A Taxonomy of Variability Realization Techniques

MIKAEL SVAHNBERG
Blekinge Institute of Technology

JILLES VAN GURP AND JAN BOSCH
University of Groningen

Development of software product lines relies heavily on the use of variability to manage the differences between products by delaying design decisions to later stages of the development and usage of the constructed software systems. Implementation of variability is not a trivial task, and is governed by a number of factors to consider. In this paper we describe the factors that are relevant to determine how to implement variability, and present a taxonomy of variability realization techniques.

Categories and Subject Descriptors: D.2.2 **[Software Engineering]** Design Tools and Techniques - *Modules and Interfaces*; D.2.2 **[Software Engineering]** Design Tools and Techniques - *Object-oriented design methods*; D.2.11 **[Software Engineering]** Software Architectures - *Domain-specific architectures*; D.2.11 **[Software Engineering]** Software Architectures - *Patterns*; D.2.13 **[Software engineering]** Reusable Software - *Domain engineering*
General Terms: Design
Additional Key words and phrases: Variability, Software Product Lines

1. INTRODUCTION

Over the last decades, the software systems that we use and build require and exhibit increasing variability, i.e. the ability of a software artefact to vary its behaviour at some point in its lifecycle. We can identify two underlying forces that drive this development. First, we see that variability in systems is moved from mechanics and hardware to the software. Second, because of the cost of reversing design decisions once these are taken, software engineers typically try to delay such decisions to the latest phase in the lifecycle that is economically defendable. One example of the first trend are car engine controllers. Most car manufacturers now offer engines with different characteristics for a particular car model. A new development is that frequently these engines are the same from a mechanical perspective and differ only in the software of the car engine controller. Thus, earlier the variation between different engine models first was incorporated through the mechanics and hardware. However, due to economies of scale that exist for these artefacts, car developers have moved the variation to the software.

The second trend, i.e. delayed design decisions, can be illustrated through software product lines [Weiss & Lai 1999][Jazayeri et al. 2000][Clements & Northrop 2002] and the increasing configurability of software products. Over the last decade, many organizations have identified a conflict in their software development. On the one hand, the amount of software necessary for individual products is constantly increasing. On the other hand, there is a constant pressure to increase the number of software products put out on the market in order to better service the various market segments. For many organizations, the only feasible way forward has been to exploit the commonality between different products and to implement the differences between the products as variability in the software artefacts. The product line architecture and shared product line components must be designed in such a way that the different products can be supported, whether the products require replaced components, extensions to the architecture, or particular configurations of the software components.

Based on our case studies [Bosch 2000][Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999b], we have found that it is not a trivial task to introduce variability into a software product line. Many factors influence the choices of how design decisions can be delayed. Influencing factors include the size of the software entity, how long the design decision can be delayed and the intended runtime environment. Another thing to consider is that variability need not be represented only in the architecture or the source code of a system, it can also be represented as procedures during the development process, making use of various tools outside of the actual system being built.

Although the use of variability techniques is increasing, research, both by others (for example, [Jacobson et al. 1997][Jazayeri et al. 2000][Griss 2000][Clements & Northrop 2002]), and by ourselves [van Gurp et al. 2001][Bosch et al. 2002][Jaring & Bosch 2002], shows that several problems exist. A major source for these problems is that software architects typically lack a good overview of the variability techniques available as well as the pros and cons of these techniques.

This paper discuss the factors that need to be considered for selecting an appropriate method or technique for implementing variability. We also provide a taxonomy of techniques that can be used to implement variability. The contribution of this is, we believe, that the notion of variability, and its qualities, is better understood, and that more informed decisions concerning vari-

Authors' addresses: M. Svahnberg, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, S-372 5 Ronneby, Sweden, e-mail: Mikael.Svahnberg@bth.se; J. van Gurp, Department of Mathematics and Computer Science, University of Groningen, PO Box 800, 9700 AV, the Netherlands, e-mail: Jilles@cs.rug.nl; J. Bosch, Department of Mathematics and Computer Science, University of Groningen, PO Box 800, 9700 AV, the Netherlands, e-mail: Jan.Bosch@cs.rug.nl

ability and variation points can be made during software development. Using the provided toolbox of available realization techniques the development process is facilitated as the consequences of a particular choice can be seen at an early stage, much as the use of Design Patterns [Gamma et al. 1995] also present developers with consequences of a particular design.

It should be noted that this paper focus on implementing variability in architecture and implementation artefacts, such as the software architecture, the components and classes of a software system. We do not address issues related to e.g. variability of requirements, managing variations of design documents or test specifications, structure of the development organization, etc. While these are important subjects, and need to be addressed to properly manage variability in a software product line, the goal of this paper is to cover the area of how to technically achieving variability in the software system. This paper should thus be seen as one piece in the large puzzle that is software product line variability. For a description of many of the other key areas to consider, please see e.g. [Clements & Northrop 2002].

The remainder of this paper is organized as follows: In Section 2 we introduce the terminology that we use in this paper. In Section 3 we describe the steps necessary to introduce variability into a software product line, and in Section 4 we go through one of these steps in further detail, namely the step where the variability is characterized so that an informed decision on how to implement it can be taken. In Section 5 we, based on the characterization done, present a taxonomy of variability realization techniques. This is intended as a toolbox for software developers to find the most appropriate way to implement a required variability in the software product. In Section 6 we briefly present a number of case studies, and how the companies in these case studies usually implement variability. Related work is presented in Section 7, and the paper is concluded in Section 8.

2. TERMINOLOGY

When reading about software product lines, features and variability, there seems to still be some amount of confusion regarding how different terms should be interpreted. To avoid confusion we present, in this section, a list of terms and phrases that we use in this paper. This is provided to allow the reader to relate the terms to whatever terminology is preferred, and is not meant to be a standard dictionary of software product line and variability terminology.

Variability. By this we denote the whole area of how to manage the parts of a software development process and its resulting artefacts that is made to differ between products or in certain situations within a single product. Variability is concerned with many topics, ranging from the development process itself to the various artefacts created, such as requirements, requirements specifications, design documents, source code, and executable binaries (to mention a few). In this paper, however, we focus on the software artefacts, involving software architecture design, detailed design, components, classes, source code, and executable binaries.

Feature. The Webster dictionary provides us with the following definition of a feature: *"3 a: a prominent part or characteristic b: any of the properties (as voice or gender) that are characteristic of a grammatical element (as a phoneme or morpheme); especially; one that is distinctive"*. In [Bosch 2000], features are defined as follows: *"a logical unit of behavior that is specified by a set of functional and quality requirements"*. The point of view taken in the book is that a feature is a construct used to group related requirements ("*there should at least be an order of magnitude difference between the number of features and the number of requirements for a product line member*").

In other words, features are an abstraction from requirements. In our view, constructing a feature set is the first step of interpreting and ordering the requirements. In the process of constructing a feature set, the first design decisions about the future system are already taken. In [Gibson 1997], features are identified as units of incrementation as systems evolve. It is important to realize that there is a n to m relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features and that a particular feature may meet more then one requirement (e.g. a functional requirement and a couple of quality requirements).

A software product line provides a central architecture that can be evolved and specialized into concrete products. The differences between those products can be discussed in terms of features (e.g. modelled as prescribed by FODA [Kang et al. 1990][Kang 1998]). Consequently, a software product line must support variability for those features that tend to differ from product to product.

[Griss et al. 1998] suggest the following categorization of features:

- **Mandatory Features.** These are the features that identify a product. E.g. the ability type in a message and send it to the smtp server is essential for an email client application.
- **Optional Features.** These are features that, when enabled, add some value to the core features of a product. A good example of an optional feature for an email client is the ability to add a signature to each message. It is in no way an essential feature and not all users will use it but it is nice to have it in the product.
- **Variant Features.** A variant feature is an abstraction for a set of related features (optional or mandatory). An example of a variant feature for the email client might be the editor used for typing in messages. Some email clients offer the feature of having a user configurable editor.

In [van Gurp et al. 2001] we add a fourth category:

- **External Features.** These are features offered by the target platform of the system. While not directly part of the system, they are important because the system uses them and depends on them. E.g. in an email client, the ability to make TCP connections to another computer is essential but not part of the client. Instead the functionality for TCP connections is typically part of the OS on which the client runs. Differences in external features may motivate inclusion of parts in the software to manage such variability.

Our choice of introducing external features is further motivated by [Zave & Jackson 1997]. In this work it is argued that requirements should not reflect on implementation details (such as platform specific features). Since features are abstractions

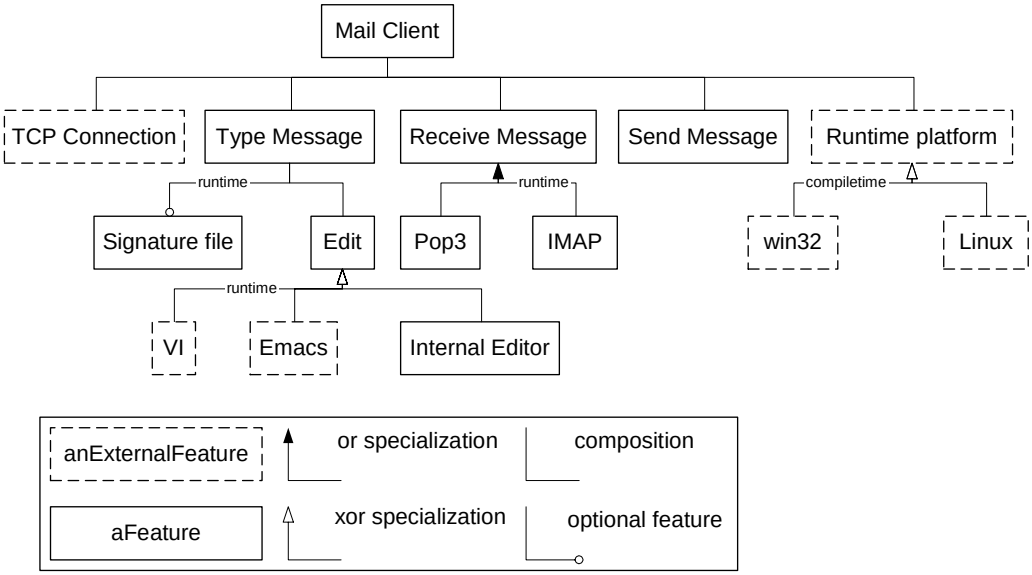

Figure 1. Example feature graph

from requirements, we need external features to link requirements to features. Using this categorization we have, in [van Gurp et al. 2001] adapted the notation suggested by [Griss et al. 1998] to support external features. In addition we have integrated the notion of binding time which we discuss in detail in Section 4. An example of our enhanced notation can be found in Figure 1. In this feature graph, the features of a email client are laid out. The notation uses various constructs to indicate optional features; variant features in that exclude each other (xor) and variant features that may be used both (or).

