Evaluating Variability Implementation Mechanisms

Claudia Fritsch, Andreas Lehn, Dr. Thomas Strohm

Robert Bosch GmbH
Corporate Research and Development
P.O. Box 94 03 50, D-60461 Frankfurt, Germany
claudia.fritsch@de.bosch.com

Abstract. Developing a software product line involves the management of variabilities. Variabilities have to be controlled during each phase of the software development process. This paper focuses on the implementation of variability in the context of embedded software for automotive systems. Implementing variability requires knowledge of variability implementation mechanisms, but catalogs of such mechanisms are still missing. We present a method to identify, document, and evaluate mechanisms to implement variability. The goal of this paper is to enable a software development team to build and use their catalog of variability implementation mechanisms.

1. Introduction

Developing a software product line requires the management of commonalities and variabilities of a set of products. A *variability* is a difference between distinct products of a product line. The possible differences are defined by a collection of *options* which are associated with a variability. Variabilities have to be controlled during each phase of the software development process. Variabilities manifest as optional features during requirements engineering, as variation points in software architecture and design, and, eventually, as different implementations: Developers must *code the options* and provide means to *select an option for a certain product*.

While feature analysis and design have been elicited, e.g., in [Hein et al. 2000], [Lalanda 1999], and [Thiel, Hein 2002], little work on the implementation of variability has been done [Svahnberg et al. 2002]. Developers who have to implement variability cannot fall back on catalogs containing variability implementation mechanisms. Moreover, different development environments may call for a diversity of mechanisms. Also, required system *qualities* (non-functional requirements) and technical constraints impose restrictions on the selection of an appropriate mechanism.

Yet variability has to be implemented and this has been done for years. Developers have been using mechanisms, self-invented or heard-about, documented or undocumented, being aware of the consequences on system qualities or not.

In this paper we propose a systematic approach: identify and document variability implementation mechanisms, identify qualities essential to your software development team, and evaluate these mechanisms according to those qualities. The result will be a catalog of evaluated mechanisms, specific to your development environment, technical constraints, and qualities essential to you. When implementing a variability, developers may then select an appropriate mechanism from this catalog. With this paper we want to encourage and enable software development teams to build a catalog of their variability implementation mechanisms.

Sections 2, 3, and 4 describe the method we have developed and have been using since: In section 2 we show how to identify and document the mechanisms. Section 3 shows how to find and describe

qualities considered relevant. In section 4 we evaluate the mechanisms according to these qualities. In each of these sections a paragraph labeled *How to proceed* follows, which suggests how the reader could implement this method in his or her development of a software product line. Section 5 summarizes the resulting artifacts and how developers may use them. In section 6 we address difficulties we encountered and how we overcame them.

The method we describe is based on the documentation of variability mechanisms in two Bosch business units. In order to enable other developers to build and use a catalog of their mechanisms we developed this method, and it got evaluated during the documentation of further mechanisms.

2. Identifying and Documenting Mechanisms

Mechanisms are architectural patterns, design patterns, idioms, or guidelines for coding. A mechanism to implement variability must offer two indispensable features:

- the implementation of the options
- the technique to select an option for a certain product

Example: Use preprocessor directives to allow for conditional compiling. E.g., enclose code fragments which deal with option A with `#if defined( A ) ... #endif`, and code fragments which deal with option B `with #if defined( B ) ... #endif`. Then define `A` when compiling the code for a product with option A.

This mechanism has certain drawbacks. It is an often used concept though, and this shows that it is worthwhile to think about the topic we are addressing.

Mechanisms we have documented include branches, dynamic binding, strategy design pattern, conditional statements, and interpreter [Fritsch et al. 2002b]. They allow to put the implementation of different options into different functions, modules, or files.

In addition to the two above-mentioned features, a mechanism to implement variability has a characteristic property, which is closely related to *the technique to select an option for a certain product:*

- the binding time

Binding time is the process step when a variability is decided.

