Main seminar 'Autonomic computing'

Operating Systems and Middleware

Reusable components in embedded systems

Marcel Kilgus, Software engineering, 8th semester
Department of distributed systems
University of Stuttgart

Talk to be given on December, 17th 2004

Abstract

The paper describes why the software development process in the embedded world often differs from the academic versions and why concepts like software reuse is not often seen there. Then it introduces some solutions that address these problems, namely the feature oriented domain analysis in form of the CONSUL configuration support library and the concept of aspect oriented programming in form of the AspectC++ language.

1. Introduction

With technology advancing everywhere so called “embedded systems” become more and more important. Embedded systems are computer systems that, unlike a PC for example, are part of a bigger engineering system. Embedded systems control your washing machine, your television set and the air bag in your car.

Although the main part of an embedded system is usually just a piece of software, traditional strategies for software engineering typically do not apply. Paradigms like “reusable components”, “object oriented development” or “data abstraction” can not be found often in the embedded area. The reasons for this are many, the main one being cost. The manufacturing of the systems is most often very high volume, and in this case every kilobyte of RAM or ROM matters. The CPUs employed are slow (but well tested and reliable) and the hardware is often very restrained in every aspect, making the software development some kind of a challenge with the result having to exactly fit into the hardware provided strait jacket. Waste is not an option.

In terms of languages, embedded system often still employ hand crafted assembler routines or C code at best. Modern software development on the other hand relies on the fact that CPU power and memory is available in abundance, the paradigm being that it's cheaper to add more resources to the hardware than to develop a more restrained software. The languages get more and more high level, relieving the programmer of the hassle of interacting with the hardware directly so that they can concentrate more on the “big picture”. This especially makes reusable components easy, one just implements (or let somebody else implement, like in standard libraries) the most powerful version one can think of and uses this solution for all occurrences of a similar problem. One size fits all. Or should fit all. In practice components often have to be reworked several times until they are truly reusable, but this is outside of the topic of this paper.

The following chapters will present some concepts that try to remedy the problem. At first the feature oriented domain analysis will be introduced and an example written for the CONSUL configuration support library will be presented. After that the concept of aspect oriented programming will be explained, with examples in the AspectC++ language. And finally the conclusions chapter will summarize and evaluate the things described in this paper.

2. One size does not fit all

The example stated in [FESC] to illustrate the point that embedded systems can have pretty specific needs is based on the cosine function, and as this example is just too good and illustrative to pass on it will also be used here.

Processors in embedded systems usually do not have a floating point calculation unit and therefore floating point arithmetics can be quite expensive. At the same time CPU power is limited, therefore a high precision cosine calculation function does not always provide the optimal trade-off. There are a few different imaginable scenarios:

1. An exact value is needed, time does not matter
2. Only a rough approximation is needed, but that very quickly
3. The function domain only includes a few discrete values, but those need to be provided fast and in high precision.

In modern software engineering all three cases would probably be served by the same function. The compiler will most likely even translate it into a single native processor instruction, in the case of the x86 architecture into an FCOS instruction, which on a Pentium 4 Northwood core only needs a blazing 240 clock cycles in *worst case* [IA32O]. Considering clocks in the range of several gigahertz this is virtually nothing.

For embedded processors whose speed is usually still measured in one or two digit megahertz values, different implementations for the 3 cases become the only option. Case one could be served by an iterative function that returns once the result becomes stable, case 2 can be solved by interpolating between known values and case 3 is an ideal prerequisite for probably the oldest software trick known to mankind: the table lookup. Example source code for all 3 variants can be found in [FESC].

Now, despite those circumstances, component reuse is highly desirable. For once, re-inventing the wheel is never a particularly good idea. It is expensive, valuable time is wasted and the resulting implementation is often not as highly tested as it would be with standard components.

A library aimed at code-reuse in embedded systems should therefore provide all 3 implementations. But this already poses some questions:

1. How does one know which implementation suits the problem best?
2. How can several implementations co-exist in one and the same library?