The example in Figure 1 demonstrates how these different constructs can be used to indicate where variability is needed. The receive message feature, for instance, is a mandatory variant feature that has pop3 and imap as its variants. The choice as to which is used is delayed until runtime, meaning that users of the email client can configure to use either variant. Making this sort of details clear early on helps identify the spots in the system where variability is needed early on. The Receive message feature might be implemented using an abstract receive message class that has two subclasses, one for each variant.

Our decomposition might give readers the impression that a conversion to a component design is straightforward. Unfortunately, due to a phenomena called feature interaction, this is not true. Feature interaction is a well-known problem in specifying systems. It is virtually impossible to give a complete specification of a system using features because the features cannot be considered independently. Adding or removing a feature to a system has an impact on other features. In [Gibson 1997], feature interaction is defined as a characteristic of *"a system whose complete behavior does not satisfy the separate specifications of all its features"*.

In [Griss 2000], the feature interaction problem is characterized as follows: *"The problem is that individual features do not typically trace directly to an individual component or cluster of components - this means, as a product is defined by selecting a group of features, a carefully coordinated and complicated mixture of parts of different components are involved."*. This applies in particular to so-called crosscutting features (i.e. features that are applicable to classes and components throughout the entire system). A further discussion on crosscutting features can be found in [Kiczalez et al.1997].

Variant. We use this as a short form to represent a particular variant of a variant feature. For example, in the e-mail example above one variant of the edit feature would be the internal editor. A single variant can consist of several software entities, collaborating to solve the functionality required for the variant feature.

Collection of Variants. A collection of variants is the whole set of variants available for one variant feature. Note that we only use the term collection of variants to refer to this set of available variant. Each of these variants, and in particular the software entities it is constituted of, is then connected to the remainder of the system using a set of variation points.

Variation Point. We use this term, in this paper, to denote a particular place in a software system where choices are made as to which variant to use. This term is further elaborated on in Section 4, but the gist of it is that a variant feature translates to a collection of variants and a number of variation points in the software system, and these variation points are used to tie in a particular variant to the rest of the system. In a larger perspective, a variation point can also involve other artefacts related to the software product line, but in this paper, we focus on the software artefacts.

Variability Realization Technique. By this we refer to a way in which one can implement a variation point. In Section 5 we present a taxonomy of variability realization techniques, i.e. a taxonomy of different ways to implement variation points.

Software Entity. A software entity is simply a piece of software. The size of a software entity depends on the type of software entity. Types of software entities are components, frameworks, framework implementations, classes or lines of code. An example of a software entity is the Emacs editor in the example in Figure 1, which is a component in a mail client. In this example, the component represent an entire variant of the variant feature "type message", whereas in other examples a variant of a variant feature is implemented by several software entities, possibly of different types. For example, if the choices for typing a message had been "plain text" and "HTML-formatted text", there might be a need for a software entity in the implementation of "send message" that re-formats HTML-formatted messages to plain text and attaches both to the e-mail before sending it.

Component. We use the same definition of a component as [Szyperski 1997] (page 34) does, namely: *"a software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Framework. In our experience, many industries do not use the kind of components as defined by [Szyperski 1997]. Rather, they use object-oriented frameworks in the style of e.g. [van Gurp & Bosch 2002], [Mattsson 2000] and [Roberts & Johnson 1996]. Such a framework consists of an abstract *framework interface*, i.e. a set of abstract classes that define the interface of the framework, and a number of concrete *framework implementations*. Each of these framework implementations, which use the same framework interface, can range in size from a few thousand lines of code up to 100 000 KLOC. Frameworks like this typically model an entire sub-domain, and the implementations represent variants of this sub-domain. An example of this is a file system framework, which has an abstract interface containing classes representing e.g. files and directories, and a number of concrete implementations of file systems for e.g. Unix, Windows, Netware, etc.

3. INTRODUCING VARIABILITY IN SOFTWARE PRODUCT LINES

While introducing variability into a software product line, there are a number of steps to take along the way, in order to get the wanted variability in place, and to take care of it once it is in place. In this section we briefly present the steps that we perceive as minimally necessary to take. These steps are also presented in [van Gurp et al. 2001].

The steps we perceive as minimally necessary are:

- Identification of variability
- Constraining the variability
- Implementation of the variability
- Managing the variability

Below, we present these four steps further.

Identification of variability. The first step is to identify where variability is needed. The feature graph notation we suggest in Section 2 might be of use for doing so, and if feature graphs are undesirable, variable features can be identified from the requirements specification. The identification of variability is a rather large field of research (see for example [Clements & Northrop 2002]), but it is unfortunately outside of the scope of this paper to investigate it further. However, there seems to be some consensus that there is a link between features and variability, in that variability can more easily be identified if the system is modelled using the concept of features (see e.g. [Becker et al. 2002][Capilla & Dueñas 2002][Krueger 2002][Salicki & Farcet 2002], as well as FODA [Kang et al. 1990] and FORM [Kang 1998]).

Constraining variability. Once a variant feature has been identified, it needs to be constrained. After all, the purpose is not to provide limitless flexibility but to provide just enough flexibility to suit the current and future needs of the system in a cost effective way. For constraining a variant feature, the following activities need to take place:

- Decide when the variant feature should be introduced into the design and implementation of the software product line and/or into the software product.
- Decide when and how variants are to be added to the system.
- Choose a binding time for each variation point, i.e. when the variation point should be committed to a particular variant of a variant feature.

After the variant features are identified, they are eventually designed as software entities, i.e. introduced into the software product line. One variant feature may result in a number of software entities of varying sizes. Moreover, places in the software system are identified where the software entities for a variant feature are tied in to the rest of the system. These places we refer to as variation points. Depending on how the variation points are implemented, they allow for adding variants and for binding during different times.

In Section 4 we describe the process of constraining variability in further detail.

Implementing variability. Based on the previous constrainment of variability a suitable realization technique can be selected for the variation points pertaining to a certain variant feature. The selected realization technique should strike the best possible balance between the constraints that have been identified in the previous step. To facilitate the selection of variability realization techniques, we provide, in Section 5, an overview of such techniques.

Managing the variability. The last step is, as with all software, to manage the variability. This involves maintenance (adaptive and corrective as well as perfective [Swanson 1976][Pigoski 1997]), and to continue to populate variant features with new variants and pruning old, no longer used, variants. Moreover, variant features may be removed altogether, as the requirements change, new products are added and old products are removed from the product line. Management also involves the distribution of new variants to the already installed customer base, and billing models regarding how to make money off new variants. As with the identification of variability, this is also outside the scope of this paper.

4. CONSTRAINING VARIABILITY

Having identified what type of variability is required, and where in the software product line it occurs, the next step is to constrain the variant features. By this we mean that the characteristics of each variant feature is determined so that a way to implement the variant feature, i.e. realize the variant feature in the software product line, can be chosen.

The aspects to consider when selecting how to implement a variant feature can be identified by considering the lifecycle of the variant feature. During the lifecycle, the variant feature is transformed in several ways during different phases, until there

Table 1: Entities most likely in focus during the different development activities

Development Activities	Software Entity in Focus
Architecture Design	Components Frameworks
Detailed Design	Framework Implementations Sets of Classes
Implementation	Individual Classes Lines of Code
Compilation	Lines of Code
Linking	Components

is a representation in software of it. Below, we briefly discuss these phases, after which we present the phases within the scope of this paper in further detail.

When a variant feature is first identified, it is said to be *implicit*, as it is not yet realized in the software product line. An implicit variant feature exists only as a concept, and is not yet implemented. Software designers and developers are aware that they eventually will need to consider the variant feature, but defer its implementation until a later stage.

A variant feature ceases to be implicit when it is *introduced* into the software product line. After a variant feature is introduced it has a representation in the design and implementation of the software product line. This representation takes the form of a set of variation points, i.e. places in the design or implementation that together provide the mechanisms necessary to make a feature variable. Note that the variants of the variant feature need not be present at this time.

After the introduction of a variant feature, the next step is to *add the variants* of the feature in question. What this means is that software entities are implemented for each of the variants available for the variant feature in such a way that they fit together with the variation points that were previously introduced. Depending on how a variation point is implemented, it is *open* for adding variants during different stages of the development, and *closed* at other times, which means that new variants can only be added at certain stages of development.

Finally, at some stage, a decision must be taken which variant of a variant feature to use, and at this stage the software product line or software system is *bound* to one of the variants for a particular variant feature. This means that the variation points related to the variant feature are committed to the software entities representing the variant decided upon.

