Composition and Construction of Embedded Software Families

Dissertation
zur Erlangung des akademischen Grades

Doktoringenieur (Dr. Ing.)

angenommen durch die
Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von Dipl.-Inform. Danilo Beuche
geb. am 2.12.1968 in Nauen

Gutachter:

Prof. Dr. Wolfgang Schröder-Preikschat
Prof. Dr. Reiner Dumke
Prof. Dr. Jörg Nolte

Promotionskolloquium:

Magdeburg, den 19. Dezember 2003

Acknowledgements

Finishing this dissertation would have been impossible without the help of many persons. I would like to express my gratitude to all of them. However, it is impossible to name all of my voluntary and sometimes involuntary helpers.

My special thanks go to Wolfgang Schröder-Preikschat who gave me opportunity and room to develop my own ideas. His research group at Otto-von-Guericke-Universität Magdeburg was the perfect place for me and my research project. Beside my advisor I have to thank my colleagues Guido Domnick, Andreas Gal, Jens Lauterbach, André Maaß, Daniel Mahrenholz, and Michael Schulze from this group. They frequently had to use the Consul program which is a substantial result of this thesis. I am grateful that they kept sending bug reports instead of deleting it from the disk right away when it still did not work as expected after the "final" bug fix. Olaf and Ute Spinczyk, who were my first colleagues in the group, had not only to live with my chaos for about five years, they also were first time readers of my drafts. Their comments and also their believe in this work were invaluable. Frank Behrens, who always tried to motivate me, is hopefully pleased with the result of his efforts.

The students who helped me implementing my ideas, Bianca Rädiger, André Herms, and Sascha Römke, did a wonderful job. I hope they will benefit from their experiences as much as I did.

Many other persons from the group and from other places have to be named. I had the opportunity to stay twice at the University of Santa Catarina, Brasil. My host there, Prof. Antonio Augusto Fröhlich provided the appropriate environment to write large parts of the dissertation there. Jürgen Lehmann always provided me with disk space, IP addresses and any other technical equipment I required. Martina Engelke, who was so unfortunate to share the office with me for more than five years, almost always ignored the high entropy of things on on my desk. I never expressed it, but I really appreciated this!

The people at pure-systems with Holger Papajewski as their head deserve my thanks since they bring my visions and ideas into real life. I am very fortunate to see this happen!

The two reviewers, Reiner Dumke and Jörg Nolte, were so fast that I could finish the last part of the dissertation in 2003. They allowed me to provide myself a very nice Christmas present. Thanks for that!

Last but not least I have to thank Birgit Schelm. She not only beared my sometimes strange behavior throughout the complete the dissertation project but also read and proof-read the manuscript at least twice and was always faster than me fixing the problems. I would not have finished this work without her.

Contents

List of Figures

List of Tables

1 Introduction

Only recently, most of the software industry and computer science community recognized the rapidly increasing importance of embedded computer systems. Up to then, embedded systems were ignored by most software engineers. Software for embedded systems was programmed mainly by electrical or mechanical engineers. One reason was that most embedded devices realized only a small set of functions that required relatively simple programs. But today the number of applications for embedded systems seems to be endless because of the availability of a wide range of embedded processors and peripheral components for almost all thinkable purposes. Embedded systems no longer only measure some data and transmit it to a gauge read by humans or turn on a simple switch. They are often highly heterogeneous distributed systems. Modern cars, for example, contain up to a hundred or more microcontrollers coupled via real-time networks like the CAN bus.[1]

It is now widely recognized that the increasingly complex design for embedded system requires advanced engineering techniques in order to deal with the challenges of today's and tomorrow's embedded systems. The important question to answer is whether and how embedded software differs from software for other systems, for example for business applications or for massive parallel applications. The following work tries to answer this question and also presents techniques that facilitate more efficient software development for embedded systems.

The problems that designers of embedded software face are manifold. In addition to the problems common to any software development, embedded software developers have to solve a number of specific problems. In many cases the software has to provide a much higher degree of safety and reliability than non-embedded software. Often the software has to provide real-time guarantees.

Despite the non-technical nature of product prices it is important to note that the cost of an embedded solution has a very strong, most likely the strongest influence on which embedded platform is used. Because many embedded systems are produced in high volumes, there is usually a strong interest in using hardware that is as cheap as possible. An indicator of this fact is that the best selling processors (in numbers) are neither 32 bit nor 16 bit but 8 bit processors. In 2000 more than 60% of all processors and microcontrollers sold had 8 bit CPU cores [Ten00]. Most of those cheap microcontrollers have but little ROM and RAM

[1] BMW 7 up to 100, Mercedes S class around 80 according to company advertisements.

available and provide only limited processing power. The technically visible implication of this phenomenon is that achieving small code size and high run-time efficiency of embedded software is a major concern for software developers.

Due to the lack of adequate software techniques and programming languages as well as the simplicity of most problems that were to be solved in the beginning of embedded systems software development, many projects were implemented using assembly language and simple imperative languages like C. Using assembly language allows, in theory, the writing of optimal efficient code. It is still a common belief among embedded software developers that only the (at least partial) use of assembly code allows to achieve the performance required for their specific applications. Over time and with the availability of more powerful embedded processors, the use of C has been increased but other, more powerful programming concepts, like object orientation, are still not yet widely used in embedded systems programming.

On the other hand, the complexity and the range of possible applications for embedded systems has grown so much that other issues require more attention. In order to reduce time-to-market, software reuse is the key factor today. Reuse comes in different flavors: It may be reuse of parts of the previously written software in a new project, it may be reuse of the same software on a new hardware platform, or it may be a combination of both. Techniques to enable software reuse are already available on the market. Although none of them is perfect and reuse is still a very hot topic for software engineering research, many successful projects have proven the benefits of reusable software.

However, there is a relatively strong resistance of the embedded software community against most advanced software engineering techniques for reuse. The problem is that most techniques that promote reuse are coupled with a more or less noticeable performance degradation. Abstractions and implementations that allow reuse often cause run-time and/or code-size overhead when compared to a specific, non-reusable implementation for a single application. The reason for the difference in performance is quite simple: reusable implementations often contain functionalities that are unnecessary and that cause unwanted overhead.

The C function `printf(const char* s,...)` is a good example to illustrate this point. Each `printf` implementation contains a simple parser that parses the string `s` for special symbols starting with `%`. The special symbols are replaced by the string versions of the remaining parameters. Usually, `printf` supports a wide range of possible conversions including characters, strings, integers, floating point numbers as well as a number of additional formatting options. For an application that prints nothing but integers, most of the functionality of `printf` is useless but causes both more code and run time than a specialized integer output function.

But without reuse, the efficient handling of a number of applications that are closely related to each other is impossible. An example could be the development of a line of car radios by one manufacturer that are based on a common hard- and software platform but differ in features as well as in the design of the front panels with their knobs, switches and displays. They have most parts in common, but there are important differences between the models. The objective is that no piece of software realizing the same function should be written twice for different models. Such sets of applications constitute *application domains*. The question is how to create software for a whole domain instead of for a single application while retaining the required efficiency.

Some answers to this question are already known. One of the first answers was proposed by Parnas back in the late seventies with his concept of *program families* [Par79]. Although the reuse problem was identified so long ago, and even a solution was proposed, program families did not get the attention they deserve until the nineties.

In the nineties, there was an important change in the view of software development. Instead of focusing only on one application during software development, some researchers believed that it pays off to look at a larger set of applications from the same application area. The main difference to the seventies was that, in the nineties, adequate programming concepts like *object orientation* were available and widely used. The advent of object-oriented programming allowed the implementation of program families based on classes and objects instead of based on procedures and modules. A good example for the use of program families is the PEACE operating system family for massively parallel systems developed at GMD FIRST [SP94] .

The research on program families and similar problems went into different directions focusing on different parts of the problem. One direction was the development of adequate languages to support family-oriented programming. One of the first language-based approaches was Neighbors' Draco [Nei84] in the early eighties. His (and many others') ideas are now known as *domain specific languages* (DSL). These languages try to simplify the problem of development (and reuse) by deploying languages that are developed for a specific (set of) domain(s) rather than for a general-purpose use like C/C++, Java or Eiffel.

Another important direction was and is the design of development processes for family-based software. Several processes like ODM [SCK$^+$96] or FAST [WL99] have been proposed for efficient development of software families.

One of the most important aspects of family-based software development is the analysis modeling of the application domain. Probably the most pioneering work in this area was FODA [KCH$^+$90], an easy to use modeling approach for domains that has been refined in many subsequent projects and is widely accepted as the most important domain modeling approach. This work set off the phase of more intensive work on family-based software development in the nineties.

For specific kinds of domains, several approaches evolved during the nineties, like Don Batory's GENVOCA [BO92] for layered software families or the DEMRAL method of Czarnecki [Cza98], which is a specialization of ODM for the development of efficient algorithmic libraries.

Many other software techniques had an important influence on software family development, like generic programming techniques and component-based software development.

However, as was stated above, software family-based development is not yet a mainstream technique due to many reasons. Some of those reasons are discussed in this work along with possible solutions.

1.1 Motivation and Goals

The insight that family-based software development in the large is not feasible without a coherent tool chain for all phases of the development process was the starting point for the work presented in this dissertation. Projects where family-based software was developed without adequate tool support could not exploit the full benefits of that approach. The main reason was that with the growth of the family it became more and more complex to select and deploy the appropriate family member.

A solution for this problem has to provide support in the areas of configuration and composition of software families. However, configuration and composition requires configurable and composable entities. The design of these entities is the result of the domain decomposition process that starts each family-based software development. Therefore, tool support should start as early as in the domain analysis phase right at the beginning of software development.

The motivation to focus this work especially on embedded systems and particularly on deeply embedded run-time systems was driven by the fact that though the idea of family-based software seems to fit well into these domains, it has not yet been widely deployed due to various reasons. One reason is the lack of tools as described above. Another reason is the prevailing demand for software that provides maximal performance at minimal cost. The cost of a product is strongly influenced by the development costs required (time and manpower) and/or by the ability to use cheap hardware with limited computing power and small memory. Techniques to solve each of these problems already existed but the glue that allows the combination of these techniques into a uniform process was missing.

To provide an environment that allows the use of many different techniques for developing and deploying family-based software was the main goal of this work.

The concentration on embedded run-time systems instead of embedded applications is motivated by the fact that the run-time system for embedded applications should obey the same principles as the whole application. Maximal performance and/or minimal cost can rarely be achieved when a family-based solution for an application domain is developed on top of a non family-based run-time system. The use of a non family-based run-time system can drastically reduce the benefits of the family-based approach. Since the applications might use only limited parts of the functionalities provided by the run-time system, there exists unused code that is never executed or, even worse, code that is executed but does not provide any service required. Another reason for the focus on embedded run-time systems is that the software complexity of run-time systems is fairly high, so the results should be transferable to other complex software systems easily.

This dissertation project started as part of the work for the project “Workbench for tailor-made operating systems” (WABE), funded by the Deutsche Forschungs Gemeinschaft (DFG) at the Otto-von-Guericke-Universität Magdeburg in 1998. The idea of WABE was to develop a solution including a tool chain for customizing operating systems according to the needs of an application. The example application domains in this project were deeply embedded applications. The application of domain engineering technology proved to be quite useful when supported by tools.

Some of the ideas where then developed further in the ITEA Project “Development process for Embedded Software Systems” (DESS). DESS was a joint project of research institutions and industrial partners from several European countries. The aim was to create a generalized methodology for developing embedded software systems based on component technology. The special requirements of deeply embedded system software if a component-based approach is used, are not met by standard component systems. The extension of tools developed in WABE proved the applicability of the DESS approach even for small deeply embedded systems.

1.2 Contributions

The main contribution presented in this work is a rule-based configuration method for component-based software families. It significantly eases the problems sketched above, especially for projects that require very fine-grained software configurability. It does so by gathering and representing configuration knowledge throughout the software development process in a uniform way. The method is based on already known software engineering techniques like domain engineering and program families but combines them in a novel way. It does not require a new software development process but can be seen rather as an addition to existing approaches for software development process models.

The introduction of domain mappings from the family application domain to component configuration domains allows a new degree of reuse for family-based software components. Instead of components that are bound to a particular application domain model of a software family, it is then possible to reuse components in other application domains. This is achieved by providing an appropriate domain mapping of the application domain to the components configuration domain.

An important point with respect to practical use of the method is that the method and its supporting tools are not bound to any particular way of implementing software families. This ensures usability in many different contexts. However, to gain maximal benefit, the method and tools are best used in combination with object-oriented programming languages. A strong focus during the development of the method and the tools was to provide a solution that is extensible and adaptable.

The description of several examples where the proposed method has been applied under different prerequisites shows its usefulness and applicability to projects in the intended application domain of embedded systems.

The re-engineering of an existing implementation, the Portable Universal Run-time Executive (PURE), shows how well the representation of application domains with feature model concepts fits into an already existing family design.

The other examples, on the other hand, show how easy it is to derive efficient implementations from the results of the feature domain analysis.

1.3 Outline

Chapter 2 characterizes the domain of embedded systems and software for embedded systems. For that, an overview about the possible application areas of embedded systems and their specific software requirements is given. A special focus presents the general abstractions an embedded application shares with other applications from the same application area. Based on this knowledge, the characteristics for embedded run-time support systems for these application areas are discussed. The results of the discussion establish the base for a comparison of available embedded run-time systems.

Chapter 3 presents a discussion of family-based software. Starting with the ideas of Parnas, different approaches to family-based software development are presented and discussed. The main focus is placed on the usability of the development methods from a practitioners point of view. For each of the presented approaches, the applicability in embedded software development is examined.

Chapter 4 concentrates on implementation techniques for family-based software like program generators, static meta-programming and special programming languages.

Chapter 5 gives an overview over the developed method and tools. It proposes a generic model for family-based software development processes. The chapter contains a description of the steps required to implement the generic family-based software development process. It covers all phases from the analysis to the deployment of the developed software.

Chapter 6 provides two different case studies of software families for embedded run-time systems. One system (PURE) is a complete run-time system for deeply embedded devices.

Its development started without tools supporting family-based software development, and it has been re-engineered later to fit into the development process described in Chapter 4. The other case study covers a thread abstraction library that has been completely developed using the process described in Chapter 4.

Chapter 7 contains a conclusion and discusses issues for future works in the area of family-based software for embedded systems.

2 Embedded Software

This chapter discusses the following questions:

- What are the differences between software for embedded systems and software for non-embedded systems?
- What are the requirements embedded software development methods must meet?

At first, the term *embedded system* is defined. Based on selection criteria for embedded systems, a classification for embedded systems is presented. Starting from this consideration, several non-embedded software technologies, concepts and deployment areas are analyzed. The goal is to point out the commonalities and differences to embedded systems software technologies.

The last part emphasizes on the special needs of embedded systems for reuse techniques that provide a high degree of efficiency of the development process as well as of the produced software.

2.1 Embedded Systems

Although embedded systems are present almost everywhere, it is not easy to give a precise definition of what an embedded system is. A single small 4 bit microprocessor coupled with a temperature sensor and regulation valve is an embedded system just like a CNC machine tool equipped with several 32 bit processors.

An embedded system

- is specifically designed to provide a given, restricted set of functionalities,
- presents hardware and software as a unit.

It is important to note that an embedded system is not defined by the technology used. Processor speed or memory size have no influence on classifying a system as embedded or not. In fact, any special-purpose system using computers is an embedded system.

However, while almost always the fastest available technologies are used for workstations and personal computers, the criteria for choosing embedded systems technologies (for hardware as well as for software) are quite different. Some of the main issues are:

Power consumption

The available energy for an embedded system is often restricted, since it is battery-powered or the heat dissipation of the system must not exceed a given value.

Timing

Many, but not all embedded systems have to meet hard real-time or soft real-time constraints. The timing is strongly related to the available processing power, that is the number of instructions a processor can execute in a given amount of time. Besides the pure execution speed, the predictability of time related issues like execution time calculation and completely deterministic behavior of the hardware are important.

Memory space

The memory space consumed by the software must fit into the available memory space. It is usually impossible to upgrade the available memory once the systems has been produced or shipped.

Cost

Last but not least, cost is one of the most important factors for embedded systems, since most embedded systems are deployed in areas where lower cost is a key success factor on the market. Therefore, the companies aim to use the cheapest hard-/software combination possible. The cost factor has an influence on all the technical factors mentioned above, because more memory or processing power usually results in higher costs and often higher power consumption as well.

Using the criteria described above, different classifications of embedded systems are possible. One of the most common categorizations, however, depends on the bit width of the microcontroller used (4 bit, 8 bit up to 64 bit). Although the bit width provides only a rough measure for the available resources, it is generally safe to assume that available memory space and processing power increase with higher bit width. This assumption is mainly based on the fact that processor cores with higher bit widths have been developed after cores with lower bit width. On the other hand, some newly developed 8 bit controllers like the Atmel AVR90Sxxxx family are much faster than some older 16 bit processors at the same clock speed.

To provide an overview over the embedded systems market, the distribution of processors produced in 2000 according to their bit width is quite helpful. The figures in Table 2.1 show that the vast majority of processors sold are still 8 bit processors (58%).

Units	**Shipments** ($\times 10^6$)					
	Technology				*subtotal*	%
	4-Bit	*8-Bit*	*16-Bit*	*32-Bit*		
μ-controllers	1680.0	4770.0	764.0	43.0	**7250.0**	**87.6**
embedded μ-processors	–	20.2	108.0	153.1	**281.3**	**3.4**
digital signal processors					**600.0**	**7.2**
μ-processors					**150.0**	**1.8**
total	20.2%	57.7%	10.5%	23.6%	8288.3	**100**

Table 2.1: Estimated production of microprocessors and -controllers in 2000 [Ten00]

Due to limited hardware capabilities of most 8 bit microcontroller units (MCU), they are mostly used in so called *deeply embedded systems*. A *deeply embedded system* performs only a very limited set of functions, often just a single, very simple function. The software of these systems uses only small amounts of memory (few bytes of RAM, few kBytes of ROM) and the available processing power is low in relation to the required calculations.

Considering the figures for 8 bit processors in Table 2.1, it becomes clear that most of today's embedded systems are *deeply embedded systems*. However, according to the definition of deeply embedded systems given above, it is very well possible to build a deeply embedded system using a 32 bit microcontroller.

Typical uses of such deeply embedded systems are simple command and control applications, for example turning on a heater at designated times if the temperature drops below a threshold value or measuring the rotation speed of a car wheel and calculating its velocity.

Deeply embedded systems can be found everywhere, one of the most popular examples are washing machines. But nowadays, almost all electrical household appliances from vacuum cleaners over microwave ovens to television sets are controlled by deeply embedded systems. Another important application area are automobiles. In modern cars, dozens of rather small microcontrollers are used to control safety critical functions like fuel injection, ABS[1] or ESP[2], and comfort functions like air conditioning.

On the other end are *high-end embedded systems* that use high-speed processors and have plenty of memory available. These embedded systems are usually derived from a workstation- or PC- like architecture. Such systems are typically used in factory automatization or for controlling, for example, chemical processes.

Many embedded systems are somewhere in between deeply embedded systems and high-end embedded systems. Although they use faster processors and more memory than a

[1] Anti blocking Brake System

[2] Electronic Stability Program

personal computer used to have but a few years ago, they are resource-restricted in that they have to meet energy-consumption constraints, because they are battery-powered (e.g. MP3-Players), or that they have to be very fast due to the amount of information they have to process (e.g. network switch controllers) for example.

The term *embedded systems* is obviously used to characterize a rather inhomogeneous set of systems that have to meet a huge variety of requirements. But common to almost all embedded systems is that the hardware sets hard limits for the software concerning resource usage like memory, energy consumption and/or processing power.

Today's software products for personal computers, on the contrary, do not have to fight against resource problems, only a few applications like games or video processing still push a modern PC to its limits. Memory space is no longer an issue, modern processors spend most of their time in an idle state, waiting for user input.

2.2 Related Software Technology Areas

The broad range of applications for embedded systems requires many different kinds of software. But common to almost all embedded systems software is the necessity to obey some given resource limit, for example memory usage, run time or energy consumption of the processor.

None of the restrictions given above are of real importance for office software or most other applications found on workstations or personal computers. For a long time, software engineering mainly focused on those systems. A large-scale development of software technologies for embedded systems did not really exist. Straightforward application of software engineering techniques for those systems to the embedded world often failed for various reasons. The Java technology gives a nice illustration of those reasons. In early statements on Java and its intended application field, Gosling [Gos95] claimed that Java would be suitable for any device from toasters to personal digital assistants. However, the concept of an interpreted byte-code language like Java needs a significantly more powerful processor than was usually found in most embedded systems at that time. Even today, many embedded systems are not powerful enough to use Java as its main programming and execution language.

In order to find suitable software solutions for embedded systems, it is necessary to determine the difference between embedded software (development) and "normal" software (development). This section examines software technologies currently in use in non-embedded contexts. For these technologies it is discussed whether they are applicable to embedded software and for what reason.

The motivation for the selection of technologies was to cover for important topics like currently evolving technologies (component-based systems), common design and implementation techniques (object orientation), technologies dealing with complex systems (distributed systems), with performance issues and also with reuse (operating systems). The selection is not complete but should make the differences and commonalities between embedded and non-embedded software technology requirements visible.

2.2.1 Other Application Domains

2.2.1.1 High-Performance Computing

The closest relation of embedded systems regarding execution speed exists to high-performance, massive parallel programs where the application speed is a critical issue to achieve meaningful results. A weather forecast needs to be in time to be useful, just like an electronic brake assistant.

Often such programs are highly dependent on high speed, low latency communication and low overhead mechanisms that are also of great interest for many embedded systems designs. High performance parallel computing, however, is more focused on the pure start-to-end run time of the application and less on the deterministic timing of the individual communication.

Of special interest are run-time systems for massive parallel systems like PEACE [SP94] or Puma [WMR$^+$94] that focus on producing almost no unnecessary overhead so as to spare as many resources as possible for use by the application.

PEACE is not a single operating system but rather a family of different operating systems that represent different levels of functionality. Depending on the needs of the application, different communication modes or application programming interfaces can be selected. To achieve this high degree of flexibility, an incremental design and implementation approach based on program families has been used.

The Puma operating system uses zero copy communication buffers to achieve a high throughput. Zero copy buffers prevent additional copying of messages, thus saving time and also extra kernel memory space for buffering.

The common aspects of embedded and high-performance software are the specialization on a specific task and the focus on resource efficiency with respect to processing power. The main difference is that the resource “memory” is much more restricted in embedded systems. While parallel systems often have several GBytes of RAM, most embedded systems have several orders of magnitude less memory available.

2.2.1.2 Distributed Systems

Another strong relation can be drawn to the area of distributed software since most of today’s embedded systems are in fact distributed systems consisting of a number of, often heterogeneous, processors connected via one or more networks. Modern middle class cars have some dozens of microcontrollers of various sizes that control engine, brakes, power-steering, ABS and comfort functions.

An interesting question is whether it it possible to reuse the existing software (concepts) for distributed systems in embedded systems.

The short answers to this question is yes, for a limited range of applications. The explanation is quite simple: Existing approaches like CORBA [Obj00], DCOM [Box98] or Java RMI [WRW96] are simply too heavyweight to run on a small 8 bit MCU. But 8 bit MCUs still account for the majority of processors found in embedded systems. Only a minority of applications where desktop like resources are available can benefit from these technologies.

Another disadvantage is that most of these technologies are not real-time capable. Although a real-time extension has been specified for CORBA [Obj99], neither DCOM nor Java RMI provide such extensions yet. This further limits the applicability of these technologies in embedded domains.

Even software following the MinimumCORBA standard [Obj02] that specifically aims at embedded systems is not suitable for use in deeply embedded systems (see Table 2.2). The interesting point is, however, what the OMG left out of the original CORBA specifications:

- Dynamic Skeleton Interface
- Dynamic Invocation Interface
- Dynamic Any
- Interceptors
- Interface Repository
- Advanced POA features
- CORBA/COM interworking

They left out almost any kind of support for dynamic changes to the system to achieve a smaller footprint. Table 2.2 gives an impression of the memory usage of MinimumCORBA implementations.

On the other hand, there are a number of research distributed systems that are aimed specifically at mobile embedded systems like DACIA [LP01] or Spectra [FNS01]. Both address real-time communication and/or energy efficiency as key issues for mobile systems. However, the mobile target systems are again relatively powerful (both use 32 bit systems) because they are intended to be used together with multimedia applications like video on demand and speech recognition.

Name	Vendor/URL	Footprint
minimumTAO	www.cs.wustl.edu/~schmidt/ACE_wrappers/ docs/minimumTAO.html	1.3 MByte
Varadhi 1.1	Sankhya www.sankhya.com/info/products/varadhi	107 kByte
K-ORB	Trinity College Dublin www.dsg.cs.tcd.ie/research/minCORBA	planned 50 kByte

Table 2.2: Memory footprints for MinimumCORBA implementations

2.2.2 Software Development Technologies

2.2.2.1 Component-Based Systems

Component-based systems encapsulate functionalities in closed entities called components and allow access only via defined interfaces and protocols. A component is “an independent unit of deployment” [Szy99], which means that it can be used in a different context as long as the defined interfaces and protocols are obeyed. Two different kinds of components can be distinguished: *source components* are combined with other source components and then compiled into the final program. *Binary components*, however, are components that are executable immediately.

To enable cooperation between components, an agreement between the communicating components has to be made about the interfaces, the way of accessing the component’s functions via the interfaces and so on. To ensure this interoperability most component approaches deploy a so called component framework that serves as infrastructure for the components.

Most commercially available component infrastructures are not well suited for the needs of embedded systems. They are focused on binary components and usually include code for almost all possible situations and needs of those components. The lack of functional scalability within frameworks according to the real needs of the components used results in code unnecessary for many applications that has to be loaded or even executed nonetheless. As memory is one of the most precious resources for embedded systems, this is rarely tolerable. Examples for such component systems are JavaBeans [Feg97] and Enterprise JavaBeans (EJB) [Suna] from Sun, DCE [Fou95] from OSF and COM/DCOM [Rog97] from Microsoft.

Microsoft claims that the new .NET environment is available for embedded systems, but it is only usable with Windows CE in fact. Windows CE supports only 32 bit systems, equipped with several MBytes of ROM and RAM and top speed embedded processors like the StrongARM from Intel or the Hitachi SH4. As discussed earlier, this does not represent the majority of embedded systems and thus is not a general opportunity for embedded software development.

Source component frameworks are not as widespread as binary component systems, because they require the exchange of proprietary source code that is often treated as a business secret between different organizations. However, within a single organization, source component frameworks can be and are used without these obstacles. Source-based component frameworks can be tailored more easily to the needs of the applications if it is possible to analyze at, or before compile time which parts of the framework are actually needed.

The *ADAPTIVE Communication Environment* (ACE) [SH01] provides basic functions for a distributed C++ component framework. According to the authors, the source code approach helped to improve the performance compared to RMI or CORBA.

Another advantage of source component frameworks occurs where real-time requirements are concerned. To ensure that an embedded system meets its real-time requirements, source code analyzing tools are frequently used. They inspect the complete software system with respect to timing aspects, for example critical execution paths or unbounded loops. In binary component systems, such an analysis is not possible, so the users have to rely on timing descriptions provided by the component producer. Since many factors like memory access times, processor speed or interrupts influence the timing, it is hard to provide accurate timing information without having access to the target hardware platform.

2.2.2.2 Object Orientation

Object orientation in its different variations like Smalltalk, Java, C++ or C# is today's most commonly used implementation paradigm[3]. However, OO itself does not ease the problem of meeting the embedded system requirements just by designing and implementing using OO. If not used carefully, it is easy to produce a functionally correct and probably nicely designed software that cannot be used on the intended embedded platform however. The following paragraphs discuss some of these issues in more detail.

Portability of software is not per se guaranteed if object-oriented methods are used. It is easy to break portability of software using almost any programming language. 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, for example consists of only a single line of code, but are called many times and from many places. If inlining of such methods is not 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.

[3] Herb Sutter, a Microsoft employee, said in an interview [Mau], that 4.5 to 5 million out of 9 million programmers world wide use either C++ or Java.

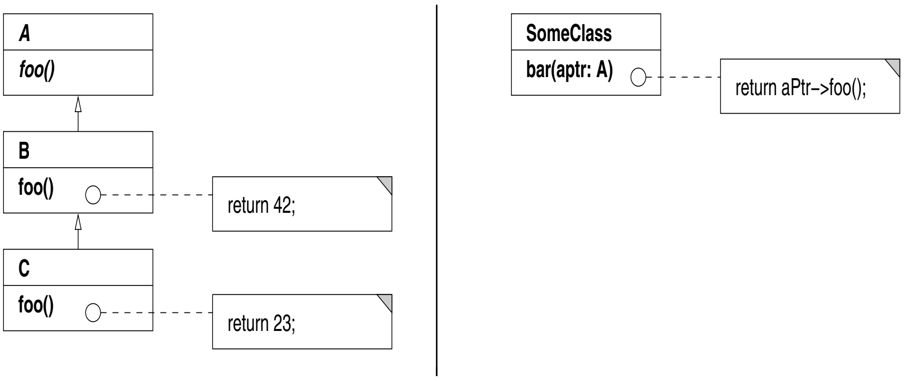


Figure 2.1: A problematic class hierarchy

Scalability is one of the most important aspects in the domain of embedded systems. The architecture of the software systems must allow down-scaling of the functionalities. Unused code that remains in the final system is not acceptable. 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, it often is not possible to detect whether an overridden method is used or not. In the example shown in Figure 2.1, it is not possible to decide whether pointer `aPtr` will ever refer to an instance of `B`. So code for method `B::foo` must always be included if an instance of a class that was derived from `B` is used. Furthermore, if exactly one specialization class is used, the run-time overhead resulting from virtual method resolution is wasted, since it will always result in the same actual method call (e.g. `C::foo`).

In fully static scenarios where every instantiated class is known at compile time, it would be possible to avoid unused code, but mostly such knowledge cannot be gathered easily. This is especially critical if any kind of dynamic loading of code is available. In this case every fragment that could possibly be loaded has to be analyzed a priori. Dynamism on the 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 [Sunb] 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 resource constraints less strong, a dynamic architecture has many benefits like loading and unloading of components on demand.

Extensibility is another important issue 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 to such changes. 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 result. The reasons for this are that the previously existing behavior may change and break other parts of the system that relied on it. A way out of this dilemma are contracts between the abstractions. Beugnard et al. [BJPW99] discuss different kinds of contracts and their usage. Another approach is to use immutable abstractions that 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 [GHJV95]. The use of the *strategy pattern* for selecting the scheduling algorithm in an operating system, for instance, allows the introduction of a new scheduling algorithm without changing any other part of the system. Unfortunately, the use of this pattern often conflicts with the efficiency requirement. Even if only a single strategy is used, the pattern code requires dynamic invocation of an implementation of that strategy. This leads both to the consumption of more processor cycles for resolving the dynamic invocation and to unnecessary code and data (dynamic dispatch code and virtual function tables) compared to a solution where the strategy would be called directly. Similar problems occur with many other patterns. The reason for this is that a single pattern may have many different implementations suited for a specific set of requirements. Often the most general "standard" implementation is used that makes only few assumptions about the surrounding system and thus contains a lot of code to handle all possible situations, most of which never occur in a specific system.

