Finegrain Application Specific Customization for Embedded Software*

Danilo Beuche, Olaf Spinczyk, Wolfgang Schröder-Preikschat
Otto-von-Guericke-Universität Magdeburg
Universitätsplatz 2
D-39106 Magdeburg, Germany
{danilo,olaf,wosch,}@ivs.cs.uni-magdeburg.de

Abstract

The paper describes techniques which have been developed to simplify the customization of the PURE operating system family for embedded systems and can be applied to almost any embedded software intended for reuse. The approach is based on feature modeling and the use of aspect-oriented programming and supported by a complete tool chain.

1 Introduction

Software engineers who are not familiar with the problems of embedded systems often wonder why so little of their nice ideas about how software should be designed make it into embedded software. This is especially true for deeply embedded systems. Object-oriented programming with 8 and 16 bit microcontroller is an exception of the rule. C and assembly are the predominant languages in this field. Reuse of software is very limited. One could say — and many do believe — , that this is caused by the fact, that most of the programmers are engineers or physicists not computer scientists.

But this is only one half of the story. The question to ask is: Could a computer scientist program nice, reusable software for the same problem in the same time which fits into the microcontroller? This is the heart of the problem: Most of the concepts for reuse developed for workstation or pc class software are not feasible for programming small microcontrollers with limited processing power and memory in the low kilobytes. The trade-off for reusable software is its runtime efficiency and/or the required memory space.

The PURE operating system family [1] is targeted at deeply embedded systems and it is implemented using the object-oriented language C++. Several publications have shown that PURE is able to meet the requirements of deeply embedded

*This work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG), grant no. SCHR 603/1-1

systems. This is mainly due to the capability of PURE of being very fine-grain configurable. However this capability also was one of the most serious problems during the development of PURE: namely, to make users able to deal with the vast number of configuration decisions. Pure allows the user to control the provided functionality on a very detailed level because only by excluding every unneeded PURE functionality it is possible to make applications fit into a small microcontroller.

This problems lead to the development of additional techniques which made the PURE system easier to use and eventually opened previously impossible ways of doing things.

Using a small example throughout the paper it will be shown that it is possible to provide programmers with reusable abstractions suitable for embedded use. The next section introduces the example and explains the relevant properties of the implementation. The following section introduces the feature-based modeling for software used to hide the reusable implementation from a deployer of a reusable abstraction. The fourth section provides ideas on the use of aspect-oriented programming in embedded contexts. The concluding section summarizes the work and presents some ideas for future work.

2 There is no right way

The problem of reuse 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.

Application 2 The second application requires a high precision of the returned cosine value, 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 3 applications using the standard iterative algorithm shown in figure 1, which returns correct values 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.[1]

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

```
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 1: Sourcecode for iterative cosine calculation

A different solution (see figure 2), which provides deterministic runtimes 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 the known values the accuracy of the result differs. Using more values consumes more data memory to store the table.

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 very easy to implement a purely table based cosine function (see figure 3). No calculation is required, especially no floating point operation at all occurs.

The code sizes for the different implementations are very different. Table 1 shows code and data space requirements for a number of different platforms ranging from 8bit controllers to 32bit processors. The application consisted of a single call to the cosine function in main. The void application is just an empty main function.

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

Table 1: Code and data sizes (bytes) for sample cosine applications