To summarize, a variant feature goes through the following phases during its lifecycle:

- It is *identified* as a variant feature.
- It is *implicit*, not yet represented in the software product line.
- It is *introduced* into the software product line, as a set of variation points.
- Variants are *added* to the system.
- The system is *bound* to a particular variant.

As stated earlier, the process of identifying variant features is outside the scope of this paper, as is the consideration of implicit features. This paper is concerned with the characteristics that must be considered in order to select a suitable realization technique of variant features, and these characteristics are the introduction time, the process of adding new variants, and the binding time. These we discuss in further detail below.

4.1 Introducing a Variant Feature

After identifying a variant feature, it should be implemented into the software product line or into the relevant software products. For this implementation, one has to consider the most suitable size of the software entities intended to represent the variant feature, and the variants for the variant feature.

The variants of a variant feature can be implemented in a multitude of ways, using a range of different software entities, such as components, sets of classes, single classes or lines of code. Because of this, variant features can be introduced in all phases of a system's lifecycle, from architectural design to detailed design, implementation, compilation and linking. Each of these different phases has a focus on different software entities. Table 1 presents the different development phases and the software entities most likely in focus during these phases. In this table, we see that starting with architectural design down to compilation, the size of the software entities in focus becomes smaller, i.e. the granularity is increased. During the linking phase the size is again increased, as it is not relevant to discuss smaller entities than components when it comes to linking.

However, in many cases the situation is not as ideal as is described above, i.e. that a variant feature, and the variants for this variant feature, maps to a single type of software entity. It may well be the case that a single variant feature maps to a set of software entities, that together constitute the desired functionality. This set of software entities need not be of the same type, but can involve for example components as well as individual classes and even lines of code. Because of this, a single variant feature typically manifest itself as a set of variation points in the implemented system, working on different abstraction levels and with software entities of different sizes. It is desirable to select the means for implementing the variant feature such that they make the resulting set of variation points as small as possible, as this increase the understanding of the source code and hence facilitates maintenance.

The decision on when to introduce a variant feature is thus influenced by a number of things, relating both to the availability of realization techniques supporting desired qualities such as when to bind and when to allow adding of new variants, relat-

ing to the sizes of the involved software entities, relating to the number of resulting variation points, and also relating to the cost of maintaining the introduced variation points. A variation point that is introduced early needs to be understood and controlled during many subsequent development phases, whereas a variation point that is introduced late need only be controlled during a shorter time. On the other hand, if the variation point is also bound early, there is no, or little, extra overhead in understanding and controlling the variation point, even if it is introduced early. Furthermore, the overhead involved in keeping track of implicit variation points not yet implemented may also be substantial.

4.2 Adding of New Variants

Having introduced the variant feature into the software product line, this means that the software product line is instrumented with appropriate variation points that together can accommodate the variants of the variant feature. Then comes the task of adding these variants, which is also governed by a number of aspects, pertaining to when to add the variants, and how to add the variants. These aspects, further discussed below, need also be considered when deciding how to implement the variation points for a variant feature.

A variation point can be *open* or *closed* for adding new variants to the collection for that variation point. This means that at any given point in time either new variants can be added or old removed, i.e. the variation point is open, or it is no longer possible to add or remove variants, i.e. the system is dedicated to a certain set of variants which means that the variation point is closed.

The time when a variation point is open or closed for adding new variants is mainly decided by the development and runtime environments, and the type of software entity that is represented by the variation point. Typically, realization techniques open for adding variations during detailed design and implementation are closed at compile-time. Realization techniques working with components and component implementations are of a magnitude that makes them interesting to keep open during runtime as well, since they constitute large enough chunks of code to easily cope with.

An important factor to consider is when linking is performed. If linking can only be done in conjunction with compilation, then this closes all mechanisms at this phase. If the system supports dynamically linked libraries, mechanisms can remain open even during runtime.

Adding variants can be done in two ways, depending on how the variation point is implemented. In the first case, the variants are added *implicitly*, which means that there is no representation of the collection of variants in the software system. The collection of variants is managed outside of the system, using e.g. simple lists of what variants are available. Moreover, an implicit collection of variants relies on the knowledge of the developers or the users to provide a suitable variant when so prompted.

In the second case, the variants are added *explicitly*, which means that the collection of variants are manifested in the source code of the software system. This means that there is enough information in the system so that it can, by itself, find a suitable variant when so prompted.

The decision on when and how to add variants is governed by the business strategy and delivery model for the products in the software product line. For example, if the business strategy involves supporting late addition of variants by e.g. third party vendors, this constrains the selection of implementation techniques for the variation points as they may need to be open for adding new variants after compilation, or possibly even during runtime. This example also impacts whether or not the collection of variants should be managed explicitly or implicitly, which is determined based on how the third party vendors are supposed to add their variants to the system. Likewise, if the delivery model involves updates of functionality into a running system, this will also impact the choices of implementation techniques for the variation points.

Also the development process and the tools used by the development company influence how and when to add variants. For example, if the company has a domain engineering unit developing reusable assets, more decisions may be taken during the product architecture derivation, whereas another organization may defer many such decisions until compile or link-time.

4.3 Binding to a Variant

The main purpose of introducing a variant feature is to delay a decision, but at some time there must be a choice between the variants and a single variant will be selected and executed. We refer to this as *binding* the system to a particular variant. This can be done at several stages during the development and also as a system is being run. Decisions on binding to a particular variant can be expected during the following phases of a system's lifecycle:

- **Product Architecture Derivation.** The product line architecture typically contains many unbound variation points. The binding of these variation points is what generates a particular product architecture. Typically, configuration management tools are involved in this process, and most of the mechanisms are working with software entities introduced during architectural design.
- **Compilation.** The finalization of the source code is done during the compilation. This includes pruning the code according to compiler directives in the source code, but also extending the code to superimpose additional behavior (e.g. macros and aspects).
- **Linking.** When the link phase begins and when it ends is very much depending on what programming and runtime environment is used. In some cases, linking is performed irrevocably just after compilation, and in some cases it is done when the system is started. In other systems again, the running system can link and re-link at will. How long linking is available also determines how late new variants can be added to the system.
- **Runtime.** This is the variability that renders an application interactive. Typically this type of binding decisions are dealt with using any standard object-oriented language. The collection of variants can be closed at runtime, i.e. it is not possible to add new variants, but it can also be open, in which case it is possible to extend the system with new variants at runtime.

Table 2: Summary of Characteristics Constraining Variability

Characteristic	Available Choices
Introduction Times	Architecture Design, Detailed Design, Implementation, Compilation, Linking
Software Entity	Components, Frameworks, Framework Implementations, Sets of Classes, Individual Classes, Lines of Code
Times for Adding new Variants	Architecture Design, Detailed Design, Implementation, Compilation, Linking
Binding Times	Product Architecture Derivation, Compilation, Linking, Runtime
Management of Collection of Variants	Implicit or Explicit
Placement of Functionality for Binding	Internal or External

Typically, these are referred to as Plug-ins, and these can normally be developed by third party vendors. Another type of runtime binding, perhaps not as interactive, is the interpretation of configuration files or startup parameters that determines what variant to bind to. This type of runtime binding is what is normally called parameterization.

Note that binding times do not include the design and implementation phases. Variation points may well be introduced during these phases, but to the best of our knowledge a system can not be bound to a particular variant on other occasions than the ones presented above.

Furthermore, there is an additional aspect of binding, namely whether the binding is done internally or externally. An *internal* binding implies that the system contains the functionality to bind to a particular variant. This is typically true for the binding that is done during runtime of the system. An *external* binding implies that there is a person or a tool that performs the actual binding. This is typically true for the binding that is done during product architecture derivation, compilation, and linking, where tools such as configuration management tools, compilers and linkers perform the actual binding.

Linking is sort of a special case since if it is done dynamically during runtime the system may, or may not, be in control of the binding, which makes linking external in some cases but internal in others.

Whether to bind internally or externally is decided by many things, such as whether the binding is done by the software developers or the end users, and whether the binding should be made transparent to the end users or not. Moreover, an external binding can sometimes be preferred as it does not necessarily leave any traces in the source code, as is the case when the binding is internal and the system must contain functionality to bind. Thus, an external binding helps in reducing the complexity of the source code.

As with the adding of variants, the time when one wants to bind the system constrains the selection of possible ways to implement a variation point. For a variant feature resulting in many variation points, this results in quite a few problems, as the variation points need to be bound either at the same time (as is the case if binding is required at runtime), or that the binding of several variation points is synchronized so that, for example, a variation point that is bound during compilation binds to the same variant that related variation points have already bound to during product architecture derivation.

When determining when to bind a variant feature to a particular variant, what needs to be considered is how late binding is absolutely required. As a rule of thumb, one can say that the later the binding is done, the more costly it is. Deferring binding from product architecture derivation to compilation means that developers need to manage all variants during implementation, and deferring binding from compilation to runtime means that the system will have to include binding functionality, and there is a cost in terms of e.g. performance to perform the binding. However, as we discussed related to adding variants to the system, the binding time may be determined by business strategies, delivery models and development processes. Naturally, this works both ways. There may be guidelines in the business strategy that binding should not be performed after a certain point, as well as a requirement that binding should be deferred until as late as possible.