Composability has two main aspects. The first aspect is to provide an architecture that allows integration of components from different sources, for example drivers for specific hardware or communication protocols. As discussed earlier in this chapter, the currently available solutions like CORBA, COM, ComponentPascal [Szy99] or JavaBeans use binary components. All these component models are too heavyweight 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 needs 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, since unnecessary code can be left out if the configuration of the execution context is known at compile time.

But a component model that is based on static compile time component composition alone is not suitable if dynamically changing systems are to be supported. While a fully static approach fits best for small deeply embedded systems, dynamic composition of varying degree can be useful for more powerful systems. Thus, an architecture that supports both static and dynamic composition of components is favored. 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. In real-time embedded systems, for instance, the behavior of a component regarding time is of particular importance. The chosen combination of components must ensure for example timeliness and also deterministic behavior. Again, as in the case of extensibility, contracts and other formal methods play an important role here.

For all general requirements mentioned above, separate solutions exist in the object-oriented world. But if those solutions are used in combination, the efficiency requirement is often violated.

2.3 Software Reuse in Embedded Systems

2.3.1 The Two Dimensions of Reuse

The term *reuse* has two different dimensions: a temporal dimension and a functional dimension. Though both kinds of reuse occur often in combination, it is important to separate them clearly.

Functional reuse means the reuse of an unmodified piece of software in a predefined context. The reuse is based on the functionality provided by that piece of software. Examples are libraries that provide a defined set of functions for many different applications. The context of reuse is defined at the production time of the software.

This kind of reuse relies on an adequate prediction of the functionalities that are reusable and especially of the combination of functionalities that are to be used by the software deployers.

Functional reuse is required when a number of products need the same functionality. This can happen at all levels, in upper application layers or at the lower layers, the run-time system or supporting libraries.

A product line of software for an ABS for car brakes, for example, shares most of the functions and these functions are reused in all systems produced at a given time (frame). But as the systems are produced for different car manufacturers and used in different cars, some parts like the communication functions cannot be reused, because they are defined differently by each car manufacturer[4].

In layered software architectures where the lower layers provide basic functions and upper layers use these functions to provide more complex functions, functional reuse at the lower

[4] Standardization in embedded systems, especially in the car industry, is far from reaching the levels of other IT domains.

layers is common. The main reason for this is that basic functions are more often reusable in a wide range of systems. There are for example a number of embedded operating systems and run-time libraries that can be reused by embedded system developers.

Temporal reuse, on the other hand, is the reuse of a piece of software within a different context. Some properties of this new context have not been anticipated when the software was initially produced. Thus changes to parts of the software may be required in order to use it in the new context. This reuse dimension is often described as *software evolution*.

Temporal reuse is very often required at the application level. When a successor product is developed, for example, as much software as possible should be reused. But usually new functional or non-functional requirements need either additional components and/or changes to existing components that may or may not have been anticipated during the original development of the components.

Dynamically changeable systems often rely on temporal reuse. In systems with high availability requirements, like telecommunication equipment, it is often impossible to stop a system and replace the software altogether. Therefore, it has to be changed during normal operation without halting the system. That is even more demanding as it requires new and old software to cooperate very tightly.

Those changes raise many issues that range from backward compatibility, if the new components should be usable within the old software environment, to the question as to when a new software development is cheaper than a reuse. Some research projects work on this subject, like the ITEA EMPRESS project [EM] or the AIT WOODES project [WO] that both focus on improved embedded software development processes.

Both kinds of reuse occur in embedded systems and both kinds of reuse are equally important, but the rather static functional reuse is the base of temporal reuse, which captures the dynamic aspects of reuse. So the problem of functional reuse has to be solved first, although in a way that permits for temporal reuse later.

2.3.2 Reuse Problems in Embedded Systems

The probably most important problem when developing reuse concepts for the embedded domain is quite simple: if an optimal solution for a problem under a given set of constraints is available, it is not necessarily the optimal solution for the same problem under a different set of constraints.

To illustrate this problem three different applications of the cosine function are introduced:

Application 1 A high precision value is required, real-time execution is not required but the available memory to store constant data is limited.

```
const double DEG2RAD = 0.01745329251994  /* (PI/180) */

double  cosine(const int degree)
{
  const double rad = (double)degree * DEG2RAD;
  double res_last, sign = fac_value = power = res = 1.0;
  double faculty = 0.0;
  double square = rad * rad;

  do
  {
    res_last = res;
    sign=(sign==1.0)?-1.0:1.0;
    fac_value *= ++faculty;
    fac_value *= ++faculty;
    power *= square;
    res = res_last + sign * (power/fac_value);
  } while (res != res_last);
  return res;
}
```

Figure 2.2: Source code for iterative cosine calculation

Application 2 A high precision of the cosine value is required as well, the angle might be any value but the calculation has to be finished fast and within a deterministic time frame.

Application 3 A sensor measures the angle only in 16 discrete values, the application has tight real-time requirements and very limited code space available.

While it is easy to provide a common cosine implementation for all three applications using the standard iterative algorithm shown in Figure 2.2 that returns correct results for every input value, this algorithm is not able to meet the additional constraints of applications 2 and 3. Its timing is hard to predict and it requires a large amount of code for its floating point operations.[5]

A different solution (see Figure 2.3) that provides deterministic run times is based on a table of known cosine values and interpolation to calculate the result for arbitrary values. The trade-off here is that, depending on the number of known values, the accuracy of the result differs. Using more values consumes more data memory to store the table.

[5] It is assumed that the processor does not have a floating point unit.

```
#include "cosine.h"
#define POINTS 24
double cosine_table[POINTS+1] = {
   1.0, 0.965925, 0.866025,
   0.707106, 0.5, 0.25881, 0.0};
// remaining table values omitted

const double pointdistance = (360.0 / (double)POINTS);

double  cosine(const int degree)
{
   double div_degree =  ((double)degree / pointdistance);
   double p1 = cosine_table[(int)div_degree];
   double diffdegree = div_degree - (int)div_degree;
   double p2 = cosine_table[(int)(diff) + 1];
   return p1 + (p2 - p1)*div_degree;
}
```

Figure 2.3: Source code for cosine calculation using interpolation

While this implementation is appropriate for many applications, for some an even more simplistic solution is possible. Because only a limited number of discrete angle values with equal distances are possible, it is easy to implement a purely table based cosine function (see Figure 2.4). No calculation is required, no floating point operation occurs at all.

The code sizes for the different implementations vary to a significant degree. Table 2.3 shows code and data space requirements for a number of different platforms ranging from 8 bit controllers to 32 bit processors. The application consists of a single call to the cosine function in `main`. The `void` application is just an empty `main` function included for comparison.

```
#define INTERVAL 15
double cosine_table[24] = { 1.0, 0.965925, 0.866025,
   0.707106, 0.5, 0.25881, 0.0}; // remaining table values omitted

double  cosine(const int degree)
{
   return cosine_table[degree / INTERVAL];
}
```

Figure 2.4: Source code for cosine calculation using a table

Processor	Appl. 1	Appl. 2	Appl. 3	void Appl.
M68HC12 (16 bit, w/o FPU)	821+233	11287+1078	13204+1448	77+50
PowerPC (32 bit, w/o FPU)	152+104	4408+284	5044+84	32+0
PowerPC (32 bit, w/ FPU)	88+96	184+240	252+40	8+0

Table 2.3: Code and data sizes (in bytes) for sample cosine applications

Taking the requirements of the applications into account, an experienced embedded systems programmer would choose implementation 1 for the first application, since it provides the best accuracy and consumes no valuable data memory (beside the required stack space). Implementation 2 fits to application 2 as it provides the required real-time characteristics. For application 3 implementation 3 is obviously best suited.

This is the crux of embedded programming: often there is not just one correct implementation. Reuse concepts for embedded systems have to take this into account.

2.4 Summary

Despite the many similarities and connections to other software application domains, embedded software is different. The ubiquitous restrictions in terms of available memory space and processing power often prevent a simple application of development concepts from other domains.

The diversity of non-functional requirements demand many different realizations for the same functional aspect of embedded software systems. Reuse in this area requires adaptability in a much finer granularity than in other domains.

The efficient realization and management of these software variants is the key factor for technically and economically successful embedded systems software development in the future.

3 Family-Based Software Development

3.1 Introduction

As discussed in the Chapters 1 and 2, for embedded software development it is important to produce both reliable software and resource efficient software in a fast and cost effective process. So the main goal of this chapter is to evaluate approaches for family-based software development with respect to their usability in embedded software development. Therefore, it is relevant which parts of the software development process are covered by a methodology, how it helps to produce the implementation(s) and that these implementations use the memory and processing time effectively. This requires not only to look at the technical aspects of the methodologies alone. It is also necessary to take the typical skills and education of embedded software developers into account.

The promoters of software families and software product lines generally claim that these are an efficient way to produce reliable software (and are able to prove this). The question of resource efficiency does rarely get so much attention, however.

This chapter starts with a general introduction to software families and product lines. The following detailed discussion of the state of the art in this area covers general methodologies, which are more focused on processes, as well as realizations techniques that can be used to actually implement software families or product lines. The concluding section discusses how the different methodologies can be combined and summarizes the main issues of this chapter.

3.2 Family-Based Software Development and Product-Line Engineering

In the sixties and seventies of the last century a phenomena called “software crisis” appeared. Faster computers made previously unthinkable applications (theoretically) possible. But while the creation of better and faster hardware even today follows Moore’s Law [Moo75], which predicts a doubling of computing speed per 18 month, the growing complexity of software development started to cause problems. More complex software results in more lines of code, more interaction between program parts, more possible errors in programs, and more time to get software produced. Therefore development of new programs could not be done at the same pace hardware evolved.

One of the reasons was that, compared to other areas of business, reuse of software artifacts in these days was limited. The emerging software engineering discipline started to think about how a system could be built from reusable parts. Parnas with his work about *program families* pioneered in an area that is one of the hottest topics in software engineering today. The term *program families* was coined by Parnas in 1976 when he discovered that many problems of software development and deployment are related to the fact that software development activities are often focused on solving only a single problem. He recognized that it may pay off to look not only at one problem at a time but to consider similar problems simultaneously.

His definition for program families, as given in [Par76], is

> We consider a set of programs to constitute a *family*, whenever it is worthwhile to study programs from the set by *first* studying the common properties and *then* determining the special properties of the individual family members.

While this definition can also be applied to already existing programs (or any set of related software artifacts), Parnas already saw the main application of the program family concept in the creation of new software. In [Par79] he described the application of the program family concept to support reuse in the area of operating system design.

The term *product line* appeared much later in the software engineering community. The focus of product-line engineering is quite different, yet the main ideas are the same for program families and product lines.

The definition for a product line, as given in Griss [Gri00], is quite close to the definition of Parnas:

> A *product line* is a set of products that share a common set of requirements, but also exhibit significant variability in requirements. This commonality can be exploited by treating the set of products as a family and decomposing the design and implementation into a set of shared components that separate concerns.

But this definition requires that a *significant* amount of variability has to be present, whereas Parnas focuses more on commonality between family members.

Another definition, given by Lopez-Herrejon et al. [LHB01], states that “A *product line* is a family of related software products . . . Different family members (product-line applications) are represented by different combinations of components”. This definition states even more clearly that while there have to be relations between the products in a product line, it does not have to be a very strong relation.

Both definitions focus on different aspects of product lines. The first definition emphasizes on the origin of a program family based on a set of requirements, whereas the second definition gives a hint on how product-lines are implemented. Both use the term *family* to describe the set of members of a product line and both fit into Parnas definition of a program family if *program* is replaced by product. Product lines are by definition not necessarily pure software products but may be any kind of product. The main difference between program families and product-lines is the view on commonalities and variabilities: A product line is defined by the needs of the customers of the product line, which is an external view, a program family is defined by its internal view on the programs.

Although the product-line term is the more popular term today, this dissertation focuses on software family development rather than product-line development for two reasons:

- The term *software family*[1] focuses on the aspect of producing software, which is an inside view, a product-line approach takes an outside view, focusing on the final results (the products). Products are not always just software, often, especially in the embedded systems area, they are a combination of hard- and software.
- Although it is possible (and sometimes even the optimal way) to implement software for product lines without using software families, software families are in general the appropriate solution.

The goal of family-based software development is to benefit from producing the common parts of a family only once and reuse them to create different family members by combining common parts and member-specific parts. Admittedly, the extraction and realization of the common parts of a set of applications may initially cost more than the creation of a small number of separate applications, but once the number of applications grows, the family-based development finally costs less, as is shown for example in [CHW98].

The Figure 3.1 shows that under though simplified yet reasonably accurate assumptions, there is a point where the high initial investment for a family starts to pay off, because the cost for creating a family member is lower than the cost for producing a new single system solution from scratch.

[1] Although Parnas used the term *program family,* the term *software family* seems to be more appropriate, because a software family is not necessarily a set of complete programs, but may be any kind of software artifact like a library or component.

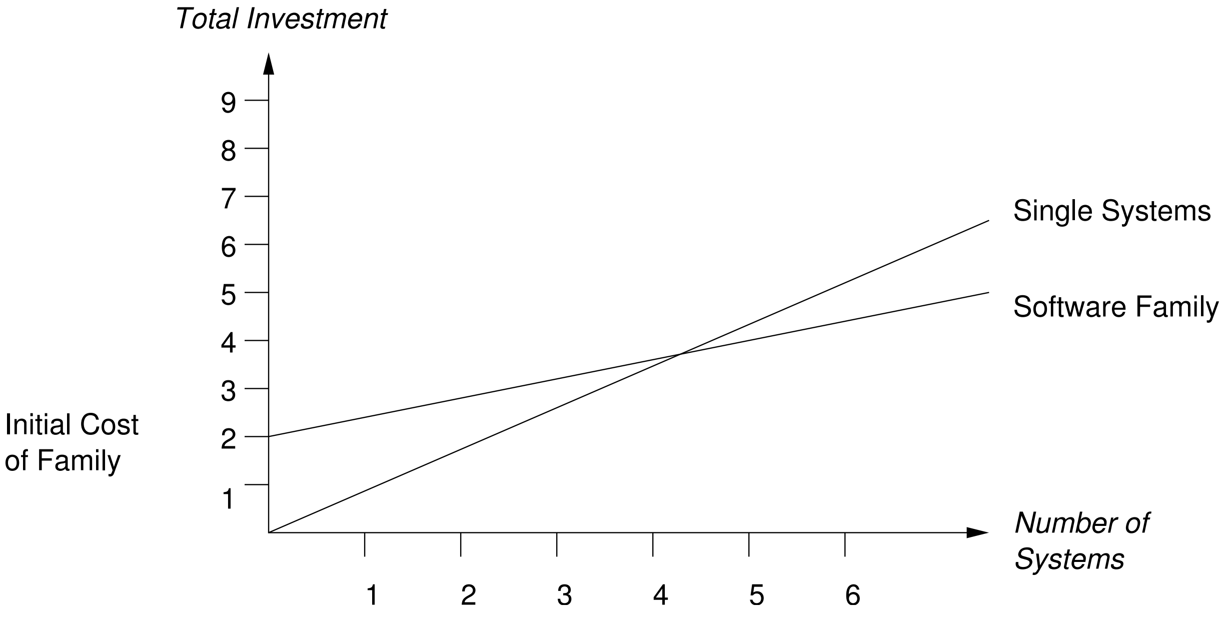


Figure 3.1: Benefits of software families versus single application development

3.3 Family-Based Software Development Process

Contrary to "traditional" software development process models like OMT [RBP$^+$91], Booch [Boo94] or the Rational Unified Process [JBR99], family-based software development is divided into two separate processes: *domain engineering* ("engineering for reuse") and *application engineering* ("engineering with reuse"). Domain engineering comprises all activities to create the reusable parts of a family. Application engineering is concerned with the construction of concrete applications from the previously created parts.

Despite the fact that family-based development process models differ in many ways, the separation between domain and application engineering always exists. The degree of separation and the ways of linking both can be quite different.

3.4 Domain Engineering

Domain engineering can be defined as "...the activity of collecting, organizing and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e. reusable work products), as well as providing an adequate means for

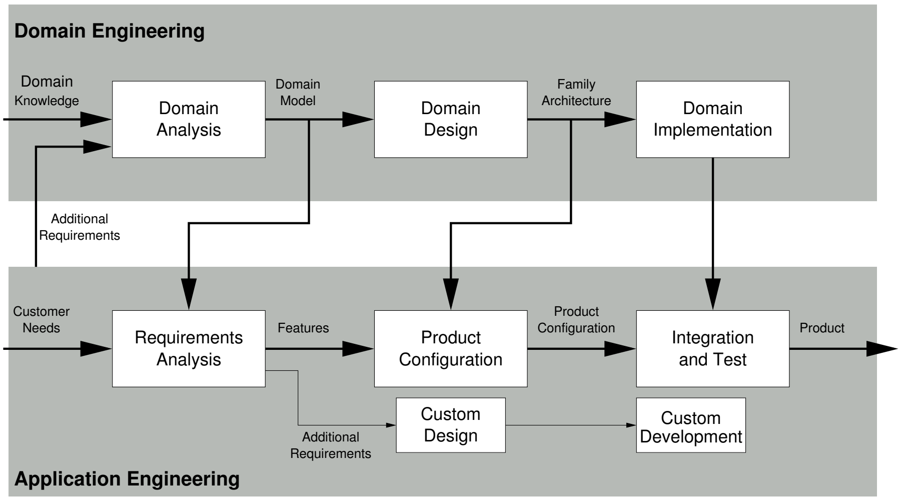


Figure 3.2: Relation between domain and application engineering (based on [SEI97])

reusing these assets (i.e. retrieval, qualification, dissemination, adaptation, assembly, and so on) when building new systems" [CE00].

The domain engineering itself can be further divided into *domain analysis*, *domain design* and *domain implementation*. Each of these parts has a strong relation to its corresponding counterparts in application engineering. The relations are depicted in Figure 3.2. It is important to point out the feedback path from the application engineering back to the domain engineering. During application engineering additional requirements that have to be considered in the domain engineering may be found. These requirements may influence the domain analysis, design or implementation.

The gathering of domain knowledge is not only an activity where external information is processed just once. Knowledge gathered during the other parts of the process may also change the view on the domain. The same holds true for all parts. Even a custom development may eventually lead to the inclusion of parts of that development into the domain design or implementation.

3.4.1 Domain Analysis

The domain analysis (DA) is the most important and probably also the most difficult step in developing of family-based software. It is mainly concerned with defining the *problem domain*. The term "problem domain" is used, because it describes the different problems the family members have to solve. The goal of DA is to model the problem domain in a way that can be used to produce a software family for it.

In order to be able to produce a model for a domain, everyone must agree on what the domain actually is. Therefore, the first activity in DA is to define the *scope of the domain*, that is, to find a measure to decide which problems belong into the domain and should be supported by family members and which do not.

DA requires an in-depth understanding of the (possible) problems of the domain to give a precise and clear definition of the domain scope. The *domain experts* are responsible for the scope definition. They have to create the definition in cooperation with all potential *stakeholders* of the domain, like end users, domain designers and implementors, application analysts and designers, and even the developing organization management.

The scoping activity, like all other activity in domain analysis, usually does not produce final results immediately. Results of the following activities during the domain analysis and also during the other parts of domain and application engineering may lead to a changed view on the domain and could eventually result in a changed and improved domain scope definition.

The next activity is to find out what differentiates the problems in the domain and what is common to all problems or subsets of them. The results of this variability and commonality (VC) analysis already give a measure to evaluate the scope definition with respect to the range of problems included.

If the number of commonalities is high and only a very limited number of variabilities exists, the scope might be too small to justify the development of a family. If the number of potentially useful family members is too small, the additional effort for creating a family could be wasted.

It is also possible that the VC analysis shows a low number of commonalities. This could indicate that the scope is too wide, there could be too many systems in the domain that have not enough in common to be effectively producible from the same set of reusable abstractions. However, since all these indicators are very soft, it is not possible to give general advice on how to interpret the results of the VC analysis. Experience from previous, comparable projects is necessary to decide this matter.

The scope and representation of the results of a VC analysis differs for different methodologies. Most methodologies use a graph-like structure to represent commonalities, variabilities and their relations. Some methods additionally include case diagrams, state diagrams and other means to represent the problems covered by the domain.

To enable discussion during the analysis and to communicate its results, it is necessary for all persons involved to understand each other. Often, different terms have the same meaning to different persons or, even worse, the same term is interpreted differently by different persons. To solve this problem most methodologies propose the creation and use of a domain dictionary. Such a dictionary holds the definitions for all special terms relevant in the domain. The domain dictionary is created at the beginning of the domain analysis phase, and extended later, if required.

The scope definition, the results of the VC analysis and the domain terminology dictionary together form a model of the domain that is then used in the subsequent development activities.

The main problem of domain analysis is that it relies mostly on soft factors like knowledge about the domain, the appropriate choice of the domain scope, commonalities and variabilities. Even small changes may result in completely different results that may fit better or worse. In general, it is not possible to decide whether a better domain representation exists.

3.4.2 Domain Design

The domain design activity translates the results of the domain analysis (the domain model) into a software design that allows to create the different family members from it. For a given domain model there might be many different designs that could be used. The choice of the design is influenced, for instance, by the skills of the designers and the available tools.

One of the most important influences is the nature of the domain itself. If, for instance, a high degree of variability must not cause avoidable overhead, the chosen design has to ensure this.

The methodologies vary extremely in the way they support the domain design activity. A number of methodologies only define the process of how to create a domain design but do not prescribe a specific design methodology (for example FAST [WL99], ODM [SCK+96]).

While this permits the use of these methodologies in a wide range of domains, the price to be paid is that external domain design methodologies have to be integrated. Some other methodologies define a specific way how the results of the domain analysis can be translated into a design more or less directly (for example GenVoca [BO92] or DEMRAL [Cza98]). Usually, these methodologies fit only for a smaller range of domains, but are easier to use and produce results in shorter time.

3.4.3 Domain Implementation

The implementation of a software family from a domain model is the last step in the creation of a software family. Though the design has a strong influence on the implementation, there is still a high degree of freedom for an implementation. Object-oriented designs can be realized with a number of different programming languages. As with the domain design, the domain implementation depends on factors like developer skills, resource constraint etc.

To implement families and especially the variability within families, it is possible to use different techniques depending on the design and other criteria like the binding time of a variability (configuration time, compile time, link time or run time). These techniques are discussed in more detail in Section 4.

3.5 Application Engineering

Application engineering is the counterpart to domain engineering. The result of the domain engineering, the software family, is deployed to construct a specific application.

3.5.1 Requirement Analysis

The requirement analysis captures the needs of the application. To be able to select a family member from the software family, the requirements have to be expressed either directly in terms of the domain model or have to be translated into those terms either manually or automatically.

The domain model can be used to guide the application engineer during the analysis, as it already should contain the relevant variation points and commonalities for the given application domain.

The requirements analysis might also detect deficiencies of the domain model if, for instance, a variability that is not represented in the domain model is found.

3.5.2 Application Design and Implementation

The application design activity creates a design that matches the requirements of the application that were identified during requirements analysis, using the appropriate family member of the software family.

The application design does not have to be strongly related to the family design. If a family exports its functionality through common interfaces like COM/DCOM interfaces [Rog97] or any other API where the design of the family is virtually hidden, it is not necessary to use the design principles of the family for the concrete application design.

Likewise, the application implementation uses the implementation of the family member but may implement its own additional functionality with different implementation techniques. If the family member is represented for example by CORBA components and implemented in C++, the application may use Java for its implementation and communicates via CORBA protocols with its family component(s).

3.6 Selected Approaches

The following section contains a selection of methodologies that are useful for the creation and use of software families. Some of the methods cover only parts of the process (for example the domain analysis) and have to be combined with other approaches to create and use families. Others provide a full solution for family development and deployment. The selection of approaches was based on their influence on the family-based software development and their usability in practice. The presentation of the different methods is organized based on the variability modeling approach used in the methodology.

3.6.1 Domain-Specific Languages-Based Methodologies

3.6.1.1 Draco

In his PhD thesis [Nei80], Neighbors presented with *Draco* the first domain engineering approach. His model for the general domain engineering process with separation of domain engineering from application engineering is still valid. Almost all newer models are only refinements of this model.

Neighbors recognized that it is important to represent a domain in a language that fits the problems of the domain rather than using a general-purpose programming language. He introduced the term *domain language*[2] for these languages. The Draco approach is based on source-to-source transformations of domain languages. The Draco tool chain allows to specify and transform the domain specific language into statements of a general-purpose language. The difference between normal compiler technology and Draco transformations is that the transformations are user-defined and can be tagged with constraints for a selection of transformations based on domain knowledge. That in turn enables domain-specific optimizations. This is achieved by including more than one possible transformation for an input language statement from which one is selected based on the given constraints.

The initial Draco system had some deficiencies regarding scalability. For each new domain, the process of developing a domain specific language and its transformations to the target language had to be redone from scratch. This has later been fixed in Draco [Nei84] with the introduction of subdomains.

A subdomain represents a part of the solution for the parent domain. It is a Draco domain itself, using its own domain language. The parent domain can use the subdomain by providing a mapping of its domain language (or parts of it) to the subdomain language. This can increase reusability significantly. Carefully designed subdomains can be reused in many different contexts under the assumption that it is easier to provide the mapping than to develop new transformations or adapt transformations from other domains.

[2] Today the term *domain specific language* is more common, but has essentially the same meaning.

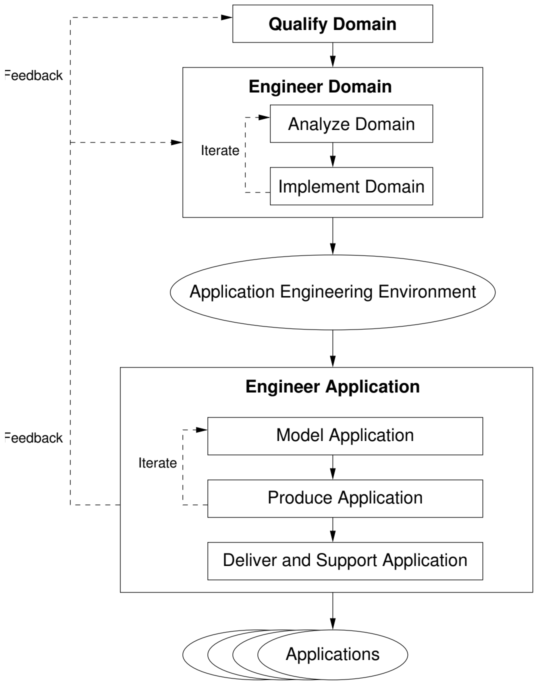


Figure 3.3: The FAST process (taken from [WL99])

3.6.1.2 FAST — Family-Oriented Abstraction, Specification and Translation

The Family-Oriented Abstraction, Specification and Translation (FAST) approach was developed by Weiss et al. [WL99]. FAST has its roots in the Synthesis [CBFO91] approach. FAST is a customizable process and artifact model for family-based software development.

The development process is described in terms of artifacts, steps to produce these artifacts and an associated role model. Each artifact has a definition where the structure, possible states and transition of each artifact are described.

The basic steps for producing a software family are shown in Figure **??**[3]. FAST starts with a *domain qualifying* activity to decide whether it is worthwhile to pay the increased initial investment for developing a family when compared to developing a number of single systems for the domain of interest. If the qualifying activity results in a positive evaluation scope, variability and commonality (VC) analysis follows.

The results from this analysis are represented as lists. For each variability a set of parameters of variation is defined. These parameters with their value ranges describe the different

[3] FAST's *domain analysis* activity includes the domain design activity.

variations possible at that point. A *decision model* is derived from the variabilities list. This model contains an ordered list of decisions and allows to differentiate the possible family members based on these decisions. As with the results of the VC analysis, it is not clearly defined how these decisions should be represented.

The decision model is used to defined an *application modeling language* (AML) for the description of the family members. An AML can be anything from simply more or less informal text, which is processed manually, up to a new language for which compilers have to be build. The decision what kind of AML representation to use depends on the nature of the domain, the development environment and also the skills of the developers involved.

The domain implementation activity consists of the implementation of the AML environment and includes the construction of tools, libraries and documentation for the AML.

The application engineer uses the environment to model the application in terms of the AML. The AML tools then generate all or parts of the application from that AML specification. Parts that cannot be generated are added to produce the final application.

The strength of the FAST process lies in its detailed process model that is supported by a process modeling approach (Process and Artifact State Transition Abstraction - PASTA). This model allows an organization to introduce family-based development easily by following the process model and changing the process according to its needs.

Due to its general approach, FAST does not give much information on how to implement a family. The representation for commonalities and variabilities is weaker and less formally defined as in FODA or other feature model based approaches. On the other hand it is possible to customize the process for the use of feature models for this representation.

3.6.1.3 P3

P3 [BCRW00] is a DSL for container data structures. It is implemented using a GenVoca generator, see section 4.5.1 for a detailed description of GenVoca. Its root is P2 [BTS94], an earlier approach for the same application domain.

P3 is not an entirely new language but extends the Java language to support easy definition of container data structures and efficient access to them. The user of P3 has to specify how the container data structures are constructed from basic types and what kind of access operations are required. The generated code provides special objects (cursor objects) for data structure access. Each cursor type has to be defined using the DSL as part of the container data structure description. A cursor can have a number of attributes, for example how often this cursor will be used, predicates defining which data items are visible using this cursor, or what operations on the data items can be performed using the cursor. From this information the implementation of the data structure itself and the associated cursors types is generated.

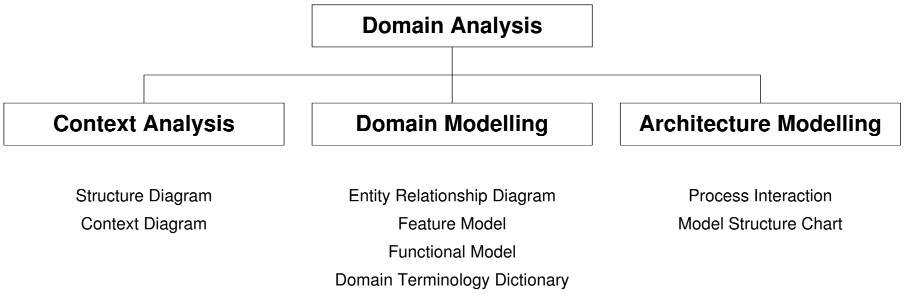


Figure 3.4: FODA activities and results (from [KCH^{+}90])

P3 is a good example where additional domain knowledge, which usually cannot be expressed easily in common programming languages, is used to provide implementations that are more efficient. The comparison with other Java-based libraries in [BCRW00] shows that using the additional knowledge leads to better performance for the DSL based approach.