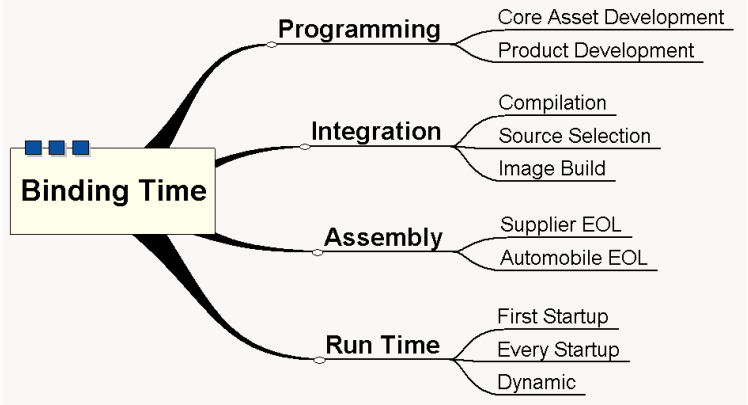

Figure 1: Binding Time. Concerning the implementation of embedded automotive system software, we consider four main phases with binding times: programming, integration, assembly and run time. In *programming,* core asset development may happen much earlier than product development, so these are different binding times. *Integration* builds the executable software from various sources (code, data, resource files, 3rd party components). During *assembly,* software may be configured at End Of Line, e.g., by overwriting parts of flash memory, first at the supplier's line and later at the automobile's line. The latest binding time happens at *run time,* e.g., software may adapt at system startup to the hardware it finds itself in, or even later software may adapt dynamically to hardware changes.

The binding time restricts the selection of a mechanism. E.g., if you need to bind a variability at run time, you can't implement it with a mechanism which is bound at compile time [Fritsch et al. 2002a]. A set of binding times for the implementation of embedded automotive system software is given in figure 1.

How to Proceed. Any mechanism which helps to produce different behavior for different products may be considered. You find them in books (e.g., [Gamma et al. 1995]) and articles ([Svahnberg et al. 2002]). However, the best source may be your own code.

Document these mechanisms, and be precise. We propose to use a template which covers

- mechanism name
- solution
- design diagram
- example code
- how options are implemented
- how an option gets selected
- binding time

You may also want to add rules for what purpose a mechanism may be used or not. The result will be a mechanism catalog which contains mechanisms and their basic properties.

The user of this catalog will need some more criteria to choose a mechanism for a certain implementation task. E.g., if performance is important, he or she must choose a mechanism which does not impede this quality.

In the next section we will identify a set of qualities which is relevant for variability implementation mechanisms. In section 4 we will show how to evaluate mechanisms according to these qualities. With this evaluation at hand, we can tell a priori which qualities a mechanism will introduce in the code and in the system we build.

3. Identifying and Documenting Qualities

A software system has to fulfill functional requirements and certain quality attributes (non-functional requirements) like performance, security, reliability, or modifiability. These qualities shape a system's architecture [Bass et al. 1998]. In the context of variability implementation mechanisms, we are interested in qualities which are *relevant to the implementation of variabilities*.

The qualities we have identified as essential concern development and run time:

- *Development time qualities* show up during the implementation of core assets and product-specific parts, during maintenance, or extending the product line by variabilities or options.
- *Run time qualities* show up when a product of the software product line is used, and cover different aspects like availability, performance, or memory consumption.

These qualities and their sub-qualities can be arranged in a quality tree [Lehn et al. 2002], see figure 2.