4.4 Summary

In summary, there are a number of aspects to consider when selecting how to actually implement a variant feature. The first of these aspects is when to introduce the variant feature in terms of variation points and variants, which ultimately depends on the size of the software entities representing the variants. Secondly, there are two aspects to consider regarding when and how to add new variants, namely when the variation points are open for adding and whether or not the collection of variants should be managed implicitly by the developers and users or whether it should be explicitly represented in the system itself. Thirdly, the binding of a system to a particular variant is governed by the two aspects when to bind, and whether the binding is done externally by developers or users (potentially using a software tool to perform the binding), or whether it should be done internally by the system itself. The characteristics and the possible choices are summarized in Table 2.

5. VARIABILITY REALIZATION TECHNIQUES

To summarize what we have presented hitherto, we have, in Section 3, presented how variability is first identified and then constrained. In Section 4 we discussed in further detail how, from an implementation point of view, variability is constrained by a number of characteristics. The next step is to use the identified aspects of a particular variant feature, i.e. the size of the involved software entities, when it should be introduced, when it should be possible to add new variants, and when it needs to be bound to a particular variant, to select which way to implement the variation points associated with the variant feature. These ways to implement variation points we refer to as variability realization techniques. In this section we present the vari-

Table 3: Variability Realization Techniques

Involved Software Entities	Binding Time			
	Product Architecture Derivation	**Compilation**	**Linking**	**Runtime**
Components Frameworks	Architecture Reorganization (Section 5.1.1)	N/A	Binary Replacement - Linker Directives (Section 5.1.4)	Infrastructure-Centered Architecture (Section 5.1.6)
	Variant Architecture Component (Section 5.1.2)			
	Optional Architecture Component (Section 5.1.3)		Binary Replacement - Physical (Section 5.1.5)	
Framework Implementations Classes	Variant Component Specializations (Section 5.1.7)	Code Fragment Superimposition (Section 5.1.13)	Binary Replacement - Linker Directives (Section 5.1.4)	Runtime Variant Component Specializations (Section 5.1.9)
	Optional Component Specializations (Section 5.1.8)		Binary Replacement - Physical (Section 5.1.5)	Variant Component Implementations (Section 5.1.10)
Lines of Code	N/A	Condition on Constant (Section 5.1.11)	N/A	Condition on Variable (Section 5.1.12)
		Code Fragment Superimposition (Section 5.1.13)		

ability realization techniques we have knowledge of and those that we have come across during our collaborations with industry. Most likely, this list is not complete, and we encourage readers to submit missing realization techniques to the authors.

The variability realization techniques are summarized in Table 3. In this table, the variability realization techniques are organized according to the software entity the variability realization techniques work with, and when it is, at the latest, possible to bind them. For each variability realization technique, there is also a reference to a more detailed description of the technique, which are presented below. There are some areas in this table that are shaded, where we perceive that it is not interesting to have any variability realization techniques. These areas are:

- Components and Frameworks during compilation, as compilation works with smaller software entities. This type of software entities comes into play again only during linking.
- Lines of Code during Product Architecture Derivation, as we know of no tools working with product architecture derivation that also work with lines of code.
- Lines of Code during Linking, as linkers work with larger software entities.

5.1 A Detailed Description of the Variability Realization Techniques

Below, we present each of these realization techniques in further detail. We present these using a Design Pattern like form, in the style used by e.g. [Buschman et al. 1996] and [Gamma et al. 1995]. For each of the variability realization techniques we discuss the following topics:

- **Intent.** This is a short description of the intent of the realization technique.
- **Motivation.** A description of the problems that the realization technique address, and other forces that may be at play.
- **Solution.** Known solutions to the problems presented in the motivation section.
- **Lifecycle.** A description of when the realization technique is open, when it closes, and when it allows binding to one of the variants.
- **Consequences.** The consequences of using the realization technique, both positive and negative effects.
- **Examples.** Some examples of the realization technique in use at the companies in which we have conducted case studies.

5.1.1 Architecture Reorganization

Intent. Support several product specific architectures by reorganizing the overall product line architecture.

Motivation. Although products in a product line share many concepts, the control flow and data flow between these concepts need not be the same. Therefore, the product line architecture is reorganized to form the concrete product architectures. This involves mainly changes in the control flow, i.e. the order in which components are connected to each other, but may also consist of changes in how particular components are connected to each other, i.e. the provided and required interface of the components may differ from product to product.

Solution. This technique is implicit and external, as there is no first-class representation of the architecture in the system. For an explicit realization technique, see Infrastructure-Centered Architecture. In the Architecture Reorganization technique, the components are represented as subsystems controlled by configuration management tools or, at best, Architecture Description

Languages. What variants to include in a system is determined the configuration management tools. The actual architecture is then depending on variability realization techniques on lower levels, for example Variant Component Specialization.

Lifecycle. This technique is open for the adding of new variations during architectural design, where the product line architecture is used as a template to create a product specific architecture. As detailed design commences, the architecture is no longer a first class entity, and can hence not be further reorganized. Binding time, i.e. when a particular architecture is selected, is when a particular product architecture is derived from the product line architecture. This also implies that this is not a technique for achieving dynamic architectures. If this is what is required, see Infrastructure-Centered Architecture.

Consequences. The major disadvantage of Architecture Reorganization is that, although there is no first class representation of the architecture on subsequent development phases, they (the subsequent phases) still need to be aware of the potential reorganizations. Code is thus added to cope with this reorganization, be it used in a particular product or not.

Examples. At Axis Communications, a hierarchical view of the Product Line Architecture is employed, where different products are grouped in sub-trees of the main Product Line. To control the derivation of one product out of this tree, a rudimentary, in-house developed, ADL is used. Another example is Symbian that reorganizes the architecture of the EPOC operating system for different hardware system families.

Table 4: Summary of Architecture Reorganization

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.2 Variant Architecture Component

Intent. Support several, differing, architectural components representing the same conceptual entity.

Motivation. In some cases, an architectural component in one particular place in the architecture can be replaced with another that may have a differing interface, and sometimes also representing a different domain. This need not affect the rest of the architecture. For example, some products may work with hard disks, whereas others (in the same product line) may work with scanners. In this case, the scanner component replaces the hard disk component without further affecting the rest of the architecture.

Solution. The solution to this is to, as the title implies, support these architectural components in parallel. The selection of which to use any given moment is then delegated to the configuration management tools that select what component to include in the system. Parts of the solution is also delegated to subsequent development phases, where the Variant Component Specialization will be used to call and operate with the different components in the correct way. To summarize, this technique has an implicit collection, and the binding functionality is external.

Lifecycle. It is possible to add new variants, i.e. parallel components, during architectural design, when new components can be added, and also during detailed design, where these components are concretely designed as separate architectural components. The architecture is bound to a particular component during the transition from a product line architecture to a product architecture, when the configuration management tool selects what architectural component to use.

Consequences. A consequence of using this pattern is that the decision of what component interface to use, and how to use it, is placed in the calling components rather than where the actual variant feature is implemented. Moreover, the handling of the differing interfaces cannot be coped with during the same development phase as the varying component, but has to be deferred until later development stages.

Examples. At Axis Communications, there existed during a long period of time two versions of a file system component; one supporting both read and write functionality, and one supporting only read functionality. Different products used either the read-write or the read-only component. Since they differed in the interface and implementation, they were, in effect, two different architectural components.

Table 5: Summary of Variant Architecture Component

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.3 Optional Architecture Component

Intent. Provide support for a component that may, or may not be present in the system.

Motivation. Some architectural components may be present in some products, but absent in other. For example, a Storage Server at Axis Communications can optionally be equipped with a so-called hard disk cache. This means that in one product configuration, other components need to interact with the hard disk cache, whereas in other configurations, the same components do not interact with this architectural component.

Solution. There are two ways of solving this problem, depending on whether it should be fixed on the calling side or the called side. If we desire to implement the solution on the calling side, the solution is simply delegated to variability realization techniques introduced during later development phases. To implement the solution on the called side, which may be nicer, but is less efficient, create a “null” component, i.e. a component that has the correct interface, but replies with dummy values. This latter approach assumes, of course, that there are predefined dummy values that the other components know to ignore. The binding for this technique is done external to the system.

Lifecycle. This technique is open when a particular product architecture is designed based on the product line architecture, but, for the lack of architecture representation during later development phases, is closed at all other times. The architecture is bound to the existence or non-existence of a component when a product architecture is selected from the product line architecture.

Consequences. Consequences of using this technique is that the components depending on the optional component must either have realization techniques to support its not being there, or have techniques to cope with dummy values. The latter technique also implies that the “plug”, or the null component, will occupy space in the system, and the dummy values will consume processing power. An advantage is that should this variation point later be extended to be of the type variant architecture component, the functionality is already in place, and all that needs to be done is to add more variants for the variant feature.

Examples. The Hard Disk Cache at Axis Communications, as described above. Also, in the EPOC Operating System, the presence or absence of a network connection decides whether network drivers should be loaded or not.

Table 6: Summary of Optional Architecture Component

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.4 Binary Replacement - Linker Directives

Intent. Provide the system with alternative implementations of underlying system libraries.

Motivation. In some cases, all that is required to support a new platform is that an underlying system library is replaced. For example, when compiling a system for different UNIX-dialects, this is often the case. It need not even be a system library, it can also be a library distributed together with the system to achieve some variability. For example, a game can be released with different libraries to work with the window system (Such as X-windows), an OpenGL graphics device or to use a standard SVGA graphics device.

Solution. Represent the variants as stand-alone library files, and instruct the linker which file to link with the system. If this linking is done at runtime, the binding functionality must be internal to the system, whereas it can, if the linking is done during the compile and linking phase prior to delivery, be external and managed by a traditional linker. An external binding also implies, in this case, an implicit collection.

Lifecycle. This technique is open for new variants as the system is linked. It is also bound during this phase. As the linking phase ends, this technique becomes unavailable. However, it should be noted that the linking phase need not end. In modern systems, linking is also available during execution.