3.6.2 Feature Model-based Methodologies

3.6.2.1 FODA — Feature Oriented Domain Analysis

The Feature Oriented Domain Analysis (FODA) was introduced by Kang et al. in [KCH^{+}90]. A revised version of the original FODA process has been included in the Model-Based Software Engineering (MBSE) method described in [SEI97].

FODA encompasses three different activities (see Figure 3.4):

Feature modeling provides a model of end-user visible features that are present in the given domain by providing a description for each feature and a relationship model for these features. A detailed discussion of feature models is given in the respective section below.

Information analysis identifies the type and structure of data relevant for the domain. The possible output of this analysis could be entity-relationship diagrams or class diagrams.

Operational analysis captures the possible flows of data and control within the applications for the domain. FODA allows the use of different representations for this kind of information, for example sequence charts or state diagrams.

Feature Models: The central element of the FODA methodology is the *feature model*. A feature model represents the commonalities and variabilities of the domain. A feature in FODA is defined as an *end-user visible characteristics of a system*.

Features are organized in the form of *feature models*. A feature model of a domain consists of

Feature description: Each feature description in turn consists of a feature definition and a rationale.

The definition explains which characteristic of the domain is described by the feature, so that an end user is able to understand the purpose of the feature. This definition may be given as informal text only or as a defined structure with predefined fields and values for some information like the binding of the feature, that means the time a feature is introduced in the system (configuration time, compile time, . . .).

The rationale gives an explanation when or when not to choose a feature.

Feature relations: The feature relations define valid selections of features for a domain. The main representation of these relations is the *feature diagram*. Such a diagram is a directed acyclic graph where the nodes are features and the connections between features indicate whether they are optional, alternative or mandatory. Table 3.1 gives an explanation on these terms and their representation in feature diagrams. Additional constraints can be expressed as *composition rules*. Possible constraints include that a given feature can only be included if two of three other features are selected as well, or that the feature may not be chosen if one of a specific set of features is selected. The original FODA method, however, defines only two constraints: *requires* and *mutex-with*.

Going back to the cosine example given in Section 2.3, it is possible to model the reuse problem described in that section with a feature model quite easily. For the cosine example, a feature model should contain a feature that allows to specify the precision required for the results (`Precision`)[4], a feature that represents whether discrete angle values are used (`ValueDistribution`), a feature to express that fixed calculation time (`FixedTime`) is required and so on. The complete feature model is shown in Figure 3.5.

A serious problem of FODA is that a feature model is focused on the end user of a family alone. Different stakeholders might have different views on the same domain that could result in different feature models for the same domain. A domain designer cannot request the inclusion of additional features that are relevant for the domain design but should not be visible to end users. In later methodologies like FORM or ODM (see sections 3.6.2.2 and 3.6.4.1), this problem was recognized and fixed.

[4] The names in parentheses are the feature names used in the resulting feature model, see Figure 3.5.

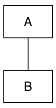
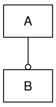
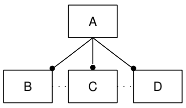
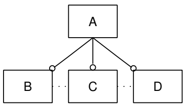
Feature type	Description	Graphical representation
mandatory	Mandatory feature B has to be included if its parent feature A is selected	
optional	Optional feature B may be included if its parent feature A is selected	
alternative	Alternative features are organized in alternative groups. Exactly one feature of the group B,C,D has to be selected if the parent feature A is selected	
or	Or features are organized in or groups. At least one feature of the group B,C,D has to be selected if the parent feature A is selected	

Table 3.1: Explanation of feature diagram elements

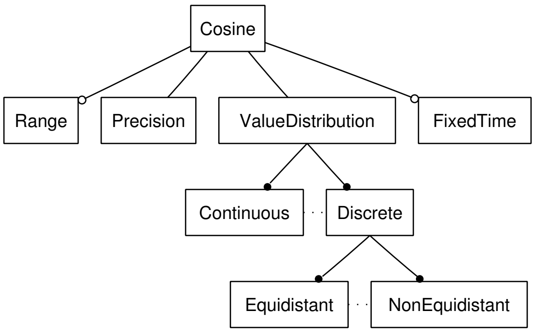


Figure 3.5: Feature model of cosine domain

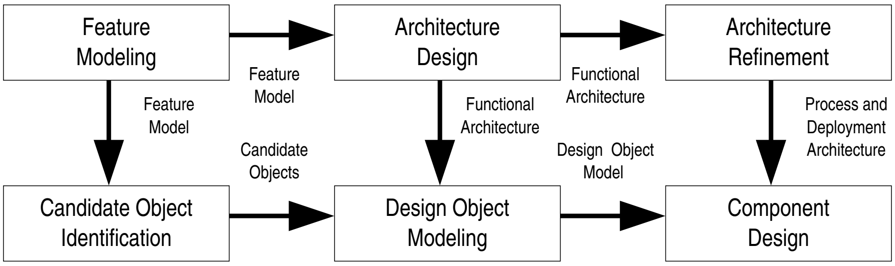


Figure 3.6: FORM development activities

Tool support for the FODA method has not been an integral part of the initial work. All information was represented in a textual or graphical way and interpreted by humans. The adaptation of tools to partially support FODA has been described in [Kru93].

FODA covers only the analysis part but its principles had a great influence on the development of many other domain engineering methodologies (for example FORM, FeatureRSBE [GFd98], ODM, MBSE) that included feature models as their main domain representation. There are also specific adaptations of FODA for specific domains like FODAcom [VA98] that provides such an adaptation for the telecommunications domain.

3.6.2.2 FORM — Feature-Oriented Reuse Method

The Feature-Oriented Reuse Method (FORM) by Kang et al. [KLLK02] is an extension of the FODA method. It extends FODA to include domain design and domain implementation based on object-oriented components.

FORM consists of guidelines for each of the six development activities shown in Figure 3.6. The first activity is the creation of a feature model for the given domain. The feature modeling activity is executed as in FODA but uses additional feature categories. FORM defines four feature categories:

Capability: Capability features describe end-user visible services and non-functional end-user visible aspects as well as so called operations, that are internal functions required to provide end-user visible features.

System environment: System environment features define the external context of the domain such as the available computing environment or protocols and interfaces to other domains/systems.

Domain technology: Domain technology features describe the domain specific technologies, patterns, methods, etc. that are used to implement capability features (that means services, operations and aspects). Such features are often not reusable in other domains (unlike the implementation technique features below).

Implementation technique: Implementation technique features are more generic than domain technology features and represent design and implementation decisions that are useful for realizing other features. Examples include design patterns, communication methods or synchronization methods.

Any feature belongs to exactly one category. In a feature model, the capability features are the top-level features.

The feature modeling guideline consists of a number of recommendations drawn from practical experiences that help the domain engineer to develop adequate models.

The architecture design guideline proposes to model the architecture in four separate views (functional, process, deployment and module) to reduce the complexity of the models. The models form a hierarchy where the functional model is the top-level model and also the base for the process model. The process model is then translated into the deployment model and finally a module model is derived.

The component design activity is used to define the large-scale components design. The input of the feature modeling activity gives an overview over the common and variable parts of the product line. Components are separated into components that capture the common features of a family. These components have to be designed to allow extension by product specific components. The product specific components capture the variability of the system. The guideline requests to design the components in such a way that the binding of components to deployment contexts is separated from the component itself as much as possible. This allows for the most flexible (re)use of a component.

For components a categorization is suggested too. Six different component categories are given:

Work-flow management components realize business processes in business applications and control, coordination and synchronization in embedded systems.

Functional components realize the core functionalities like data transformations, transactions etc.

Domain technology components realize functionalities that enable functional components to provide their functionalities. An example is the transmission of financial data in an e-commerce system that can be changed independently from the data to be transmitted, which comes from functional components.

Interface components realize the higher level part of communication (protocols, data conversion) between components or devices outside the system.

Data sharing components realize data repositories (database, ...) .

Communication/synchronization components provide communication and synchronization abstractions for interaction between components. These components are often used to abstract from the middle-ware or operating system interfaces.

FORM is intended for the implementation of large-scale systems. The guidelines allow an engineer to benefit form a relatively detailed categorization of features and components that help to organize the architecture and components design.

The main addition of FORM to the original FODA method are the feature categories that allow the representation of not end-user related features in the model. But FORM does not answer the question of how to implement the component architecture. FORM does not include tool support for any of its activities.

3.6.2.3 Other Approaches

FeatuRSEB — Featured Reuse-Driven Software Engineering Business [GFd98] combines the use case-based RSEB methodology with feature modeling as an element for managing the variability. The feature model is the central model to which use cases are attached in order to illustrate behavior of different members or member groups.

3.6.3 Decision Model-Based Methodologies

MRAM — Method for Requirements Authoring and Management Mannion et al. proposed in [MKKW99] the MRAM approach for representing the requirements of an application in terms of application family requirements. The application family requirements represent the domain model of the family. The domain model is a hierarchical organized tree of requirements with the same expressiveness as found in feature models.

A requirement can be mandatory, optional, a single adaptor (= *alternative feature* in feature models) or a multi adaptor (= *or feature* in feature models). The difference between feature models and the MRAM model is that MRAM prescribes an order in which the user traverses the domain model. Starting from the root requirement, a depth-first search is performed and if a non-mandatory requirement is found, the user has to make a choice whether or which requirement to choose. Requirement branches that lie below an unselected requirement are not considered for the remainder of the search.

The authors claim that this strategy reduces the number of decisions to be made by the user and believe that this is superior to what they call “free selection” as it is used in the feature-model based approaches. But this claim does not hold in general, since every feature model has the same expressiveness as a MRAM model with the same hierarchical structure and can be used with the same algorithm. The ability to view and explore the whole feature model can be a great advantage. The user might see a feature and the path to that feature in the model, so he or she knows which other features are required in order to obtain a specific feature. This is not possible with the MRAM approach that tries to hide the model from the user.

Tool support for defining and using a MRAM model exists and is described in [MKKW99].

3.6.3.1 Other Approaches

PuLSE has been developed at Fraunhofer IESE [BFK$^+$99]. It consists of a set of domain engineering methodologies that can be combined according to the needs of the customer and the skill level within the customer's organization. The part of PuLSE related to variability and commonality modeling (PuLSE Customizable Domain Analysis), uses a table based variability model. Each variability is listed with its permissible binding times, value ranges and constraints.

As there is no tools support for PuLSE available, the process of deriving a family member from the variability model has to be performed manually. This makes larger variability models hard to manage and use.

3.6.4 Generic Methodologies

3.6.4.1 ODM — Organizational Domain Model

The ODM [SCK$^+$96] is a domain engineering methodology that integrates concepts from other approaches, even from non-software domains, into a complete guide for the domain engineering process. ODM, like FAST, intends to be complete on the one hand, but open to specialization and customization on the other hand.

ODM uses a different terminology for its activities but fits in the general model for the domain engineering process. The activities *plan domain* and *model domain* correspond to the domain analysis, whereas the *engineer asset base* activity includes the domain design and implementation.

ODM's highlight is the introduction of a more elaborate view on many issues already present in other methodologies:

Stakeholders: The analysis results (domain models) of ODM are related to stakeholders. For each stakeholder, for instance, his or her feature(s) of interest can be traced.

Feature: To support the concept of stakeholders, which do not have to be end users, features do not have to be visible to end users in the sense of FODA but have to be of interest for at least one stakeholder.

Domain types: ODM characterizes domains by a type that explains the nature of the domain. A domain type is built from a combination of attributes like native or innovative domain, encapsulated or diffused domain, etc. The domain type definition permits specialized processes for different domain types.

Asset base scoping: ODM explicitly allows the partial implementation of domain models explicitly by first prioritizing features from the domain model during the implementation activity. Features that do not have enough priority are then not implemented even if they are part of the model.

ODM has to be customized (that means instantiated) before it can be used in an organization. Therefore, the initial effort when introducing ODM is quite high as well as the risk of failure.

Other Approaches

DEMRAL — Domain Engineering Method for Reusable Algorithmic Libraries

DEMRAL [Cza98] is not an entirely new approach but a specialization of ODM in combination with a domain implementation approach. The methodology allows to develop highly efficient algorithmic libraries based on generative programming concepts.

3.7 Summary

The numerous approaches presented in this chapter are not as different as they might appear. Most approaches use feature model like variability and commonality models (FODA, FORM, FeatureRSBE, MBSE) or models that can be easily transformed into feature models (MRAM, PuLSE). The advantage of feature model based concepts is that the models are easy to understand and scale quite well with larger domains. A drawback of feature models, and similar models, is the reduced reusability of the family artifacts due to the tight coupling to problem domain specific feature models.

Approaches based on domain specific languages can provide an appropriate way of expressing the family members and generate those family members from the member specification. However, the effort for building a DSL is rather high, since it requires experienced teams. The use of DSL is probably not justified in most cases, especially not in smaller projects.

	Draco	FODA	FORM	FeatureRSBE	MRAM	FAST	ODM	Pulse	MDA
Variability model	DSL	FM	FM	FM	DM[5]	DSL	any	VL	n/a
Model size scalability	o	++	++	++	+	o	n/a	-	n/a
Covers		DA	DA,DD	DA	DA	DA-DI	DA-DI	DA-DI	DD,DI
Design and implementation support		no	partial	no	no	no	no	no	partial
Effort for new domain	high	low	med	med	low	high	high	high	n/a
Tool support available	no	no	no	no	yes	no	no	partial	yes
Domain specific tools required	yes	no	no	no	no	yes	no	no	n/a
Required skill level	very high	medium	medium	high	high	very high	very high	high	high

Table 3.2: Overview FBSE approaches

The main focus of most methodologies is the organization of the development process and the domain analysis activity. Besides FORM and partially FAST, no methodology gives detailed instructions regarding design and implementation. While these phases are covered by all presented methodologies, with the exception of FODA, to some extent, there is no focus on deriving the software design or implementation from the results of the analysis directly.

Tool support is a another weak point of almost all approaches. Dedicated tool support as an integral part of the methodology is only available for Draco and FAST, for MRAM a modeling tool exists. While it is consensus that tool support is a critical factor for the success of a methodology, all other methodologies neither require nor provide tools or tool chain integration.

While these approaches contain many valuable ideas, their practical usability in embedded software development is quite limited because of the missing tool support, mismatch between the knowledge and skill level that is required and available, and the investments required for domain engineering. Table 3.2 gives a summarized overview. The table shows that feature modeling related methodologies like FODA and FORM are good candidates for further investigations, since they provide a good scalability of the model, do not require highly specialized computer scientist skills in order to use them, and the effort of switching to a new problem domain is relatively low.

4 Configuration Techniques in Domain Design and Implementation

Most domain engineering methodologies presented in Chapter 3 are not complete. They focus mainly on the overall process model for family-based software development (FAST, ODM, PuLSE). Some of them (ODM, FODA, FORM, FeatuRSEB) include a detailed description of how to perform the *domain analysis* and how to represent its results. But almost none, with the exception of FORM, give advice on how to design and implement a family using the artifacts produced during the domain analysis. Nevertheless, designs have to be developed and implemented.

The following section analyzes techniques available for software family realization. In order to understand when to use a certain technique, it is important to know its limitations and implications. Because the domains vary in size and nature there is no "fits all" technique available for implementing variability. Some techniques are very easy to use but do not scale well, others require extensive initial investments in terms of time and manpower, but can be used with very complex domains, others in turn are useful only for certain types of domains.

The survey focuses on

Variations that are supported by the technique. This includes the point of variation and the time at which the variation is resolved (binding time). See the Tables 4.1 and 4.2 for the variation points and binding times considered.

Performance impacts of variation point implementations. Different kinds of variation points and binding times require a certain amount of memory and processing time that might be a reason for choosing a specific implementation technique.

Combination of techniques, that means whether a certain techniques can be combined with other techniques.

Understandability of the system implementation.

Supported domains that means for what kind of problem domains a solution based on this technique is adequate.

Binding time	**Description**
configuration time	The variation point is bound by generating an implementation where the variation is already removed.
compile time	The implementation language compiler binds the variation point to a variation. This binding cannot be changed later. The binding relies on the implementation language features, for instance template evaluation.
load time	The variation point is bound at load time. One of the possible variations is included in the executable. This is different from compile time variation because it is based on object code rather than on source code selection. Examples are shared libraries.
run time	The application is able to change the variation during run time, possibly more than once.

Table 4.1: Binding times for variations

Variation point type	**Description**
Extension	Several variants are supported at the same time.
Alternative	Only one variant is supported at the same time.
Option	A variant may or may not be used.

Table 4.2: Types of variation points

When design and implementation activities for single system software development and software family development are compared, it is obvious that family-based software requires a higher degree of variability.

The sole use of programming concepts and languages that are only designed for single system development to realize software families often yields results that are hard to understand [SC92] or do not meet the performance requirements [HU96].

Although many different programming paradigms could be used for implementing software families, only object-oriented programming is investigated further, since other paradigms do not play an important role (for example logic programming and functional programming) or can be seen as a (very) limited subset of object-oriented programming (imperative programming).

4.1 Preprocessors

Some of the most widely used programming languages (C and C++) feature a built-in preprocessor, many others can be used in conjunction with an external preprocessor like m4, the standard Unix preprocessor[1].

Preprocessors perform textual manipulations on program source code before compilation. This allows to exchange placeholder strings by concrete values. Many preprocessor even allow to write macros that can be executed and in turn generate program source code.

Preprocessors are widely used to implement variations at compile time. Based on configuration information that is fed into the preprocessor, macros may generate different program code or set different values for constants.

The use of preprocessor languages bears some risks. Usually, the error checking capabilities are very limited. In order to verify if the preprocessor produces the intended results, it is necessary to check the output of the preprocessing state. This check involves in most cases human inspection of arbitrary large preprocessor-generated output, which makes these checks ineffective and error-prone.

4.2 Object-Oriented Languages

The majority of programs are nowadays implemented using object-oriented languages like C++, C#, Java or Smalltalk. According to Wegner [Weg86], an object-oriented language provides

[1] The GNU project provides an open source implementation of m4. For more information see [Gnu].

- encapsulation of data structures and operations on the data within classes,
- inheritance of class data structures and operations (method) ,
- objects that are instantiated from a class.

While not all programmers use these features in their OO programs, those basic mechanisms provide a good base for implementing variation. [2]

4.2.1 Inheritance

The inheritance mechanism allows the implementation of *extension variation* points. A base class provides a common interface that is shared by all extensions. Each extension is implemented as a subclass of this base class and provides specialized implementations.

The specialization of an implementation can be done in several ways:

Interface change: The interface of the base class is extend by an additional method that provides a new operation, or a method is removed from the interface[3].

Method change: An already existing method is replaced by a changed implementation.

Data addition: Additional data members of the class are introduced.

This variation can be bound at any time because it is possible to introduce new subclass implementations at configuration time (source code), at link time or load time (object modules) and at run time (dynamic loading of object code) as well. Depending on the programming language, not all binding times are supported by the language directly but require additional support for example from the run-time system. Java, for instance, does support dynamic loading of new classes into the running program by its class loader, whereas C++ has no built-in support for this.

Alternative variation points can be implemented like the extension variation point but only one of the available extensions is actually used. In this case, the system must ensure that only the correct extension is used all the time. The alternative must not be bound more than once.

An *option variation point* can also be realized by using a subclass that implements the optional part. If an object is instantiated from the base class it does not support the option. If it is instantiated from the subclass it supports the option.

[2] Some of the concepts can also be emulated in imperative languages like C or Pascal with additional code and less strict checking of type systems etc.

[3] While Eiffel [Mey92] supports the removal of methods, most other object-oriented languages do not.

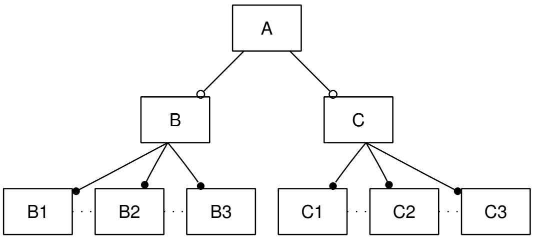


Figure 4.1: Feature model exhibiting dependent variability of features

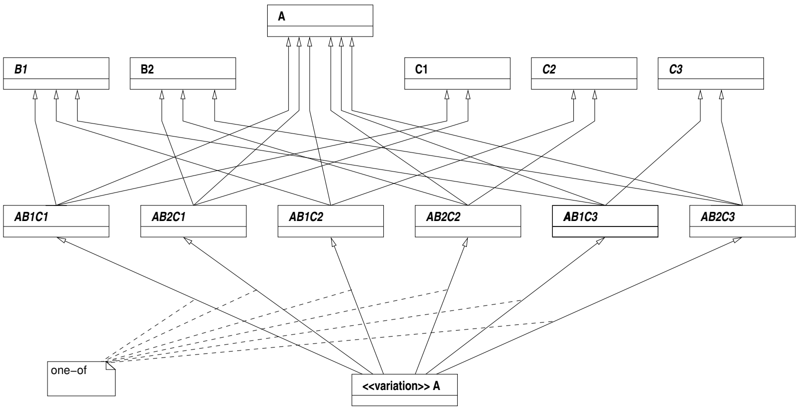


Figure 4.2: Class number explosion problem

If the variation points are sub-variations of another variation, inheritance is an efficient way to implement these variations. The feature model in Figure 4.1 can be implemented easily using any object-oriented language.

However, inheritance faces serious problems when different variation points that are implemented via inheritance are combined. Figure 4.2 demonstrates the problem.

The feature model requests two different variation points that could be implemented using inheritance. Both extension variations are independent from each other and can be realized by a single method. The classes B1-B2 and C1-C3 implement these methods. Using inheritance, it is necessary to combine them into a huge class hierarchy with 6 classes[4].

Object-oriented languages that do not support multiple inheritance are not able to implement more than one independent variation point with inheritance, but some languages like Java allow to simulate multiple inheritance via interface inheritance mechanisms, see the section

[4] 6 = (number of variants of class B) * (number of variants of class C).

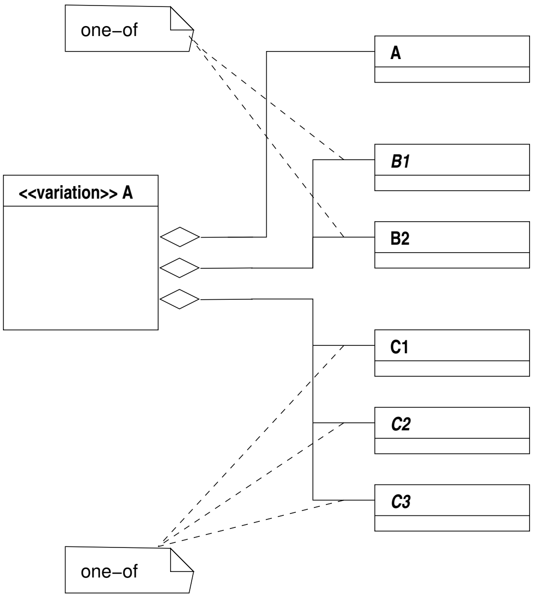


Figure 4.3: Aggregation-based implementation of dependent feature variability

on aggregation below.

Application of inheritance-based techniques are described in [SB00] (C++) and [BC90] (CLOS).

4.2.2 Aggregation

Object aggregation is the combination of instances of different classes as data members in another class. The aggregating class ties all the different aggregated classes together by its own methods. Aggregation allows the combination of independent extensions and alternatives even for OO languages that support only single inheritance but still faces the problem of class number explosion as described in Section 4.2.1. The realization of the feature model from Figure 4.2 using aggregation is shown in Figure 4.3.

The use of generic or parameterized types offered by some object-oriented languages[5] allows to avoid the re-implementation of the aggregating class for each possible combination. The generic implementation of the aggregation class is parameterized with the possible combinations of classes that implement the variations.

[5] C++ supports generic types with its template mechanism, for Java several language extensions for generic types have been proposed by several authors [BOSW98, MBL97, Vir01].

4.2.3 Uses-Relation

The uses-relation is another way to implement variation points. Instead of aggregating the object instances in the class, only a reference to an object of a given class type is stored. The reference is resolved at run time and the referenced object is accessed. By changing the reference to point to another object, alternative objects can be accessed. Each of those objects might represent a variation. The difference between aggregation and the uses-relation is that for an aggregated object, the variation is bound at instantiation time, whereas a variation implemented via uses-relations can be changed during the whole lifetime of the object.

4.2.4 Parameters

Parameters are often the most easiest way to realize a variation. Parameters are given to objects at instantiation time or supplied by a method call. The instantiation code or the method implementation evaluates the parameter during run time and behaves differently for different parameter values, thus realizing variation. Alternatives and options can be realized by boolean decisions based on the parameter value, extensions use a discriminator function to decide which extension is going to be used.

The drawback of using parameters is that the binding time is usually resolved at run time every time the variation point is accessed. This may cause serious overhead when such a variation is accessed very often. Some languages and compilers (C++,Eiffel) have limited support for constant propagation and partial evaluation at compile time, but most object-oriented languages do not support this.

4.3 Meta-programming

For a programming language, rules exist that define for each basic statement what will happen if this statement is executed. Meta-programming allows to change the semantic by introducing a way to modify the way a language works. It could be possible, for instance, to change the way a method call is carried out for all (or certain) classes. This can be very useful for profiling applications or to make remote method calls.

Some languages are based on the concept of meta-programming directly, like Smalltalk [GR83] or Beta [LKMM94], for others it is possible to define an environment that enables meta-programming to some extent. With the C++ template mechanism [CE00, Frö01], for example, meta-programming at compile time is possible but meta-programming at run time is not.

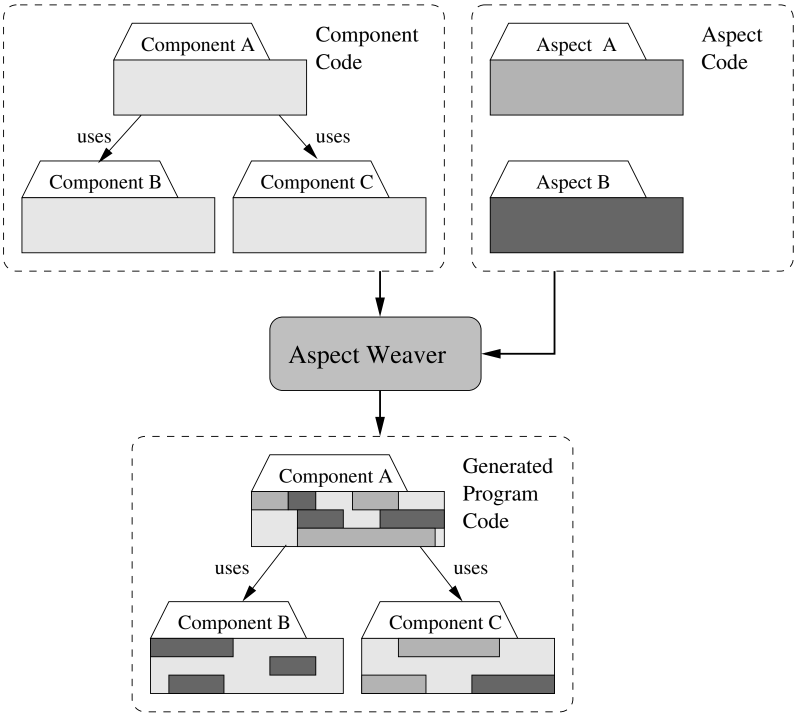


Figure 4.4: Aspect weaving process

4.4 AOP — Aspect-Oriented Programming

While many features of a system family can be implemented in clearly defined locations like a method, class or component, some of the features are cross-cutting the systems implementation. These cross-cutting features are called non-functional features or *aspects* whereas the other features are functional features of the system.

The realization of non-functional features like synchronization influences the implementation of functional features. AOP [KLM$^+$97] suggests a separation of concerns. Functional features are still implemented in a normal fashion, but the additional non-functional features (aspects) are expressed in aspect languages that allow to express the cross-cutting aspect(s) in an aspect-specific language. The aspect code is integrated into the functional feature implementation during a *weaving* process by the *aspect weaver*. Figure 4.4 illustrates this process. The weaving might take place before compile time using a generator or during run time using an interpreter.

The example in Figure 4.5 shows how a simple sensor read operation `read_sensor()`, written in C or C++, is either made multiprocessor safe or multi-thread safe with the help

```
// activated in multiprocessor case
aspect MPSafe {
  advice execution(“% read_sensor()”): around()
   {
     do_spinlock() ;
     int retval = tjp->proceed(); // call orginal function
     release_spinlock();
     return retval;
   }
};
// activated in multithreaded case
aspect MTSafe {
  Semaphore mtSema(1); // semaphore shared by all instances
  advice execution(“% read_sensor()”): around()
   {
     mtSema.wait() ;
     int retval = tjp->proceed(); // call orginal function
     mtSema.signal();
     return retval;
   }
};
// conditionally to be synchronized function
int read_sensor() { ...; return value; }
```

Figure 4.5: Modeling synchronization using aspects

of AspectC++ [AC]. The aspect `MPSafe` adds synchronization using spinlocks to the code, while the `MTSafe` aspect introduces semaphore based locking. The resulting implementations after weaving show how strong the influence of these aspects on the functional implementation is: there are more lines of code concerning the synchronization than are necessary for the actual functionality.

Functional features are relatively separated and independent so it is easy to handle variability on a functional level. Including a functional feature means inclusion of its implementation. For non-functional features, this is not possible because their addition may modify/influence the implementation of all other features. Family-based software development can benefit from AOP, because AOP makes it possible to handle many non-functional features in the same way as functional features because each non-functional feature has an isolated implementation.

AOP is a relatively new field, so there are only a limited number of mature aspect languages available. Most notably are AspectJ [AJ], which allows to weave Aspects with Java programs, and AspectC++ [GSPS01a, AC], which provides the same for C++ programs. Weaving can be performed not only at compile time but also at load time, like AspectJ 1.1, or at run time [SP02]. However, such ideas are still in their early stages and not yet widely used.

4.5 Generators

Domain specific languages can be used to express domain knowledge more easily than general-purpose programming languages. But often the effort to build a compiler that generates the complete machine-executable system from a DSL is not feasible. The use of standard programming languages and environments as base for a DSL is often much more economical.

Domain specific languages allow to hide the complexity of the implementation of variability from the user and allow for optimizations based on application knowledge. Standard compilers have their strength in the fine-grained optimizations of program code including domain knowledge about the target processor platform. Only the combination of both makes it possible to generate high-performance systems.

