Managing Variability in Software Architectures[1]

Felix Bachmann*
Carnegie Bosch Institute
Carnegie Mellon University
Pittsburgh, Pa 15213, USA
fb@sei.cmu.edu

Len Bass
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pa 15213, USA
ljb@sei.cmu.edu

ABSTRACT

This paper presents experience with explicitly managing variability within a software architecture. Software architects normally plan for change and put mechanisms in the architecture to support those changes. Understanding the situations where change has been planned for and recording the options possible within particular situations is usually not done explicitly. This becomes important if the architecture is used for many product versions over a long period or in a product line context where the architecture is used to build a variety of different products. That is, it is important to explicitly represent variation and indicate within the architecture locations for which change has been allowed.

We will describe how the management of variations in an architecture can be made more explicit and how the use of variation points connected to the choices a customer has when ordering a product can help to navigate to the appropriate places in the architecture.

General Terms

Documentation, Design

Keywords

Software architecture, product lines, requirement analysis, strategic reuse, variability, variation points

1. INTRODUCTION

Designing the architecture for a product or a family of products means to create the software structures that enable the system to achieve its quality goals. One very common goal is to prepare the software for change. This is important to minimize efforts required for maintenance and especially when an architecture for a family of products is designed.

An architect preparing a software architecture for change, will try to predict likely changes and will design components in the architecture that encapsulate the change in a single component. For the other components of the architecture rules like strong cohesion and loose coupling are used to increase the probability that unpredicted changes only affect a single component. At the end, there will be a documentation of the architecture, probably as a "boxes and lines" diagram, where the implicit assumption is that the boxes are the points of variations. In reality, an architecture also must support designs in which a single variability is spread across multiple components. The variability may also involve the relations among multiple components. Usually, relations do not carry a notation for variation and this adds additional, normally undocumented dependencies. I.e., the dependencies between components and their relations that need adaptation to support a specific type of change.

The architectural documentation does not necessarily reflect the thought and efforts the architect put into the design in order to achieve support for variability. At least for the predicted changes, the architect might have thought about possible alternatives and might have had a concept for what to do if the change actually occurs. As long as this architect is involved in the analysis of how to support a required variation, the result will likely be of high quality. Sometimes however, especially in a product family [2] context, the adapters of the architecture are not the creators. To enable the adapters to achieve high quality requires an explicit documentation of planned variability in the architecture is required.

During the lifetime of an architecture it is also very likely that changes occur that were not planned. In this paper we will not discuss those kind of variability because unplanned changes cannot have an explicit architectural solution (beside documenting rationale and providing traceability links).

In this paper we will first describe the different types of variability that are visible in an architecture description. Then we will show some possibilities of how to represent those types of variabilities. We will also describe at what times in the development cycle adaptations can be done and finally we show features that could be added to the architecture description to make it easier to find the places in the architecture that support adaptations.

The examples shown in this paper are based on a product line architecture designed by the Robert Bosch GmbH in cooperation with the Software Engineering Institute.

[1] This work was supported by Robert Bosch GmbH and the United States Department of Defense

* Felix Bachmann is employee of Robert Bosch GmbH

2. CAUSES OF VARIABILITY

Two of the cases where there is a requirement to represent different possibilities within an architecture are:

1. During design, a collection of alternatives may exist and capturing these alternatives may be necessary if a decision among them is deferred.
2. An architecture for a product line may encompass a collection of different alternatives for dealing with the variation among products. Capturing these alternatives and the rationale for each alternative enables the team constructing a product to have a list of potential solutions to choose from.

These two situations are really the same. That is, at the point in time when the representation is being generated, it is not known which alternative is going to be chosen for a product and so the alternatives have to be captured. The alternatives are best captured in the context of the remainder of the architecture and so the architecture representation, itself, should support variability.

We begin by discussing some sources of variation and then we discuss how support for these variations could be represented within an architecture representation.

Variation in function. A particular function may exist in some products and not in others. For example, consider a car radio/navigation system within an automobile. Some automobiles may have a radio and no navigation, others navigation without the radio and still others may have both. The characteristics of the radio will vary across different products as well. This situation may also arise within a single product if the requirements are not known as the design proceeds.