Consequences. This is a fairly well developed variability realization technique, and the consequences of using it are relatively harmless.

Examples. For Linux, the web browser Konqueror can optionally use the web browsing component of Mozilla instead of its own web browsing component in this fashion.

5.1.5 Binary Replacement - Physical

Intent. Facilitate the modification of software after delivery.

Motivation. Unfortunately, very few software systems are released in a perfect and optimal state, which creates a need to upgrade the system after delivery. In some cases, these upgrades can be done using the variability realization techniques at

Table 7: Summary of Binary Replacement - Linker Directives

Introduction Times	Architecture Design
Open for Adding Variants	Linking
Collection of Variants	Implicit or Explicit
Binding Times	Linking
Functionality for Binding	External or Internal

variation points already existing in the system, but in others, the system does not currently support variability at the places needed.

Solution. In order to introduce a new variation point after delivery, the software binary must be altered. The easiest way of doing this is to replace an entire file with a new copy. To facilitate this replacement, the system should thus be organized as a number of relatively small binary files, to localize the impact of replacing a file. Furthermore, the system can be altered in two ways: Either the new binary completely covers the functionality of the old one, or the new binary provides additional functionality in the form of, for example, a new variant feature using other variability realization techniques. In this technique the is collection implicit, and the binding external to the system.

Lifecycle. This technique is bound before start-up (i.e. before runtime) of the system. In this technique the method for binding to a variant is also the one used to add new variants. After delivery (i.e. after compilation), the technique is always open for adding new variants.

Consequences. If the new binary does not introduce a “traditional” variation point, the same technique will have to be used again the next time a new variant for the variant feature in question is detected. However, if traditional variation points are introduced, this facilitates future changes at this particular point in the system. Replacing binary files is normally a volatile way of upgrading a system, since the rest of the system may in some cases even be depending on software bugs in the replaced binary in order to function correctly. Moreover, it is not trivial to maintain the release history needed to keep consistency in the system. Furthermore, there are also some trust issues to consider here, e.g. who provides the replacement component, and what are the guarantees that the replacement component actually does what it is supposed to do.

Examples. Axis Communications provide a possibility to upgrade the software in their devices by re-flashing the ROM. This basically replaces the entire software binary with a new one.

Table 8: Summary of Binary Replacement - Physical

Introduction Times	Architecture Design
Open for Adding Variants	After Compilation
Collection of Variants	Implicit
Binding Times	Before Runtime
Functionality for Binding	External

5.1.6 Infrastructure-Centered Architecture

Intent. Make the connections between components a first class entity.

Motivation. Part of the problem when connecting components, and in particular components that may vary, is that the knowledge of the connections is often hard coded in the required interfaces of the components, and is thus implicitly embedded into the system. A reorganization of the architecture, or indeed a replacement of a component in the architecture, would be vastly facilitated if the architecture is an explicit entity in the system, where such modifications could be performed.

Solution. Convert the connectors into first class entities, so the components are no longer connected to each other, but are rather connected to the infrastructure, i.e. the connectors. This infrastructure is then responsible for matching the required interface of one component with the provided interface of one or more other components. The infrastructure can either be an existing standard, such as COM or CORBA [Szyperski 1997], or it can be an in-house developed standard. The infrastructure may also be a scripting language, in which the connectors are represented as snippets of code that are responsible for binding the components together in an architecture. These code snippets can either be done in the same programming language as the rest of the system, or it can be done using a scripting language. Such scripting languages are, according to e.g. [Ousterhout 1998], highly suitable for “gluing” components together. The collection of variants is, in this realization technique, either implicit or explicit, and the binding functionality is internal, provided by the infrastructure.

Lifecycle. Depending on what infrastructure is selected, the technique is open for adding new variants during a shorter or longer period. In some cases, the infrastructure is open for the addition of new components as late as during runtime, and in other cases, the infrastructure is concretized during compile and linking, and is thus open for new additions only until then. However, since the additions are in the magnitude of architectural components or component implementations, it becomes

unpractical to talk about adding new variants during, for example, the implementation phase, as components are not in focus during this phase. This realization technique can be seen as open for adding new variants during architectural design, and during runtime. If this perspective is taken, it is closed during all other phases, because it is not relevant to model this type of variation in any of the intermediate development phases. Another view is that the variability realization technique is only open during linking, which may be performed at runtime. The latter perspective assumes a minimalistic view of the system, where anything added to the infrastructure is not really added until at link-time. The technique binds the system to a particular variant either during compilation time, when the infrastructure is tied to the concrete range of components, or at runtime, if the infrastructure supports dynamical adding of new components.

Consequences. Used correctly, this realization technique yields perhaps the most dynamic of all architectures. Performance is impeded slightly because the components need to abstract their connections to fit the format of the infrastructure, which then performs more processing on a connection, before it is concretized as a traditional interface call again. In many ways, this technique is similar to the Adapter Design Pattern [Gamma et al. 1995].

The infrastructure does not remove the need for well-defined interfaces, or the troubles with adjusting components to work in different operating environments (i.e. different architectures), but it removes part of the complexity in managing these connections.

Examples. Programming languages and tools such as Visual Basic, Delphi and JavaBeans support a component based development process, where the components are supported by some underlying infrastructure. Another example is the Mozilla web browser, which makes extensive use of a scripting language, in that everything that can be varied is implemented in a scripting language, and only the atomic functionality is represented as compiled components.

Table 9: Summary of Infrastructure-Centered Architecture

Introduction Times	Architecture Design
Open for Adding Variants	Architecture Design Linking Runtime
Collection of Variants	Implicit or Explicit
Binding Times	Compilation Runtime
Functionality for Binding	Internal

5.1.7 Variant Component Specializations

Intent. Adjust a component implementation to the product architecture.

Motivation. Some variability realization techniques on the architectural design level require support in later stages. In particular, those techniques where the provided interfaces vary need support from the required interface side as well. In these cases, what is required is that parts of a component implementation, namely those parts that are concerned with interfacing a component representing a variant of a variant feature, needs to be replaceable as well. This technique can also be used to tweak a component to fit a particular product's needs.

Solution. Separate the interfacing parts into separate classes that can decide the best way to interact with the other component. Let the configuration management tool decide what classes to include at the same time as it is decided what variant of the interfaced component to include in the product architecture. Accordingly, this technique has an implicit collection, and external binding functionality.

Lifecycle. The available variants are introduced during detailed design, when the interface classes are designed. The technique is closed during architectural design, which is unfortunate since it is here that it is decided that the variability realization technique is needed. This technique is bound when the product architecture is instantiated from the source code repository.

Consequences. Consequences of using classes are that it introduces another layer of indirection, which may consume processing power (Although today, the extra overhead incurred by an extra layer of indirection is minimal.). Nor may it always be a simple task to separate the interface. Suppose that the different variants require different feedback from the common parts, then the common part will be full with method calls to the varying parts, of which only a subset is used in a particular configuration. Naturally this hinders readability of the source code. However, the use of classes like this has the advantage that the variation point is localized to one place in the source code, which facilitates adding more variants and maintaining the existing variants.

Examples. The Storage Servers at Axis Communications can be delivered with a traditional cache or a hard disk cache. The file system component must be aware of which is present, since the calls needed for the two are slightly differing. Thus, the file system component is adjusted using this variability realization technique to work with the cache type present in the system.

Table 10: Summary of Variant Component Specialization

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.8 Optional Component Specializations

Intent. Include or exclude parts of the behavior of a component implementation.

Motivation. A particular component implementation may be customized in various ways by adding or removing parts of its behavior. For example, depending on the screen size an application for a handheld device can opt not to include some features, and in the case when these features interact with others, this interaction also needs to be excluded from the executing code.

Solution. Separate the optional behavior into a separate class, and create a "null" class that can act as a placeholder when the behavior is to be excluded. Let the configuration management tools decide which of these two classes to include in the system. Alternatively, surround the optional behavior with compile-time flags to exclude it from the compiled binary. Binding is in this technique done externally, by the configuration management tools or the compiler.

Lifecycle. This technique is introduced during detailed design, and is immediately closed to adding new variants, unless the variation point is transformed into a Variant Component Specialization. The system is bound to the inclusion or exclusion during the product architecture derivation or, if the second solution is chosen, during compilation.

Consequences. It may not be easy to separate the optional behavior into a separate class. The behavior may be such that it cannot be captured by a "null" class.

Examples. At one point, when Axis Communications added support for Novel Netware, some functionality required by the filesystem component was specific for Netware. This functionality was fixed external of the file system component, in the Netware component. As the functionality was later implemented in the file system component, it was removed from the Netware component. The way to implement this was in the form of an Optional Component Specialization.

Table 11: Summary of Optional Component Specialization

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Implicit
Binding Times	Product Architecture Derivation
Functionality for Binding	External

5.1.9 Runtime Variant Component Specializations

Intent. Support the existence and selection between several specializations inside a component implementation.

Motivation. It is required of a component implementation that it adapts to the environment in which it is executing, i.e. that for any given moment during the execution of the system, the component implementation is able to satisfy the requirements from the user and the rest of the system. This implies that the component implementation is equipped with a number of alternative executions, and is able to, at runtime, select between these.

Solution. Basically, there are two Design Patterns [Gamma et al. 1995] that are applicable here: Strategy and Template Method. Alternating behavior is collected into separate classes, and mechanisms are introduced to, at runtime, select between these classes. Using Design Patterns makes the collection explicit, and the binding is done internally, by the system.

Lifecycle. This technique is open for new variations during detailed design, since classes and object oriented concepts are in focus during this phase. Because these are not in focus in any other phase, this technique is not available anywhere else. The system is bound to a particular specialization at runtime, when an event occurs.