Creating a new DSL for a domain can be a difficult task, because it involves the creation of the language itself, the tools to translate the language into another language understood by the following tools, and a supporting environment (run-time libraries, graphical tools, debuggers, ...). The use of compiler generation technologies can be very helpful in this case.

The implementations generated usually deploy some of the techniques presented above like static meta programs, AOP, OO, etc. but hide their complexity from the user.

4.5.1 GenVoca

GenVoca allows to model a family in terms of incremental extensions of a family, described by equations. The valid equations are described by a grammar. Using this grammar, it is possible to describe each family member by its equation.

The example given below shows the basic idea of GenVoca. The features a and b represent basic features that can be used stand-alone. The features $c(X)$ and $d(X)$ are features that extend another family member X. A family member with the features a and c is described by $c(a)$. This family member can be extended by the feature d using the equation $d(c(a))$. The grammar allows to specify another family member with the same features a, c and d but a different equation $c(d(a))$.

Compared to a feature model, GenVoca adds the ability to describe a structure in which features are to be combined and allows to attach semantics to the structure. That means that a family member with the same set of features but a different equation may or may not be represented by the same implementation. Although GenVoca not necessarily describes a layered architecture, it is quite easy to derive such a layered architecture from the equations. In this case, the creation of the grammar also develops the system design.

It is not required that a GenVoca grammar is processed by a tool to generate a family member. In [CE00] the grammars are used to derive a matching system architecture implemented by static meta-programming in C++. Each grammar rule is represented by a C++ template and systems are built by combining the templates.

Although this approach is sufficient for some domains, many others can benefit from generators that process a GenVoca equation and generate the appropriate family member from it. Batory et al. [BTS94], [BCRW00] have developed a number of generators supporting different kinds of implementation languages for family features. They have recently implemented the JTS [BLS98] that allows to generate GenVoca generators for Java from a GenVoca grammar.

The limitations of GenVoca are obvious: if the family architecture can not be modeled easily be a layered or stacked design, GenVoca cannot be used. GenVoca is not a standalone family-based software development approach. It lacks the domain analysis. Though Batory et al. give hints on what to look at during the analysis activity, the construction of GenVoca grammars relies on a domain analysis that allows to model the domain in terms of layers of features.

4.6 Summary

All techniques presented in this chapter may be used to provide the different types of variation points. However, there is no single technique that can be used, but careful combination of the appropriate techniques is the key. Some are very easy to use in the first place but get very problematic in more complex scenarios. Table 4.3 gives an overview over the presented techniques and their characteristics.

The C/C++ preprocessor is a good example for this issue. It is very easy to learn, available in every C/C++ compiler and can be very helpful but also introduces very nasty problems due to missing type checking and the like. Similar problems on a different level can be observed when static template meta-programming is used. It provides very interesting ways of compile-time configuration of C++ applications but programmers, on the other hand, tend to produce “write-once” code. Static template meta programs can be written by an experienced programmer and may provide the expected results. The intention of a meta program, however, often cannot be easily grasped by another programmer, or even by the original programmer after a while. Discussions with users [Frö02] and Krzysztof Czarnecki, one of the inventors of static template meta-programming [Cza03], attested to this perception. However, in combination with code generators there is a wide range of applications that can benefit from static meta-programming techniques. Dynamic meta-programming techniques usually require very specialized languages that are not main stream in embedded systems development and, even more importantly, require more processing power and code size due to the necessary meta-programming run time of the target application.

While parameterization, together with preprocessors, is the most common technique in embedded applications, it may cause many problems. Performance depends heavily on the "cleverness" of the compilers, for example if for constant parameters appropriate partial evaluation is performed at compile time. Realizing cross-cutting variations is also a weak point of parameterization. However, for local run time, decision parameters are a good choice.

The object-oriented techniques presented above provide a good ratio between ease of use, performance impacts and applicability. While most embedded programs are still not written in object-oriented languages, there is a trend towards C++ (EC++) and also Java. The use of aspect-oriented techniques will also benefit from this trend. AOP is a good compromise between the full power of meta-programming (MP) and the problem of performance (for dynamic MP)/understandability (static MP) when it comes to the implementation of cross-cutting variations. Drawbacks are the still limited tool support. However, with AspectJ and AspectC++ two general-purpose AO-languages are available.

Last but not least, code generators are especially in the embedded world a well-known and already applied concept to hide the complexity of certain program parts from the application programmer. However, the creation of new code generators is a demanding task and only feasible for larger application domains. Code generators themselves may use all of the techniques presented above in their generated code, as it is (mostly) hidden from the users.

The main conclusion is that most of the techniques are relatively complicated and not directly usable for typical embedded software developers. The use of standardized programming languages (C, C++, Java) together with additional techniques like code generators and, to a limited degree, also general-purpose aspect-oriented languages (AspectC++, AspectJ) seems to be the most sensible approach. An additional benefit, when standardized techniques are used, is the transfer of developer knowledge and skills into new projects, which increases quality and reduces cost.

		Preprocessor	Inheritance	Aggregation	Uses-Rel.	Parameterization	Meta-programming	Generators	AOP
Binding time		Config	Comp.-Run	Comp.	Comp.-Run	Comp.-Run	Comp.+Run	Config	Comp.-Run
Code size	Alternative	0	o to +	+	+	0 to ++	0 to ++	0	0
	Extension	0	0 to +	+	+	0 to ++	0 to ++	0	0
	Option	0	0	+	+	o to ++	0 to ++	0	0
Run-time Performance	Alternative	0	0 to +	0	+	0 to ++	0 to +++	0	0
	Extension	0	0 to +	0	+	0 to ++	0 to +++	0	0
	Option	0	0	0	+	0 to ++	0 to +++	0	0
Cross-cutting Variation		no	limited	limited	limited	limited	yes	limited	yes
Comprehensibility		low	med.	good	med.	good	low	good	med.
Implementation cost		med.	med.	low	low	low	high	very high	med.

Table 4.3: Overview Implementation Techniques

5 Tool-Based Construction and Composition

Based on the discussion presented in the previous chapters, it is obvious that successful application of the software family approach requires adequate tool support. However, most existing approaches do not come with tool support. The lack of tool support for the domain of embedded systems development was the main motivation for the work presented in this chapter.

The approach for embedded software production presented in this chapter is a combination of

Family-based domain engineering — to produce reusable software components with the required level of functional variability.

Object-oriented implementation patterns — to design and implement the components in an efficient way.

An integrated tool chain — to connect domain models with design and implementation and aid the user of components.

The explicit goal here was not to invent a completely new solution for domain engineering but to reuse the best of the many already existing methods and to improve the parts that need it. Special attention has been paid to the practicability of the methods in embedded development contexts. It was not intended to develop a universal method that is able to support every family-based development optimally but rather to provide a solution for restricted sets of problem domains in the embedded context.

The basic idea of CONSUL[1] is to provide a set of tools from a common base for each phase of the family-based software development process. To ensure the ability to integrate the CONSUL tools with other tools, it has to be designed as an open framework that integrates the data sources and other tools used throughout the family-based development process, like the modeling tool for VC analysis, object-oriented modeling tools, code generators, compiler, UML or SDL descriptions, documentation, source code, etc.

[1] CONSUL = CONfiguration SUpport Library

Figure 5.1 shows the four corner stones of family-based software development and the models used in CONSUL to represent the necessary information. The problem domain is represented by hierarchical feature models, an extended form of the original FODA feature models. For the solution domain, that means the concrete design of a software family and its implementation model, a new modeling language, the CONSUL component family model (CCFM) has been developed. The reuse of an existing modeling language was not feasible, since those languages do not support the required flexibility of artifacts in an easy way (UML) and/or are too specialized on specific software architectures (GenVoca). The two modeling languages for the deployment of families during application engineering are complementary to the modeling languages used for domain engineering. The feature set represents a single problem inside the problem domain with features and associated values. The concrete component model describes a concrete member derived from the solution family.

The CONSUL methodology combines the knowledge captured in these models to provide tool support for the different roles within a family-based software development process:

- The domain analyst uses a CONSUL-based feature model editor to build and maintain the problem domain model.
- The domain designer uses a component family model editor to describe the family architecture and to connect it via appropriate rules to the feature models.
- The application analyst uses a tool that allows him or her to explore the problem domain and to express the problems to be solved in terms of selected features and additional configuration information.
- The application developer gets a member of the solution family generated from the concrete component model with the help of the CONSUL transformation engine.

Though the tools have to be tailored to the different needs of the different roles, they all use the model evaluation and manipulation facilities provided by CONSUL. The base for successful cooperation of these different roles are that both the models and the tools provide the necessary expressiveness and support for describing the relevant data and solving the related tasks (for example modeling a problem domain or describing a software solution).

To get an idea how a CONSUL application could look like, Figure 5.2 shows an example for an interactive application based on CONSUL. The consul@gui application is a general-purpose editor for CONSUL models and is also able to generate family members. More specialized CONSUL applications include a Java applet for interactive configuration of existing models via a web browser and a command line client that evaluates predefined models. Other possible applications include seamless integration of plug-ins for integrated development environments or client/server applications.

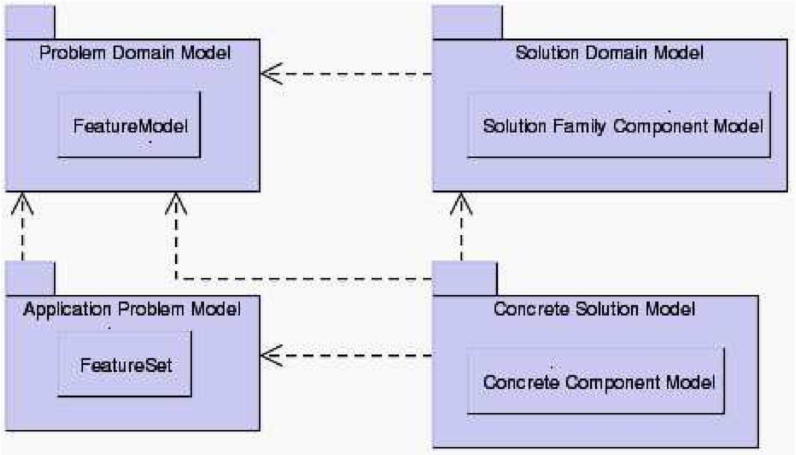
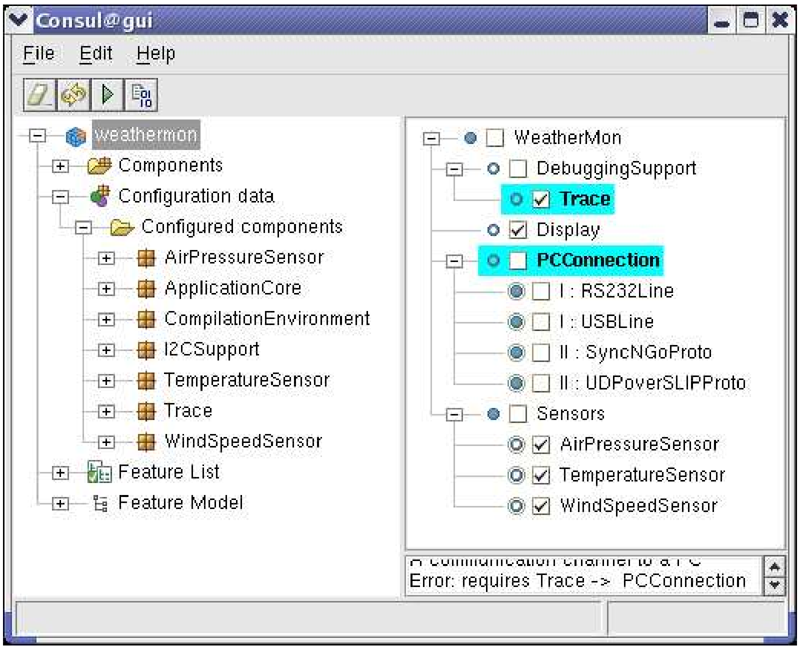


Figure 5.1: Overview of CONSUL models



- ● □ WeatherMon
  - ○ □ DebuggingSupport
    - ○ ☑ Trace
  - ○ ☑ Display
  - ○ □ PCConnection
    - ◉ □ I : RS232Line
    - ◉ □ I : USBLine
    - ◉ □ II : SyncNGoProto
    - ◉ □ II : UDPoverSLIPProto
  - ● □ Sensors
    - ◎ ☑ AirPressureSensor
    - ◎ ☑ TemperatureSensor
    - ◎ ☑ WindSpeedSensor

```
A communication channel to a PC
Error: requires Trace ->  PCConnection
```

Figure 5.2: consul@gui, an interactive CONSUL model editor

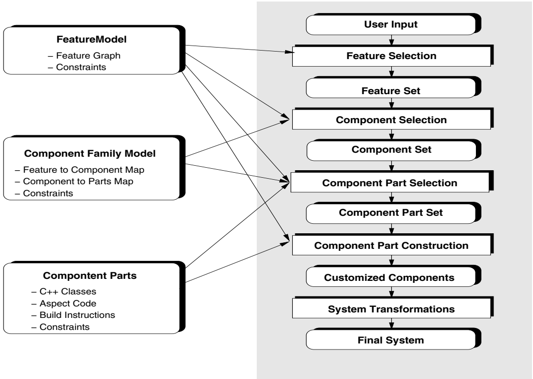


Figure 5.3: Overview of CONSUL method data flow

A general overview over the flow of data during the model evaluation in CONSUL is shown in Figure 5.3. Except for the introduction into some general conceptual aspects of CONSUL in the next section, the remainder of this chapter is structured along this data flow. The first block, starting in Section 5.3, introduces the feature modeling concepts (feature models and feature sets) used in problem domain analysis and application analysis to provide abstract definitions of the problems to be solved. The second block in Section 5.5 deals with the modeling of solution domains with components (component family model and concrete component model) and the transformation of the knowledge captured in feature models into solutions.

5.1 Foundations of CONSUL

5.1.1 Software Family Hierarchy

Almost every problem in software development with a certain complexity is usually solved by decomposing it into smaller units (problems) that can be solved independently. This ability to decompose problems is also one of the fundamentals of reuse, since if a problem is decomposed into smaller problems, it is more likely that a solution for the smaller problems already exists or can be reused in a later project. Software families are no exception from this rule. A software family for a specific domain may be built from a set of reusable software families that were built from other reusable software families themselves and so on.

The CONSUL approach proposes a three level hierarchy of software families:

Application family level: Application families are end-user visible systems, for example complete embedded systems. The number of family members is relatively small, as each family member represents an end product. A CONSUL application family can be built from any number of component families.

Component family level: A component family is composed of a set of components and the component architecture. A component family provides functionalities that can be used in different applications/application families. The number of component configurations (that means family members) is high.

Component implementation family level: Each component in turn can be a family in itself, providing different implementations for use in different component families and component family configurations. Most components have a small to medium number of different implementations (that means family members).

Each hierarchy level has different characteristics and requirements. Therefore, there are different ways to deal with them. In general, it is not possible to analyze and develop a whole application the way a single component is analyzed and developed. One of the main reasons for this lies in the different reuse characteristics for each family type.

A single component with different implementations tends to be reused by different application domains, providing its defined set of functionalities. This type of reuse is called functional reuse (see Section 2.3.1). Application families, however, are usually extended by new members with changed functionalities during the family life time. Often, sometimes too often, the changes required are quite extensive. This is a temporal reuse. Component families lie somewhere in between. They can be used in a number of different projects relatively unchanged, but their functionalities may also evolve over time. The relation between the reuse dimensions and these hierarchy levels is illustrated in Figure 5.4

To support reuse based on hierarchical software families to the full extent in a development approach, it is necessary to provide the ability to combine hierarchical models of different problem domains and also to support the composition of several solution domains.

Most family-based methodologies do not provide such support. Often they use a single combined problem domain and family model that is not reusable by definition (GenVoca is one example).

5.1.2 Restrictions

Restriction expressions play a central role in CONSUL. They are used in almost every model in many different places. A restriction expression is an arbitrary expression that evaluates to a boolean value. Usually, the boolean value `true` indicates success and the

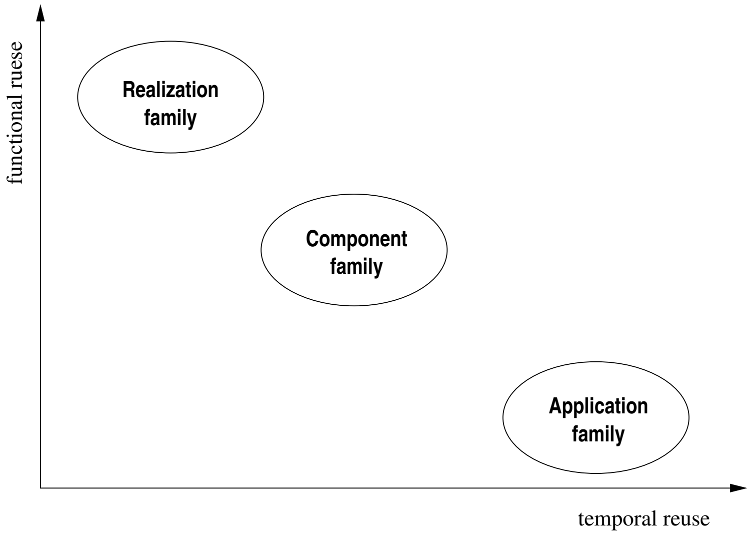


Figure 5.4: Relation between reuse dimensions and development hierarchy levels

associated element or evaluation operation can be performed. The value `false` indicates failure and is interpreted in different ways. If a restrictions is the left hand side of an implication, the implication as a whole returns `true`. In most other cases, a failed restriction indicates some error state.

The actual language of restrictions is not part of the CONSUL models, which allows to use different languages here. The current version of CONSUL, however, allows to use two different languages: the Object Constraint Language (OCL) and Prolog.

Prolog is a general-purpose logical programming language and described in many books, for example in [CM87]. CONSUL uses Prolog as the core language for model checking/evaluation. The CONSUL Prolog API allows access to any CONSUL model element and a number of different predefined Prolog clauses allows evaluation of specific conditions in the models. The `has(feature('X'),_NT)` clause, for example, can be used to check for the existence of a specific feature X in the feature set stored in _NT.

OCL is part of UML. It is a general-purpose constraint language used in UML to specify conditions for UML model elements. CONSUL uses only a limited subset of OCL. It serves mainly as easy-to-use alternative to Prolog and is internally translated into Prolog upon evaluation. The functions available for accessing the models are the same as in Prolog. Because OCL has a different syntax, however, they look a little bit different. The check `has(feature('X'),_NT)` becomes `has(feature('X')).`

As the OCL is much easier to learn and understand, it will become the standard language for restrictions in future versions of CONSUL.

5.2 The CONSUL Feature Modeling Language

The following section introduces the CONSUL feature modeling language. Figure 5.5 shows an overview over the various elements of the language.

Each problem domain model has one or more related feature models that in turn consist of a set of features. The parent/child relation between features is expressed by feature groups that collect references to related child features (Section 5.2.1). Additional relations between features (for example mutual exclusion) are expressed by feature relations (Section 5.2.2). The specification of arbitrary feature properties (for example an associated value) is supported by the possibility to attach any number of feature attributes (Section 5.2.3). The interconnection between models is done via model mappings (Section 5.2.4).

The feature set (Section 5.2.5) represents the application domain model and contains references to features and additional information, like feature attribute values, to specify the application domain in terms of features.

Many elements of the feature model may have a restriction expression (Section 5.1.2) that is evaluated before the respective item is accessed. If the evaluation of a restrictions yields the boolean value `false`, the restricted element is not accessible. The element context defines what actions will follow this inaccessibility.

5.2.1 Feature Model Structure

The basic structure of a CONSUL feature model is a tree of unique features. The child features are collected in feature groups. A feature group has a type, indicating the parent/child relation of the parent feature and the features referenced in the feature group. In addition to the three feature group types of the original FODA method (mandatory, optional and alternative), CONSUL supports the *or feature* as describe for example in [CE00] (see Table 3.1 on page 50). Although there are other feature types discussed in the literature, Czarnecki shows in [CE00] that most of those other feature types can be represented by these four feature types, so there is no need to support other feature types.

A feature or a feature group may have a guarding restriction. If a restriction expression exists and this expression evaluates to `false`, the feature/features of the respective group (and any child features) cannot be members of a valid feature set.

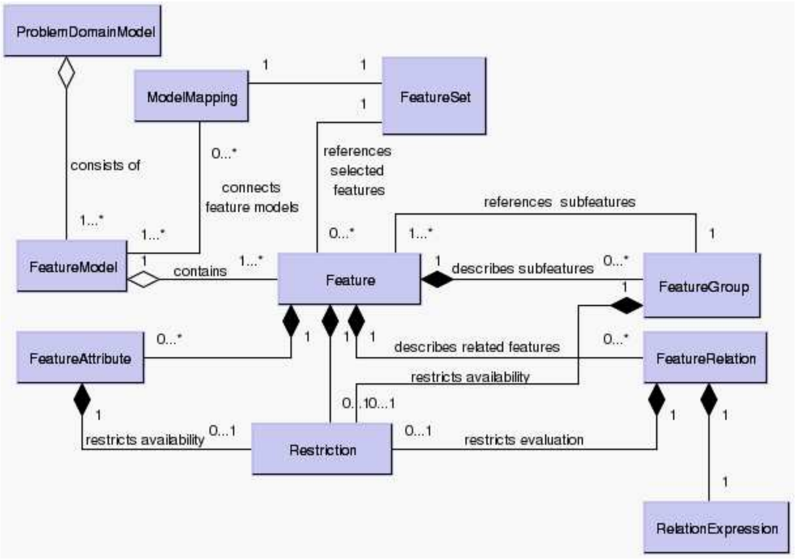


Figure 5.5: Simplified CONSUL problem domain model class diagram

5.2.2 Feature Relations

For representing additional relations (besides the parent/child relation) between features, CONSUL provides the feature relations. Each feature relation belongs to a feature that may itself have any number of feature relations. A feature relation defines a typed relation between this feature and the features mentioned in the relation expression. CONSUL comes with a set of predefined relations, see below, that can be extended via a simple mechanism to contain arbitrary relations.

Feature relations may have a guarding restriction. If the guarding restriction evaluates to `false`, the relation is not checked. Thus the restriction and the relation form an implication. If the guarding restriction is not there or evaluates to `true`, the relation must be fulfilled.

The four predefined relations are:

requires(f_{list}) The feature can only be selected if at one least feature contained in f_{list} is also selected.

conflicts(f_{list}) The feature cannot be selected if any feature of f_{list} is selected.

recommends(f_{list}) The feature should only be selected if at least one feature contained in f_{list} is also selected.

discouraged(f_{list}) The feature should not be selected if any feature of f_{list} is selected.

The difference between the first and the last two relations is that recommends/discouraged never fails. They allow the user to interpret the information that something might be wrong in both directions (ignore/correct problem).

These predefined relations are almost identical to the original FODA style constraints as far as their expressiveness is concerned. However, due to the structural constraints of the expressions, it is not only possible to check whether a certain feature set is allowed by the model and its relations, but also to detect and solve conflicts automatically. Such an automated conflict detection and solving mechanism is able to decide whether it is possible to find a conflict free solution at all, and if so, can even find it.

The user defined relation expressions can be more complex, namely any boolean expression in either a restricted subset of OCL or Prolog.

If possible, the user defined relation expression should be avoided because automatic problem solving is not possible for any of these user-defined relation expressions. This is caused by the fact that these relations may be of arbitrary complexity, allowing users to express undecidable problems that, naturally, cannot be solved by CONSUL.

However, using the additional expressiveness is useful for an important extension to FODA style feature models that is available in CONSUL, namely the feature value.

5.2.3 Feature Attributes

Some features of a domain cannot be easily expressed by normal feature semantics that require a fixed description of the feature and allow only inclusion or exclusion of the feature. For many features this is perfectly suitable, but some variabilities cannot be expressed in an efficient manner using this simple approach.

An example from the operating systems domain is the number of processors the operating system must support. There are two possibilities to model this: each possible number of processors is represented as a separate feature of an alternative feature group, or to assign the chosen value to a feature. Figure 5.6 shows both ways. For larger sets of values, representing the use of one feature per value is not very easy to use and mostly provides no additional information compared to the use of a single feature with a numeric feature attribute.

CONSUL uses so called feature attributes to specify additional information associated with a feature. A feature attribute is a typed and named element that can represent any kind of information (according to the values allowed by the type). A feature may have any number of feature attributes.

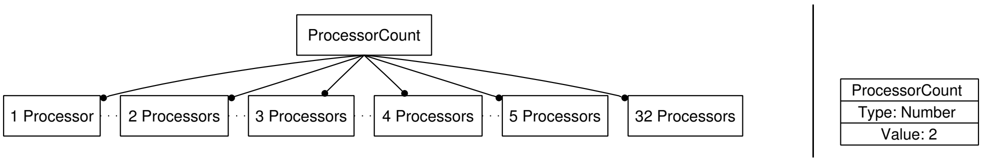


Figure 5.6: Alternative feature group versus feature attribute value

Feature attributes may be global or local. The difference between a global and a local attribute this the visibility of the associated information. The value of a global feature attribute is stored together with the feature inside the feature model and is considered to be part of the feature model. A local feature attribute value is stored in the feature set, so the value may be different in another feature set.

The types currently available for attributes are integer values and strings, but any type (like enumerations or even structures) could be added easily. A feature attribute may have a default value that is used when the feature is selected and no value has been specified.

The availability of a feature attribute to the model evaluation process may be restricted by a guarding restriction. If the restriction evaluates to false, the attribute is considered not to be available and may not be accessed during model evaluation.

Although feature attributes are a deviation from the original feature model approach and some authors argue against such extensions (again [CE00]), careful and selective use of feature attributes makes feature models much more readable and usable. Besides the scenario sketched above, feature attributes can have many different uses. The optional feature deployment model, for example, (Section 5.3.9) makes use of feature attributes to provide information on the deployment characteristics of a feature (binding time, etc.).

The main drawback is that for checking the feature attribute values, the simple requires, conflicts, recommends and discouraged statements are not sufficient. If value checks are necessary, for example to determine whether a value within a given range conflicts with another feature, OCL/Prolog level restrictions are required.

5.2.4 Model Mapping

To connect different feature models, CONSUL uses the model mapping expressions. A mapping expression is basically an implication. The left hand side of the implication describes a condition. If the condition is true, the right hand side specifies a set of features that must be selected. This can be used to simply map feature names or to introduce a feature that represents a combination of features in another model.

The left hand side is expressed using the normal restrictions, the right hand side is a simple list of features. So the mapping expression $has(M1.A)- > M2.Y, M3.Z$ defines that if A

Mapping functions:

```
has(feature('M1.A'))
implies feature('M2.Y')
```

Feature set:
{M1.A }

Mapping graph:

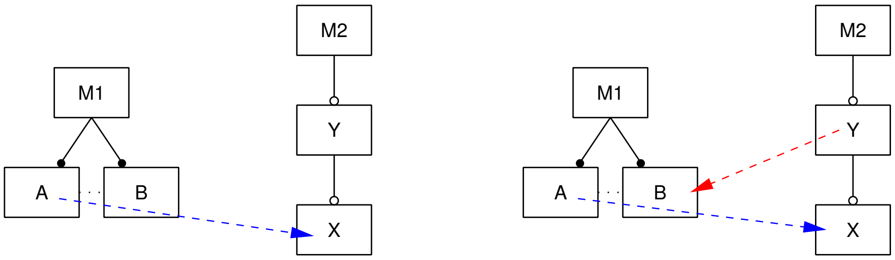
Mapping functions:

```
has(feature('M1.A')
implies feature('M2.Y')

has(feature('M2.X'))
implies feature('M1.B')
```

Feature set:
{M1.A }

Mapping graph:



valid model dependency

invalid model dependency

Figure 5.7: Valid and invalid dependency graph for model mapping

in Model M_1 is in the set of selected features, the features `Y` in Model M_2 and `Z` in Model M_3 should be selected too.

Figure 5.7 shows two different mapping functions for the same feature model and same feature set. The resulting dependency graphs are valid (left side) or invalid (right side, the selection of two alternatives is not possible).

The mapping allows the combination of the advantages of a single model (clarity, user oriented vocabulary) and the advantages of multiple models (reuse, increased model stability). The drawback is the additional effort to create and maintain the mapping functions. Especially the consistency between models is a critical issue. When a model is changed (a feature is removed or renamed for example), all related models with their mapping functions have to be rechecked. To support this, CONSUL is able to trace all changes to a model and makes it possible to detect most changes automatically.

Except for small and non-critical changes like feature renaming, the consistency check cannot be automated and is to be executed by humans. However, most changes do not require immediate action. If the relations of features within a model change due to modified restriction rules or changes in the parent-child feature hierarchy, this can lead to the following situations:

- At least one possible mapping results in an invalid feature selection. If this occurs, the user cannot proceed and has to consult the model developers to fix the problem. This may or may not be acceptable for the user, but at least no invalid selection is possible.
- No possible mapping generates an invalid selection, so there is no problem at all.

An automated check whether there are invalid selections caused by a feature mapping is usually not feasible, because of the high number of possible feature selections[2].

5.2.5 Feature Sets

A feature set describes the specific problem inside the problem domain. It contains the list of selected features and local feature attribute values. The list of selected features records for each feature the reason and the source of its selection. Possible reasons for selection are:

User selected: A user has selected this feature. It will be never removed from the list of selected features except by user request.

Mapped: A mapping expression caused inclusion of this feature. The responsible mapping expression is recorded as source. Once the reason (the mapping restriction evaluates to `true`) is no longer valid, the feature can be removed.

Auto: An automated problem solver included that feature in the list. The feature may be removed when the reason (the detected problem) no longer exists. A typical case is that a `requires`() feature relation caused the problem solver to select a feature. When the `requires`() is no longer present, the feature can be removed from the selection.

Implicit: When a feature is selected, all parent features and its mandatory child features are implicitly selected too. When a parent feature is not already selected by other means, it is recorded as an implicit selection.

5.2.6 Language Representation

The CONSUL feature modeling language uses a partial graphical representation for user interaction. The feature model tree representing the main sub-feature relations (mandatory, optional, alternative, or), for example, is shown in graphical form. Additional information like values and restrictions are shown in textual form. The graphical notation is a slightly

[2] The worst case is that all n features are optional. That leads to 2^n combinations.

