only a small number of, e.g., 20 parameters available for software configuration [4]. As a consequence, configuration variability would be handled at the expense of FlexRay potentials and in favour of a less complex approach to generate ECU application/system software.

2.2. Feature Modeling

Feature modeling is understood as “the activity of modeling the common and the variable properties of concepts and their interdependencies and organizing them into a coherent model referred to as a *feature model*. [7]” Goal is to come up with directives for and a first structure of a design of a system that meets the requirements and constraints specified by the features.

Common is a graphical representation of the feature model in terms of a *feature diagram*. The diagram is of tree-like structure (fig. 1), with the nodes referring to specific feature categories. Four fundamental feature categories (represented by different kinds of arcs) are defined: *mandatory*, *optional*, *alternative*, and *or*. A feature diagram describes the options and constraints that exist within a system. Once features have been mapped to software artefacts (such as program constants, variables, procedures, modules), a feature diagram models the variable and fixed properties of a family of programs implementing that system.

Feature modeling of PURE resulted in a feature diagram of about 250 features allowing for about 2^{105} different valid feature combinations. The smallest possible PURE feature set comes up with just three features (CPU, target platform, and compiler), leading to the selection of 20 C++ classes in the configuration process. A feature set for a typical PURE configuration (with preemptive multitasking) has about 20 features. This set describes all the (functional/non-functional) properties of a given member from the PURE nucleus subfamily.

The PURE nucleus concept allows for three fundamental configurations as defined by the two or-features “thread concept” and “interrupt concept” (fig. 1):

1. If only the thread-concept feature is required, the nucleus configuration excludes any means of interrupt handling. In this configuration, interrupt handling is entirely up to the application program if needed.

2. If only the interrupt-concept feature is required, the nucleus configuration excludes any means of thread handling. In this configuration, thread handling is entirely up to the application program if needed.

3. If both features are required, the nucleus configuration comes with thread handling as well as interrupt handling.

Thus, according to the meaning of an or-feature, once the nucleus-concept feature gets to be included in the final sys-

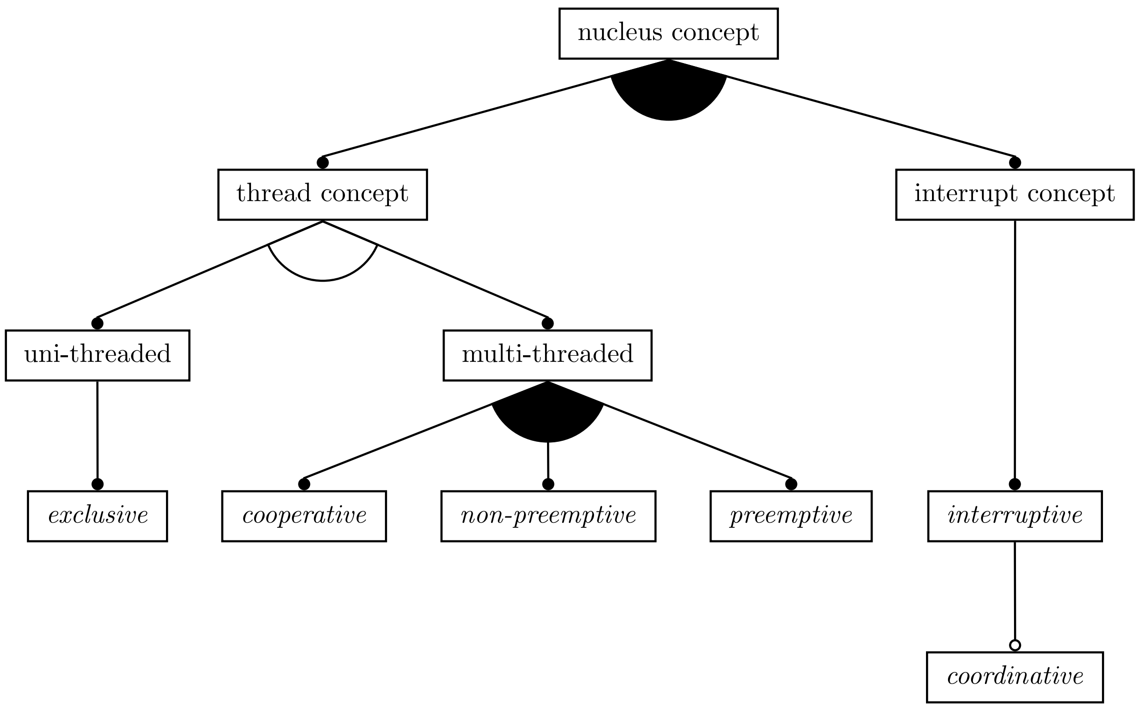

Figure 1. Feature diagram of the PURE nucleus family. Emphasized feature names indicate the different operating modes provided by the family.

tem configuration any non-empty subset of features from its set of or-features is also included.