Consequences. Depending upon the ease by which the problem divides into a generic and variant parts, more or less of the behavior can be kept in common. However, the case is often that even common code is duplicated in the different strategies. A hypothesis is that this could stem from quirks in the programming language, such as the self problem [Lieberman 1986].

Examples. A hand-held device can be attached to communication connections with differing bandwidths, such as a mobile phone or a LAN, and this implies different strategies for how the EPOC operating system retrieves data. Not only do the algorithms for, for example, compression differ, but on a lower bandwidth, the system can also decide to retrieve less data, thus

reducing the network traffic. This variant need not be in the magnitude of an entire component, but can often be represented as strategies within the concerned components.

Table 12: Summary of Runtime Variant Component Specializations

Introduction Times	Detailed Design
Open for Adding Variants	Detailed Design
Collection of Variants	Explicit
Binding Times	Runtime
Functionality for Binding	Internal

5.1.10 Variant Component Implementations

Intent. Support several concurrent and coexisting implementations of one architectural component.

Motivation. An architectural component typically represents some domain, or sub-domain. These domains can be implemented using any of a number of standards, and typically a system must support more than one simultaneously. For example, a hard disk server typically supports several network file system standards, such as SMB, NFS and Netware, and is able to choose between these at runtime. Forces in this problem is that the architecture must support these different component implementations, and other components in the system must be able to dynamically determine to what component implementation data and messages should be sent.

Solution. Implement several component implementations adhering to the same interface, and make these component implementations tangible entities in the system architecture. There exists a number of Design Patterns [Gamma et al. 1995] that facilitates in this process. For example, the Strategy pattern is, on a lower level, a solution to the issue of having several implementations present simultaneously. Using the Broker pattern is one way of assuring that the correct implementation gets the data, as are patterns like Abstract Factory and Builder. Part of the flexibility of this variability realization technique stems from the fact that the collection is explicitly represented in the system, and the binding is done internally.

The decision on exactly what component implementations to include in a particular product can be delegated to configuration management tools.

Lifecycle. This technique is introduced during architectural design, but is not open for addition of new variants until detailed design. It is not available during any other phases. Binding time of this technique is at runtime. The binding is done either at start-up, where a start-up parameter decides which component implementation to use, or at runtime, when an event decides which implementation to use. If the system supports dynamic linking, the linking can be delayed until binding time, but the technique work equally well when all variants are already compiled into the system. However, if the system does support dynamic linking, the technique is in fact open for adding new variations even during runtime.

Consequences. Consequences of using this technique are that the system will support several implementations of a domain simultaneously, and it must be possible to choose between them either at start-up or during execution of the system. Similarities in the different domains may lead to inclusion of several similar code sections into the system, code that could have been reused, had the system been designed differently.

Examples. Axis Communications uses this technique to, for example, select between different network communication standards. Ericsson Software Technology uses this technique to select between different filtering techniques to perform on call data in their Billing Gateway product. The web browsing component of Mozilla, called Gecko, supports the same interface that enables Internet Explorer to be embedded in applications, thus enabling Gecko to be used in embedded applications as an alternative to Internet Explorer.

Table 13: Summary of Variant Component Implementations

Introduction Times	Architecture Design
Open for Adding Variants	Detailed Design
Collection of Variants	Explicit
Binding Times	Runtime
Functionality for Binding	Internal

5.1.11 Condition on Constant

Intent. Support several ways to perform an operation, of which only one will be used in any given system.

Motivation. Basically, this is a more fine-grained version of a Variant Component Specializations, where the variant is not large enough to be a class in its own right. The reason for using the condition on constant technique can be for performance reasons, and to help the compiler remove unused code. In the case where the variant concerns connections to other, possibly

variant, components, it is also a means to actually get the code through the compiler, since a method call to a nonexistent class would cause the compilation process to abort.

Solution. We can, in this technique, use two different types of conditional statements. One form of conditional statements is the pre-processor directives such as C++ ifdefs, and the other is the traditional if-statements in a programming language. If the former is used, it can actually be used to alter the architecture of the system, for example by opting to include one file over another or using another class or component, whereas the latter can only work within the frame of one system structure. In both cases, the collection of variants is implicit, but, depending on whether traditional constants or pre-processor directives are used, the binding is either internal or external, respectively. Another way to implement this variability realization technique is by means of the C++ constructs templates, which is, in our experience, handled as pre-processor directives by most compilers we have encountered. (Granted, it is a long time since we had a chance to work with C++, and evolution of what one can do with templates has moved forward, so our knowledge of this may be a bit rusty. Templates may today be a variability realization technique in its own merit.)

Lifecycle. This technique is introduced while implementing the components, and is activated during compilation of the system, where it is decided using compile-time parameters which variation to include in the compiled binary. If a constant is used instead of a compile-time parameter, this is also bound at this point. After compilation, the technique is closed for adding new variations.

Consequences. Using ifdefs, or other pre-processor directives, is always a risky business, since the number of potential execution paths tends to explode when using ifdefs, making maintenance and bug-fixing difficult. Variation points often tend to be scattered throughout the system, because of which it gets difficult to keep track of what parts of a system is actually affected by one variant.

Examples. The different cache types in Axis Communications different Storage Servers, that can either be a Hard Disk cache or a traditional cache, where the file system component must call the one present in the system in the correct way. Working with the cache is spread throughout the file system component, because of which many variability realization techniques on different levels are used, including in some cases Condition on Constant.

Table 14: Summary of Condition on Constant

Introduction Times	Implementation
Open for Adding Variants	Implementation
Collection of Variants	Implicit
Binding Times	Compilation
Functionality for Binding	Internal or External

5.1.12 Condition on Variable

Intent. Support several ways to perform an operation, of which only one will be used at any given moment, but allow the choice to be rebound during execution.

Motivation. Sometimes, the variability provided by the Condition on Constant technique needs to be extended into runtime as well. Since constants are evaluated at compilation, this cannot be done, because of which a variable must be used instead.

Solution. Replace the constant used in Condition on Constant with a variable, and provide functionality for changing this variable. This technique cannot use any compiler directives, but is rather a pure programming language construct. The collection of variants pertaining to the variation point need not be explicit, and the binding to a particular variant is internal.

Lifecycle. This technique is open during implementation, where new variants can be added, and is closed during compilation. It is bound at runtime, where the variable is given a value that is evaluated by the conditional statements.

Consequences. This is a very flexible realization technique. It is a relatively harmless technique, but, as with Condition on Constant, if the variation points for a particular variant feature are spread throughout the code, it becomes difficult to get an overview.

Examples. This technique is used in all software programs to control the execution flow.

Table 15: Summary of Condition on Variable

Introduction Times	Implementation
Open for Adding Variants	Implementation
Collection of Variants	Implicit or Explicit
Binding Times	Runtime
Functionality for Binding	Internal

5.1.13 Code Fragment Superimposition

Intent. Introduce new considerations into a system without directly affecting the source code.

Motivation. Because a component can be used in several products, it is not desired to introduce product-specific considerations into the component. However, it may be required to do so in order to be able to use the component at all. Product specific behavior can be introduced in a multitude of ways, but these all tend to obscure the view of the component's core functionality, i.e. what the component is really supposed to do. It is also possible to use this technique to introduce variants of other forms that need not have to do with customizing source code to a particular product.

Solution. The solution to this is to develop the software to function generically, and then superimpose the product-specific concerns at stage where the work with the source code is completed anyway. There exists a number of tools for this, for example Aspect Oriented Programming [Kiczalez et al.1997], where different concerns are weaved into the source code just before the software is passed to the compiler and superimposition as proposed by [Bosch 1999b], where additional behavior is wrapped around existing behavior. The collection is, in this case, implicit, and the binding is performed externally.

Lifecycle. This technique is open during the compilation phase, where the system is also bound to a particular variation. However, the superimposition can also simulate the adding of new concerns, or aspects, at runtime. These are in fact added at compilation but the binding is deferred to runtime, by internally using other variability realization techniques, such as Condition on Variable.

Consequences. Consequences of superimposing an algorithm are that different concerns are separated from the main functionality. However, this also means that it becomes harder to understand how the final code will work, since the execution path is no longer obvious. When developing, one must be aware that there will be a superimposition of additional code at a later stage. In the case where binding is deferred to runtime, one must even program the system to add a concern to an object.

Examples. To the best of our knowledge, none of the case companies use this technique. This is not surprising, considering that most tools for this technique are at a research and prototyping stage.

Table 16: Summary of Code Fragment Superimposition

Introduction Times	Compilation
Open for Adding Variants	Compilation
Collection of Variants	Implicit
Binding Times	Compilation Runtime
Functionality for Binding	External

5.2 Summary

In this section we present a taxonomy of variability realization techniques. These techniques make use of various implementation techniques, as identified by [Jacobson et al. 1997]: inheritance, extensions, parameterization, configuration and generation. The variability realization techniques are categorized by a number of characteristics, as summarized in Table 17.

6. CASE STUDIES

In this section we briefly present a set of companies that use product lines, and how these have typically implemented variability, i.e. what variability realization techniques they have mostly used in their software product lines.

The cases are divided into three categories:

- Cases which we based the taxonomy of variability realization techniques on.
- Unrelated case studies conducted after the initial taxonomy was created, which were used to confirm and refine the taxonomy.
- Cases found in literature, that contains information regarding how variability was typically implemented.