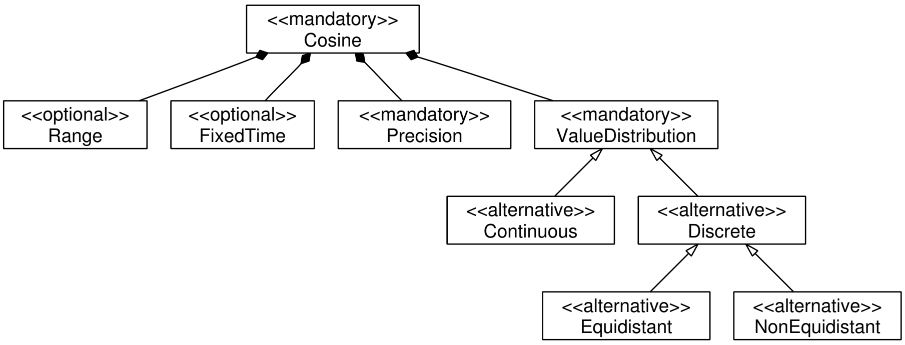


Figure 5.8: UML version of the cosine feature model

modified form of the original FODA style graphical elements and the ideas presented in [CE00].

A UML conform representation is possible, see [Cla01], but does not achieve the compactness of the original notation and makes it more complicated to understand. However, if integration with standard UML tools is required, these notation could be used. Figure 5.8 shows the feature model from Figure 3.5 (page 50) in the proposed UML notation.

The internal CONSUL representation used for storage is based on XML for easy manipulation, exchange and also for simple integration of future extensions.

5.3 Feature-Based Problem Analysis

A good and complete problem analysis is the base for successful software projects. Without precise knowledge about the problems to be solved, it is impossible to build a good solution.

Although these statements are true for any software development, they are *the* key to any kind of family-based software development. Family based development in fact combines a number of related problems and tries to form synergies by reusing parts of solutions in different contexts (that means family members).

The main difficulty of problem analysis is that it is mostly a creative process that involves many soft factors like the knowledge and experience of the people involved (analysts, customers, developers, etc.). It is impossible to define a practicable method that neglects these “human” factors within the problem analysis entirely. Some methodologies leave the issue completely open and just require a “problem analysis” but do not define how to perform it, for example in Extreme Programming (XP). Other methodologies like GenVoca and most domain specific language-based approaches require the result to be a very formalized description of the problem domain in terms of the language grammar.

Like most family-based development approaches, CONSUL lies between these two camps. Depending on the nature of the project, the results have more or less formal character.

The required results are:

- Precise textual problem domain definition (Section 5.3.1).
- Variability and commonality analysis as (set of) feature model(s) (Sections 5.3.7 and 5.3.8).
- Feature sets and composition rules that describe the possible family members based on feature model(s) (Section 5.4).

Additional results can be interaction diagrams between the external environment and family members (sequence charts, timing descriptions etc.), a list of referenced standards, common non-functional requirements, etc. The amount and character varies from project to project and with the problem domain level, see next section. Although these results are also important, their production lies not within the scope of the work presented here.

5.3.1 Problem Domain Definition

The CONSUL *problem domain definition* activity corresponds to the *plan domain* phase of ODM with its three sub phases (set objectives, scope domain and define domain). The results are the domain definition and the initial feature set (starter set) for the domain models.

Instead of repeating the ODM tasks, this section will present only issues specific to embedded projects that have to be included into the problem domain definition activity as well as CONSUL specific issues.

5.3.1.1 Domain Attributes

ODM already assigns attributes to the problem domain (vertical versus horizontal, encapsulated versus diffused, native versus innovative). As already discussed, CONSUL requires the intended family deployment scenario of the problem domain (application family, component family, component implementation family) as part of the domain definition. This characterization is later used to decide on the appropriate way to model the problem domain based on features. This is discussed in detail in the Sections 5.3.7 and 5.3.8.

5.3.1.2 Embedded Specifics

Embedded software is often coupled with specific hardware. To be able to choose the appropriate design and implementations, it is important to define the characteristics of the potential hardware platforms regarding available memory sizes and types (ROM, RAM, flash), processing power, supported hardware devices, etc. precise as possible.

For development on component implementation level, it is often complicated to gather this kind of information, since a component may be used in very different contexts with very different hardware resources. Furthermore, the actual share of a single component implementation of resource usage is often not very large, especially if a high number of components is involved. Usually, the goal on component implementation level is just resource efficiency without specific resource constraints.

But the higher the family level, the more important precise knowledge about hardware limits gets. For an application family, it can be crucial to know the limits to decide about the design or even the domain model, to decide, for instance, whether to reuse existing models/design/implementations with known resource usage.

5.3.2 Feature Starter Set

Building a feature model for the problem domain requires knowledge about which features represent the commonalities and variabilities in a domain. Candidates for these commonalities and variabilities can be derived from the domain definition. The boundary definition, which allows to decide what is inside a domain and what not, has to describe features common to all members of the domain.

Variable features can be derived from the initial description of different family members. A feature that is present in some but not all family members is a variable feature.

These lists of common and variable features are used as a starting point for building a feature model in the next step, the domain modeling activity. The feature lists do not need to be complete but rather represent the knowledge at this point. During later activities, more domain knowledge is gathered and is used to refine the model and possibly the domain definition as well.

5.3.3 From Problem to Model

The process to build a feature model for a problem domain is strongly influenced by the characteristics of the domain. An innovative domain has to be handled differently than a native domain, because the knowledge of the domain analysts about the domain will probably change to a greater extent for an innovative domain than for a native domain where the analysts have a lot of experience.

Another important issue is that there is not a single "valid" model for a given problem domain, but there exists a large number of possible models, even many good ones. The CONSUL methodology, as all other methodologies, can only provide guidelines to produce models but will not give a guarantee that the produced model is a good model.

5.3.4 Building a Feature Model

What is a feature? The definition of a feature as given in Section 3.6.2.1 states that a feature is something visible to the end user. Thus finding a feature requires a knowledge about the end user. The CONSUL domain definition allows to identify the intended end user by looking at the domain level. Each domain level (application family, component family, component implementation) has a different end user and in turn has a different specialized definition of a feature.

Application family features: The end user of an application family is someone who wants to chose between the different members of the family according to their functional characteristics without bothering how a member is realized. A feature describes a *what* not a *how*. A feature can summarize certain different functionalities when they are used as a package in the application family.

As an example, Figure 5.9 shows a feature model for a web browser application family with its application features. In the figure, there are only a small number of features present, describing two different main operation modes of the browser (text versus graphic) and the possible alternative operating systems platforms. There are many more requirements for the browsers represented by this family, but they are part of the common base and therefore not present in the feature model.

Component family features: End users of component families are application engineers who want to build applications from a set of component families. So features are the functionalities provided by the component family and also non-functional properties important for application development in the given domain. Each functional and non-functional discriminator should be represented as an independent feature.

Feature relations should be independent from component implementation issues. A relation between two features should indicate only general relations, not a relation that is only valid for the available implementations of the component family. Otherwise, changes to the implementation would force changes to the component family model, which should be avoided.

A fictive HTML viewer component family model is given in Figure 5.10. It exposes a lot more details than an application family, because a component family should be as flexibly reusable as possible by providing a scalable set of functionalities.

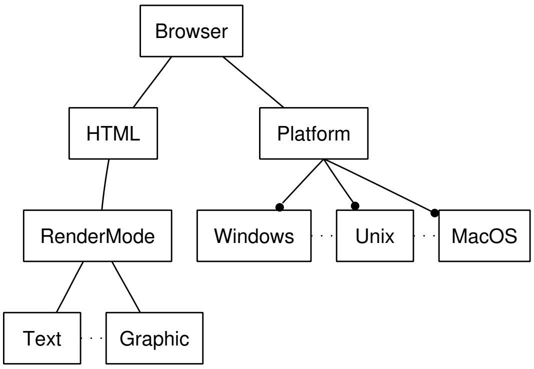


Figure 5.9: Feature model of a web browser application family

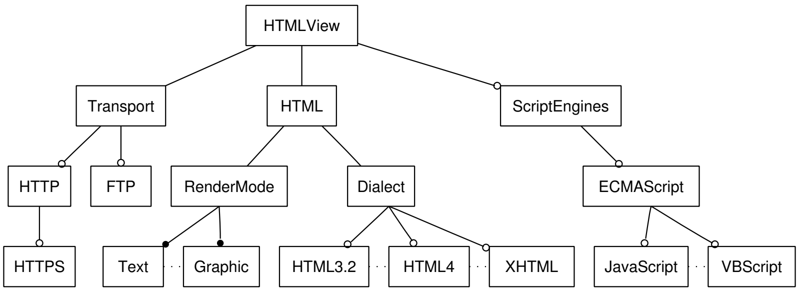


Figure 5.10: Feature model of an HTML viewer component family

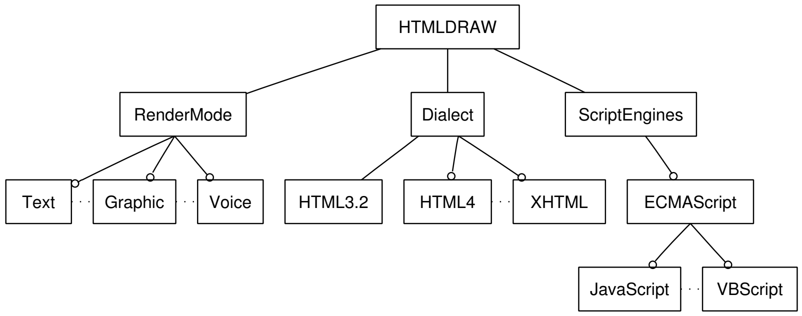


Figure 5.11: Feature model of an HTML drawing component implementation family

Component implementation family feature: Components are integrated in one or more component families, therefore their end users are the implementors of component families. Features expose the functional and non-functional implementation details like the implementation languages used or component interface standards supported.

In contrast to the feature models in the other levels, component implementation models are tied to a specific component implementation and may change if the component implementation changes.

Figure 5.11 shows a feature model for an HTML drawing component that has different implementations for different output media like graphical screen, text mode or even voice mode.

5.3.5 Feature Types versus Feature Relations

A feature may have several relations to different features but can have just one parent feature and feature type. Therefore, any other relation has to be expressed by feature relations.

The decision which relation to express by a parent/child feature relation, is influenced by a number of factors, like the probability of a relation change and the position of the feature within the model.

The lower the probability that the relation between two features changes, the better is it to express this relation by the parent/child relation. Feature relations with a high likelihood of change should be expressed as restrictions, because it is much easier to change a feature relation than to change a parent/child relation. This is especially relevant if the child is not a leaf feature but a parent itself.

During the feature finding activity and model building, all known relations should be marked as *stable* or *instable*. For model building, the stable relations are used for determining the parent/child structure first. Remaining stable and instable relations are added as feature relations later on.

5.3.6 From Feature Relations to a Feature Model

Building a feature model is an incremental process. A feature model as central element in the development of a family may be changed several times due to new insights. To simplify the building of the first version of a feature model, the following algorithm has been successfully used in CONSUL-based projects.

The starting point is to use the list of features from the starters set. Starting with this list, a model can be built executing the following steps:

1. List all known features and mark the features as stable or instable.
2. List all known relations between features and mark them as stable or instable.
3. Identify stable groups of alternative features and or features.
4. Find all stable features/feature groups that have no stable relation to other stable features and model them as optional or mandatory features of the top-level feature representing the complete domain.
5. Find all features/feature groups that have just one stable relation to a feature in the model or have only relations to features that have a parent/child relation. Integrate them as child feature(s). Repeat until no more feature/feature groups with such relations are available.
6. Decide which stable relation for the remaining stable features should be expressed via a parent/child relation and repeat the steps 4 to 6.
7. Repeat steps 4 to 6 for instable features with their stable relations.
8. Express instable relations as restrictions.
9. Review of feature relations.

This algorithm provides an initial feature model that is the base for further refinements and optimizations. Possible refinements are the introduction of mandatory parent features for groups or subgroups of features to provide better readability and understandability of the model. An optimization/improvement that is often possible, is the replacement of an alternative group of just two features by an optional feature representing a specialization of the parent's feature.

Leaf features or nodes with a small set of children are much easier to remove or change without causing problems than a feature within the tree. The distinction between stable and instable features and the delayed addition of instable features and relation positions helps to increases the model stability by positioning instable features closer to the outer border of the feature model.

Example: Cosine component implementation domain The domain analysis provided the following list of starter features:

`Cosine` The top level feature is the domain itself.

`FixedTime` A feature that requires the implementation to execute in a deterministic way and thus to provide a known time limit.

`Precision` A feature that describes the required precision of a calculated cosine value. It is a feature with a value and therefore represents in fact an alternative group of all possible precision values.

`EquidistantAngleValues` All input values can be expressed as $n*baseangle$ where $baseangle$ is a constant and n an integer. The feature value represents the $baseangle$.

`NonEquidistantAngleValues` Possible input values are not expressible in the form $n * baseangle$. The feature value represents the number of possible input values.

The features `Precision` and `FixedTime` are not likely to change as the component is intended for the real-time domain that relies on both features. The two other features are initially marked as instable, because it is not yet known if they are important at implementation level. These two features are mutually exclusive and therefore have a stable conflict relation with each other.

Figure 5.12 shows the resulting initial feature model after applying the presented algorithm to the initial list of features. The starting model consists of `Cosine` only. In the next step, `FixedTime` is added as an optional feature and `Precision` as a mandatory feature. There are no more stable features available. The features `EquidistantAngleValues` and `NonEquidistantAngleValues` form a group of alternative features where both members are instable. This group is now added as children to the root feature, since there are no other relations between the group's features and the other features.

Obviously, the model differs from the model in Figure 3.5 (page 50). This is not surprising as the model here is built on the initial knowledge from the problem analysis. Refinements of the initial model and gathering additional knowledge led to the final feature model. For improved readability, for example, the mandatory feature `Value distribution` with

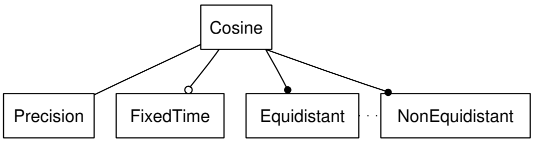


Figure 5.12: Building of the initial model for the Cosine domain

the alternative group as child was introduced. Additional investigations of the characteristics of the implementation led to the introduction of an additional alternative for the value distribution `Continuous` and an additional structuring mandatory feature `Discrete`. The optional feature `Range` is introduced, because the resource usage of some of the implementations used is influenced by the angle range for which the values have to be calculated. Again, this is a feature with a value attached that represents the numerical range of angles.

5.3.7 Single Domain Model Approach

All feature model-based domain analysis approaches use a single feature model to describe the domain of interest. CONSUL supports this approach as well. However, since using a single model approach has several limitations, CONSUL supports also domain models based on more than one feature model.

The single model approach as described above, has problems when it comes to unexpected changes of core features in the model. This can happen for innovative domains, if for example the domain knowledge is not yet mature and evolves a lot during the development activities.

Reuse of the feature model is also limited in a single domain approach. The feature model tries to represent a given domain as closely as possible. The usage of such a specialized model in a different context, which means in a different domain, is problematic.

Single model approaches are good if

- The domain is very well known and analyzed by the model designers.
- The domain borders are fixed and not likely to change.
- The model will not be reused in other contexts.

Those three requirements are often met in application family development. An application family model is not likely to be reused and it should be easy to define its borders. Components implementation families, too, are often relatively static and small enough to be completely analyzed by the domain analysts.

A good example is the domain of real-time run-time systems for which it is relatively easy to model the domain with a single model, because the requirements within this domain are stable and well analyzed by many experts. Section 6.2 gives a detailed review on building a domain model for this domain.

Difficulties arise if the conditions given above are not met, since it is then problematic to find an exact and stable description of commonalities and variabilities expressed in a single model.

5.3.8 Multiple Domain Model Approach

It is not always easy to build a stable feature model for a given domain. Some reasons have been discussed in the previous section. One other reason is the complexity of models for larger domains with a high number of features (>100). Such domains often exhibit complex relations between features as well.

Such models are hard to develop, hard to maintain and hard to reuse. The problems sketched above are not unique to feature models but occur wherever complex scenarios are to be represented. The most common software engineering approach to handle those problems is decomposition into smaller problems that are easier to handle. The *multi domain model* approach of CONSUL allows just that for feature models, unlike other approaches like FODA, FORM or FeatuRSEB that use only a single feature model.

CONSUL supports two different decomposition approaches based on multiple feature models.

5.3.8.1 Domain Decomposition

Closer examination of a problem domain will always reveal that it is in fact composed of a number of smaller subdomains with a specialized focus. Of course, different decompositions are possible, because some features may belong to more than one domain.

Looking at the HTML viewer component family example already introduced (Figure 5.10), a sample decomposition is given in Figure 5.13. Most features of the model belong to more general problem domains. Four domains have been identified here. The `File Access`, `Document Rendering`, `Document Representation` and `Scripting Languages` domains contribute to the HTML viewer domain. The HTML viewer domain is an intersection of these domains. However, the relations between features within the intersected domains remain the same or are more restricted. For example in the general model a feature HTTP is probably an option or part of an or-feature group. In the HTML viewer component it is mandatory. But HTTPS is a sub-feature of HTTP in both models. Additionally more domain models may contain more features (shown with different color in Figure 5.13), since they covers different problems not relevant in the HTML viewer case.

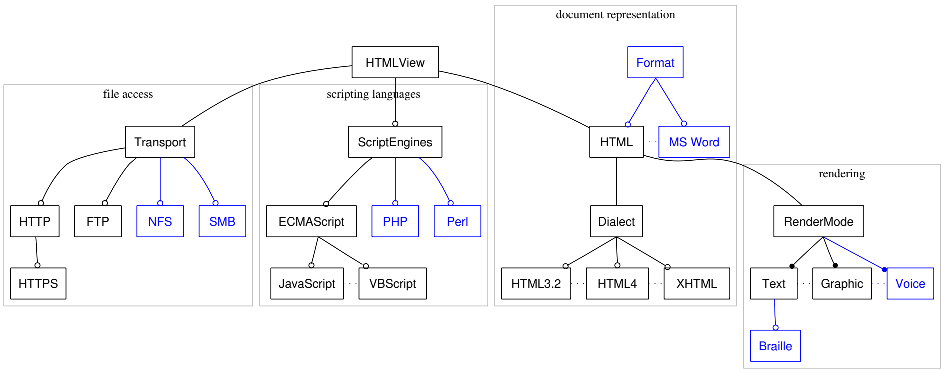


Figure 5.13: Decomposition of a domain: HTML viewer component family domain

```
feature('ScriptingLanguages.JavaScript') implies
 conflicts(feature('DocumentRepresentation.HTML3_2'))
```

Figure 5.14: Sample cross model feature relation for decomposed HTML viewer domains

Such a decomposition seems to have not much use in the first place, but the benefits become obvious when a similar problem is to be solved and therefore to be modeled. For a word processor a similar model could be used based on the same four domains but with a different intersection result. For example a different document representation would be probably used, but the rendering domain would remain the same.

As long as those models are independent from each other, their handling is easy. Often, there will be some kind of relationships between these models, although it is advisable to minimize these relations. CONSUL supports the connection of different models via their feature relations and model mapping expressions.

Instead of referring to features local to the model, it is also possible to include features from other models in the relation expression. For this, to each model a unique name is assigned that is used as prefix for its feature names. Figure 5.14 shows an exemplary rule for the domain based on the decomposition given in Figure 5.13. The left-hand side, before the colon, shows the name of the feature to which the relation expression is attached, the right-hand side shows the actual relation. In this example, the feature `JavaScript` of the `ScriptingLanguages` feature model conflicts with the feature `HTML3_2` in the model `DocumentRepresentation`, that means that they cannot selected for the same feature set.

```
has(feature(’HTMLViewer.Text’)
  implies feature(’DocRendering.Text’)
has(feature(’HTMLViewer.Graphic’))
  implies feature(’DocRendering.Graphic’)
has(feature(’HTMLViewer.HTML3_2’))
  implies feature(’DocRepresentation.HTML3_2’)
...
```

Figure 5.15: Mapping the HTML viewer domain

5.3.8.2 Domain Composition

Decomposing the domain of interest into a number of smaller domains makes it more maintainable, easier to change and also easier to reuse, but on the flip-side is the loss of clarity for the family deployer. With just a single model, it is easier to visualize the most important relations between features as a parent/child relation. Between independent models no such relation exists. Additionally, when models are reused, the vocabulary (feature) used in one or more of the models may not be the desired one. There could be many features in a domain, for example, that have no relevance to the specific case. There are two options: the use of a modified feature model, or the hiding of the problems.

Depending on the situation, one approach has to be chosen. The use of a new model needs no further explanation and also no special support by CONSUL.

CONSUL supports the later approach by *feature mapping*.

The example in Figure 5.15 shows some mapping functions of the original HTML viewer domain to the two other domain models.

5.3.8.3 Hierarchical Domain Models

The combination of the multi domain modeling with CONSUL’s three levels of software family domains presented in Section 5.1.1, leads to a hierarchal domain model structure.

The starting point is a domain model for the application family that may be mapped to other, reusable domain models on the application family level. These models are in turn mapped to the models that describe the component families used to implement the application families. On the lowest level, the component family models are mapped to component specific models.

Figure 5.16 shows the previously introduced HTML browser application family with its feature model hierarchies. In this example, only the component family `HTML Viewer`

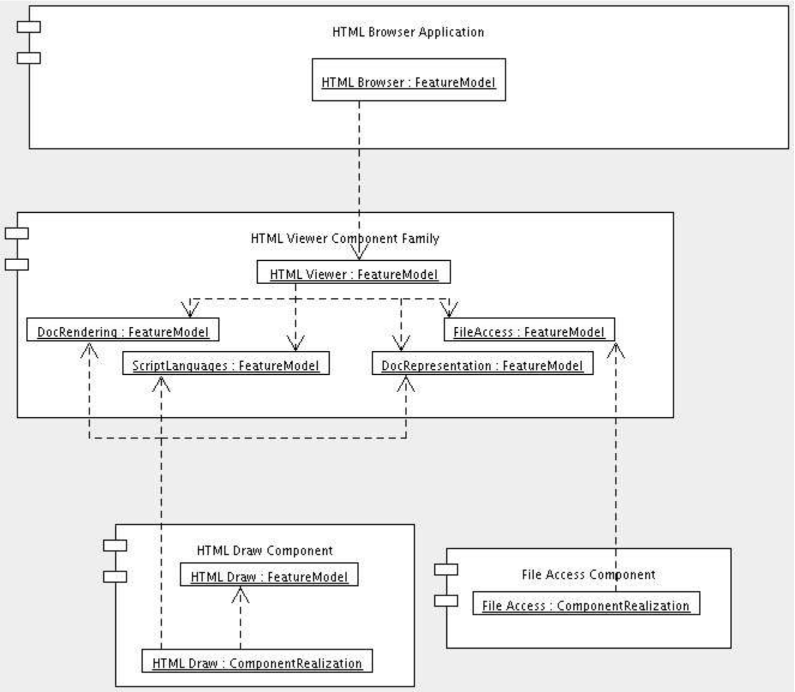

HTML Browser Application
HTML Browser : FeatureModel
HTML Viewer Component Family
HTML Viewer : FeatureModel
DocRendering : FeatureModel
ScriptLanguages : FeatureModel
DocRepresentation : FeatureModel
FileAccess : FeatureModel
HTML Draw Component
HTML Draw : FeatureModel
HTML Draw : ComponentRealization
File Access Component
File Access : ComponentRealization

Figure 5.16: Structure of hierarchical domain models

uses multiple models. The features selected in the `HTML Browser` are mapped to the viewer model that in turn is mapped to the four sub-models (mapping expressions (Figure 5.17)). The mapping expressions for the sub-model mapping use some special expressions. The expression $has(feature('Model1:*'))$ is expanded to match several expressions, one for each feature in the model. The $feature('Model2.\%')$ is the counterpart and references the matched feature on the left-hand side. Thus, if both parts are combined, the expression maps each feature of the first model to the same feature of the second model.

This top-down view on the model structures is only possible after the completion of the application family development. In reality, the development of an application family from reusable component families is composed of two different, complementary processes: The top-down process that analyzes the application domain and builds/reuses models for it, and the bottom-up process of matching existing reusable component families against the application domain requirements. The same applies to the component family/component levels. The result of these parallel processes is the completed, hierarchical model structure.

Compared to approaches that rely on a single model, the hierarchical model approach of CONSUL allows easier integration of existing software families into new families. Using a single model for each application family would force the developers of component families to maintain the features relevant to their component family in all application domain models. With feature model mapping, only the links between the models have to be maintained.

5.3.9 Feature Deployment Model

For realizing efficient family designs and implementations, it is not sufficient to build feature models representing the functional and non-functional properties of a problem domain. Usually, more design and implementation specific information is required to support the design process and allow more detailed selection of the properties of family members, like binding time of variations, influence of a feature to the system, etc. The use of this kind of information can result in more efficient implementations. Other feature model-based methodologies lack support for a uniform representation of such information. The CONSUL method introduces a new additional model, the feature deployment model, to solve this problem.

The feature model captures *which* features represent variation points within the family, the *family deployment model* (FDM) is used to express *how* the variability has to be present in the family and in a specific family member.

The aggregated information builds an additional model used in the design and deployment of a family. It is a separate model that is derived from the feature model and contains additional information about the problem domain and design constraints. If such a model would be an integral part of the feature model itself, it would limit the reuse of the feature model, since it is possible to build many feature deployment models for the same feature

```
HTML Browser -> HTML Viewer

has(feature(’Browser.Text’))
implies feature(’HTMLViewer.Text’)

has(feature(’Browser.Graphic’))
implies feature(’HTMLViewer.Graphic’)

has(feature(’Browser.Windows’))
implies
feature(’HTMLViewer.VBScript’)

has(feature(’Browser.MacOS’)) || has(feature(’Browser.Linux’))
implies feature(’HTMLViewer.JavaScript’)

true
implies feature(’HTMLViewer.{HTML3.2,HTML4,XHTML,HTTPS}’)
```

```
HTML Viewer -> sub-models

has(feature(’Viewer.Text’))
implies feature(’DocRendering.Text’)
has(feature(’Viewer.Graphic’))
implies feature(’DocRendering.Graphic’)

has(feature(’Viewer.ScriptLanguages:*’))
implies feature(’ScriptLanguages.%’)

has(feature(’Viewer.DocRepresentation.*’))
implies feature(’DocRepresentation.%’)

has(feature(’Viewer.FileAccess:*’))
implies feature(’FileAccess.%’)
```

Figure 5.17: Mapping the HTML browser domain

model. Therefore, the FDM is associated with a concrete design. For the same feature models with a different design, there can be a different FDM necessary.

The FDM lies at the border between domain analysis and domain design, as it reflects design related issues from the perspective of the domain analysis.

An FDM describes for each feature the impact when and how it is introduced into a family member. The following list describes the elements of a feature deployment model entry:

1. Locality of change

   Self contained: A self contained feature does not require changes to the rest of the system, it is implemented locally.

   Cross-cutting: The implementation of a cross-cutting feature needs modifications in many places of a system. For example if the feature `synchronization` is selected for an operating systems, this requires changes in all places where synchronization has to be used.

2. Kind of change

   Addition: The implementation of the feature does not require modifications to other parts of the system. The introduction of a new class or even a new component are examples for such a feature implementation.

   Modification: To realize the feature, it is necessary to change existing parts of the system. A modifying feature can be realized as a wrapper to existing parts of the system for example by an overwritten method in an object-orientend design. The modification type can be further subdivided into

   - **Wrapping:** All existing parts remain unchanged. The changes are realized by providing a wrapper/adapter that changes the system or parts of the system but still uses the already existing parts.
   - **Replacement:** The system is modified by replacing parts of the system with new parts with different behavior/requirements/characteristics.

3. Time of binding

   Compile time: The variation expressed by this feature is resolved at compile time.

   Load time: The variation expressed by this feature is resolved at load time.

   Run time: The variation expressed by this feature is resolved at run time. If combined with the property “dynamic”, the feature may be loaded and unloaded during run time.

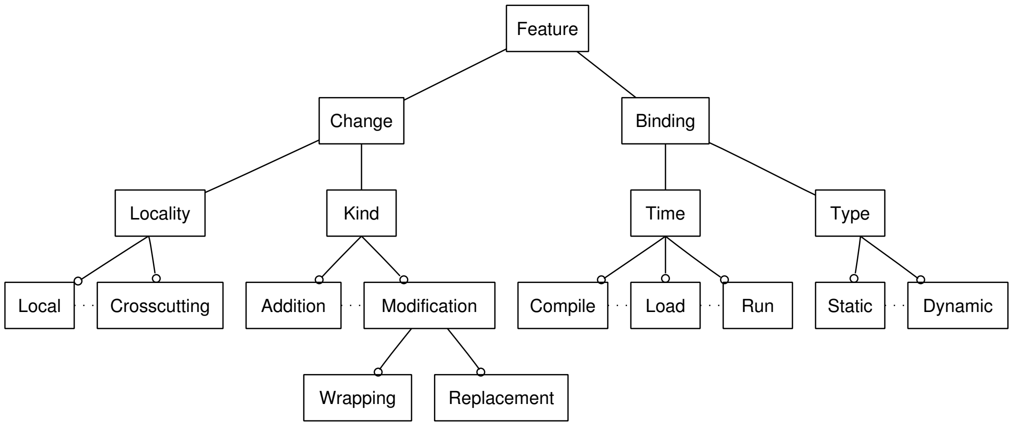


Figure 5.18: Feature model of design criteria derived from a feature

4. Binding type

   Static: The feature is present in the resulting system and used all the time.

   Dynamic: The feature is not required to be present and usable during the whole run time of a system.

Figure 5.18 shows these criteria in form of a feature model. This model illustrates that any combination of feature criteria is possible in principle. The analysis of these criteria has to be performed for each possible family member, because the same feature may influence different family members in a different way. Especially the binding time may vary. In some cases, the binding can be done at compile time, in other cases it must be done at run time.

In the CONSUL tools, the FDM is technically realized as predefined feature attributes that are associated with each feature.

Table 5.1 shows the model for the cosine example domain and the design used in Chapter 2.

The underlying feature model of the FDM (see Figure 5.18) can be extended by more features according to the needs of the concrete design and implementation methodology and strategy. In a large scale commercial development, for instance, it might be relevant to exclude features from being released in products, but they should be present in the model. To express this the feature, deployment model could be extended by an optional feature “not for release”.

The FDM properties of the features of a feature model can be specified by different stakeholders and at different times. It may be possible to fix all properties for the features during

Features	Criteria
Cosine, Range, Precision, FixedTime	cross-cutting, compile time, static, modification
ValueDistribution, Continuous, Discrete, Equidistant, NonEquidistant	self contained, compile time, static, addition

Table 5.1: Feature deployment model for the cosine example implementation

design and implementation time of the family as in the example given above. It is also possible, however, to set some or even all properties during the application analysis when the application engineer decides on the feature selection for the desired family member. This usually requires a much more flexible design and implementation but allows a very fine-grained adaptation of the family member to the needs of the application.