```
#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: Sourcecode for cosine calculation with interpolation

```
#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 3: Sourcecode for cosine calculation with table

Taking the requirements of the applications into account, a good embedded programmer would choose to use implementation 1 for the first application, because it provides the best accuracy and consumes no valuable data memory (beside the used stack space). Implementation 2 goes with the application 2 as it provides the required real-time characteristics. For application 3 obviously the implementation 3 is best suited without discussion.

Three times cosine, three different implementations. This is the crux of embedded programming: in most cases there is not just one right way to do something. Reuse concepts for embedded systems have to take this into account.

3 Putting the puzzle together

The most simple solution to the reuse problem is to provide a library with all three implementations (or more). But here the problems already start:

1. How to choose the right implementation from the library?

2. How to use more than one implementation in the same system?

Using an informal description of library function properties for each function is only feasible for a small number of functions. Larger libraries designed for reuse should provide more support for the user of the library in her or his search for an appropriate implementation.

The remainder of the section describes the approach used to make the PURE operating system family easier to (re-)use. As a demonstration example the cosine problem will be used.

3.1 Step 1 — Analysing and Modeling of the Problem Domain

The first problem is that on one side there is a user who needs a function with a set of properties. On the other side there are a number of implementations which fit more or less to the needs of the user. To bring both sides together it is necessary to establish a common language to describe the requirements and properties of software components (functions, classes, modules, ...).

The approach chosen for PURE was feature modeling. Feature modeling is a relatively simple approach for modeling the capabilities of a software system introduced by Kang et al. [4]. A feature model represents the commonalities and variabilities of the domain. A feature in FODA[2] is defined as an *end-user visible characteristic of a system*.

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

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 what this feature is about. This definition may be given as informal text only or in a defined structure with predefined fields and values for some information like the binding of the feature, i.e. 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 value Each feature can have an attached type/value pair. This allows to describe non-boolean features more easily.[3]

feature relations The feature relations define valid selections of features from 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 2 gives an explanation on these terms and shows its representation in feature diagrams.

[2] Feature-oriented Domain Analysis

[3] Typed features with values are not part of the original feature model proposal. However, this extension is required to describe many domains and has been proven to be very useful.

Feature Type	Description	Graphical Representation
mandatory	Mandatory feature B has to be included if its parent feature A is selected	A — B
optional	Optional feature B may be included if its parent feature A is selected	A — ○B
alternative	Alternative features are organized in alternative groups. Exactly one feature of such the group B,C,D has to be selected if the groups parent feature A is selected	A — B C D

Table 2: Explanation of feature diagram elements

From characteristics of the problem a domain analyst derives the features relevant for the problem domain. 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 4.

The feature model of a problem domain (in our case the cosine world) can be presented to an application engineer and she or he should be able to select the feature the application requires. If necessary feature values have to be set.

The CONSUL[5] environment developed in our group allows to check the selection interactively and shows whether there are errors in the selection (invalid feature combinations) or open selections (i.e. open alternative feature groups). The evaluation of feature models and associated rules are done with help of a Prolog interference engine.

3.2 Step 2 - Mapping the Features to Implementations

Once the feature selection is finished it must be mapped to an implementation. The number of valid feature combinations is in most cases too high to provide specific implementations for each possible selection.

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

[5] COnfiguration SUpport Library

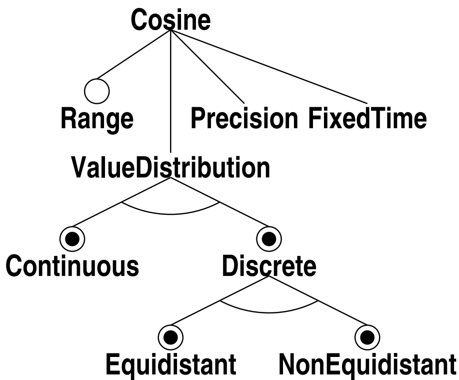

Figure 4: Feature model of cosine domain

In the given example only three different implementations are available. Implementation 1 should only be used if `FixedTime` is not selected. If the input angle values are equidistant implementation 3 seems to fit best. But even with equidistant values when the number of input values gets very high (e.g. > 360) implementation 2 or 1 could be used to save memory. If values are not equidistant implementation 2 can be used if `FixedTime` is selected and implementation 1 otherwise. Further enhancements could be done when more information about the target system is available. For instance, if it would be possible to detect whether the platform has a hardware floating-point unit, the cosine function of the floating-point unit should be called directly in every case.

The question is how to describe these dependencies. The description must be easy to use and expressive enough to allow complex dependency rules. Acknowledging the fact, that embedded programming requires the use of different languages like assembly, C , C++ and others, it has to be a language independent way.

The CONSUL component description language allows to attach such dependency rules to any construction element of a software system (component, class, function, makefile, object, file etc.). Like the feature model the component description is evaluated using Prolog and the result is a description of the software system to be generated. In turn a generator interprets this description to build the actual system from it.

Figure 5 shows a simplified[6] component description for the example.

3.3 Step 3 - Taking Benefit from Aspect-Oriented Programming

Looking at the given implementations, it is obvious that the feature `Range` is not implemented by any of the alternatives. It shouldn't do any harm, because each implementation is able to cope with any angle. But if a check of the angle is

[6]To save space more complex configuration rules have been omitted. Nevertheless it is a complete description.