We provide a brief presentation of the companies within each category, and how they have typically implemented variability. The cases from the first category are presented to give a further overview of the companies behind the examples in the taxonomy. The second category is presented to give further examples of which we have in-depth knowledge and have had full insight in the development process of, and which have confirmed or confuted our taxonomy. The third category is included to extend the generalizability of the taxonomy further, by means of increasing the statistical power of our findings.

In the first category, the taxonomy of variability realization techniques, and indeed the identification of the relevant characteristics to distinguish between different variability realization techniques, was created using information gathered from four companies. These companies are:

- Axis Communications AB and their storage server product line [Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999b][Bosch 2000] (presented in Section 6.1)
- Ericsson Software Technology and their Billing Gateway product [Mattsson & Bosch 1999a][Mattsson & Bosch 1999b][Svahnberg & Bosch 1999a] (presented in Section 6.2)
- The Mozilla web browser [Mozilla][Oeschger 2000][van Gurp et al. 2001] (presented in Section 6.3)
- Symbian and the EPOC Operating System [Symbian][Bosch 2000] (presented in Section 6.4)

In the second category we have case studies conducted by the research groups of the authors of this paper. These case studies were not conducted with the purpose of neither creating nor refining the taxonomy of variability realization techniques, but during these studies we have had the opportunity to see and understand their software product lines to such a degree that we can also make confident statements regarding how these companies choose implementation strategies for their variant features, and what these implementation strategies are. The companies in this category are:

- NDC Automation AB [Svahnberg & Mattsson 2002] (presented in Section 6.5)
- Rohill Technologies BV [Jaring & Bosch 2002] (presented in Section 6.6)

In the third, and final, category, we include examples of case studies described in literature, where these descriptions are of sufficient detail to discern what types of variability realization techniques these companies typically use. The cases in this category are:

- Cummins Inc. [Clements & Northrop 2002] (presented in Section 6.7)
- Control Channel Toolkit [Clements & Northrop 2002] (presented in Section 6.8)
- Market Maker [Clements & Northrop 2002] (presented in Section 6.9)

6.1 Axis Communications AB

Axis Communications is a medium sized hardware and software company in the south of Sweden. They develop mass-market networked equipment, such print servers, various storage servers (CD-ROM servers, JAZ servers and Hard disk servers), camera servers and scan servers. Since the beginning of the 1990s, Axis Communications has employed a product line approach. This Software Product Line consists of 13 reusable assets. These Assets are in themselves object-oriented frameworks, of differing size. Many of these assets are reused over the complete set of products, which in some cases have quite differing requirements on the assets. Moreover, because the systems are embedded systems, there are very stringent memory requirements; the application, and hence the assets, must not be larger than what is already fitted onto the motherboard. What this implies is that only the functionality used in a particular product may be compiled into the product software, and this calls for a somewhat different strategy when it comes to variation handling.

Table 17: Summary of Variability Realization Techniques

Name	Introduction Time	Open for Adding Variants	Collection of Variants	Binding Times	Functionality for Binding
Architecture Reorganization	Architecture Design	Architecture Design	Implicit	Product Architecture Derivation	External
Variant Architecture Component	Architecture Design	Architecture Design Detailed Design	Implicit	Product Architecture Derivation	External
Optional Architecture Component	Architecture Design	Architecture Design	Implicit	Product Architecture Derivation	External
Binary Replacement - Linker Directives	Architecture Design	Linking	Implicit or Explicit	Linking	External or Internal
Binary Replacement - Physical	Architecture Design	After Compilation	Implicit	Before Runtime	External
Infrastructure-Centered Architecture	Architecture Design	Architecture Design Linking Runtime	Implicit or Explicit	Compilation Runtime	Internal
Variant Component Specializations	Detailed Design	Detailed Design	Implicit	Product Architecture Derivation	External
Optional Component Specializations	Detailed Design	Detailed Design	Implicit	Product Architecture Derivation	External
Runtime Variant Component Specializations	Detailed Design	Detailed Design	Explicit	Runtime	Internal
Variant Component Implementations	Architecture Design	Detailed Design	Explicit	Runtime	Internal
Condition on Constant	Implementation	Implementation	Implicit	Compilation	Internal or External
Condition on Variable	Implementation	Implementation	Implicit or Explicit	Runtime	Internal
Code Fragment Superimposition	Compilation	Compilation	Implicit	Compilation or Runtime	External

In this paper we have given several examples of how Axis implements variability in its software product line, but the variability realization technique they prefer is that of variant component implementations (Section 5.1.10), which is augmented with runtime variant component specializations (Section 5.1.9). Axis use several other variability realization techniques as well, but this is more because of architectural decay which has occurred during the evolution of the software product line.

Further information can be found in two papers by Svahnberg & Bosch [Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999b] and in our co-author's book on software product lines [Bosch 2000].

6.2 Ericsson Software Technology

Ericsson Software Technology is a leading software company within the telecommunications industry. At their site in Ronneby, in the same building as our university, they develop their Billing Gateway product. The Billing Gateway is a mediating device between telephone switching stations and post-processing systems such as billing systems, fraud control systems, etc. The Billing Gateway has also been developed since the early 1990's, and is currently installed at more than 30 locations worldwide. The system is configured for every customer's needs with regards to, for instance, what switching station languages to support, and each customer builds a set of processing points that the telephony data should go through. Examples of processing points are formatters, filters, splitters, encoders, decoders and routers. These are connected into a dynamically configurable network through which the data is passed.

Also for Ericsson, we have given several examples of how variability is implemented. As with Axis Communications, the favoured variability realization technique is that of variant component implementations (Section 5.1.10), but Ericsson has managed to keep the interfaces and connectors between the software entities intact as the system has evolved, so there is lesser need to augment this realization technique with other techniques.

For further reading, see [Mattsson & Bosch 1999a][Mattsson & Bosch 1999b] and [Svahnberg & Bosch 1999a].

6.3 Mozilla

The Mozilla web browser is Netscape's Open Source project to create their next generation of web browsers. One of the design goals of Mozilla is to be a platform for web applications. Mozilla is constructed using a highly flexible architecture, which makes massive use of components. The entire system is organized around an infrastructure of XUL, a language for defining user interfaces, JavaScript, to bind functionality to the interfaces, and XPCOM, a COM-like model with components written in languages such as C++. The use of C++ for lower level components ensures high performance, whereas XUL and JavaScript ensure high flexibility concerning appearance (i.e. how and what to display), structure (i.e. the elements and relations) and interactions (i.e. the how elements work across the relations). This model enables Mozilla to use the same infrastructure for all functionality sets, which ranges from e-mail and news handling to web browsing and text editing. Moreover, any functionality defined in this way is platform independent, and only require the underlying C++ components to be reconstructed and/or recompiled for new platforms. Variability issues here concern the addition of new functionality sets, i.e. applications in their own right, and incorporation of new standards, for instance regarding data formats such as HTML, PDF and XML.

As described above, Mozilla connects its components using XUL and XPCOM. In our taxonomy, this would translate to the use of an infrastructure-centered architecture (Section 5.1.6).

For further information regarding Mozilla, see [Mozilla], [Oeschger 2000] and [van Gurp et al. 2001].

6.4 Symbian - Epoc

EPOC is an operating system, an application framework, and an application suite specially designed for wireless devices such as hand-held, battery powered, computers and cellular phones. It is developed by Symbian, a company that is owned by major companies within the domain, such as Ericsson, Nokia, Psion, Motorola and Matsushita, in order to be used in these companies' wireless devices. Variability issues here concern how to allow third party applications to seamlessly and transparently integrate with a multitude of different operating environments, which may even affect the amount of functionality that the applications provide. For instance, with screen sizes varying from a full VGA screen to a two-line cellular phone, the functionality, and how this functionality is presented to the user, will differ vastly between the different platforms.

Symbian, by means of EPOC, does not interfere in how applications for the EPOC operating system implement variability. they do, however, provide support for creating applications supporting different operating environments. This is done by dividing applications into a set of components handling user interface, application control and data storage (i.e. a Model-View-Controller pattern [Buschman et al. 1996]). The EPOC operating system itself is specialized for different hardware environments by using the architecture reorganization (Section 5.1.1) and variant architecture component (Section 5.1.2) variability realization techniques. Mainly, different hardware environments are related to differences in screen sizes.

More information can be obtained from Symbian's website [Symbian] and in [Bosch 2000].

6.5 NDC Automation AB

NDC Automation AB develops general control systems, software and electronic equipment in the field of materials handling control. Specifically, they develop the control software for automated guided vehicles, i.e. automatic vehicles that handle transport of goods on factory floors. NDC's product line consists of a range of software components that together control the assignment of cargo to vehicles, monitor and control the traffic (i.e. intelligent routing of vehicles to avoid e.g. traffic jams) as well as steering and navigating the actual vehicles. The most significant variant features in this product line concern a variety of navigation techniques ranging from inductive wires in the factory floor to laser scanners mounted on the vehicles and specializations to each customer installation, such as different vehicles with different loading facilities, and of course different factory layouts.

The variability realization techniques used in this software product line is mainly by using parameterization, e.g. in the form of a database with the layout of the factory floor, which translates to the realization technique "condition on variable" described in Section 5.1.12. For the different navigation techniques, the realization technique used is mainly the "variant architecture component" (Section 5.1.2), which is also aided by the use of an infrastructure-centered architecture (Section 5.1.6).

For further information about NDC Automation AB, see [NDC] and [Svahnberg & Mattsson 2002]. For a further introduction to the domain of automated guided vehicles, see [Feare 2001].