5.4 CONSUL Component Models

The representation of solution domains is a crucial factor in the design process of software families. Existing standard software description techniques, namely UML, do not support the description of variable software family architectures [SRPC02]. It was necessary to develop an additional modeling language for that purpose.

CONSUL aims to support component-based family design in a way that is open to any component architecture. For CONSUL a component encapsulates a configurable set of functionalities. CONSUL supports the description of components as configuration entities but does not rely on a specific component framework like CORBA or COM. Figure 5.19 gives an overview over the hierarchical structure of the component-based family model supported by CONSUL. The concrete component model has the same basic structure but uses only a subset of the component family modeling language, since it describes instances of the family model with all variation points removed. Therefore, the following description applies to both, in the concrete component model, however, there are no restrictions allowed/generated.

As a consequence of its openness to any component model, CONSUL does not check interfaces of connected components by itself. To provide such a functionality, it is possible to introduce user-definable checks appropriate for the intended framework/architecture into the CONSUL tool chain.

This approach is reflected in the CONSUL component family model that mainly describes the internal component structure of a family and its configuration dependencies. The model is complementary to models like OMG's CORBA IDL or Microsoft's COM IDL that focus on the external view of a component.

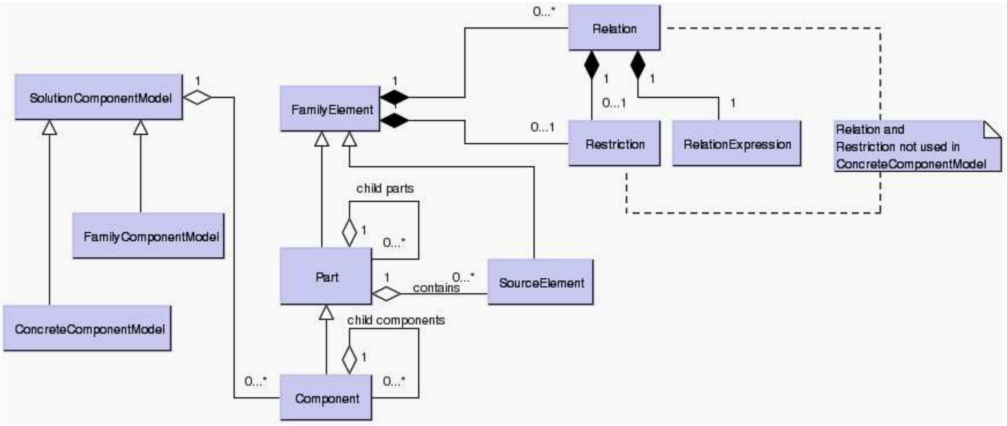


Figure 5.19: Structure of the CONSUL solution component models

However, different configurations of a CONSUL component can also provide different interface descriptions, because functional changes often, but not always are also reflected in the external interfaces of the components. In this case, the CONSUL component description language allows to provide the support for managing these changes.

A small example of the language[3] is given in Figure 5.20. It shows a simple component realizing the cosine example domain with three different implementation files. Depending on the selected features, the appropriate `cosine_?.cc` implementation file is chosen. If there are no real-time constraints to be meet, for example, the implementation in `cosine_1.cc` is selected, regardless of the other feature's settings. This is correct, because the general-purpose implementation meets all functional requirements (precision, works for any angle, etc.) and it is not necessary to optimize the run time of the cosine function in a non real-time environment. In real-time contexts the implementation is selected according to the input characteristics of the angle values. It is obvious that when this small family is used in conjunction with the cosine domain feature model presented in Figure 3.5 (page 50), there are several features that are not used for configuration purposes. This is permitted as long as the implementations do not conflict with any of the selected but unused features. It is not permitted, for example, to provide just a single implementation like the one found in `cosine_1.cc` , because this implementation would conflict with any feature selection that includes `FixedTime`.

[3] In the example Prolog is used instead of OCL for restrictions, but OCL could be used too.

```
Component("Cosine") {
  Description("Efficient cosine implementations")
  Parts {
     function("Cosine") {
       Sources {
         file("include", "cosine.h",def)

         file("src", "cosine_1.cc",impl) {
           Restrictions { Prolog("not(has_feature('FixedTime',_NT))")}}

         file("src", "cosine_2.cc",impl) {
           Restrictions { Prolog("has_feature('FixedTime',_NT),
                                  has_feature('NonEquidistant',_NT")}}

         file("src", "cosine_3.cc",impl) {
           Restrictions { Prolog("has_feature('FixedTime',_NT),
                                  has_feature('Equidistant',_NT")}}
       }
    }
  }
  Restrictions { Prolog("has_feature('Cosine',_NT)") }
}
```

Figure 5.20: (Simplified) component description for cosine component

5.4.1 CONSUL Family Model Structure

The CONSUL family model represents a family as a set of related components. The inter-component relation of these components is not fixed. That means that both hierarchical component structures like the OpenComponent model [GSPS01b] or normal independent components can be modeled by a family model. The CONSUL family description language (CFDL) is the textual representation of the model.

Components: A component is a named entity. Each component is hierarchically structured into *parts* that in turn are built from *source elements*. A component may specify a parent component. A component is only included in the concrete model if its parent component is included.

Parts: Parts are named and typed entities. Each part belongs to exactly one component and consists of any number of *source elements*.

A part can be an element of a programming language like a class or an object but also any other key element of the inner and external structure of a component, for example an interface description. CONSUL provides a number of predefined part types, like `class`, `object`, `flag`, `classalias` or `variable`. The model is open to the introduction of new part types, depending on the needs of the users. Table 5.2 gives a short description of the available part types in the current CFDL version.

Source elements: A part as a logical element needs some physical representation(s) that are described by the *source elements*. A source element is an unnamed but typed element. The type of a source element is used to determine the way to generate the source code for the specified element. Different types of source elements are supported, like `file` that simply copies a file from one place to a specified destination. Some source elements are more sophisticated like the `classaliasfile`. Table 5.3 lists the currently available source element representations.

The actual interpretation of these source elements is handed over to the CONSUL transformation backend. To allow the introduction of custom source elements and generator rules, CONSUL is able to plug in different transformation engines that interpret the generated concrete component model and produce a physical system representation from it.

Currently, two different transformation engines exist. One is implemented in Prolog and operates directly on the Prolog knowledge database representation. The second uses an XML based approach.

The advantage of the Prolog-based approach is its speed and the ability to use the power of Prolog. However, it requires a decent knowledge of Prolog to change or add source element

Part type	Description
`interface` (X)	Represents an external component interface X.
`class` (X)	Represents a class X with its interface(s), attributes and source code.
`object` (X)	Represents an object X.
`classalias` (X)	Represents a type-based variation point in a component. A classalias is an abstract type name that is bound to a concrete class during configuration.
`flag` (X)	Represents a configuration decision. X is bound to a concrete value during the configuration. Depending on the physical representation chosen for the flag, it can be represented as a makefile variable, variable inside a class or a preprocessor flag.
`variable` (X)	Similar to a flag, but a variable should not represent direct variation points rather additional configuration information like buffer sizes or the number of resources available.
`project`(X)	Represents anything that cannot be described by the part types given above.

Table 5.2: Overview of CFDL part types

generators. The other approach, described in [Roe02], uses XML to describe the transformations. This allows users to integrate own special-purpose modules into the systems via an easy-to-use module concept. This enables users to introduce their own family specific transformation engines without any need to change the core CONSUL tools.

Using restrictions in CFDL: A key element, which makes the CFDL different from other component description languages, is the support of flexible rules for the inclusion of components, parts and sources. For each of these elements, rules of inclusion can be specified by `Restrictions`.

Each element may have any number of restrictions. At least one of them has to be true to include the element into the system. If there is no restriction specified, an element is always included.

5.5 Domain Design and Implementation

Family design is design with change and for change. Design with change refers to the known and anticipated changes within the family that represent the differences between family members. Design for changes refers to the changes that may happen when more domain knowledge is gathered and/or the domain scope is changed.

Source element	Description	Part types
`file` (Dir,$Name$:,$Type$)	Represents a file that is used unmodified. Dir is the name of the target directory, $Name$: the filename, and $Type$ can be used to describe the type of the file, e.g. if it contains an implementation (`impl`) or a definition of a class (`def`.) For unknown types the special type `misc` should be used.	any
`file` (Dir,$Name$:,$Type$,$OSource$)	Same as the previous `file`, but the original file can have a different name specified via $OSource$.	any
`flagfile` (Dir,$FileName$,$Name$:)	Represents a C++ preprocessor flag $Name$: stored in the file $FileName$ in directory Dir. This is a generated source, it is produced by CONSUL during configuration. The value of the flag is determined by `Value` statements given inside the part description.	`flag`, `variable`, `classalias`
`makefile` (Dir,$FileName$,$Name$:)	Represents a makefile variable $Name$: stored in the file $FileName$ in directory Dir. This is a generated source, it is produced by CONSUL during configuration. The value of the variable is determined by `Value` statements given inside the part description.	`flag`, `variable`, `classalias`
`classalias` (Dir,$FileName$,$Name$:)	Represents a C++ typedef variable $Name$: stored in the file $FileName$ in directory Dir. This is a generated source, it is produced by CONSUL during configuration. The value of the flag is determined by `Value` statements given inside the part description.	`flag`, `variable`, `classalias`

Table 5.3: Overview of CFDL source element representations

Family design requires additional flexibility incorporated into the design in contrast to single system design where each design represents just the equivalent of a family member.

Not only for embedded systems is it important that this flexibility in the family design does not cause excessive resource usage in the target system. Flexible design parts, where the flexibility can be removed prior to run time should not impose any run-time overhead.

5.5.1 Variable Designs

Although it is not possible to derive the family design from the results of the design phase automatically, especially the feature model can help to find appropriate designs for the given problem domain.

CONSUL itself does not require the developers to follow any special design approach. The developers can use any design strategy suitable for the concrete family domain. However, a detailed analysis of the feature model and feature deployment model helps to find a matching design for the problem domain.

Flexibility always comes at a price, which is one of the reasons why many component architectures are not suitable for embedded projects. In the attempt to support all kinds of problem domains with just one implementation, the flexibility of those architectures is usually heavily based on dynamic binding at run time. This introduces much overhead, since dynamic changes are usually not required for every part of the system. The complete dynamic approach is often used anyway, since, from a pure functional point of view, this is equivalent to a design that is (at least) partially static.

Especially the feature deployment model allows to reflect on the required levels of flexibility of the design, which is very important especially for embedded projects. The availability of those models enables the developers to build designs with just the amount of flexibility required, because for each feature the binding time for the variation is known in advance. The rule of thumb is that the earlier the binding is done, the lower the cost in terms of memory and processor cycles at run time.

But even if these binding times are determined for each feature during the domain analysis, the solution design does not necessarily implement them unconditionally. It is often possible to delay binding of a feature to a later time without breaking the functionality of the system at the expense of speed and memory usage.

5.5.2 Family Variation versus Family Member Flexibility

One of the main problems of family-based software designs is that, with a family-based approach, there are two levels of flexibility or variation in the design. On one hand, the “usual”

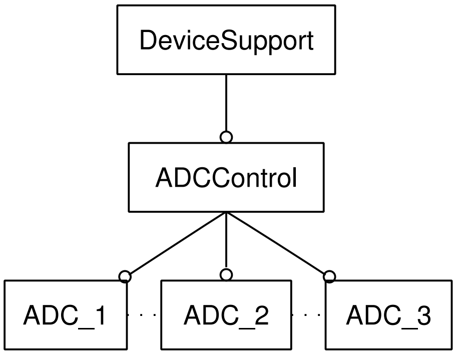


Figure 5.21: Partial feature model for the ADC example

flexibility a family member or a single application has to provide and, on the other hand, the variation inside the family to provide different family members. Both levels cannot be totally separated in a design. The same design can often represent both family variation and member flexibility.

The following example will illustrate this problem and also give an idea how CONSUL can be used to deal with it in a way that suits the needs of embedded systems.

An important service of any operating system is to provide access to the hardware connected to the processor. Depending on the hardware configuration and/or the needs of the application software, an operating system has to provide software components and interfaces to different sets of hardware devices. In embedded systems this application centric view is often found, since software support for unused hardware devices fills only memory in the best case but has an impact on performance or stability of the system in the worst case. Therefore, a (minimal) family approach is almost a must for device drivers and operating systems for embedded systems.

Example Scenario: The example is based on a fictitious hardware that has three different types of analog/digital converters (ADC) available. The goal is to provide a software design and implementation that is scalable without having to program different versions of the device drivers for different family members. The scalability shall be achieved by using the services of CONSUL.

Figure 5.21 shows the relevant part of the feature model. When ADC support via the feature `ADCControl` is selected, any combination of support for the three different ADC types can be requested. Therefore, there are seven combinations[4] of functional support for ADCs

[4] Three with a single ADC, three with two ADCs out of three ADCs, and one with all three ADCs.

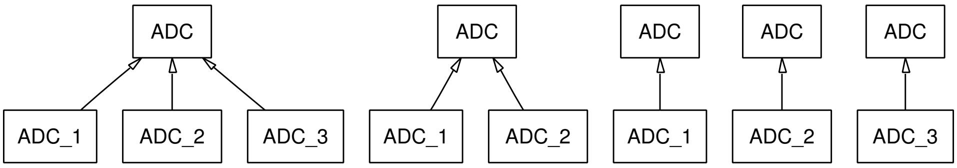


Figure 5.22: Class hierarchies for different members

possible. In some application it is known in advance which ADC(s) are going to be used, so compile-time binding should be possible, but there can be also applications that will bind an ADC at load time, and some will defer the decision until run time and may request access to different ADCs over the time.

The drivers shall be realized within a single component. All ADCs must provide the same interface to be able to switch between different ADCs easily.

Domain Design and Implementation: This setting seems to be a classical example for the use of an abstract base class, defining the common interface and three different subclasses for the concrete implementations of the interface. In many configurations as shown in Figure 5.22, however, the base class is not necessary, as there is only one class derived from it in use. Though the use of abstract base classes is good for modeling and communicating interfaces to users and developers, it also uses resources during run time. To implement the variability, C++ as well as other object-oriented languages rely on tables associated with each class. Each table stores the location of the method implementations for the common interface. In C++ these tables are usually called *virtual method tables*. The use of such tables costs memory for storing the table and also run time, since for each call to an abstract method the corresponding table is consulted.

The measurements for an abstract/concrete class pair with just one virtual method (see Table 5.4[5]) clearly prove the increased memory usage. Of special importance is the use of data memory. Without virtual methods, no data memory is used. Many embedded microcontrollers have separate code and data memories. The data memory is often very small (few bytes to one kByte), so wasting a few dozen bytes of data memory can be a real problem, especially since there can be many parts in the system that look alike and may also use virtual methods. A skilled embedded programmer will avoid using virtual methods whenever possible[6]. In the proposed family, the family members that provide support for only a single ADC controller should avoid them as well.

[5] Compiler: gcc 2.96 for x86, gcc 2.95.2 for avr.

[6] Today, most do that by not even using object-oriented languages for embedded systems programming.

Hierarchy	Processor	Code/Bytes	Data/Bytes
non-virtual	x86	32	0
virtual	x86	206	140
non-virtual	AVR90Sxxxx	80	0
virtual	AVR90Sxxxx	284	42

Table 5.4: Memory consumption of abstract and non-abstract classes

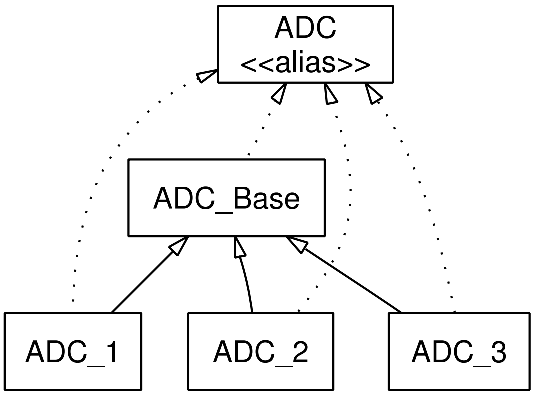


Figure 5.23: Variable class hierarchy for ADC component

CONSUL provides the `classalias` to allow description of flexible, statically changeable class relations. Figure 5.23 shows a new class hierarchy where the external component interface `ADC` can be mapped to any of the `ADC_?` classes.

The corresponding component description is shown in Figure 5.24. The class to which the alias should be set is determined by the four `Value` statements given inside the `classalias` definition. The first statement for which the rule given as second argument evaluates to `true` is used to calculate the value. The first argument of this statement sets the value. The CONSUL PROLOG clause `is_single(X,_NT)` is true if only `X` is selected from its or-feature group. If more than one ADC is selected from the group, the abstract base class is used.

To solve the problem whether the `ADC_?` classes have to be derived from an abstract base class, the class `ADC_Base` has two different declarations, one as an abstract class while the other is just an empty class definition.

The description of class `ADC_1` is straightforward. It is included in the component whenever support for `ADC_1` is requested. For the other two classes the descriptions look alike.

```
Component("ADCControl")
{
  Description("ADC Controller Access")
  Parts {
     classalias("ADC") {
       Sources {
         classaliasfile("include", "ADC.h","ADC") }
       Value("ADC_1",Prolog("is_single(’ADC_1’,_NT)"))
       Value("ADC_2",Prolog("is_single(’ADC_2’,_NT)"))
       Value("ADC_3",Prolog("is_single(’ADC_3’,_NT)"))
       Value("ADC_Base",Prolog("true"))

     }
     class("ADC_Base") {
       Sources {
         file("include", "ADC_Base.h",def,"include/ADC_Base_virtual.h") {
           Restrictions {
     Prolog("not(selection_count([’ADC_1’,’ADC_2’,’ADC_3’],1,_NT))")
           }
         file("include", "ADC_Base.h",def,"include/ADC_Base_empty.h") {
           Restrictions {
     Prolog("selection_count([’ADC_1’,’ADC_2’,’ADC_3’],1,_NT)")
           }
         }
       }
     }
     class("ADC_1") {
       Sources {
         file("include", "ADC_1.h",def)
         file("src", "ADC_1.cc",impl)
           { Restrictions { Prolog("has_feature(’ADC_1’,_NT)") } }
       }
     }
     ...
    }
  }
  Restrictions { Prolog("has_feature(’ADCControl’,_NT)") }
}
```

Figure 5.24: CFDL for ADC component

This example illustrates that the mechanisms used to implement customizable components are available, even without CONSUL. Changing a class hierarchy could be easily accomplished by a conditional `#include` resolved by the C++ preprocessor according to a compiler argument that is defined in a makefile. With the CFDL, however, there is one single place to manage the customization process. The information what and how to configure is not spread out over different files in different languages. It separates the structure of systems and components from the source files in which they are implemented.

The extensibility of the CFDL through its customizable backend makes the introduction of new high-level description elements easy.

Combining CONSUL with AOP: Going back to the example given above, it was necessary to deal with the problem of having different base classes of `ADC_?`. To be able to leave the class definitions unmodified, a fake base class for the cases where the abstract base class should not be used had to be provided.

The aspect language AspectC++ [AC] allows to write aspects that are able to introduce new base classes to arbitrary classes. To make this available in the CFDL, a new part description element named `Baseclass`, which takes two arguments, had to be defined, carrying the name of the intended base class and the privilege level (`private, public, protected` for C++). This could be accomplished by adding a new XML transformation description for the `baseclass` statement to the backend transformation library. Figure 5.25 shows the modified component description and Figure 5.26 the generated aspect code.

The generated aspect code is later woven with the C++ source code of the class `ADC_1` and results in a modified declaration of this class with an additional base class `ADC_Base.` The benefits are obvious: the use of a “dummy” class is no longer necessary and the CFDL expresses the design variation more explicitly.

5.6 Family Deployment

A critical point in software family-based development is the deployment of the previously developed families. Deployment of software families happens in different ways. A single member of a software family might be used in the development of a software system, or the development target is itself a software family and integrates other software families.

An application (or even a family of applications), for instance, might always require the same set of operating system functionalities. In this case, an appropriate member of an operating system family can be chosen and used in the application development like other existing non family-based software artifacts. If the deployment of the operating system family happens in a family-based development, however, it might be different. If several

```
Component("ADCControl")
{
  Description("ADC Controller Access")
  Parts {
    ....
     class("ADC_Base") {
       Sources {
         file("include", "ADC_Base.h",def,"include/ADC_Base_virtual.h") {
           Restrictions {
     Prolog("not(selection_count(['ADC_1','ADC_2','ADC_3'],1,_NT))")
           }
         }
       }
     }
     class("ADC_1") {
       Sources {
         file("include", "ADC_1.h",def)
         file("src", "ADC_1.cc",impl)
 // introduce new base class for multiple ADCs
 baseclass("ADC_Base","public")
           { Restrictions { Prolog("not(is_single('ADC_1',_NT))") } }
        }
     }
     ...
  }
  Restrictions { Prolog("has_feature('ADCControl',_NT)") }
}
```

Figure 5.25: CFDL for ADC component using the `baseclass`() source element

```
aspect consul_ADC_1_ADC_Base
{
  advice classes("ADC_1"): baseclass("public ADC_Base");
};
```

Figure 5.26: Aspect code generated for the CFDL `baseclass` source element

members of an embedded operating system family are required depending on the different scenarios the new family is going to be used in, then it is an integration process of one family into another.

CONSUL supports both ways of deployment easily. The integration of an existing family into a new family development is done by providing a mapping of the new family feature model(s) to the existing family using the CONSUL mapping language. Of course, the "normal" activities, like providing matching interfaces, analyzing the interaction between the parts of the implementation and so on, have to be done as well.

In most cases of component-based developments, such a family integration will lead to a hierarchical structure, as discussed in the previous part of the chapter (Section 5.3.8.3) and is therefore supported by CONSUL as part of the family development process in the domain engineering phase.

The deployment of a single family member in an application development is the last phase in any family-based development. Based on the application requirements specification the variation points in the software family are resolved. For CONSUL based software families this is done by selecting a set of features, assigning feature values if necessary and specifying the family deployment information.

5.6.1 From feature model to feature sets

As already discussed earlier, using a family-based software for development does not only have benefits. There are also problems associated with such an approach. Using a family instead of a fixed solution requires the specification of the needs of the intended application in terms of the software family. This customization and configuration process needs to be easy enough, so that it really pays off in terms of development time and/or resource usage to do it. Figure 5.27 shows the important issues here. The complexity and number of decisions to be made before a family/family member can be used in the application is a critical point for the success of a method. Software developers are not willing to invest too much effort into the selection process if it is too complex and takes too much time. If the process is easy but does not offer the expressiveness to specify application requirements, developers cannot use it.

CONSUL (and therefore all other feature model-based approaches) uses feature models as a main instrument in this decision making process. The relatively simple language of those models makes it easy to understand. One problem seems to be the number of features one has to consider. However, unlike approaches that use decision models to guide users trough the configuration process, feature models allow the exploration of all features on any hierarchy level without the need to find out which answers are required to reach a specific point (feature) in the tree.

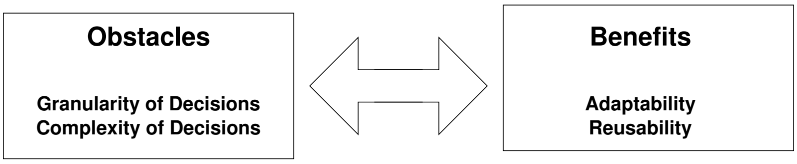


Figure 5.27: Obstacles versus benefits of configuration models for software families

CONSUL enables a user to select any feature she or he likes, and it can immediately execute checks whether there are open decisions (that means unbound variation points) or unfulfilled restrictions. Figure 5.28 shows the cosine feature model where a problem is signaled by a color indication. In this example the user selected a feature (`Trace`) for which an additional required feature is defined (`PCConnection`, see lower right part of the window) . The user must now either deselect the `Trace` or select `PCConnection` as well.

The number of features selected from a model is usually relatively small compared to the number of features in the model (see Section 6.2.3). One reason for this is that through the selection of a feature all its parent features up to the root node are included in the configuration automatically.

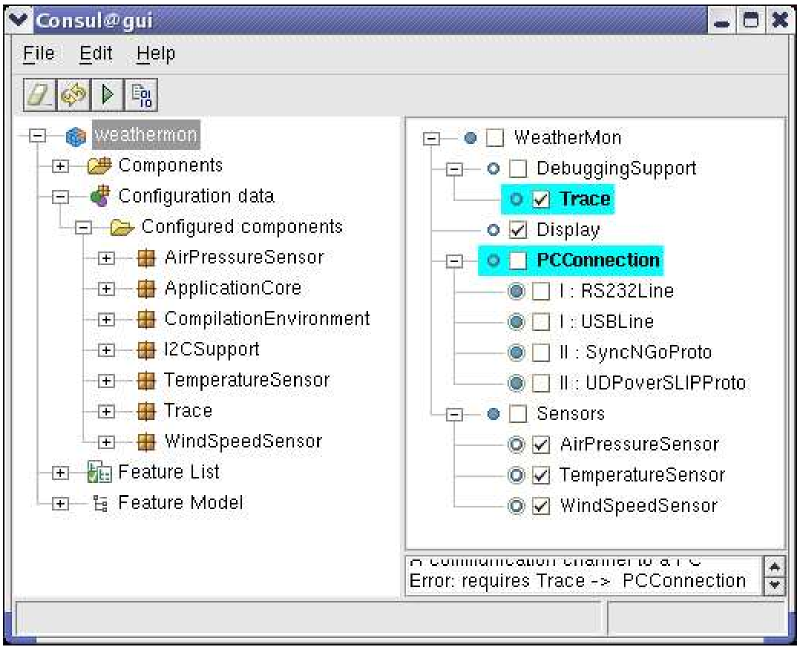
5.6.2 From Feature Sets to Family Members

The final step in using a family member in an application development is the generation of the family member itself. CONSUL uses the three elements *feature model*, *component model* and *feature set* as input for the member generation process. Depending on the generation process used, either source code or binary code is generated from these inputs and can be used by the application.

5.6.2.1 Feature Sets versus Family Members

As already mentioned earlier in this chapter, CONSUL does not require a distinct family member for each valid feature set. This might seem like a problem but is really a benefit. A feature set describes the requirements of the application in terms of the problem domain, a family member is one of the available members of the solution domain. There is not necessarily a one-to-one mapping between valid features sets and family members. Many different feature sets might be realized using the same family member[7]. There might be also feature sets that have no associated family member, that means that there is no realization

[7] In the cosine example (Figure 5.20 on page 104) all feature sets without the feature ’FixedTime’ use the same implementation, that is the same family member.



- □ WeatherMon
  - □ DebuggingSupport
    - ☑ Trace
  - ☑ Display
  - □ PCConnection
    - □ I : RS232Line
    - □ I : USBLine
    - □ II : SyncNGoProto
    - □ II : UDPoverSLIPProto
  - □ Sensors
    - ☑ AirPressureSensor
    - ☑ TemperatureSensor
    - ☑ WindSpeedSensor

A communication channel to a PC
Error: requires Trace -> PCConnection

Figure 5.28: The interactive CONSUL feature model checker showing a model with an invalid feature selection

known for the given requirements. Depending on the feature model and the software family, there might also be family members that cannot be mapped to any of the possible valid feature sets.

Although it should be ensured that as much feature sets as possible are mapped to existing family members, it is also possible to wait for a user to really select a feature set and start the process of providing a solution, that is a family member, only then. This is also called the reactive approach (Charles Krueger [Kru02]).

CONSUL can handle this problem gracefully, it is possible, for instance, to notify the developers automatically if for a given feature set no working realization[8] could be found.

5.7 Summary

The described CONSUL method for feature model-based development and deployment of software families was introduced in this chapter. The CONSUL method is inspired by the early feature model-based methods FODA and FORM. The chapter introduced the newly developed concepts of multi domain feature models and the hierarchical feature models that allow a much better reuse of feature models. The presented approach goes beyond the FODA and FORM approach by also giving support for the domain design/implementation via the component family model. This model remains neutral to programming languages and specific component models and therefore provides a universal description of software family implementations.

The openness of the presented method itself is also supported by the tools that implement the infrastructure for family development and deployment. The presented approach uses Prolog for knowledge representation and conflict resolution. Users can extend the knowledge easily to incorporate a domain specific knowledge base. The generation process is also controlled by tools that allow all necessary customization by the users of CONSUL.

Some design issues specific to embedded systems were discussed, and it was shown how the CONSUL method can be used to solve these problems efficiently. Also the combination of CONSUL with aspect oriented programming was introduced briefly.

[8] The meaning of the term “working realization” in this context is defined as “The CONSUL system cannot provide a family member in the form requested by the user”. For obvious reasons, it cannot be checked/guaranteed by CONSUL that a generated family member works in the users application context as expected. This is the field of formal verification techniques not included in CONSUL.

6 Rule-based Configuration of Embedded System Software

An important practice during the development of CONSUL was the continuous experimentation with the ideas, models and tools. The immediate feedback of the users lead to many changes, additions and corrections. The following chapter presents the application of CONSUL to several projects in the embedded domain. The experiments discussed are all based on Pure [BGP$^+$99], an object-oriented operating system family for deeply embedded systems.

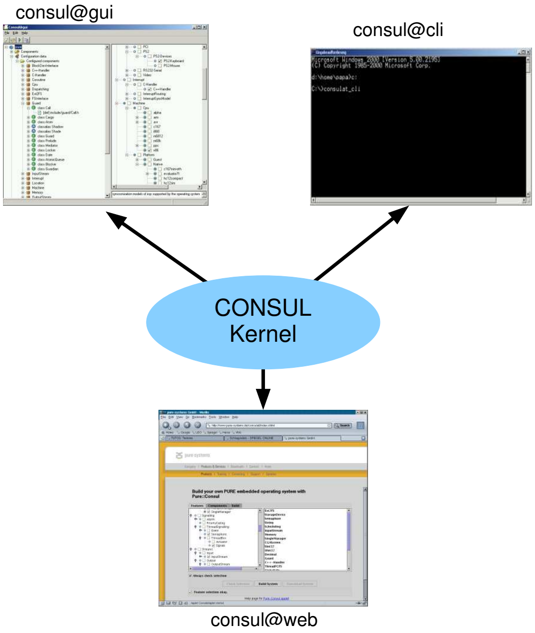
The combination of Pure and CONSUL was a reengineering experiment, since Pure already existed by the time CONSUL was developed. Based on initial experiences of using CONSUL with Pure, several Pure extensions were created with the help of CONSUL. Section 6.3 discusses the improved reusability when multiple feature models are used. The context is an extension of Pure to provide a standardized operating system application programming interface (API) for the automotive domain. Last but not least, the application development with CONSUL (Section 6.3) is discussed. The object of experimentation was the development of a complete embedded system, a small weather station.

Before the discussion begins, however, a brief overview of the CONSUL tools developed is given below.