The interrupt concept defines the mandatory feature *interruptive*, which means that it must be selected if the interrupt concept was selected. This nucleus family member enables the reactive execution of tasks purely on interrupt handling basis. So that there is no necessity for threading in order to bring in concurrency into the system. At this level of abstraction, interrupt synchronization is entirely up to the application program if needed.

Optional feature *coordinative* stands for a member of the nucleus family that supports interrupt transparent synchronization [19] of the reactive execution of the tasks. Because being optional, that feature may be included if the interrupt concept (i.e., *interruptive*) was included in the configuration. Interrupt synchronization is considered a “minimal system extension” and introduced as functional enrichment of the “minimal subset of system functions” defined by the interruptive PURE nucleus.

The thread concept comes in two different flavours, as described by the alternative features “uni-threaded” and “multi-threaded”. This feature category allows for a nucleus configuration that supports either a single-threaded or a multi-threaded mode of execution of the application program. That is to say, if the thread-concept feature was selected, there must be a decision for exactly one of its alternate features. Mandatory feature *exclusive* results in a nucleus configuration that leaves processor control entirely up to the application program. The or-features *cooperative*, *non-preemptive*, and *preemptive* describe the properties of the thread-scheduling subsystem of the nucleus. Depending on the application needs, the PURE nucleus may be run in cooperative, non-preemptive, or preemptive mode of operation, or in any other combination except the empty set.

In the OSEK extension of PURE, a *basic task* requires the *exclusive* feature and may rely on feature *interruptive* for background event processing. In contrast, an *extended task* may go with the features *cooperative*, *non-preemptive*, or *preemptive*, depending on the type of events (hardware- and/or software-initiated) that need to be processed. The OSEK conformance classes (BCC1, BCC2, ECC1, and ECC2) correspond to different members of the OSEK branch of the PURE family.

2.3. Family-Based Design

The initial modeling process also laid the basis for designing the functional hierarchy of building blocks that implement the features described by the feature diagram. Figure 2 gives an example of the coarse-grain family-based

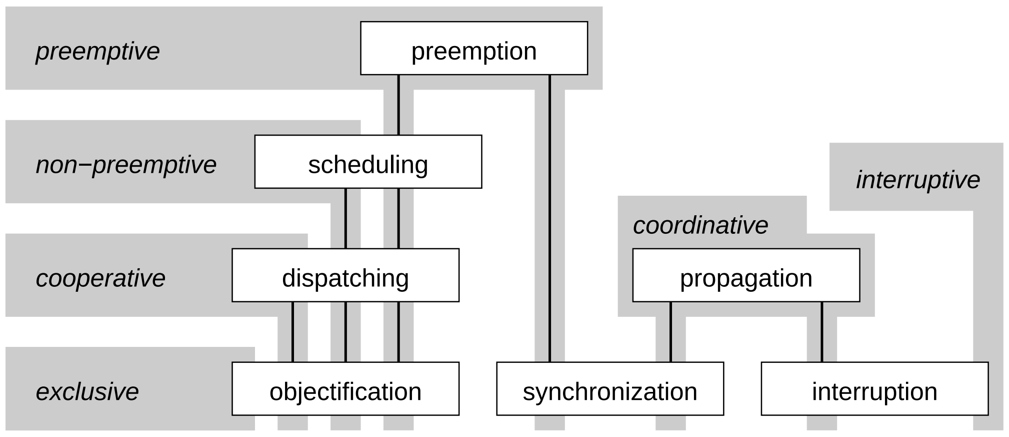

Figure 2. Functional hierarchy (coarse grain) of the PURE nucleus family.

design of the PURE nucleus. This design declares family member *cooperative* to be a “minimal system extension” to the “minimal subset of system functions” provided by family member *exclusive*. Similar holds with the family members *non-preemptive* in relation to *cooperative* and *preemptive* in relation to *non-preemptive*.

Of particular interest in this functional hierarchy is family member *preemptive* as it is the cause of the non-functional property “thread synchronization”. This property represents a PURE cross-cutting concern. Its implementation has been separated from the functional code by means of aspect-oriented programming (AOP) using AspectC++ [20, 2]. The PURE nucleus realizes preemptive scheduling through asynchronous, event-triggered invocations of the fundamental scheduling functions provided by family member *non-preemptive*. There is absolutely no functional difference between the two levels, except that the preemption building block (in contrast to scheduling) executes in a synchronized mode.