Variation in data. A particular data structure may vary from on product to another. For example, assume in a call center application two components exchange information about a customer. This information contains among other things the mailing address, which is realized as an unstructured text string. To support a feature in another version of the call center application (e.g. a structured display of the customers mailing address) the format of the mailing address has to be different. Variation in data in most cases is a consequence of variation in function.

Variation in control flow. A particular pattern of interaction may vary from on product to another. For example, assume there is a notification mechanism between components in place that informs interested components that some data values have been changed. One possible implementation is that all the components get notified in sequence within a single control flow. In a particular product some of the components to be notified may actually be 3rd party components, which may have some unknown behavior. For reliability reason it might be a good idea to direct the control to a component that is able to do an error recovery in case a notified component does not return the control.

Variation in technology. The platform (OS, hardware, dependence on middleware, user interface, run-time system for programming language) may vary in exactly the same fashion as the function. A particular piece of middleware may be required in one product and not in another. The OS or the hardware may vary from product to product. For example, a sensor may be connected directly to the controller whose software is being designed or it may be connected over a communication line. If the sensor is connected directly to the controller, then sensor management software is needed, if it is connected over a communication line then communication line management software is needed.

Variation in quality goals. The particular quality goals important for a product may vary. For example, the coupling between a producer and consumer of a data item may be achieved via a publish subscribe mechanism or via a permanent connection. The choice of one or another of these two options embodies a choice of the importance of performance and modifiability and this choice may be different in different products.

Variation in environment. The fashion in which a product interacts with its environment may vary. For example, a particular piece of middleware may be invoked from either C++ or Java. The invocation mechanism may vary from one product to another.

Since the focus of this paper is on how to describe variability on architecture level, we will not describe how to deal with variability that is not visible in the architecture. For example, if the architecture describes an application that uses an operating system, but the operating system is not described because it is not of interest, then variability in the operating system cannot be described. Or, if a specific variability was implemented by creating a component that realizes all possible variations, then no differences in the architecture would appear between different products/versions. Thus, this variability is not architecturally relevant.

3. TYPES OF VARIATION

No matter what causes the variation, any variation has a type. That is:

- A variation can be optional
- A variation can be an instance out of several alternatives
- A variation can be a set of instances out of several alternatives.

A variation is *optional* if, for example, a specific functionality is contained in one product but not in another. Of specific interest here are relations that other functions have to the optional functionality and what happens to those relations in case the optional functionality is not included in the product. In section 7 we will discuss possible mechanisms that deal with optionality.

A variation can be an *alternative*. This means that the architecture provides a placeholder in which one of several alternatives can be inserted. For example the architecture may provide a place for “cruise control” functionality. In high-end cars this might be realized by an “adaptive cruise control” (adapts car speed to the car driving in front), while low-end cars use the normal cruise control (constant speed). The latter can be realized with less memory and less computing power, thus at lower cost.

A variation can be a *set of alternatives*. This means that the architecture provides a placeholder in which multiple instances of different alternatives can be inserted. For example a software system may have the ability to communicate with the outside world by using different communication protocols in parallel. Thus, one product could be built using one set of communication protocols while another product has a different set. In this case also the relations that other functions have to the set of

alternatives are of interest since a binding to the instances of the collection needs to be done. Section 7 discusses possible mechanisms for this.

4. PUTTING VARIATION IN CONTEXT

Our concern is how to represent variation of the three types we have identified. The first question is what, more precisely, do we mean by variation. Let us examine variation in function in more detail to begin to answer that question. Figure 1 shows a module view of two systems. There are modules and they are related by "depends on". The first product shown (Product 1) includes module B, which may have the "radio" functionality. In another product (Product 2) the radio is substituted by a high-end radio shown as module D.