6.1 CONSUL Tool Family

As already discussed earlier in this dissertation, good tool support is one of the key elements for a successful software engineering methodology for software family development. The CONSUL and its tool family (Figure 6.1) is no exception from this rule. Much effort has been spent in providing an environment that was accepted by the users.

An important point was the availability of a set of different tools for different purposes. The needs during family development differ from the needs during family deployment. The result was a (small) family of tools centered around a common kernel. The kernel contains the complete model evaluation and handling up to the generation of family members. The tools differ in the user interfaces and also the set of functionalities available.



```
Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

d:\home\papa>c:

C:\>consulat_cli
```

Figure 6.1: CONSUL tool family

The main tool is consul@gui that provides an interactive, graphical user interface. It can be used during all phases of family development and deployment. It enables the users to design and explore feature models, create feature sets, and to generate family members.

For the deployment of software families such a powerful tool is not always necessary. To support a “read-only” use of the developed models, the consul@web interface has been designed that allows users to interactively select feature sets and generate the respective family member. It uses a client/server model. All work is done by the server. The results provided by the server are presented using a Java applet.

The third member is consul@cli, a command line interface of the CONSUL kernel. This tool can be integrated into makefiles or automated test scripts. It allows the use of predefined information that created using the consul@gui or consul@web programs for example.

As expected, mainly the consul@gui and consul@cli have been used during the experiments described below, since the developers had to change and extend models frequently and wanted to experiment with the changed software family.

6.2 Pure — Reverse Engineering a Program Family with CONSUL

The starting point for the discussion is Pure itself, the first application for CONSUL. Because of the sheer size of the Pure family and the ongoing development by several researchers and students, it provided an ideal ground for experimentation of usability, practicability and also soundness of CONSUL.

6.2.1 Pure Basics

Pure is an acronym for *Portable Universal Runtime Executive*. Pure is in many ways a successor of PEACE, a parallel operating system family developed in the early nineties at GMD FIRST.

Because many ideas in Pure are heavily influenced by PEACE experiences and because Pure also shares some of its deficiencies, it pays off to have a look at PEACE first, and then give a more detailed introduction to Pure.

Pure's Ancestor PEACE: PEACE [SP94] was designed with the idea to provide the best possible support for parallel programmers. It was designed as a portable operating system. Several different execution environments were supported, namely the SUPRENUM and the MANNA platforms, developed inhouse at GMD FIRST, and a number of so-called guest level implementations on top of other operating systems (for example Windows and unix-like operating systems).

Parallel programs are special, because they are more sensitive to certain kinds of overhead than most other programs. Most of the time, a parallel program is either calculating some results, which does not involve interaction with other nodes, or communicating/waiting (for) a result, which means that communication latencies have a big influence on the performance. As communication handling (device drivers, protocol stacks) is usually seen as an operating system task, PEACE had to provide this support. Different programs have different communication needs that require different protocols implemented by PEACE.

The different possibilities for communication support were one dimension of variability in PEACE. Another dimension was the threading model. Some parallel applications used a one-to-one mapping of thread to node, so virtually no thread support was needed, in other cases several threads had to be executed on a single node, sometimes even in separate address spaces. Additional variability came through the support for different multiprocessor architectures (single processor, symmetric multi processing, asymmetric multi processing).

At the time, no tool support for the configuration of the several members of PEACE was used. The number of available family members was 32 on top of each supported hardware platform (native and guest level). In reality, only a much smaller number was actually deployed by the users. The hardware abstractions available on a platform that were to be supported were more or less identical since just the main memory and a communication device were of actual interest for most parallel applications.

The design of PEACE was based on functional hierarchies and incremental design (this will be elaborated in Section 6.2.2.1) and implemented using C++. To achieve both the configuration of the different family members and the required execution performance, a compile-time approach was used mainly[1].

Because of the non-existence of tools for this purpose[2], the configuration was based on the well known but also notoriously problematic macro mechanism of C/C++ [SC92]. According to the setting of several so called fame (FAmily MEmber) flags that represented either true or false, different parts of the source code were actually included in the compilation process. So choosing a family member meant setting the appropriate fame flags. Since the number of fame flags was relatively low this could be done quite easily.

From the Parallel World to the Real-time World: A first step into a new application domain for PEACE was that adaptation for massively parallel computation in a real-time application during the METRO project in the mid-nineties. PEACE was ported to a relatively powerful dual processor board using a hybrid PowerPC/Transputer architecture.

From the Real-time World to the Embedded World The successor of PEACE goes one step further. The Pure operating systems family was designed with the results of the PEACE experiment in mind but as an operating system family for deeply embedded systems. The goal was to provide very efficient yet ultra portable run-time systems that suit as many different embedded applications as possible. Since most embedded systems are rather badly equipped with respect to memory space and processing power, similar to parallel systems efficient resource usage is a key issue.

However, there are several key differences of embedded systems compared to parallel systems :

- The number of different hardware platforms is very large.
- The number of configurations of a single hardware platform (available devices, memory organization) is even larger.

[1] Henning Schmidt describes a PEACE extension to allow dynamic changes in his PhD thesis[Sch95].

[2] And also because a strong but understandable skepticism of PEACE developers and users against the use of “just another” tool

- The needs of applications are much more diverse. Some use only a small part of the available hardware but need to be fast. Some do not have to be fast but must fit into a very limited memory And some have to be both fast and small.
- Purely static configuration of the operating system is not always possible. However, what can be done statically should be done statically if it saves run time and memory space.

The design and implementation of Pure was based on the same principles as PEACE: an incremental design was used with C++ as implementation language. In the first step, only a statically configurable family was developed.

6.2.2 Pure Architecture

To understand the architecture of Pure, it is important to understand the ideas of *incremental design* and *functional hierarchies*.

6.2.2.1 Incremental Design

The approach known as *incremental design* was introduced by Habermann [HFC76]. Incremental design is based on the notion of a *minimal base* and *minimal extensions* to that base. The minimal base defines the set of functions common to all family members. The minimal extensions are the extensions required to extend this base with sufficient functionality to build a family member. An extension is considered to be minimal if no unnecessary functionality is introduced by this extension into any possible family member which uses it. The result of an incremental design process is a functional hierarchy build from the base and the extensions. This design represents only functional aspects of the family, non-functional aspects like memory usage or real-time capabilities are not represented in such a hierarchy.

Functional Hierarchies and Object-Orientation: The resulting functional hierarchy of a software family can be implemented in several ways. Figure 6.2 shows the functional hierarchy for the control flow and interrupt support of the Pure operating system family. A functional hierarchy does not imply a specific implementation design as Habermann [HFC76] said:

> It is the system design which is hierarchical, not its implementation.

The Pure family with its mostly static variation binding and compile-time decisions uses class inheritance extensively to represent the functional hierarchies. The base class corresponds to the minimal base and possibly many different subclasses represent the minimal

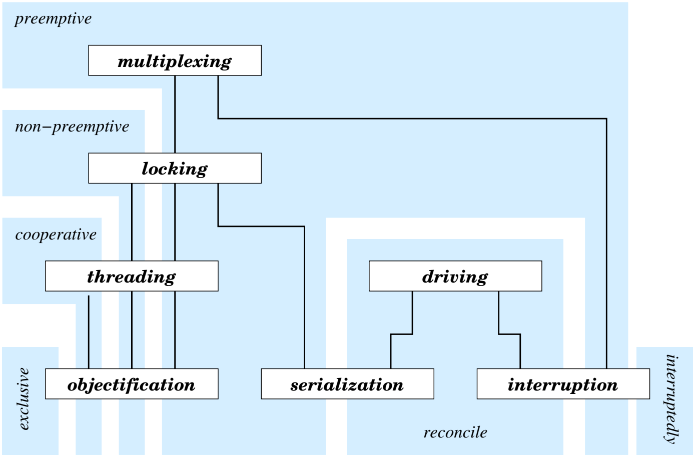


Figure 6.2: Functional hierarchy of Pure threading support

extensions (Figure 6.3). Figure 6.4 shows the Pure class hierarchy for Figure 6.2. The advantage of the inheritance approach is the efficiency of the resulting systems [BGP+99].

One of the downsides is that the implementations tend to have a large number of classes in an inheritance path. This makes it complicated for users and developers of a software family to get a complete overview over a family member, because many different classes and methods have to be explored in order to understand the interfaces and inner workings of the family[3]. The class representing the initial thread `Genius`, for instance, may be composed from up to 20 classes from a set of over 45 classes. To understand which operations (methods) the `Genius` class in a given family member supports it is necessary to go through the complete inheritance path and check which methods are available.

This problem, however, would occur with other approaches in a similar fashion. In a design where the functional hierarchies are implemented using modules of which each implements a set of minimal extensions, the same problem could be observed. This kind of problem is not technical a problem since the design delivers the required performance figures. It is a problem of maintainability and understandability. In order to use and extend such a system efficiently, a developer has to be able to understand the system easily.

By relying more or less entirely on inheritance, it is also necessary that at some points in the class hierarchy, several different class implementations are used depending on the configuration. The darker shaded boxes in Figure 6.4 mark these mutual exclusion points for classes, that means that the `Schemer` abstraction uses one of six classes depending on

[3] Tool support for class interface exploration could limit the problem sketched above to some degree. However currently available tools are not able to cope with flexible class hierarchies, where the base classes are chosen depending on the configuration.

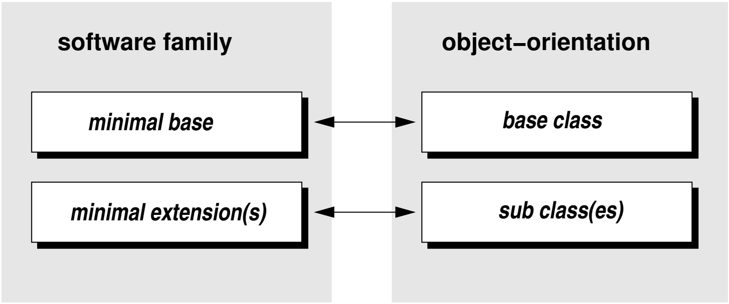


Figure 6.3: Object orientation and software families

the required functionality. This is necessary to avoid either code duplication[4] or the use of indirection techniques like aggregation. Code duplication is a night mare for maintenance and would definitely lead to even more confusion. Using aggregation instead of inheritance would not solve the problems either, as the type of the aggregated object would have to be resolved at compile time as well and, additionally, the aggregate object needs access to the functionality of the base classes of `Schemer`, which would incure more run-time overhead because of the cost of indirection.

It is important to note, that not the incremental design itself is the problem here but the implementation using inheritance as a means to express functional variability and to compose parts of a system. Besides the problem of deep class hierarchies, a second problem is that the efficient implementation of such mutual exclusion points via inheritance is not very well supported by object-oriented languages. Alternatives to the use of inheritance are not always available, because, as already discussed in Chapter 4, other approaches have increased resource requirements, which is often problematic.

An additional design constraint were the tools available at the time of the creation of Pure. Most C++ compilers, for instance, were not able to handle templates in the mid-90s. Therefore, approaches, for example static template meta-programming or template based aggregation/composition could not be used.

Realizing Inheritance Variation with C++ The control of these variation points in Pure is handled by C++ preprocessor definitions just like in PEACE. Figure 6.5 shows the relevant configuration files from the Pure sources. Selecting a specific configuration was handled by choosing the right value in `fameScheme.h` and uncommenting it.

Several critical issues with the chosen approach can be seen here:

[4] The subclass hierarchy of the `Schemer` representations would have to be cloned in a separate name space for each of the six `Schemer` variants.

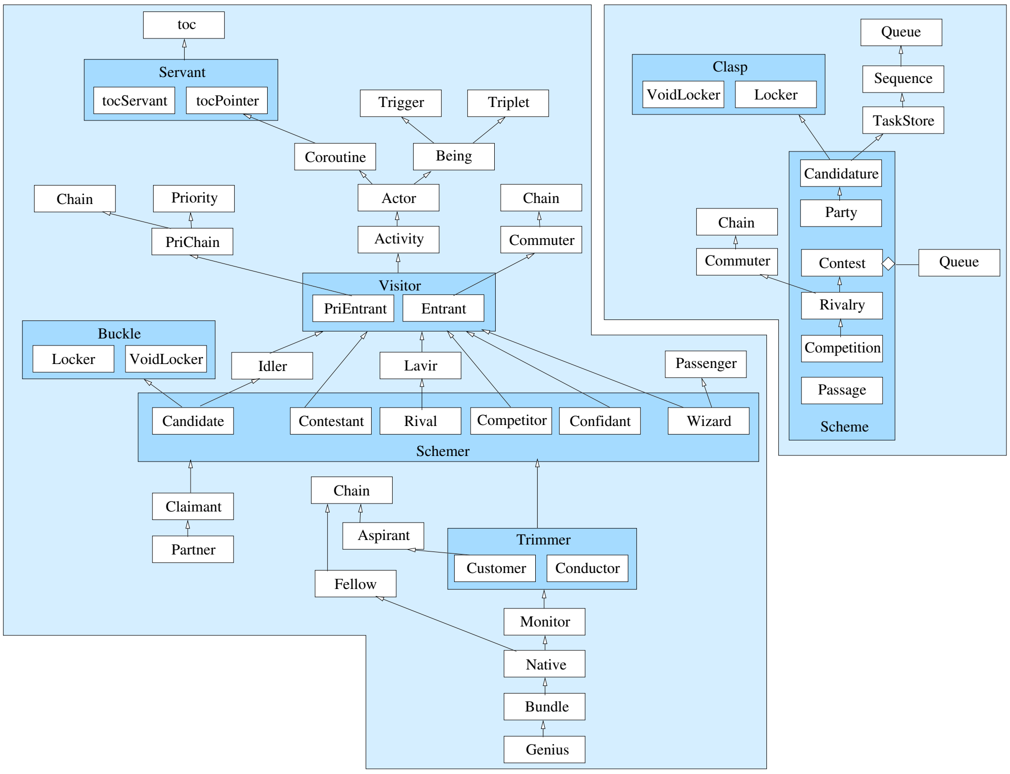


Figure 6.4: Implementation of the functional hierarchy of the Pure threading support

```
#ifndef__Scheme_h__
#define__Scheme_h__

#include "thread/fame/fameScheme.h"

#ifdef fameLCFS_thread
#include "thread/Confidant.h"
#define SchemerConfidant
#define Scheme Confidant
#endif

#ifdef fameFCFS_thread
#include "thread/Contestant.h"
#include "thread/Contest.h"
#define SchemerContestant
#define Scheme Contest
#endif

#ifdef fameLCFS_bundle
#include "thread/Rival.h"
#include "thread/Rivalry.h"
#define SchemerRival
#define Scheme Rivalry
#endif

#ifdef fameFCFS_bundle
#include "thread/Competitor.h"
#include "thread/Competition.h"
#define SchemerCompetitor
#define Scheme Competition
#endif

#endif




//#definefameLCFS_thread
#definefameFCFS_thread
//#definefameLCFS_bundle
//#definefameFCFS_bundle
//#define famePriority_non
//#define famePriority_pre
```

Figure 6.5: Expressing an inheritance variation using the preprocessor

Maintainability: A new variation of the inherited base class requires changing existing source code. The configuration information is scattered across the system and hidden in header files. It is not obvious that a file named `Schemer.h` is in fact a variation point.

Understandability: To understand how the system is configured, it is necessary to check every source file and examine which configuration is currently used and how a specific configuration information is used. A close look at the example given above for instance reveals that there are two configuration values `famePriority_none` and `famePriority_pre` that are not handled in the corresponding `Scheme.h` file. Obviously this is done elsewhere.

Scalability: As the number of configuration points grows, the correct combination of the different configuration points into valid configurations gets exponentially harder. Every new configuration point at least doubles the number of possible configurations, but not all configuration value combinations are necessarily valid. Preprocessor macros do not allow to check for valid combinations.

At a certain point Pure consisted of about 200 classes, distributed in over 600 files and was configured using 64 preprocessor flags. The result was that most of the time only three or four different configurations were in use, others were used infrequently, if they were used at all. At this point, a new way of handling the family was needed.

What was required was a way to model the family members and their relationships in a user-friendly way. Furthermore, it should be possible to derive the configuration from an abstract specification of the system rather than forcing the user to edit source files manually.

6.2.3 (Re-)modeling Pure

The main problem to be solved was the ability to express arbitrary relations between both the implementation elements (for example class implementations that cannot be used together for some reason) and also on the user-requirement level (for example the inability to support both single tasking and multi tasking at the same time). In addition, changes to the existing implementations should be avoided.

Among the modeling approaches evaluated were some in which both kinds of relations are represented together (annotated source code, UML/OCL) and some that separated the problems in different models (GenVoca, decision models and feature models).

The mixture of relation information and source code increases the source code complexity and leads to reusability problems, since the domain model cannot be separated easily from the implementation specific model parts. First experiences with such an approach [Beu97, Beu98] showed that it is difficult to maintain a consistent model when the model information is scattered throughout the implementation.

A UML-based solution would have required quite complex tool support, including OCL evaluation, which was not available at the time of the initial modeling (1997). Additionally, using graphical UML models directly as a way to communicate configuration options to the user is problematic for larger problems since UML models tend to take large space on screen. A separate graphical modeling language would have been required anyway.

GenVoca as an approach that uses user-defined domain specific languages and, usually, a code generator is problematic in a reengineering scenario. Adapting the grammar of the language and the respective code generator is a matter of GenVoca experts. Contributions to the model require changes to language and generator by the GenVoca expert(s), so either the domain experts and developers become GenVoca experts as well or model changes can be made only by a small number of people.

Decision models and feature models are quite similar in their expressiveness. However, the idea of free exploration of available variation points is easier with feature models. Decision models prescribe the way to a specific configuration, while feature models express only relations between features. Both (decision models and feature models) cannot be used directly to express relations between implementation elements. This requires additional models.

Besides the simplicity of their basic concepts, feature models were seemingly suitable to model functional hierarchies quite well. The search for an adequate modeling language for the requirement level thus eventually lead to the selection of a feature model based approach in combination with a modeling language for implementation descriptions.

To verify this issue, the first step was to try to model the functional hierarchy of the thread support from Pure with a feature model. Figure 6.2 shows the basic functional hierarchy of this part of Pure as well as the interrupt support.

The model shown in Figure 6.6 was the result. The features that represent non-functional issues in the model are shown with a gray border. Several interesting observations can be made. The feature model resembles in fact the decision model a programmer has to go through when deciding which kind of thread support his or her application really needs. The first question is to decide whether really more than one thread is needed. If not, a single thread model is sufficient (`objectification` in Figure 6.2). Otherwise, the next question is whether a simple cooperative scheme is sufficient (Coroutine) or some kind of thread management has to be provided by the operating system (Dispatching). Embedded software developers with prior knowledge about real-time operating systems can easily find the right set of features sufficient for the desired application.

Feature Model Observations:

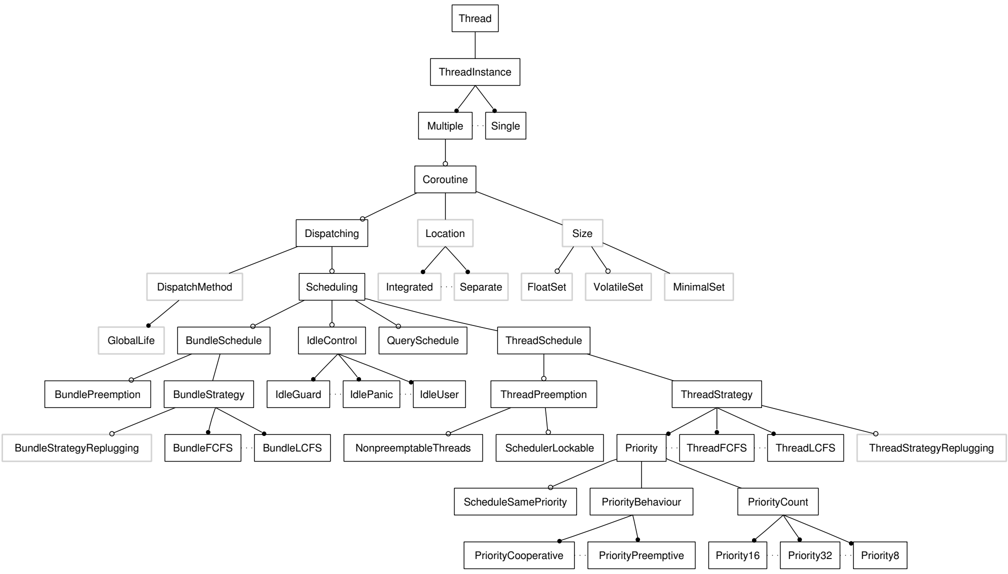


Figure 6.6: The Thread domain feature model

- The *resulting feature model not related to Pure*, most of the thread subsystems of today's operating systems could be described by the given set of features. The Linux scheduling, for example, is described by `PriorityCooperative`, `SchedulerLockable`, `Integrated`, `VolatileSet`, `FloatSet`, `GlobalLife`; the late Win 3.x by `ThreadFCFS`, `Integrated`, `VolatileSet`, `FloatSet`, `GlobalLife`.

- Another observation is that the *non-functional features are grouped at the outer border of the tree*. No functional feature is a sub-feature of a non-functional one. This is not very surprising, as non-functional features tend to lie orthogonal to functional features.

- Relatively surprising was the *number of additional restrictions required* to model the thread domain. *Actually none* was used. This domain is so well structured that it could be expressed by the four basic relations of features (mandatory, optional, or and alternative).

The last step was to provide a corresponding family model that describes the implementation in terms of files, constants, etc. The CONSUL family description language provides constructs rich enough to completely express Pure using it.

In most cases the parts[5] were simple `class` or `object` parts. The main configuration

[5] Part refers to the CONSUL CCFM part here.

```
...
    flag("fameClasp_lock")
    {
        Sources { flagfile("include/thread/fame","fameClasp.h","fameClasp_lock") }
        Value(1,Prolog("has_feature(’SchedulerLockable’,_NT),!"))
    }
...
```

Figure 6.7: Part of the Pure family model, describing a fame flag

instrument, the fame flags inside the Pure source, were not always replaced. If they were used to enable or disable parts of the source code, the `#ifdef` based on flags were left untouched. However, the setting of the fame flags is now done automatically during the family member generation, according to rules specified in the family model. Figure 6.7 shows the family rule that sets the flag `fameClasp_lock` based on the selection of the feature `SchedulerLockable`. The `Value` statement sets the value of the flag to 1 if the PROLOG rule is satisfied, otherwise the flag remains unset. The file containing the flag is generated in both cases, because it may be referenced by other source files to get the value of the flag. To prevent the generation of the file, a restriction rule could be used, which was not necessary here.

However, the variation points shown in Figure 6.4 were no longer represented by flags. To make the variation more explicit, the `classalias` part was used. In Pure each of the variation points already had a name, like `Schemer`, `Scheme` or `Servant`. The configured class was referenced by this name (which was set to the actual name of the class via `#ifdef` and `#define`). The `classalias` makes the variation point explicit in the family model. A classalias is basically a variable `typedef` that is resolved at member generation time.

Figure 6.8 shows the definition for the `Servant` classalias. The concrete class is calculated from the values of the features `Integrated` and `Separate`[6] in the feature selection of the member. The advantage of the use of classaliases instead of the mechanisms shown in Figure 6.5 is obvious. A single location inside the family model expresses the variability in an easily understandable way.

The results for the other parts of Pure were similar. It was relatively straightforward to build the feature model if the functional hierarchy was used as a starting point. The outcome was a feature model with about 220 features that were realized by about 57 components in the family model. The number of features is relatively high for various reasons. The first reason is that Pure supports many different processors and hardware platforms, each modeled by a

[6] These features control the place where the memory space for saving processor registers on a context switch is allocated.

```
...
classalias("Servant")
{
   Sources { classaliasfile("include/machine","Servant.h","Servant") }
   Value("tocServant",Prolog("has_feature('Integrated',_NT)"))
   Value("tocPointer",Prolog("has_feature('Separate',_NT)"))
}
...
```

Figure 6.8: Part of the Pure family model, describing a classalias

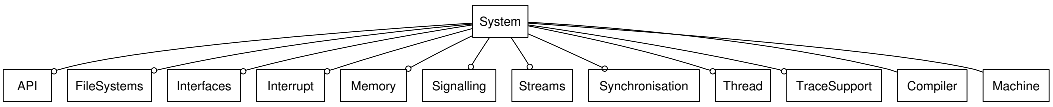


Figure 6.9: The top level of the Pure feature model

feature (43 features), has a highly configurable streamed Input/Output model (33 features) and the scheduling contributes 44 features as was already shown. The largest amount of features, however, comes from the modeling of hardware devices: 74 features. In total, almost half of the features describe the hardware platform.

In the current version of Pure the number of features has increased significantly. There are now about 411 features. Although the number of features is much higher than the number of previously existing fame flags (64), the selection of a family member has become much easier, since a typical feature set has 21 user selected features on average[7].

6.2.3.1 Multi Domain Models for Pure

For several reasons, the modeling of Pure was not based on multi-domains initially. The most prominent one is that the concept of multi domain models was developed later[8]. However, it is easy to see that there are several, relatively loosely coupled domains inside each operating system. Thread handling, memory management, device control or IO streams have almost no relation to each other.

In the feature model of Pure this is visible by the fact that these domains are all optional parts of the model and also direct sub-features of the top-level feature `System` (see Figure 6.9).

[7] The typical feature sets were represented by the collection of approved Pure test case feature sets in the Pure source tree (currently 193 sets). The standard deviation for the number of features was around 7.4 .

[8] The experiences with Pure were a significant reason for developing multi domain models.

The only mandatory features at this level are `Machine` and `Compiler`. The `Machine` feature and its descendants describes the target system, and the `Compiler feature` tree is used to describe the application development environment.

So the model of Pure consists in fact of several independent domain models. In a next step the model was split up and reorganized accordingly.

6.3 Reusing Pure — PureOSEK

An example for the usefulness of splitting up a domain model into several (smaller) models is given below in the context of an extension of Pure in terms of support for a specific operating system interface standard.

Many different run-time systems and operating systems for deeply embedded systems have been developed independently with different application programming interfaces (API). Changing the hardware platform often meant rewriting the applications because of changed operating systems APIs. Several companies in the automotive industry tried to develop standardized APIs for automotive operating systems to avoid this problem. One of the most successful standards in this area is OSEK. OSEK is an abbreviation for the german term “Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug” (“Open Systems and their Corresponding Interfaces for Automotive Electronics”). Several car manufacturers (BMW, DaimlerChrysler, VW) were among the founders of the OSEK Group in 1993. Renault and PSA joined the consortium relatively fast and contributed their VDX (Vehicle Distributed eXecutive) standard, changing the official name to OSEK/VDX. This group developed a number of standards for the various basic software functions in automotive real-time applications (operating system, communication, network management).

In a research project the Pure operating system was to be extended by an implementation of the OSEK operating system API. A detailed analysis of the OSEK OS standard showed that most of the required functionality of OSEK OS was already present in Pure. The main difference was the application level API itself and the configuration process of OSEK OS. The API of OSEK is procedural, so a thin layer to map the object-oriented Pure world to OSEK was needed.

The interesting part was the configuration. OSEK OS is completely statically configured. It uses a configuration language, OIL[9] , to describe the application and uses this information to generate both the application and also configuration information for the operating system. Figure 6.10 shows the principal way of development with OSEK OS.

Tasks, resources used by the tasks, error events, timing information, application source files, etc. can be described using OIL. The system generator produces a linkable version of the

[9] OIL = OSEK Implementation Language.

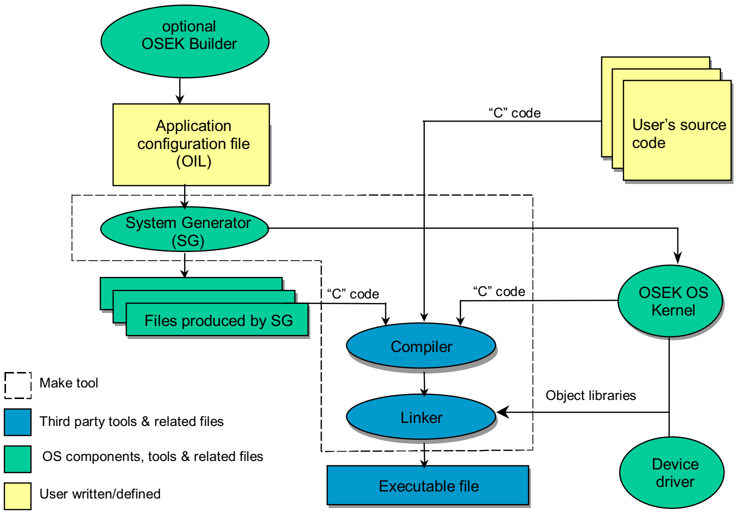


Figure 6.10: An example for the OSEK development process (from [OSE01])

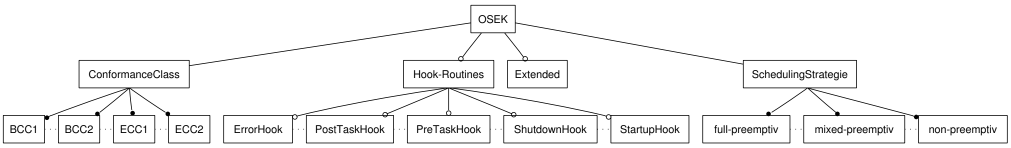


Figure 6.11: The OSEK OS domain feature model

application and also provides information how to configure the operating system, which is configured separately in an operating system vendor specific way.

In principle, it would have been sufficient to just provide a Pure specific system generator that produces the appropriate feature set for the intended application. The problem is that OIL does not include any specification of hardware platforms (processor type, etc.). This is left open to the application developer who has to provide an appropriate hardware abstraction layer. Another issue is that for some scenarios, OIL is not used, so the automatic configuration was not possible.

It was decided to develop a specialized feature model that could be used by the system generator as well as by humans with the appropriate knowledge of OSEK OS (not Pure!) to configure the PureOSEK.

Figure 6.11 shows the OSEK OS domain according to the OSEK specification. It defines so

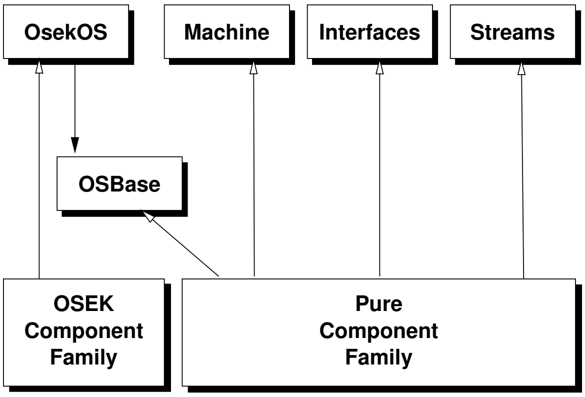


Figure 6.12: The OSEK OS domain feature model

called conformance classes that control basic features of the operating system. Both BCC1 and BCC2, for instance, do not allow event based activation of tasks. The other features allow the developer or the generator to provide additional information about the services the application requires.