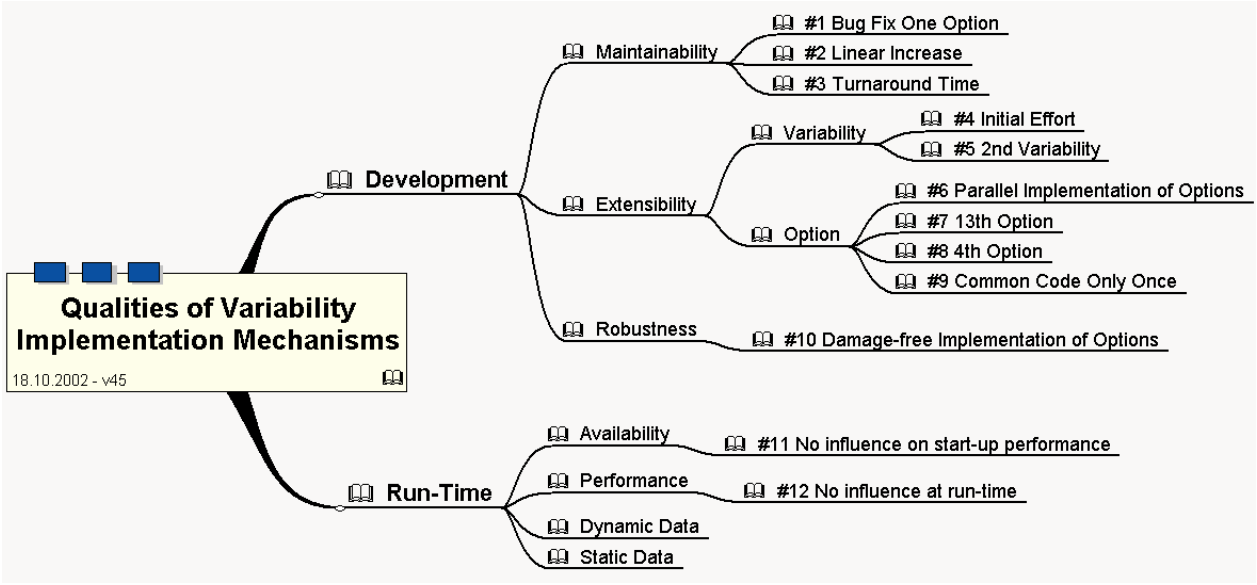

Figure 2: Quality Tree. Qualities may be arranged hierarchically in a quality tree. Here, development gets refined by maintainability, extensibility, and robustness. Extensibility – as a major quality of variability implementation – is even further refined. The leaves of the tree contain the names of quality scenarios which, eventually, define the aspects of a quality precisely.

Quality Scenarios. Our goal is to evaluate the mechanisms according to the qualities. In order to be precise, we have captured the leaf qualities as *quality scenarios*. A quality scenario describes

- context in which the stimulus arrives
- stimulus
- the artifact influenced by the stimulus
- response of the system to the stimulus
- response measures

Thereby we roughly followed the template provided by Bass and Bachmann [Bass, Bachmann 2002].

E.g., quality #5 in the quality tree (figure 2), *2nd Variability,* is refined by: *A module (class, component, ...) has one variability <context>. A second variability has to be implemented <stimulus>. The second variability is independent of the first one, but in the same module <artifact>.* (Independent *means, no code deals with options of both variabilities.) Introducing the second variability <response> is as easy as introducing the first <response measure>.*

For this scenario, it is easy to tell whether a mechanism supports or torpedos it.

How to Proceed. Check the quality tree provided. It should include the situations relevant to you; expand it if necessary. Add all quality scenarios you consider to be important as leaves of the quality tree. Think of the situations you could find yourself in after a couple of months or after three years. Be precise – the above template will help you. The more precise you are, the easier it will be to rank the mechanisms.

4. Evaluating Mechanisms According to Qualities

After documenting the mechanisms and qualities, the mechanisms get *evaluated* according to these qualities. Mechanisms are measured relatively to each other. Usually there is no absolute measure. We found it worthwhile to look at a certain quality and ask for a set of mechanisms: Which mechanism supports it the most? Which mechanism torpedos it the most? In this way, one finds a "good" and a "bad" mechanism. When this is completed, the other mechanisms can be ranked for this quality. The result is a sequence of mechanisms, all sorted according to a certain quality.

Presenting the evaluation. The ranking of the mechanisms should be presented in such a way that a developer, given an implementation task, can choose the best mechanism. We provide two possibilities [Lehn, Strohm 2002]:

1. Kiviat diagram. Kiviat diagrams, also called radar charts, may be used to graphically present the qualities of the mechanisms (figure 3). For each mechanism we draw one diagram. Each axis represents one quality, and each Kiviat diagram has the same axes. On each axis the point gets marked which represents the ranking of this mechanism according to this quality: A point on the innermost ring means, this mechanism is one which supports this quality the least, while a point on the outermost ring means that none of the other considered mechanisms provides better support on this quality.

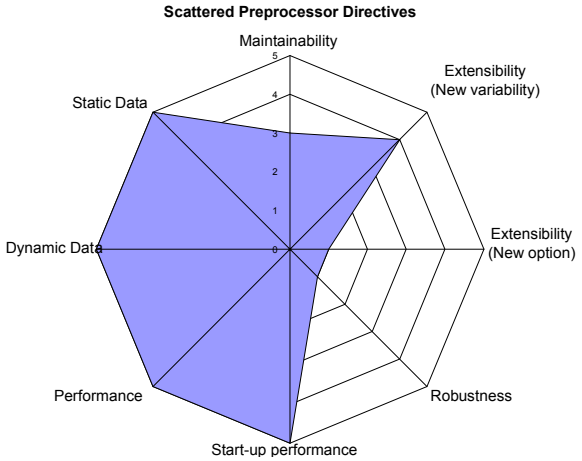

Figure 3: Example Kiviat Diagram. The axes are the same for all mechanisms, each represents a certain quality. For a mechanism we apply its ranking for the different qualities as points on the axes. The figure results from connecting these points. Kiviat diagrams make use of our ability to recognize and remember facial patterns.

We provide a Kiviat diagram for each mechanism. In such a way visualized, the reader can realize the qualities of a mechanism and differences between mechanisms regarding a certain quality at a glance.

2. Mechanism-Quality Matrix. Additionally, we show the quality values of all our mechanisms in a *matrix* with mechanism rows and quality columns. In the fields we put the quality value shown in the Kiviat diagram. The matrix gives an overview of all mechanisms and qualities, and allows to find the best mechanism for a certain quality.