2.4. Operating-System Workbench

PURE was turned into an object-oriented implementation using C++ and domain-specific configuration decisions were enforced partly with AOP on the basis of AspectC++. Table 1 shows how the individual PURE nucleus products (for Alpha, ARM, AVR, C167, i860, M6812, M68K, PPC, and x86 processors) scale with respect to the functions they provide to an application program.

Feature modeling of PURE allowed for a compact and precise specification of interdependencies of functional and non-functional properties of fairly complex (hardware and software) configurations. Basing on `pure::variants`, a tool which aids the construction process of a feature model and supports the mapping of features to implementations [3], automated generation of highly specialized PURE systems became possible. Construction kit plus tools establish a workbench for design and development of tailor-made embedded operating systems.

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

Table 1. Memory footprint of the PURE nucleus family (x86 port)

3. Lessons Learned and Future Work

Reaching better separation of concerns in software systems was and is the driving factor for the development of AOP technologies. One of the big hopes associated with the application of AOP is to get a clearly modularized implementation of even “hardly observable” non-functional properties such as *robustness*, *dependability*, or *timeliness*. Even if non-functional properties have no impact on the primary functionality of the software system, they have a big impact on its applicability.

Architecture Transparency A system that provides perfect functionality, but works terribly slow or unpredictable, may be just unusable. Non-functional properties are therefore important concerns, their controllability is crucial in real-time embedded systems. Due to their inherent emergence it is, however, not possible to tackle them directly by decomposition techniques like AOP.

Non-functional properties are strongly influenced by the operating system's architecture. By configuring architectural properties such as synchronization, isolation, and protection it might be possible to fulfill varying requirements on non-functional properties indirectly. On the other hand, lessons learned from applying AOP principles to PURE and eCos [11] indicated that it is very hard, and sometimes impossible, to implement an *ex post* configurability of architectural properties. The reason is that architectural properties often do not only lack characteristic code patterns in the component code (which are addressable by aspects), but also lead to a number of *implicit constraints* that are not visible in the code. The conclusion from this experience is that an (embedded) operating system has to be designed specifically for *architectural transparency* and, thus, being able to make the integration of the implementation of architectural properties a configuration decision that indirectly allows us to perform an application-specific tailoring with respect to non-functional properties [13].

This is where CiAO will continue the PURE approach, namely to apply aspect orientation to the modeling, design, and implementation phase of a feature-based configurable *operating-system product line*. In addition to consequently applying AOP throughout the entire system-software development process, CiAO also aims at further tool support for synthesising real-time systems from so called *atomic basic blocks* (ABBs, [18]). Goal of the ABB approach is to make even the distinction between time-triggered and event-triggered real-time architectures a configuration decision, just as the non-functional property timeliness.

Time- and Event-Triggered Mode of Operation ABB-based software synthesis aims at finishing embedded (application and) system software with respect to the selected real-time operating mode. This step is mainly concerned with reorganization, compilation, and optimization of software packages. The idea of the step before in the configuration process is to interweave aspect programs for time- or event-triggered mode of operation with reusable assets from the CiAO family implementing the functional properties required by a given application scenario. Roughly speaking, the software base to be synthesized first gets adapted with respect to polling mode (for time-triggered operation) or interrupt mode (for event-triggered operation) before being subjected to ABB treatment.

This adaptation affects not only device driver software but also all other software modules attached to the same control and data flow path related to I/O in general. The simple case is, for example, to omit all functional units dealing with synchronization when software for time-triggered mode of operation is assembled using feature-based configuration. In this mode, the clocked control at task-scheduling level implicitly synchronizes all task activities. It is up to an off-line scheduler to create a static schedule free of race conditions between the individual tasks. The difficult case is to add proper hooks to the reused assets in order to take care for *indexed* software functions to be executed in a cyclic manner without being aware of the particular operation principle: in time-triggered mode of operation, polling becomes a cross-cutting concern of the system software.

Similar holds for event-triggered mode of operation, but in the opposite way round i.e. without those hooks and with explicit synchronization statements added to the assets. In addition to synchronization, system software must be functionally enriched by buffering capabilities to compensate for non-clocked control: in event-triggered mode of operation, interruption becomes a cross-cutting concern of the system software.

Validation Providing support for architecture transparency regarding real-time operating modes will be a medium-term goal tracked by the CiAO project. The capabilities of the members of the CiAO family will be validated on the basis of real-time experiments.[2] For comparison purposes, these experiments will be conducted using COTS real-time operating systems of class OSEK on the one hand and homemade members of the CiAO family on the other hand. Centralized and distributed (FlexRay-based) time- and event-triggered control will be considered.