One could use some informal description in the function headers to distinguish between the different version and give the functions different names, like CosIterate, CosAverage and CosTable. On first glance this could solve both problems, but even with this very limited example the naming scheme already looks ugly and impractical and the whole method is not feasible for a library with a higher function count. A C programmer of course could counter point one with the software equivalent of a hammer, the preprocessor:

```
#define Cos(x) CosTable(x)
```

This way one can still write code independent from the actual implementation used, but this too is only feasible with a limited amount of functions, as it involves a lot of manual work and care.

3. Feature oriented domain analysis

Lets have a deeper look at problem number one, how to know which implementation is best suited for the problem at hand. On one side there is the application programmer with his specific needs for a certain problem, on the other side there are a number of implementations that hopefully fit his needs more or less. To build a bridge between the two sides a common language is necessary that

can describe the prerequisites and properties of a software component (function, class, module etc.).

One way to do this is called “feature modeling”. Already described in [FODA] in 1990 this method is far from new, but still not very widely known. To introduce the concept some definitions have to be explained first:

- **problem domain:** several problems that share some common capabilities and/or data. In the example above these would be the different cosine implementations. All 3 variants calculate the cosine, therefore reside in the same problem domain, while implementing it differently with different capabilities and restrictions.
- **feature:** a characteristic of a system that can be seen by the end-user. For example one feature of all the cosine implementations above is, believe it or not, that they calculate the cosine of a given angle. Another feature could be the range the functions accept or the precision they do the calculation with.
- **feature model:** the model contains all the features the members of a problem domain have in common and all the features that distinct them from each other. It is made up of feature descriptions, feature values and feature relations.
- **feature description:** consists of a feature definition and a rationale. The definition explains which aspect of the problem domain is modeled by the feature so the user can understand what this feature is about. It may be in form of a human readable informal text or in a standardized structure of field/value pairs that represent common aspects like the binding time of the feature (configuration time, compile time...).
  The rationale should help the user decide whether to use the feature or not.
- **feature value:** every feature can have an additional type/value pair. With those pairs non-boolean properties can be described more easily.
- **feature relations:** this defines a valid set of features of a domain. It is mainly represented by so called *feature diagrams*. A feature diagram is a directed acyclic graph with the nodes being features and the connections being indicators of whether the feature is optional, mandatory or an alternative. Table 1 shows the possible relations (graphics copied from [FESC]):

Feature type	***Description***	***Graphical representation***
mandatory	Feature B must always be included if its parent feature A is selected.	
optional	Feature B is optional and may be included if feature A is selected.	

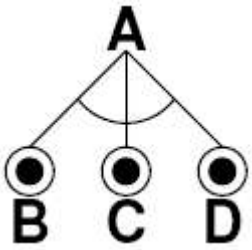
Feature type	*Description*	*Graphical representation*
alternative	Exactly one of the features B/C/D has to be selected if the parent feature A is selected (in other words one of the feature group B/C/D is mandatory and they are mutually exclusive!).	

Table 1: Explanation of feature diagram elements

When analyzing the domain of the cosine problem above one could come up with the feature diagram shown in Figure 1. The root is obviously “cosine”, because that's what all implementations calculate. The implementations could however differ in the angles they accept (range or distribution), the precision they deliver or the runtime they need to calculate the result.

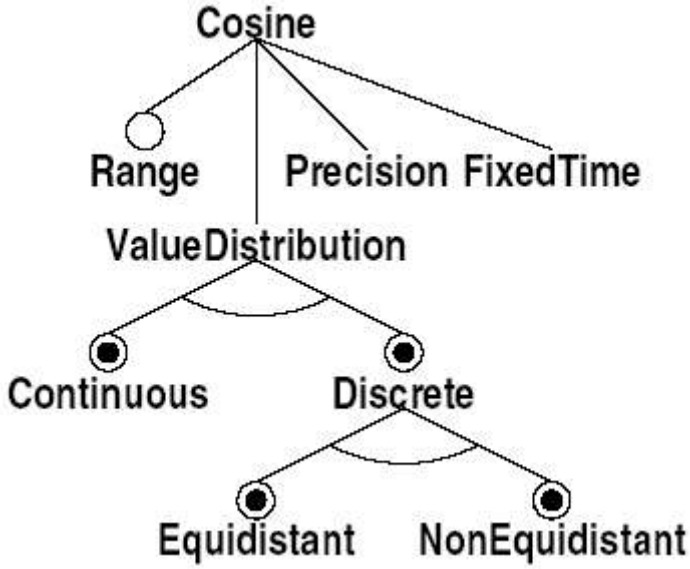


Figure 1: Feature model of cosine domain [FESC]

Of course in most cases there would be too many valid feature combinations to match the actual number of implementations.