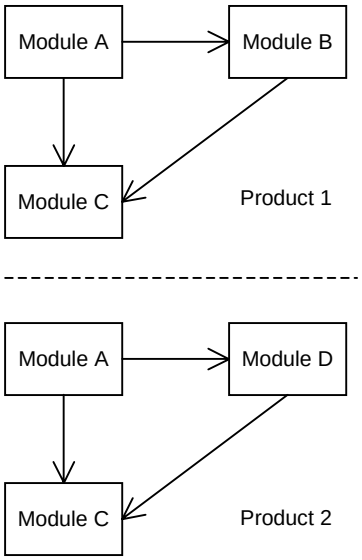


Figure 1 Alternative architectures for two products

In this example, the differences between the two products are located in one module (Module B is replaced by Module D). To express an architecture that supports this variation the two diagrams shown in Figure 1 need to be collapsed into one diagram.

Figure 2 shows the two architectures in a single picture. The box "Variant A" in this picture now has different semantics. It describes a variation point and not a specific functionality, as a module would do. Therefore a different notation (here a shaded box) should be used to clearly express this fact.

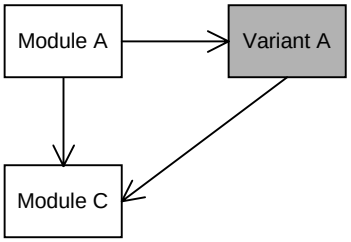


Figure 2 Variation in architecture

The problem now becomes how to keep track of possible implementations of Variant A. This leads to Figure 3.

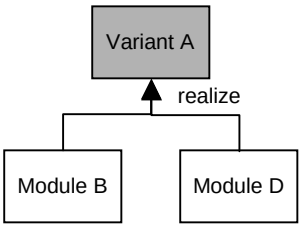


Figure 3 Alternatives for Variant A

Variations also can occur for relations between components of an architecture. Figure 4 shows again a module view of two possible products. In this example the variation occurs between Module B and Module C. In product 1 module B, a low cost radio, issues a command "Display Frequency" to module C, a user interface. In product 2, the high-end version, the name of a radio station is available and has to be displayed.

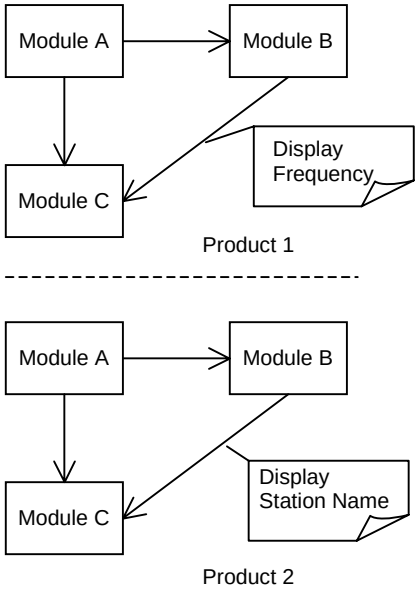


Figure 4 Variation of relation

A closer look on variations of relations reveals that this normally includes variation of the connected components. If the relation is now an information exchange of an the station name, then at least the producer of that information has to be different. The information has to be extracted from the signal the station sends. Most likely, also the consumer of the information varies. Therefore a variation in a relation can be described as shown in Figure 5. In addition, Figure 6 shows the dependencies between the alternatives for variant A and B. That is, both Module B and Module C together realize the feature to display the frequency whereas Module D and Module E together realize the display of the station name.

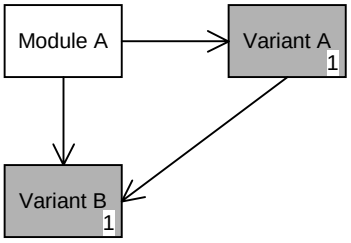


Figure 5 Variation of relation as variation of components

Note that the relation between module B and C as well as the relation between module D and E are realizations of the relation between variant A and B.

A variation in function may affect multiple modules in various portions of the architecture, similar to the case shown in Figure 6.