```
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: (Simplified) component description for cosine component

required e.g. for security reasons an additional optional subfeature `CheckedRange` of `Range` could be introduced.

The implementation of this feature is quiet easy, a simple comparison will do. But it has to be introduced in all implementations only if the corresponding feature is selected. The common solution for this scenario in C/C++ programs is to use a preprocessor macro here, which inserts the required code if a preprocessor variable is set. Using the CONSUL component description it would be easy to do it this way.

But closer examination reveals, that this approach is quite problematic: each implementation has to be changed and the implementation itself gets more complicated. Help comes from a relatively new programming concept, aspect-oriented programming (AOP). AOP was introduced by Kiczales [6] and is about separation of concerns. An *aspect* allows to implement a feature of a software system in a modular way that crosscuts different parts or, in our case, different configurations. In the cosine example the range check feature is obviously independent from the rest of the cosine implementations, but the implementation effects alle three versions of the function.

The AspectC++ language [7] developed in our group enriches the language C++ with the aspect concept. Figure 6 illustrates how an aspect can be used in our example to implement the range check feature throughout all versions of the cosine function. By adding this aspect to the component description (figure 5) it can be

directly enabled and disabled by selecting the CheckedRange feature.

```
aspect CosRange {
   pointcut cosfct (const int arg) = args (arg) &&
                                     execution ("double cosinus(...)");
public:
   advice cosfct (arg) : void before (const int arg) {
      // ARGMIN and ARGMAX are ``feature values''
      if (arg < ARGMIN || arg > ARGMAX)
         appropriate_action ();
   }
};
```

Figure 6: Sourcecode for the CosRange aspect

It is not necessary to change the three cosine function implementations because the AspectC++ language guarantees that the code following the keyword advice is always run before cosinus is executed. This is achieved by binding the advice code to the pointcut cosfct. A pointcut describes the points in the program code were an aspect can interfere. The pointcut arguments can be used to expose context information from these points[7].

4 Related Work

Software engineering techniques like feature modeling are getting more and more interest from the embedded community. Kang who was one of the developers of the FODA methodology presented with FORM[8] [5] an approach to use feature model based techniques for distributed, component based command and control systems. FODAcom [8] applies feature modeling to the telecom domain. All these approaches lack a complete tool chain for the developers which allows to generate systems from a feature model. They use feature models mainly to organize the design and implementation work.

The FAST method of Weiss and Lai [9] describes the process to develop and deploy customizable software families and requires the implementation of generators. However the modeling approach is not based on feature models.

5 Current State and Future Development

The feature modeling and aspect-oriented programming techniques described in the previous sections have been implemented in our group in prototypical form [3], [2]. The resulting tools have been used to build an interactive configuration enviroment for the PURE operating system family. The resulting feature model consists

[7]Interested readers should visit www.aspectc.org

[8]Feature-Oriented Reuse Method

of about 220 features which are used to configure the 57 components consisting of some 350 classes.

Although these tools made the configuration of PURE much easier, there are still a number of unresolved issues. Reuse of parts of a feature model across problem domains is not yet possible. To be able to reuse feature models in different domains it should be possible to merge feature models but no currently available feature modeling approach allows this.

The AspectC++ environment is still in its early stages but has already been used successfully with PURE, however many C and C++ constructs are not yet fully supported.

References

[1] 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 *IEEE Proceedings ISORC'99*, 1999.

[2] D. Beuche, W. Schröder-Preikschat, O. Spinczyk, and U. 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.

[3] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. AspectC++: Language Proposal and Prototype Implementation. In *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa, Florida, Oct. 2001.

[4] 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, Nov. 1990.

[5] K. C. Kang, K. Lee, J. Lee, and S. Kim. Feature Oriented Product Lines Software Engineering Principles. In *Domain Oriented Systems Development — Practices and Perspectives*, UK, 2002. Gordon Breach Sience Publishers. to appear.

[6] 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.

[7] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: An Aspect-Oriented Extension to C++. In J. Noble and J. Potter, editors, *Proceeding of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, Feb. 2002.

[8] 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.

[9] D. M. Weiss and C. T. R. Lai. *Software Product-Line Engineering: A Family-Based Software Development Approach*. Addison-Wesley, 1999. ISBN 0-201-69438-7.