In the example above we've seen 3 different approaches to calculate the cosine. Approach one (the iterative version) is only appropriate if *FixedTime* is not selected. A table based approach is only possible if *Discrete* is true and even then it probably only makes sense when the number of discrete angles is comparatively small. Otherwise the interpolation code might be a good compromise.

Now the problem is how to express all these dependencies. To address this issue the OS research group at the University of Magdeburg has developed the CONSUL component description language [CONSUL]. In CONSUL the component description is evaluated using Prolog and a description of the software system is generated. This in turn is read by an interpreter that generates the actual system.

Figure 2 shows a simplified component description for the cosine example. The header "cosine.h" will be included in all cases by the generator, the actual source file with the code will be chosen according to the Prolog terms in the Restrictions statements.

Looking back at Figure 1 there is one feature that has not been mentioned yet: the Range feature. According to the diagram the feature is optional, in other words the user has the choice to restrict the range of the cosine function, for example for security reasons.

The check itself is of course very easy, a simple comparison of the range to the allowable interval will do. But for performance reasons this check should only be implemented when it's really required by the user. The C programmer will immediately bring out his hammer, the preprocessor, and write a macro that includes code on demand. The problem with this approach is that the feature is universal to all 3 implementations and therefore all 3 source files would have to be altered. Beware, source code redundancy ahead! A logistical nightmare in any bigger project. Therefore in this case something called "aspect oriented programming" enters the stage.

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

        file("src", "cosine_iterate.c", impl) {
          Restriction {Prolog("not(has_feature('FixedTime',_NT))")}}

        file("src", "cosine_average.c", impl) {
          Restriction {Prolog("has_feature('FixedTime',_NT),
                               has_feature('NonEquidistant',_NT)"}}

        file("src", "cosine_table.c", impl) {
          Restriction {Prolog("has_feature('FixedTime',_NT),
                               has_feature('Equidistant',_NT)"}}
      }
    }
  }
  Restrictions {Prolog("has_feature('Cosine',_NT)")}
}
```

Figure 2: Example of component description of cosine implementations [FESC]

4. Aspect oriented programming

First introduced by Kiczales [AOP] in 1997 aspect oriented programming addresses the problem of *cross-cutting concerns*. These concerns are secondary functionalities several software components have in common, like in above example where all 3 codes may share the "Range" feature. Checking the range is not part of the main functionality of the cosine calculation. It is however a fixed piece of code that can be put at the beginning of each function without affecting the main code block. In aspect oriented programming the "Range check" would be called an *advice* and the start of the different cosine functions a *point-cut*. Those two together, a point-cut and an advice, form an *aspect*.

With those concepts in mind the people from the OS department at the University of Magdeburg once again went ahead and created AspectC++ [ASPC], the C++ answer to the existing AspectJ from Palo Alto Research Center PARC [ASPJ]. Figure 3 shows how the Range feature could be implemented using the AspectC++ language.

```
Aspect CosRange {
  pointcut cosfct(const int angle) = args(angle) &&
                                     execution("double cosine(...)");