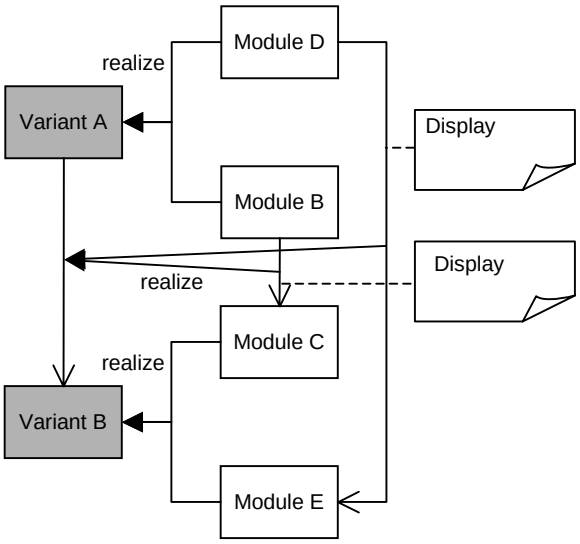


Figure 6 Constraints between alternatives of variants

This same picture suffices for variations caused by platform, control flow, or quality as well. It also would suffice for multiple variations rather than just the two we have hypothesized. Thus, in general, we can situate variants in the architecture by identifying those modules that are in the architecture for all of the variants and those modules that will vary based on some cause of variation. Furthermore, if we can describe a single variant within its context then we can describe all of the variants. This limits our scope of concern when discussing variation to representing a single variation.

We now examine the relation between a variant and decomposition. In the radio example we used so far it is obvious the two different types of radios (low-end and high-end) still have a lot in common. It is not very realistic to assume that module B and D, which are alternative implementations of the variant A, are totally different and independent from each other. When decomposing both modules we can be more explicit where the variation really occurs. Some parts of the decomposition are common for all alternatives and some differ. This suggests that there is a decomposition of a variant such as it is shown in 0. Variant A actually consists of the modules A1 and A2, which are common for all alternatives of variant A, and it has a variation point A1 with two alternative solutions, module A3 and A4.

This opens two possibilities to describe and implement alternatives of a variation. The alternatives can be described on the more abstract level as shown in Figure 2 and Figure 3 with the composition rule that module B is composed using module A1, A2, and A3 whereas module D is composed using module A1, A2, and A4. The second possibility is to describe the variation of the detailed, the decomposed level as shown in 0. What to use strongly depends on the mechanisms chosen to realize the different products, as well as the decision of the organization on what level of granularity the product definition should be handled.

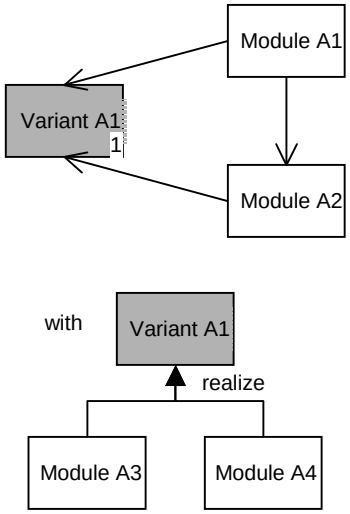


Figure 7 Decomposition of Variant A

We now introduce a basic notation for the different types of variants for the purpose of better understanding of examples in this document. As shown in 0 we distinguish between:

- Optional variant (Variant A), which means that there exists exactly one implementation that could be included in a product.
- Alternative variant (Variant B), which means that there exist multiple realizations of this variant and exactly one must be included in the product.
- Set of alternative variants (Variant C), which means that there exist multiple realizations of this variant and at least one must be included in the product.
- Optional alternative (Variant D), which means that there exist multiple realizations of this variant and one of it could be included in the product.
- Optional set of alternatives (Variant E), which means that there exist multiple realizations of this variant and a collection of it could be included in the product.

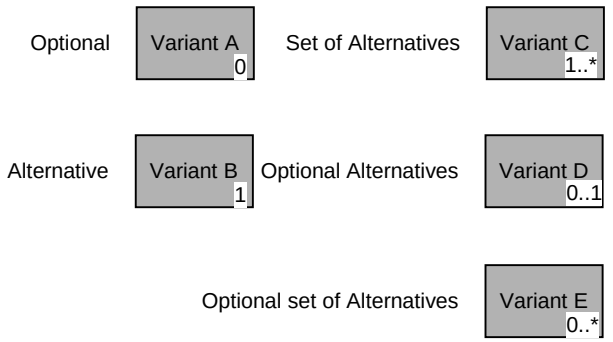


Figure 8 Notation for types of variants

Using this notation, Figure 5 is redrawn into Figure 9. Variant A and B are alternative variants each with two possible implementations as shown in Figure 6, which describes that variant A and B have alternative solutions.