This model was integrated into a hierarchical domain model, which is schematically shown in Figure 6.12. The `OsekOS` model is mapped to the `OSBase` that is basically the original Pure feature model minus the `Machine`, `Interface` and `Streams` sub-model. Those three models are also exposed to the user and/or the system generator. The `OSEK` component family consists almost entirely of the API interface layer. The hiding of `OSBase` features with the `OsekOS` model reduces the amount of features the user may select by 74% (16 versus about 100), the number of possible configurations is reduced even further.

This proves the usefulness of multi domain models, both in terms of reusability and usability.

6.4 The Weather Station — a Pure Application

While the previous examples were centered on the component or component family level, the final example is a small application family that shows again the usefulness of hierarchical feature models. In a project the goal was to develop a complete weather station based on a small experimental microcontroller board equipped with an ATMEL ATMEGA103. The board was equipped with several sensors (air pressure, temperature, wind speed) and has an LCD display, a serial controller and a USB controller for output and input purposes. Although the ATMEGA103 has 128 kByte of code memory, for the project the memory was limited to 8 kBytes (Figure 6.13)

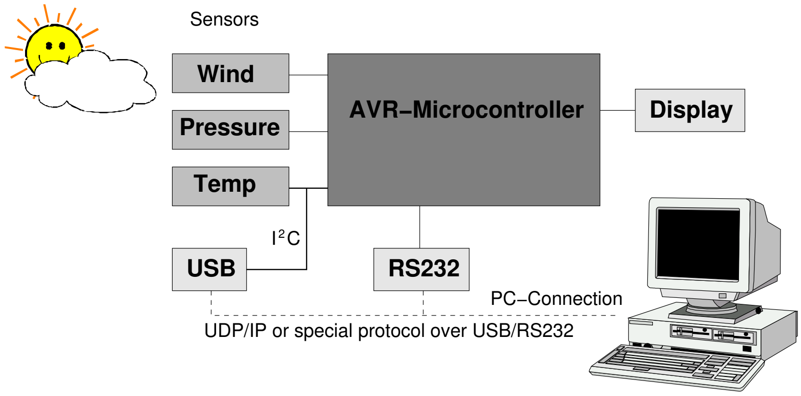


Figure 6.13: Weather station

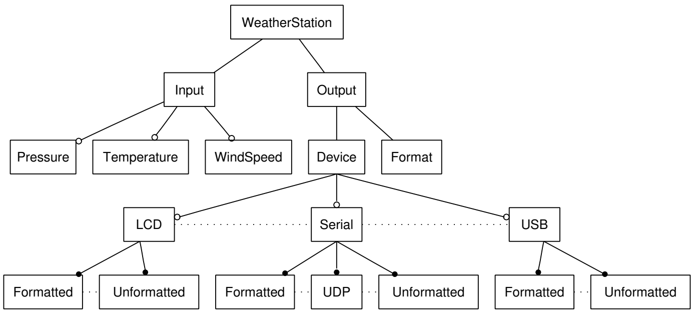


Figure 6.14: The weather station application feature model

Several input and output options had to be implemented. Each of the sensors was optional, the output options were formatted or unformatted output for the LCD display and for serial output. Another output option was the use of SLIP[10] UDP[11] packets in a predefined way. One resulting feature model for this application family is shown in Figure 6.14. The model was completely independent from the implementation platform used (in this case Pure) and could be reused without any changes on top of any other operating system. Only the mapping functionality had to be exchanged.

The implementation of the weather station relied heavily on the combination of simple C++ components, realizing separate functions, and AspectC++ aspects that provided the “glue” for combining the independent components. Features were used to select functional

[10] Serial Line Internet Protocol
[11] User Datagram Protocol

```
#ifndef __Pressure_ah__
#define __Pressure_ah__
#include "PressureSensor.h"
#include "WeatherData.h"
aspect Pressure {
  // measure the pressure, when the weather data should be updated
  advice args (weather) && execution ("void Sensors::measure(...)") :
    after (WeatherData &weather) {
      weather.pressure = PressureSensor::measure ();
    }
  // advise the display to print the wind speed after each update
 advice args (weather) && execution ("void Display::update(...)") :
   before (WeatherData &weather) {
     thisJoinPoint->that ()->print ("Pres", "hPa",
                                    850 + (int)weather.pressure);
    }
};
#endif // __Pressure_ah__
```

Figure 6.15: Aspect connecting pressure sensor and output components

components like pressure sensor driver, LCD output driver and the connecting aspects (see Figure 6.15).

The combination of functional C++ code and aspects proved to be easy to handle. The aspects were quite small, the functional components remained uncluttered from the different variations, the sensors, for example, did not know whether and how their output was used for displaying or transmitting via USB. Other techniques, like virtual methods/abstract base classes, would have allowed the same effect but require more resources, as shown in Table 5.4 on page 111.

6.4.1 Feature Model Templates

The feature model of the weather station exhibits an interesting problem. Each of the different output devices (USB, Serial and LCD) have almost identical sub-features. This is not very surprising, as all devices can be used for the same functionality. The resulting replication is a general problem of feature models. The alternative of representing the formatting features in a separate and independent sub-model is not possible, because then it is not defined which formatting option has to be supported by which output device. All devices would have to support all output options. The output formatting has to be set separately for each output device.

The most intuitive solution would be a matrix of $DeviceType \times FormatOption$. The feature model shown is in fact a flattened matrix. However, the creation for larger numbers of features involved can be very expensive. A feature model template mechanism where the

```
model  FormatOption
  { alternative(Formatted,Unformatted,UDP) }

template WeatherStationTemplate<LCDFO,SFO,USBFO>

{ ...
  Device: or(LCD,USB,Serial)
  LCD: LCDFO
  Serial: SFO
  USB: USBFO
}

model WeatherStation<LCD: FormatOption,
                         Serial: FormatOption,
                         USB: FormatOption>
```

Figure 6.16: Proposal for feature model templates

parameters are models that can be inserted into specif iced locations would make this quite easy to handle. Based on the feature model notation introduced in [vDK02], a template notation is proposed (Figure 6.16). To avoid name conflicts when the same model or template is inserted, an individual prefix can be specified for a template parameter at instantiation time. CONSUL does not yet implement this extension but will do so in the near future.

6.5 Summary

The chapter presented examples of the application of CONSUL in different scenarios. It showed that CONSUL can be used in reengineering tasks as well as for new developments. The ability to perform reengineering is an important success factor in embedded software development contexts, since even when a complete switch from single system development to family-based development is made, third-party software or legacy software has to be integrated into the development (process).

The easier configuration model compared to approaches based on simple mechanisms, like makefile variables and preprocessor directives, increased the number of Pure configurations deployed by users significantly. This in turn increased the quality of the software indirectly: the more possible configurations are deployed, the better the independent components are tested. Upon detection of invalid combinations, the knowledge could be expressed in terms of the CONSUL feature model and component family model.

The OSEK example showed that through the multi domain model approach, increased reusability of feature model based software developments can be achieved, since it allows the integration of previously created models in other contexts.

The functionality and expressiveness of CONSUL seems to be sufficient for the cases shown that reflect many relevant scenarios from the embedded software development domain. However, with respect to user interaction and increased usability some potential improvements have been identified, most notably the need for feature model templates.

Some of the presented issues and examples are discussed in more detail in [BSSP02, BSPSS00, BPSP03]

7 Conclusions

This thesis presented the CONSUL approach for family-based software development. This approach follows a rather pragmatic way, using many concepts already known like feature models and component oriented software development. It then combines these with new elements to make it suitable for the needs of both the embedded system software domain and the embedded software developers. The former requires fine-grained variability in the software architecture, to achieve an efficient resource usage and reusability of the resulting software. The latter requires models that are easy to understand and scale well even for larger software projects and permit as much automation as wanted[1].

The CONSUL approach supports both goals by providing:

- Reusable domain modeling with hierarchical feature models (Sections 5.2 and 5.3) .
- A flexible and user adaptable component model (Sections 5.4 and 5.5).
- Tool support for every CONSUL related activity in all phases of the development process (Section 6.1).

For successful use of almost any software engineering method for the embedded software domain, it is important to provide the ability for reengineering, since it is not feasible to start new software developments from the scratch. The examples given in Chapter 6 proved that even complex software like a full-fledged operating system family for embedded systems can be modeled and used relatively easy with CONSUL tools.

An important issue for tool support was the interactivity concept. The tools should be as interactive as possible so as to allow exploration of the models. This was a relevant constraint that had to be dealt with. Too complex models would require too much computation power to be evaluated in a short time frame, breaking the concept of interactiveness. The use of relatively simple models like the feature model allows, with current desktop computers, interactive model evaluation even for complex scenarios.

[1] It is important to distinguish between the terms “as wanted” and “as possible”. Any successful tool should support the user only where it is necessary and wanted. Developers sometimes tend to do some things manually even when it could be done automatically and reject tools that leave no choice in this matter.

7.1 Comparison with Related Work

The issues addressed by CONSUL get increasingly more attention in both the software industry and software engineering research. Naturally, other projects and products try to solve the same or similar problems. The following section discusses some of those projects most closely related to CONSUL that provide at least limited tool support.

There are not many tools available for language-independent, cross-level management of software variability. The company BigLever with their product GEARS [Kru02] is one of the few. GEARS operates on the file system level to manage variability. It allows to specify conditions for the inclusion of a specific file into a resulting system. However, there is no complete domain model but several independent sets of parameters are used to describe those conditions. Although this might enhance reusability, it restricts the description of cross-component dependencies. The GEARS approach does not provide a separation between the problem domain models and the solution domain models. Thus independent reuse or exchange of these models is not possible, which limits the usability of GEARS.

Several other approaches use feature models for domain modeling [GFd98, KLLK02]. Most of them, however, do not use an explicit feature modeling tool. Feature modeling without tool support restricts the size of the models to a small number of features, rendering feature models almost useless for practical purposes. In [vDK02] a tool is described that evaluates feature models and is able to generate Java class skeletons from feature models. Another tool for drawing feature models is AmiEdit , which was developed as part of a diploma thesis at the University of Applied Sciences in Kaiserslautern. However, this tool supports drawing only, there is no connection to other models or a model evaluator.

The transformation process in CONSUL, which produces the customized implementation from component descriptions has some similarities to frame-based source generators like COMPOST [Aßm02] or XVCL [JZ01]. The idea of frames blends perfectly into the concepts of CONSUL. The open model of the CONSUL tools allows the integration of such a generator into the transformation process, and the parameterization of the generator is controlled by the feature model and the component family model constraints.

A different approach to solve the problem of providing efficient and yet adaptable software is program specialization. Program specialization uses compiler technologies to transform generic programs into specialized programs. Based on already know input data that is bound to variable elements (function parameters or object attributes) in the generic program a transformation tool generates partially evaluated code where the already established knowledge about already input data is incorporated into the program source. The [MCE02] describes an interesting tool for the C programming language. A separate solution model, however, is not part of program specialization. Because of the nature of program specialization, those tools usually support just one specific programming language.

Program specialization techniques like frame processors could be used to implement (parts

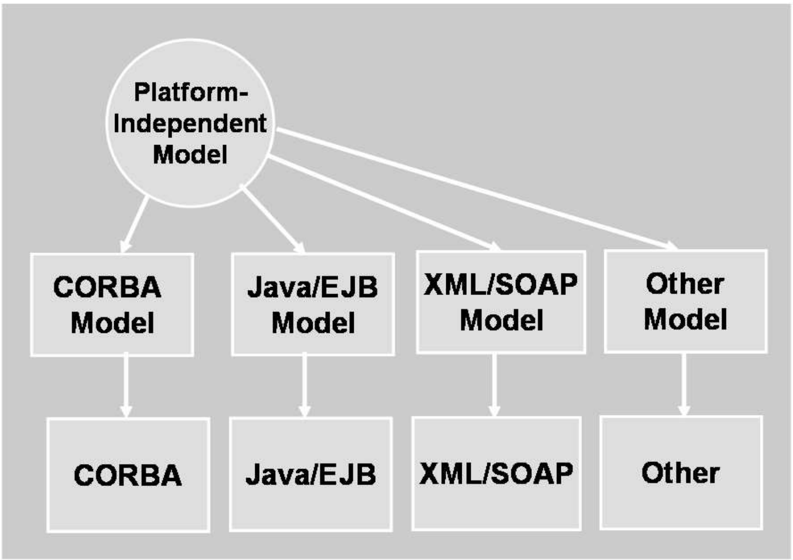


Figure 7.1: MDA: From PIM to platform specific applications (taken from [Sol01])

of) solution domains. To be useful for software family development, they have to be combined with adequate modeling approaches.

To solve the reusability problem, the Object Management Group (OMG) proposes the use of the Model Driven Architecture Approach (MDA). MDA is mainly concerned with software portability. It provides two different models: the Platform Independent Model (PIM) and the Platform Model (PM). The intention is to model the logic of a program in a platform [2] independent manner. The mapping to the desired platform(s) is the task of tools that get the PIM of an application and the PM of the target platform(s) as input and produce the concrete applications from this information (Figure 7.1) mostly automatically . In fact, MDA applications can be seen as simple product-lines where the variability is mainly the platform.

This might be feasible for relatively homogeneous domains like business applications with only a small set of platform variants[3]. Problems arise when the number of variants is high and the platforms itself are inhomogeneous as it is the case in embedded systems.

The Generic Modeling Environment (GME) [LMB$^+$01] is a tool set that allows the definition and evaluation of arbitrary modeling languages based on objects and relations between

[2] In terms of the OMG the middleware layer and/or operating system layer define a platform.

[3] A platform is mainly by the combination of middleware platforms (CORBA, J2EE, WebService), operating systems (Windows, Unix) and databases (Oracle, MySQL, ...). The number of relevant members of each of these elements is not very high.

objects. Therefore, it would be possible to define the CONSUL modeling languages within GME and use the GME for basic model evaluation. However, more complex operations like the CONSUL feature model mapping cannot be modeled directly. The GME would have to be extended to enable such operations. Anyway, the availability of a tool like GME at the beginning of this dissertation project would have been beneficial, since it allows to realize tools for model evaluation quite quickly.

7.2 Open Problems

Though CONSUL in its current form is able to ease the handling of a number of pressing problem in embedded software development, many open problems remain. Some of the problems could be solved with extended or improved CONSUL methods and tools, some are outside the scope of CONSUL and have to be solved separately.

The following section discusses first a list of open problems, then concludes with remarks about the future work related to CONSUL.

Temporal Reuse / Evolution: The main goal of the development of CONSUL was to provide support for functional reuse in software families. Temporal reuse, that means evolution, is not yet covered explicitly. Evolution can happen in all phases, but the earlier the changes are introduced the more complex the realization of these changes can be if the changes have not been anticipated. When changing the relation between two features in a feature model from “alternative” to “or”, for instance, it is necessary to trace the influence of the change through all connected models. There might be model parts where the alternative existence of these features was assumed implicitly. The detection of such problematic model parts is an important step into the usability of CONSUL in large-scale developments where requirements and consequently the results of the domain analysis may change often and in ways that were not always anticipated.

Version Control and Management: Another aspect of evolution is the versioning of both the models and the component implementations. Existing version control systems like Rational ClearCase, CVS or Telelogic CMSynergy are used to record the temporal changes of a system over the time, usually at the granularity of files.

A clear distinction between variability in the family (handled by CONSUL) and versions (handled by a version control system) has to be made. Concepts for cooperation of these variation dimensions have to be developed.

Standards: CONSUL uses own modeling languages and models for its purposes. There are widely accepted modeling standards in the software development community, however, for instance UML oder SDL. The existence of mappings between CONSUL models and those standard models would allow the use/reuse of already existing models for example expressed in UML.

Testing: Testing, especially automated testing, of software is an important issue for embedded software developers. The degree of variability in software families/software products-lines opens a new dimension. If a complete test of each possible Pure family member would take one second, it would take approximately $6 * 10^{37}$ years[4]. Test approaches that can deal with these complex tests scenarios have to be developed and coupled to CONSUL. An appropriate way could be the generation of customized tests for a given family member on demand, when a configuration has been selected.

7.2.1 Future Work

There are many interesting topics for future work related to the CONSUL approach and the prototypical implementation of support tools. However, since the ideas and tools realized as part of this dissertation are a key element of a commercial tool development project of the pure-systems GmbH [5], future work has a strong focus on usability in today's development processes and less on theoretical problems.

With respect to practicability and usability of CONSUL in the real world, the integration of CONSUL into existing development processes, reengineering of existing processes and software products is *the* problem to be solved. Since almost no project starts from scratch, CONSUL tools must be able to blend into the existing tool chain and should not cause unnecessary work. Because of the manifold development processes (there are probably not two organizations with the same process) this requires a large amount of flexibility built into the tools and the method. However, if the customization effort is too high it will limit the practicability and usability all the same. The concepts of CONSUL with its XML-based representations of models and its external communication interfaces, and the modular transformation process should be a good base for achieving this goal.

Another important part of future work will be the gathering of more knowledge about scalability and use in different development domains besides the embedded domain. Tool-supported feature modeling has not been evaluated in many domains because of the lack

[4] Based on the variability present in the current feature model, which results in approximately $2 * 10^{45}$family members.

[5] supported by the Bundesministerium für Wirtschaft und Arbeit (German Federal Ministry of Economy and Work)

of feature modeling tools. Probably better visualizations and tool support is required when handling larger models with several thousand features.

Bibliography

[AC] AspectC++ Homepage. see `http://www.aspectc.org`.

[AJ] AspectJ Homepage. see `http://www.aspectj.org`.

[Aßm02] Uwe Aßmann. Beyond Generic Component Parameters. In Judith Bishop, editor, *Proc. of the Component Deployment IFIP/ACM Working Conference*, volume 2370 of *Lecture Notes in Computer Science*, pages 141–154, Berlin, Germany, June 2002. Springer.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Programming*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[BCRW00] Don Batory, Gang Chen, Eric Robertson, and Tao Wang. Design wizards and visual programming environments for GenVoca generators. *IEEE Transactions on Software Engineering*, 26(5), May 2000.

[Beu97] Danilo Beuche. Konfigurierung objektorientierter Programmfamilen in C++. Master's thesis, Technische Universität Berlin, 1997. English title: "Configuring object oriented program families in C++".

[Beu98] Danilo Beuche. An approach for managing highly configurable operating systems. In *Object-Oriented Technology – ECOOP'97 Workshop Reader*, LNCS 1357. Springer Verlag, 1998.

[BFK$^+$99] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J.-M. DeBaud. PuLSE: A Methodology to Develop Software Product Lines. In *Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99)*, pages 122–131, Los Angeles, CA, USA, May 1999.

[BGP$^+$99] 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 International Symposium on Object-oriented Real-time distributed Computing (ISORC) 1999*. IEEE Press, 1999.

[BJPW99] Antoine Beugnard, Jean-Marc Jézéquel, Noël Plouzeau, and Damien Watkins. Making components contract aware. *Computer*, pages 38–44, July 1999.

[BLS98] Don Batory, Bernie Lofaso, and Yannis Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Int. Conference on Software Reuse*, Victoria, Canada, June 1998.

[BO92] Don Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, October 1992.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2nd edition, 1994.

[BOSW98] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998.

[Box98] Don Box. *Essential COM*. Addison-Wesley, Reading, MA, 1998.

[BPSP03] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability Management with Feature Models. In *Proceedings of the Software Variability Management Workshop*, pages 72–83, University of Groningen, The Netherlands, February 2003. Technical Report IWI 2003-7-01, Research Institute of Mathematics and Computing Science.

[BSPSS00] Danilo Beuche, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Streamlining Object-Oriented Software for Deeply Embedded Applications. In *Proceedings of the TOOLS Europe 2000*, pages 33–44, Mont Saint-Michel, Saint Malo, France, June 5–8, 2000.

[BSSP02] Danilo Beuche, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. Finegrain Application Specific Customization for Embedded Software. In *Proceedings of the International IFIP TC10 Workshop on Distributed and Parallel Embedded Systems (DIPES 2002)*, Montreal, Canada, August 2002. Kluwer Academic Publishers.

[BTS94] D. Batory, J. Thomas, and M. Sirkin. Reengineering a Complex Application Using a Scalable Data Structure Compiler. In *Proceedings of ACM SIGSOFT*, 1994.

[CBFO91] G. Campbell, N. Burkhard, J. Facemire, and J. O'Connor. *Synthesis Guidebook*, 1991. SPC-91122-MC.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming—Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[CHW98] J. Coplien, D. Hoffman, and D. Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6):37–45, November 1998.

[Cla01] Matthias Clauss. Generic modeling using UML extensions for variability. In *Proceedings of OOPSLA Workshop on Domain-specific Visual Languages*, pages 11–18, Tampa, FL, USA, 2001.

[CM87] W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Verlag, 3rd. edition, 1987.

[Cza98] Krysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. PhD thesis, Department of Computer Science and Automation, Technical University Illmenau, Germany, October 1998.

[Cza03] Krysztof Czarnecki. Personal communication, 2003.

[EM] Project Homepage Evolution Management and Process for Real-time Embedded Software Systems (EMPRESS). See `http://www.empress-itea.org`.

[Feg97] Jalal Feghhi. *Web Developer's Guide to Java Beans*. The Coriolis Group Books, 1997.

[FNS01] Jason Flinn, Dushyanth Narayanan, and M. Satyanarayanan. Self-tuned remote execution for pervasive computing. In *Proceedings of 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloß Elmau, Germany, May 2001.

[Fou95] Open Software Foundation. *Introduction to OSF/DCE*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Frö01] Antônio Augusto Medeiros Fröhlich. *Application-Oriented Operating Systems*. PhD thesis, Technische Universität Berlin, 2001.

[Frö02] Antônio Augusto Medeiros Fröhlich. Personal communication, 2002.

[GFd98] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating Feature Modeling with the RSEB. In *Proc. of the 5th International Conference on Software Reuse*, pages 76–85, Victoria, Canada, June 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995. ISBN 0-201-63361-2.

[Gnu] GNU m4 Macro processor Homepage. See `http://www.gnu.org/software/m4`.

[Gos95] James Gosling. Java: an overview. 1995. `http://java.sun.com/people/jag/OriginalJavaWhitepaper.pdf`.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[Gri00] Martin L. Griss. Implementing product-line features with component reuse. In *International Conference on Software Reuse*, pages 137–152, 2000.

[GSPS01a] Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, October 2001.

[GSPS01b] Andreas Gal, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Open Components. In *Proc. of the First OOPSLA Workshop on Language Mechanisms for Programming Software Components*, Tampa, Florida, October 2001.

[HFC76] A. N. Habermann, L. Flon, and L. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

[HU96] Urs Hölzle and David Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(4):355–400, 1996.

[JBR99] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[JZ01] S. Jarzabek and H. Zhang. XML-based Method and Tool for Handling Variant Requirements in Domain Models. In *Proc. of 5th IEEE International Symposium on Requirements Engineering RE01*, pages 116–173, Toronto, Canada, August 2001. IEEE Press.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, November 1990.

[KLLK02] Kyo C. Kang, Kwanwoo Lee, Jaejoon Lee, and Sajoong Kim. Feature Oriented Product Lines Software Engineering Principles. In *Domain Oriented Systems Development — Practices and Perspectives*, UK, 2002. Gordon Breach Sience Publishers. to appear.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, June 1997.

[Kru93] Robert W. Krut. Integrating 001 Tool Support into the Feature-Oriented Domain Analysis Methodology. Technical Report CMU/SEI-93-TR-11, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[Kru02] Charles Krueger. Variation Management for Software Production Lines. In *Proc. of the 2nd International Software Product Line Conference*, volume 2379 of *LNCS*, pages 37–48, San Diego, USA, August 2002. ACM Press. ISBN 3-540-43985-4.

[LHB01] Roberto E. Lopez-Herrejon and Don Batory. A standard problem for evaluating product-line methodologies. In *Third International Conference on Generative and Component-Based Software Engineering*, 2001.

[LKMM94] M. Löfgren, J. Lindskov Knudsen, B. Magnusson, and O. Lehrmann Madsen. *Object-Oriented Environments - The Mjølner Approach.* Prentice-Hall, 1994.

[LMB+01] Akos Ledeczi, Miklos Maroti, Arpad Bakay, Gabor Karsai, Jason Garrett, Charles Thomason, Greg Nordstrom, Jonathan Sprinkle, and Peter Volgyesi. The generic modeling environment. In *Proceedings of the Workshop of Intelligent Signal Processing WISP'2001*, Budapest, Hungary, May 2001. IEEE Computer Society.

[LP01] Radu Litiu and Atul Prakash. Dacia: A mobile component framework for building adaptive distributed applications. *ACM Operating Systems Review*, 35(1-3), 2001.

[Mau] Chris Maunder. Interview with Herb Sutter. see `http://www.codeproject.com/interview/herbsutter3032002.asp`.

[MBL97] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for Java. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 132–145, New York, NY, 1997.

[MCE02] Anne-Francoise Le Meur, Charles Consel, and Benoit Escrig. An environment for building customizable software components. In *Proceedings of the IFIP/ACM Conference on Component Deployment*, Berlin, Germany, June 20–21 2002.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, New York, NY, 1 edition, 1992.

[MKKW99] Mike Mannion, Barry Keepence, Hermann Kaindl, and Joe Wheadon. Reusing single system requirements from application family requirements. In *International Conference on Software Engineering*, 1999.

[Moo75] Gordon E. Moore. Progress in digital integrated electronics. In *Proceedings IEEE Digital Integrated Electronic Device Meeting*, pages 11–13, December 1975.

[Nei80] James M. Neighbors. *Software construction using components*. PhD thesis, Department Information and Computer Science, University of California, Irvine, USA, 1980.

[Nei84] James M. Neighbors. The Draco Approach to Constructing Software from Reusable Components. *IEEE Transactions on Software Engineering*, 10(5):564–573, September 1984.

[Obj99] Object Management Group. Real-time CORBA Joint Revised Submission, March 1999. OMG Technical Document orbos/99-02-12.

[Obj00] Object Management Group. The Common Object Request Broker: Architecture and Specification Revision 2.4, 2000. OMG Technical Document formal/00-10-60.

[Obj02] Object Management Group. Minimum CORBA V1.0, 2002. OMG Technical Document formal/02-08-01.

[OSE01] OSEK/VDX Steering Committee. OSEK/VDX OIL: OSEK Implementation Language Version 2.3, September 2001. `http://www.osek-vdx.org/`.

[Par76] D. L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, SE-5(2):1–9, 1976.

[Par79] D. L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, 1979.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall International Editions, New York, NY, 1991.

[Roe02] Sascha Roemke. XML-Based Modular Transformation System. Master's thesis, Computer Science Faculty, University Magdeburg, Magdeburg, Germany, 2002. In German.

[Rog97] David Rogerson. *Inside COM*. Microsoft Press, Redmond, WA, 1997.

[SB00] Yannis Smaragdakis and Don Batory. Mixin-based programming in C++. 2000. `http://www.netobjectdays.org/node00/de/Conf/publish/talks.html`.

[SC92] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C news. In *Proc. of the Summer 1992 USENIX Conference*, pages 185–198, San Antionio, Texas, 1992.

[Sch95] Henning Schmidt. *Dynamisch veränderbare Betriebssystemstrukturen*. PhD thesis, Universität Potsdam, 1995.

[SCK+96] M. Simos, D. Creps, C. Klinger, L. Levine, and D. Allemang. STARS Organizational Domain Modeling (ODM) Version 2.0. Technical report, Lockheed Martin Tactical Defense Systems, Manassas, VA, USA, 1996.

[SEI97] SEI. *Model-Based Software Engineering*. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA, 1997. `http://www.sei.cmu.edu/mbse`.

[SH01] Douglas C. Schmidt and Stepen D. Huston. *C++ Network Programming: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Reading, MA, 2001.

[Sol01] Richard M. Soley. Model Driven Architecture: An Introduction, 2001. OMG Presentation at `http://www.omg.org/mda/mda_files/MDA_Seminar_Soley6.pdf`.

[SP94] Wolfgang Schröder-Preikschat. *The Logical Design of Parallel Operating Systems*. Prentice Hall International, 1994. ISBN 0-13-183369-3.

[SP02] Wolfgang Schult and Andreas Polze. Aspect-Oriented Programming with C# and .NET. In *Proceedings of International Symposium on Object-oriented*

Real-time distributed Computing (ISORC) 2002, pages 241–248, Crystal City, VA, USA, 2002. IEEE Press.

[SRPC02] Andreas Speck, S. Robak, Elke Pulvermüller, and Matthias Clauss. Version-based approach for modeling software systems. In *Proceedings of ECOOP Workshop on Model-based Component Systems*, Malaga, Spain, 2002.

[Suna] Sun Microsystems Computer Corporation. Enterprise JavaBeans Homepage. `http://java.sun.com/products/ejb`.

[Sunb] Sun Microsystems Computer Corporation. JavaCard Technology Page. http://www.javasoft.com/products/javacard/.

[Szy99] Clemens Aldon Szyperski. *Component Software: Beyound Object-Oriented Programming*. Addison-Wesley, 1999.

[Ten00] David Tennenhouse. Proactive computing. *Communications of the ACM*, 43:43–50, May 2000.

[VA98] A. D. Vici and N. Argentinieri. FODAcom: An Experience with Domain Analysis in the Italian Telecom Industry. In *Proc. of the 5th International Conference on Software Reuse*, pages 166–175, Victoria, Canada, June 1998.

[vDK02] Arie van Deursen and Paul Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, pages 1–17, 2002.

[Vir01] Mirko Viroli. Parametric polymorphism in Java: an efficient implementation for parametric methods. In *Selected Areas in Cryptography*, pages 610–619, 2001.

[Weg86] P. Wegner. Classification in Object-Oriented Systems. *ACM, SIGPLAN Notices*, 21(10):173–182, 1986.

[WL99] David M. Weiss and Chi Tau Rober Lai. *Software Product-Line Engineering: A Family-Based Software Development Approach*. Addison-Wesley, 1999. ISBN 0-201-69438-7.

[WMR$^+$94] Stephen R. Wheat, Arthur B. Maccabe, Rolf Riesen, David W. van Dresser, and T. Mack Stallcup. PUMA: an operating system for massively parallel systems. In *Proceedings of the 27th Hawaii International Conference on System Sciences*, volume II, pages 56–65, Wailea, HI, USA, 1994.

[WO] Project Homepage Workshop for Object Oriented Design and Development of Embedded Systems (WOODES). see `http://woodes.intranet.gr`.

[WRW96] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the Java system. *Computing Systems*, 9(4):265–290, 1996.