public:
  advice cosfct(angle) : void before (const int angle) {
    // ARGMIN and ARGMAX are "feature values"
    if (angle < ARGMIN || angle > ARGMAX)
      // Some appropriate reaction (exceptions anyone?)
  }
};
```

Figure 3: Source code for the CosRange aspect, adapted from [FESC]

The *CosRange* aspect here consists of the pointcut *cosfct* and the advice for this pointcut. The *execution("double cosine(...)")* expression sets the pointcut to the function called "cosine" with an undetermined amount of parameters. The *args(angle)* term then maps the first argument of the "cut" cosine function to the parameter "angle".

The advice is defined as *before*, therefore it will be run before the execution of the cosine function it cuts into. Also possible declarations would be *after,* which would obviously be run after the function, and *around*, which completely encapsulates the original routine into the advice [ACLR].

By adding the file with the aspect code to the component description shown in Figure 2 the code can automatically be injected into the cosine function when the Range feature is requested.

The AspectC++ compiler is by the way not really a compiler but a complicated pre-processor that analyzes the given code, weaves the advices into the pointcuts and throws the end result at a real compiler, like GCC or Microsoft VisualC++. The clear advantage of this strategy is the independence one buys with it, both of the development platform and the target platform. Also developers don't have to leave their favorite compilers behind: not taking away your programmer's favorite tools is an important part in keeping your programmer happy.

Actually the concept of aspect oriented programming is even more powerful than the example above. Pointcuts can not only inject code at the beginning or end of function calls. Advices could also be "triggered" by accesses to variables for example (*gets*/*sets*, however not yet implemented in current AspectC++ prototype implementations). Triggered is written in quotes here because as mentioned this is not really a dynamic process, instead all the aspect functionality is done statically before the actual compilation of the source files.

Another way to trigger an advice is for example *calls(expression)*, which finds all calls to methods whose signature match the expression. c*lasses(expression)* matches all classes against the expression and *objects(expression)* obviously does the same for all objects. For more variants refer to [ACLR].

```
pointcut listAccess() =
  calls("void List::insert(...)") ||
  calls("void List::remove(...)") ||
  calls("void List::get%(...)") ||
  calls("void List::set%(...)");
```

Figure 4: Example pointcut that matches some list methods

Aspect oriented programming is also predestined to be used for optionally including logging facilities, for example for debugging purposes.

This example in Figure 4 matches the *insert* method, the *remove* method and all methods that start with either *get* or *set* (% is a wildcard in AspectC++). This could then be used with an advice that logs changes to the list. A simple and admittedly not too useful example is given with this advice:

```
advice listAccess(): void around() {
  cout << "The list will be accessed." << endl;
  proceed();
  cout << "The list has been accessed." << endl;
}
```

Figure 5: Matching advice for the listAccess pointcut

It uses the *around* term, i.e. it is called *instead* of the original function. The advice can then call the original code using the *proceed()* statement.

Perhaps one more example, this time a bit more useful:

```
advice calls("void GraphElement::setPos(x, y)"):
  void before(int x, int y) {
    cout << "A graphical element should be moved to position ("
         << x << ", " << y << ")." << endl;
}
```

Figure 6: Combined advice and pointcut

In this case the pointcut is given directly in the advice declaration.

5. Conclusion

With CONSUL and AspectC++ the OS department of the University of Magdeburg has provided a complete tool chain to fine-tune code and libraries to the needs of embedded applications development. The PURE operating system, consisting of about 220 features, 57 components and 350 classes was implemented using these tools. Here the software configuration can even be done graphically using the CONSULAT application (see Figure 7), which is a very convenient way to tailor the system to the needs of the developer.

At the same time the tools are still under heavy development. AspectC++ is in an early stage and CONSUL is so early that it is not even publicly available, though copies for evaluation purposes can be obtained. Therefore it's difficult to tell whether these implementations will catch on and experience a wider circulation and use.

Also, the tools are aimed at C++ as the development language, which is still not the norm in embedded development. On the other hand AspectC++ is already quite an interesting new development by itself and could also be useful for normal application development outside of the embedded world.

Last but not least, it is said that most embedded system developers are not computer scientists but engineers of other disciplines and there's certainly some truth to that. Whether these people would be willing to give up their normal development process in favor of new concepts like the domain analysis remains to seen. Big companies like Nokia claim that the introduction of domain specific modeling has increased the productivity tenfold [RTC], therefore it might well be worth looking into. However, the initial effort can be very high and managers only concerned about short term revenues might not be willing to invest that much, even though the long-term gains could very well outweigh these costs.

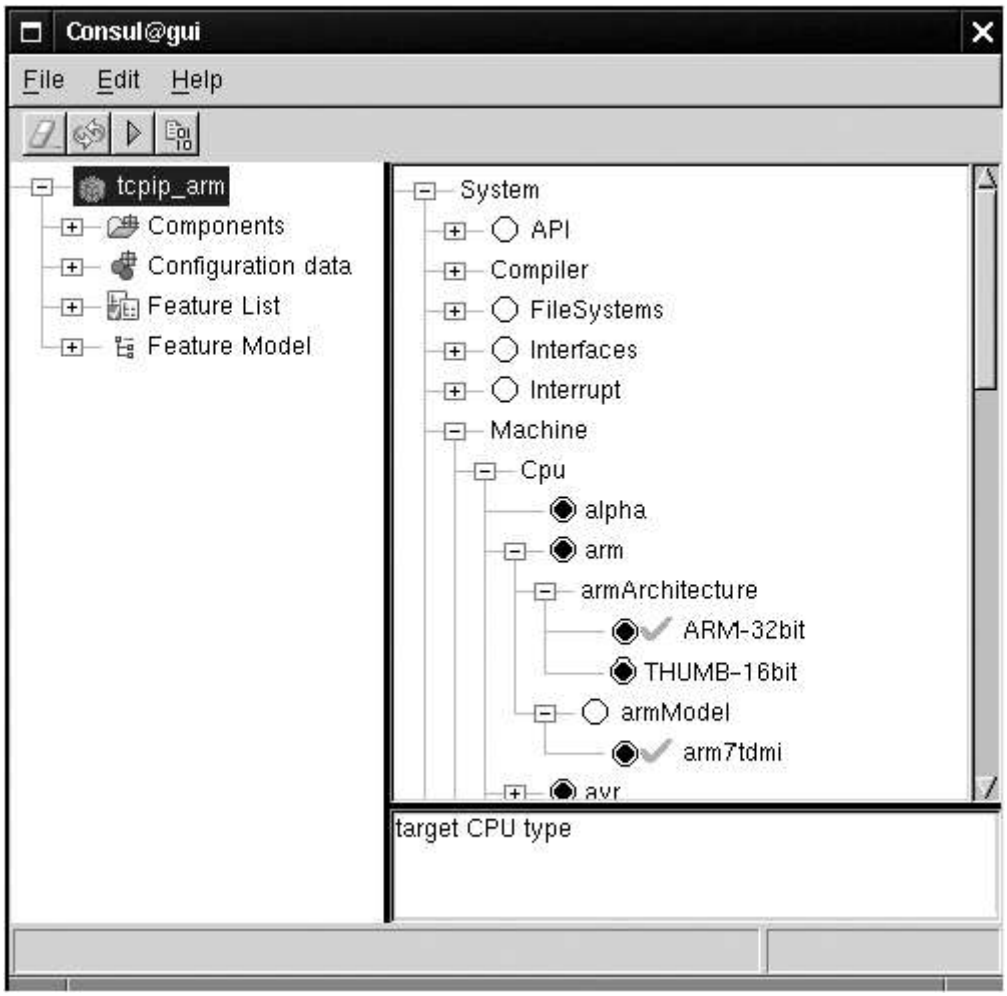


Figure 7: Consul configuration application for PURE [PURE]

6. Bibliography

ACLR: Matthias Urban, Olaf Spincyk, AspectC++ Language Reference, http://www.aspectc.com/ac++languageref.pdf

AOP: G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin: Aspect-Oriented Programming, 1997

ASPC: AspectC++ development group, The Home of AspectC++, www.aspectc.org

ASPJ: Palo Alto Research Center, aspectj project, http://eclipse.org/aspectj/

CONSUL: OS Research Group at the University of Magdeburg, Homepage of CONSUL - Configuration Support Library, http://wwwivs.cs.uni-magdeburg.de/~consul/

FESC: Danilo Beuche, Olaf Spinczyk, Wolfgang Schröder-Preikschat: Finegrain Application Specific Customization for Embedded Software, 2002

FODA: K. Kang, S. Cohen, J. Hess, W. Nowak and S. Peterson: Feature Oriented Domain Analysis Feasibility Study, 1990

IA32O: Intel, IA-32 Intel Architecture Optimization, 2004

PURE: PURE Documentation Team, PURE Documentation, http://ivs.cs.uni-magdeburg.de/~pure/manual.pdf

RTC: Risto Pohjonen, Boosting Embedded Systems Development with Domain-Specific Modeling, http://www.rtcmagazine.com/home/article.php?id=100224