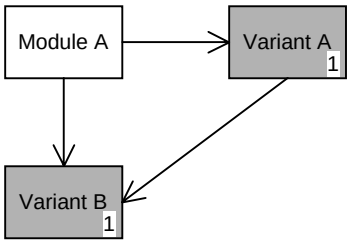


Figure 9 Basic problem statement

5. RELATIONS FROM AND TO VARIANTS

The relations between variants and/or modules of the architecture are of special interest if the architecture is designed to support change. Changing the variants almost always influences the relations among those variants and modules.

Relations that point from a variant to a module are easy to handle. A specific implementation of a variant may or may not use that relation. As long as the relation is used by a module as intended no specific action is required.

More difficulties arise with relations that point to a variant, like the relation from Module A to Variant A or from Variant A to Variant B in Figure 5. The simplest case is when an alternative variant was designed. The only condition is that all implementations of that variant have to comply with the interface required by the relation that points to the variant.

Optionality requires specific actions by modules pointing to an optional variant. Since optional means that a variant may not have an implementation, all relations to this variant must also be optional. Having two different implementations of the module that uses an optional variant could do this. However, this would automatically change a module into a variant. This is not really what is desired. Variations should be kept as local as possible and such kind of ripple effect through the architecture should be avoided. A possible solution is to do the adaptation during development time by using techniques like generators or compiler switches (see section 7 for more information about how to implement variations). Those techniques hide the adaptation on the architecture level.

Another difficulty to be dealt with occurs when a variant is designed as a set of alternatives. Modules that use this set have to be able to deal with a varying set of implementations. This normally requires some sort of distribution mechanism. This, in turn, means that requests to the set of alternatives are not sent directly to them but are routed through a distributor of some kind. A “factory pattern” would be an example of such a distributor.

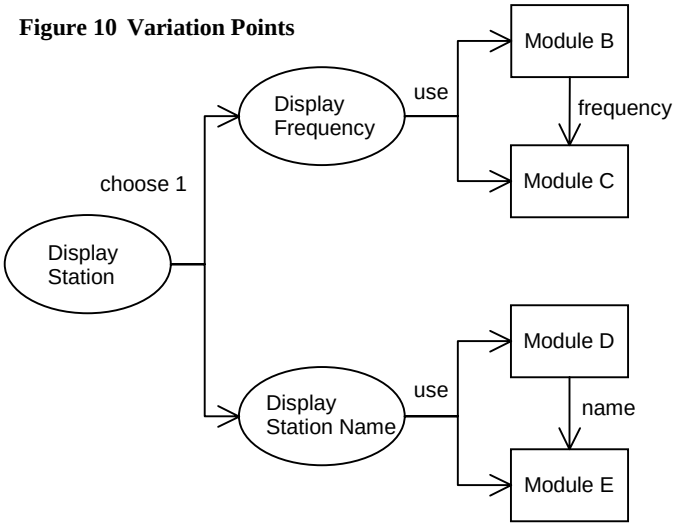
The distributor clearly has to be an indirection in the relation to the set of alternatives. Generally speaking there would be three possible places to locate the distributor. The distributor can be a) part of the module that requests the service, b) it can be a separate module between the module and the set or c) it can be a “wrapper” around the set. Alternative a) would very likely change a common module into a variant, alternative b) would require a change in the architecture, and alternative c) would add the adaptation to the different sets into the variant itself. Thus, for the management of variants point of view, alternative c) is the preferred one since it limits the number of variants.

6. CONNECTION TO REQUIREMENTS

Representing the variation in the architecture solves one part of the problem. The other problem is that those variations have to be found in the documentation. Even the architecture of a midsize system can contain many modules with a fairly high number of them designed to be variants. Several of those variants may also have dependencies among each other to fulfill a single customer requirement. For example, allowing a customer to make a single choice of having either a radio that displays the frequency or the station name as shown in Figure 6 actually created two variants (Variant A and B) in the architecture and these two variants depend on each other.