The goal is not only to demonstrate that the CiAO approach leads to highly reusable and yet customizable system-software components but also to prove that CiAO bears comparison with COTS solutions for the embedded systems market. However, emphasis of CiAO is more on the software-engineering side of embedded systems development and less on the operational side. By conducting experiments that reproduce real-world control problems to some extent, CiAO as well as its underlying software-engineering process undergo continuing evaluation with respect to actual questions when building system software for deeply embedded systems. This is to make sure that CiAO does not lose sight of requirements coming up with (deeply) embedded applications.

4. Conclusion

Central theme in the development of PURE was to postpone design and implementation decisions as far as possible. PURE is a family of embedded operating systems.

[2]Such as a "ring-the-bell" game controlled by eight magnetic coils, voltage meters, and photo sensors to manage a metal cylinder vertically through a perspex-lined pipe or remotely controlled model trucks coupled by electronic drawbars.

A couple of members of the PURE family were designed with respect to the very specific demands of deeply embedded systems. The PURE family is made of about 350 C++ classes implemented in over 990 compilation units. PURE runs on nine different processor types ranging from 8- to 64-bit technology.

Feature modeling was used to express the commonalities of and differences amongst the various members of the PURE family. The instantiation of a specific application-aware PURE configuration is controlled by a feature model and supported by a feature-based configuration tool. To a limited extend, cross-cutting concerns of non-functional properties are dealt with by means of AOP (using AspectC++) and automated aspect weaving. CiAO goes a step further and consequently employs AOP in order to maintain non-functional code separate from software components and, thus, improve reusability of the latter. In addition, the ABB concept aims at a smooth transfer between time- and event-triggered CiAO variants.

The PURE development shows that the design and implementation of highly reusable and yet specialized operating-system abstractions or functions must not be a contradiction in terms. Key to success was understanding an operating system as a program family. The outcome was a solution that scales with the demands of many embedded systems. PURE demonstrates that feature-based development of an operating-system family is a very promising approach in order to master the increasing functional complexity of embedded systems in spite of utmost resource scarceness. CiAO extends the PURE approach and focuses on an operating-system product line that can be (semi-) automatically derived from a software construction kit in form of a program family.

References

[1] eCos homepage. `http://ecos.sourceware.org`.

[2] AspectC++ homepage. `http://www.aspectc.org`.

[3] `pure::variants` homepage. `http://www.pure-systems.com`.

[4] *Ingolstädter FlexRay Kolloquium*, Audi AG, Ingolstadt, Mar. 30/31, 2006.

[5] 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.

[6] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Science of Computer Programming*, 53(3):333–352, Dec. 2004.

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

[8] FlexRay Consortium. FlexRay communications system. Protocol Specification Version 2.1, Revision A, `http://www.flexray.com`, Dec. 2005.

[9] 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, Feb. 1997.

[10] J. Koetz. Personal communication. Audi AG, 2006.

[11] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *EuroSys 2006 Conference*, pages 191–204, Leuven, Belgium, Apr. 18–21, 2006. ACM.

[12] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. Functional and non-functional properties in a family of embedded operating systems. In *The Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2005)*, Sedona, USA, Feb. 2–4, 2005. IEEE Computer Society.

[13] D. Lohmann, W. Schröder-Preikschat, and O. Spinczyk. On the configuration of non-functional properties in operating system product lines. In *4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2005)*, pages 19–25, Chicago, IL, USA, Mar. 14, 2005. Northeastern University, Boston.

[14] OSEK/VDX Steering Commitee. OSEK/VDX time-triggered operating system. Specification Version 1.0, `http://www.osek-vdx.org`, July 2001.

[15] OSEK/VDX Steering Commitee. OSEK/VDX system generation—OIL: OSEK implementation language. Specification Version 2.5, `http://www.osek-vdx.org`, July 2004.

[16] OSEK/VDX Steering Commitee. OSEK/VDX operating system. Specification Version 2.2.3, `http://www.osek-vdx.org`, Feb. 2005.

[17] D. L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[18] F. Scheler and W. Schröder-Preikschat. Synthesising real-time systems from atomic basic blocks. In *Work-in-Progress Session of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*, San Jose, CA, USA, Apr. 4–7, 2006. IEEE Computer Society.

[19] F. Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. On interrupt-transparent synchronization in an embedded object-oriented operating system. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000)*, pages 270–277, Newport Beach, California, Mar. 15–17, 2000. IEEE Computer Society.

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

[21] M. Stümpfle. Personal communication. DaimlerChrysler AG, 2003.

[22] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, 1999.