How to Proceed. Rank your mechanisms according to your qualities. Use your knowledge and experience to do the ranking. Start with one quality and look for the mechanisms that provide the most and least amount of support. Rank the other mechanisms relatively. Proceed with the next quality. Produce Kiviat diagrams and a mechanism-quality matrix.

5. Resulting Artifacts

Following our proposal, you will have in hand and may provide the development team with:

- Mechanism Catalog
- Quality Tree
- Kiviat Diagrams
- Mechanism-Quality Matrix

A developer who is looking for a mechanism to implement a variability will then

- use the quality tree to decide which qualities the mechanism should support
- draw the corresponding Kiviat diagram or keep it in mind
- browse the provided Kiviat diagrams to determine a set of appropriate mechanisms
- check these mechanisms' properties in the mechanism catalog
- select the most appropriate mechanism – this may be a trade-off decision
- implement the variability, guided by the mechanism description.

6. Difficulties and How We Dealt with Them

Mechanisms. If we apply certain principles and best practices of design to certain mechanisms, they will gain a lot of quality. E.g., if you restrict the use of `#if`s by a set of rules, the code will tend to remain readable and maintainable. Whereas, if you don't restrict it, the code will become unmaintainable sooner or later, i.e., the mechanism loses quality. Therefore, some mechanisms appear in two variants in our catalog, and they differ in ranking for some qualities.

Qualities. We started with qualities like "Extensibility with respect to variability" but soon found out that this was too woolly for evaluating a mechanism. We had to be more precise. So we defined qualities by quality scenarios, and evaluation on this works better.

Evaluation. Most of the qualities cannot be measured, e.g., there is no general scale according to which one can say "this mechanism ranks 1, this one ranks 4". It is difficult to tell for any given quality how well a mechanism deals with it. What we found feasible was, to look at a certain quality and ask: Which mechanism fulfills it the best? Which mechanism torpedos it the most? These questions were easier to answer. Finally, we sort the other mechanisms for this quality, which gives us the ranking.

References

[Barbacci et al. 1995] M. Barbacci, T. Longstaff, M. Klein, C. Weinstock: *Quality Attributes,* SEI Technical Report CMU/SEI-95-TR-021

[Bass et al. 1998] L. Bass, P. Clements, R. Kazman: *Software Architecture in Practice*, Addison Wesley 1998

[Bass, Bachmann 2002] L. Bass, F. Bachmann: *Specifying and Achieving Non-Functional Requirements*, ECOOP 2002[Buschmann et al. 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *A System of Patterns: Pattern-Oriented Software Architecture*, John Wiley & Sons 1996

[Clements et al. 2002] P. Clements, R. Kazman, M. Klein: *Evaluating Software Architectures,* Addison Wesley 2002

[Clements, Northrop 2001] P. Clements, L. Northrop: *Software Product Lines,* Addison Wesley 2002

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns,* Addison Wesley 1995

[Fritsch et al. 2002a] C. Fritsch, A. Lehn, T. Strohm: *Binding Time*, Robert Bosch GmbH (internal slides)

[Fritsch et al. 2002b] C. Fritsch, A. Lehn, R. Rashidi, T. Strohm: *Variability Implementation Mechanisms – A Catalog*, Robert Bosch GmbH (internal paper)

[Hein et al. 2000] A. Hein, J. MacGregor, M. Schlick: *Requirements and Feature Management for Software Product Lines, 1. Deutscher Software-Produktlinien Workshop (DSPL-1)*, Kaiserslautern, November 2000

[Lalanda 1999] P. Lalanda, *Product-line Software Architecture,* CEC Deliverable No. P28651-D2.2 (ESPRIT Programme)

[Lehn et al. 2002] A. Lehn, C. Fritsch, T. Strohm: *Qualities of Variability Implementation Mechanisms,* Robert Bosch GmbH (internal paper)

[Lehn, Strohm 2002] A. Lehn, T. Strohm: *Evaluation of Variability Implementation Mechanisms,* Robert Bosch GmbH (internal paper)

[Svahnberg et al. 2002] M. Svahnberg, J. van Gurp, J. Bosch: *A Taxonomy of Variability Realization Techniques,* preprint 02/2002

[Thiel, Hein 2002] S. Thiel, A. Hein: *Systematic Integration of Variability into Product Line Architectural Design,* in: G. J. Chastek (ed.): Software Product Lines (Proceedings of 2nd Software Product Line Conference (SPLC-2), San Diego, CA, USA; Lecture Notes in Computer Science LNCS 2379, Springer-Verlag, pp. 130-153, 2002