To fully utilize the designed variations a mechanism for finding them is needed. A possibility is to introduce “variation points” [3]. Those variation points build the connection between the customers’ requirements/features where variation can occur (if planned for) and the places in the architecture that are designed to support those variations. Variation points also contain the information needed to build the required architecture for the specific product, such as decisions to be made, implementation techniques to be used, rationale why the variation was build that way, etc. An example of how to represent a variation point in the architecture is shown in Figure 10. This example shows the alternatives from Figure 6 in combination with a possible choice for a customer who can choose between a radio that only displays the frequency or one that displays the station name. It shows that a customer can choose the display type of a station. The example also shows that the choice is designed to be an alternative. Exactly one of the alternatives has to be chosen.

Figure 10 Variation Points



This notation should not be confused with the description of alternatives of modules as described in 0. Variation points describe variations in requirements and state the customer choices for a specific product, while the description in 0 states possible implementations for a variant. Choices in the requirements may or may not lead to choices in the architecture. That actually depends on the way the architecture is designed. For instance, the architecture could have exactly one component that is able to deal with both types of addresses. A configuration file may define which type actually was chosen.

Variation points can be used as input for generators in case that creating the product specific architecture is actually done by the generator technique. For more information see section 7. In this case the syntax of a variation point is determined by the generator to be used.

The example shown in Figure 10 suggests that the documentation of variation points is part of the architecture documentation. This is not the only possibility for documenting variation points. Variation points build the link between the requirement documents and the architecture description. Therefore variation points could be part of the requirement documents, the architecture documents, or could even be a separate document. What is important is that there is a means for the users of the documentation (hopefully with tool support) to easily find the places of variations in the architecture.

7. IMPLEMENTATIONS OF VARIATIONS

After the places where variation can occur are designed in the architecture the decision has to be made which implementation technique to use to implement the variation. Two basic implementation techniques are possible, which are *module replacement* and *data controlled variation*.

Module replacement

Module replacement is the technique of having multiple code-based versions of a particular module and choosing the correct one. The interfaces to all versions of the module are compatible and, thus, the modules that depend on the variable module do not need to be modified based on the choice of variants. This choice can be done at execution time or at an earlier time. Within the module, there can be a portion that is common to all variants (possibly empty) such as adaptation to the communication mechanism used, and a portion that is variant dependent.

Module replacement supports very easily the different cases of alternative variants, as shown in 0. To deal with optionality is a little bit more difficult. A possible solution would be to implement a module that delivers fixed responses that describe the case as if the module wouldn't be there. This would change an optional variant into an alternative variant were one alternative is an almost empty module that acts as a placeholder to satisfy references to the variant.

Data controlled variation

Data controlled variation is the technique of maintaining the variation information in a data structure and having a single module that understands how to navigate the data structure to determine the correct actions. A simple example of data controlled variation is to have a parameter to the module that determines the variant to execute. A more complicated example is to have the variation incorporated in a data structure that might be generated from a parser and have the module be an interpreter for the language underlying the parse tree. Intermediate examples are to incorporate collections of parameters or control information into a data structure. In any case, the module, itself, remains unchanged from variant to variant but the data on which it operates will control the variant.

Data controlled variation can actually lead to a design where a variant is described in the architecture, which has exactly one implementation, which would be the variant itself. If that mechanism were used, the example shown in Figure 9 and Figure 10 would change as shown in 0.

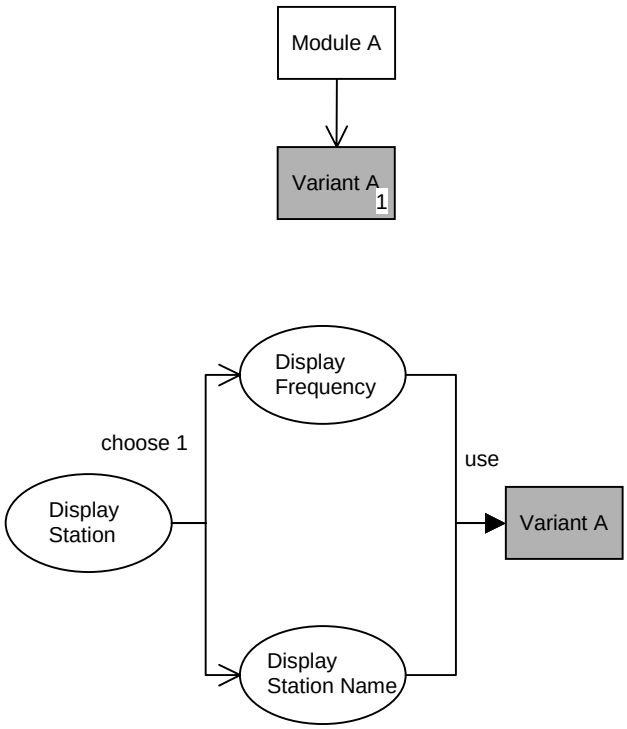


Figure 11 Data controled variation

Realization mechanisms

There are several mechanisms [1] available that can be used to build the customer specific product. Among those are:

- *Generators* can be used if the goal is to generate a product specific architecture that contains only what is needed for the product. Although this technique leads to very understandable designs, it requires the implementation of the generator, which is a cost factor that has to be considered. A generator can nicely deal with alternatives and optional modules as well as the relations among them.
- Using a *Configuration Management System* to build a customer specific product would enable the assembly of a product that only contains needed modules. The configuration management system takes the designed variants and the variation points as input and provides the set of configuration items that form the product. This mechanism can easily deal with alternative implementations. It is difficult to deal with optional components on the level of the configuration management. To build a configuration management that supports the described variation still is costly, but probably not as costly as it is to build a generator.
- When using *Compilation* as a mechanism to manage variation, the variations are implemented on code level. This implementation uses "complier switches" so that a specific set of parameters, which are input for the compilation produces the product specific object code. The implementation of the "gnu compiler", which needs to be adapted to a variety of platforms is an example for the use of this mechanism.
- A very common form to realize variations is the *adaptation during start-up*. The software normally is implemented so that it can handle all possible variants. By reading a configuration file of some sort the software learns how to react. The easiest way to support this mechanism is to implement the modules as data controlled. To use module replacement requires a dynamic load mechanism either provided by the implementation language or by the infrastructure (operating system) the software is running.
- *Adaptation during normal execution* could also be done. "Plug-ins" for a program, such as an Internet browser, are examples for adding and/or removing modules while the program is running. Since the decision which module to use to support a variant is deferred until a user actually uses a program, this mechanism offers the most flexibility for users but also requires an implementation that ensures consistency which normally would be provided by a configuration management system or a compiler.

8. CONCLUSIONS

To utilize the potential for variation that has been placed into a software architecture requires an explicit link between the requirement analysis/management discipline and the architecture development. An explicit design of variants in the architecture and the definition of variation points that support finding those variants in the architecture enables developers who are not necessarily the creators of the architecture to fully employ designed variations. Placing advice on how to implement variants within a variation point description also helps to clarify the instantiation of a product.

To make this happen in an industrial environment however needs tool support. The tools available today focus more on either the requirements side or the design side. They offer some basic support to handle variations. Mostly additional semantics to cope with variations have to be added by the users of the tools. Requirement and design tools also support some rudimentary connection among them to at least exchange some basic information. This is by far not sufficient to support the described variation points. Developers who want to use the described techniques will end up implementing their own "variation point tool".

9. REFERENCES

[1] Jan Bosch, Design & Use of Software Architectures, Addison Wesley, 2000.

[2] Paul Clements and Linda Northrop, A Framework for Software Product Line Practice – Version 3.0. http://www.sei.cmu.edu/plp/framework.html.

[3] Michael Coriat, Jean Jourdan, Fabien Boisbourdin, The SPLIT Method, Software Product Lines, Kluwer Academic Publishers, August 2000 147-166.