6.6 Rohill Technologies BV

Rohill Technologies BV is a Dutch company that specializes in product and system development for professional mobile communication infrastructure, e.g. radio networks for police and fire departments. One of their major product lines is TetraNode, a product line of trunked mobile radios. In this product line, the products are tailored to each customers' requirements by modifying the soft- and/or hardware architecture. The market for this type of radio systems is divided into a professional market, a medium market and a low-end market. The products for these three markets all use the same product line architecture, designed to support all three market segment. The architecture is then pruned to suit the different product architectures for each of these markets.

Rohill identifies two types of variability: anticipated (domain engineering) and unanticipated (application engineering). It is mainly through the anticipated variability that the product line is adjusted to the three market segments. This is done using license keys that load a certain set of dynamic linked libraries, as described in the variability realization technique "binary replacement - linker directives" (Section 5.1.4). The unanticipated variability is mainly adjustments to specific customers' needs, something which is needed in approximately 20% of all products developed and delivered. The unanticipated variability is solved by introducing new source code files, and instrumenting the linker through makefiles to bind to these product specific variants. This variability is, in fact, using the same realization technique as the anticipated variability, i.e. the binary replacement through linker directives (Section 5.1.4), with the difference that the binding is external as opposed to the internal binding for anticipated variability.

For further information regarding Rohill Technologies BV and their TetraNode product line, see [Jaring & Bosch 2002].

6.7 Cummins Inc.

Cummins Inc. is a USA-based company that develops diesel engines and, for this paper more interestingly, it also develops the control software for these engines. Examples of usages of diesel engines involve automotives, power generation, marine, mining, railroad and agriculture. For these different markets, the types of diesel engines varies in a number of ways. For example, the number of horsepowers, the number of cylinders, the type of fuel system, air handling systems and sensors varies between the different engines. Since 1994, Cummins Inc. develops the control software for the different engine types in a software product line.

Cummins Inc. use several variability realization techniques, ranging from the variant architecture components (Section 5.1.2) to select what components to include for a particular hardware configuration, to #ifdefs, which translates to the realization technique condition on constant (Section 5.1.11), which is used to specify the exact hardware configuration with how many cylinders, displacement, fuel type, etc. that the particular engine type has. The system also provides a large number of user-configurable parameters, which are implemented using the variability realization technique condition on variable (Section 5.1.12).

The company Cummins Inc. and its product line is further described in [Clements & Northrop 2002].

6.8 Control Channel Toolkit

Control Channel Toolkit, or CCT for short, is a software asset base commissioned by the National Reconnaissance Office (in the USA), and built by the Rayethon Company under contract. The asset base that is CCT consists of generalized requirements, domain specifications, a software architecture, a set of reusable software components, test procedures, a development environment definition and a guide for reusing the architecture and components. With the CCT, products are built that command and control satellites, typically one software system per satellite. Development on CCT started in 1997.

The CCT uses an infrastructure-centered architecture (Section 5.1.6), i.e. CORBA, to connect the components in the architecture. Within the components, CCT provides a set of standard mechanisms: dynamic attributes, parameterization, template, function extension (callbacks), inheritance and scripting. Dynamic attributes and parameterization amounts to the variability realization technique condition on variable (Section 5.1.12). Templates are, by the C++ compilers we have had experience with, handled as a condition on constant realization technique (Section 5.1.11). Inheritance is what we refer to as runtime variant component specializations (Section 5.1.9). Scripting is another example of an infrastructure-centered architecture (Section 5.1.6). We have not found sufficient information regarding function extension to identify which variability realization technique this is.

Further information on CCT can be found in [Clements & Northrop 2002].

6.9 Market Maker

Market Maker is a german company that develops products that presents stock market data, and also provides stock market data to users of its applications. Their product line includes a number of functionality packages to manage different aspects of the customers' needs, such as depot management, trend analysis, option strategies. It also consists of a number of products for different customer segments, such as individuals and different TV networks or TV news magazines. In 1999 a project was started to integrate this product line with another product line with similar functionality but with the ability to update and

present stock data continuously, rather than at specified time intervals (six times/day). This new product line, the MERGER product line, is implemented in Java, and also includes salvaged Delphi code from the previous product line.

Market Maker manages variability by having a property file for each customer, that decides which features to enable for the particular customer. This property file translates to the variability realization technique condition on variable (Section 5.1.12). Properties in the property file are used even to decide what parts of the system to start up, by also making use of Java's reflection mechanism in which classes can be instantiated by providing the name of the class as a text string.

For further information about Market Maker and its MERGER product line, see [Clements & Northrop 2002].

7. RELATED WORK

Software Product Lines. In the past few years, there have been a number of publications on how to design and implement software product lines such as, for instance, [Weiss & Lai 1999][Jazayeri et al. 2000][Clements & Northrop 2002]. These and other publications such as [Bass et al. 1997], our co-author's book [Bosch 2000] and conferences such as SPLC 1 [Donohoe 2000] and the upcoming SPLC 2 conference have increased interest in and use of software product lines.

Empirical research such as [Rine & Sonnemann 1996], suggests that a software product line approach stimulates reuse in organizations. In addition, a follow up paper by [Rine & Nada 2000] provides empirical evidence for the hypothesis that organizations get the greatest reuse benefits during the early phases of development. Because of this we believe it is worthwhile for software product line developing companies to invest time and money in performing methods for determining and implementing variability.

In [Bass et al. 1997], the authors define a software product line as *a collection of systems sharing a managed set of features from a common set of core software assets*. This is entirely in line with our view that using feature models is an important way of identifying and managing variability [van Gurp et al. 2001].

A case study presented by [Dikel et al 1997] recommends that a focus on simplification, clarification and minimization is essential for the success of software product line architectures. However they also warn not to over simplify since the architecture needs to be adaptable to future needs. In a case where the architecture was over simplified, the time needed to introduce a new feature tripled. Clearly the use of variation techniques is needed to be adaptable and our taxonomy can help selecting the right techniques so that the architecture can be both adaptable and not be too complex. In addition identifying the need for variation using for example feature diagrams (such as in our earlier work in [van Gurp et al. 2001]). Other methods that may be of use in doing so are the FAST and PASTA methods discussed in [Weiss & Lai 1999] and FODA [Kang et al. 1990].

In [Jazayeri et al. 2000], a number of variability mechanisms are discussed. However it fails to put these mechanisms in a taxonomy like we do. In addition, variability is not linked to features. This is an important characteristic of our approach as it is an important means for early identification (i.e. before architecture design) of variability needs in the future system.

A comprehensive work on software product lines is [Clements & Northrop 2002]. This book presents what a software product line is and is not, the benefits gained by using a product line approach, and a wide range of practice areas, covering aspects in software engineering, technical management and organizational management. This book also presents, in great detail, three cases studies of companies using software product line solutions.

Variability. There appears to be a lot of consensus that domain analysis and feature diagrams in particular are suitable for identifying and documenting variability. FODA [Kang et al. 1990], for instance, introduces a feature diagram notation that includes things like optional, mandatory and alternative features. In [Kang 1998], which discusses the FODA derived FORM method, feature diagrams are identified as a means of identifying commonality between products. Related to FODA is FeatureRSEB [Griss et al. 1998], which extends the use-case modelling of RSEB [Jacobson et al. 1997] with the feature model of FODA. Also related is the FAST method described in [Weiss & Lai 1999] which also includes analyzing variability. The use of such techniques to organize requirements is also recommended in [Clements & Northrop 2002]. This book presents a number of practices and patterns for the development of software product lines.

In [Griss 2000], it is observed that typically changes in a system can be related to individual features or small groups of features. Griss also states that *"Starting from the set of common and variable features needed to support a product-line, we can systematically develop and assemble the reusable elements needed to produce the customized components and frameworks to implement the product"*.

A good overview of domain analysis and engineering methods is provided in [Czarnecki & Eisenecker 2000]. In this book, the authors also include a chapter on feature modeling and the relation of feature models to various generative programming techniques such as inheritance and parametrization. These techniques can be regarded as variability realization techniques as well.

In [Wallnau et al. 2002] methodology for using COTS (Commercial Of The Shelf) components is discussed. The discussion also includes what the authors refer to as *alternative refinements.* These alternative refinements can be seen an instance of our variant architecture component technique.

Variability realization techniques. In [Jacobson et al. 1997], five ways to implement variability are presented, namely: inheritance, extensions, parameterization, configuration and generation. Most of the variability realization techniques we present are based on these implementation techniques. Our contribution is that we explore when it is more suitable to select one technique over another, and what the consequences are of a particular technique. Moreover, we present more than one way in which one can use these implementation techniques.

The two major techniques for variability, as identified in our taxonomy are configuration management and design patterns. Configuration management is dealt with extensively in [Conradi & Westfechtel 1998], presenting the common configuration

management tools of today, with their benefits and drawbacks. Design patterns are discussed in detail in [Gamma et al. 1995] and [Buschman et al. 1996], where many of the most commonly used design patterns are presented.

Configuration management is also identified as a variability realization mechanism in [Bachmann & Bass 2001]. This paper primarily focus on how to model variability in terms of software modules, and is as such a complement to the feature-graphs as discussed above. It does, however, also include a section on how to realize variability in the software product line, which includes techniques such as generators, compilation, adaption during start-up and during runtime, and also configuration management. Our work complement this work by providing further detail on when to introduce variability, when it is possible to add new variants, and when it is possible to bind to a particular variant. We provide a comprehensive taxonomy that brings these things together into the decision of which realization technique to use, rather than just focusing on one of these aspects.

Another technique for variability, seen more and more often these days, is to use some form of infrastructure-centered architecture. Typically these infrastructures involve some form of component platform, e.g. CORBA, COM/DCOM or JavaBeans [Szyperski 1997].

During recent years, code fragment superimposition techniques have received increasing attention. Examples of such techniques are Aspect-, Feature- and Subject-oriented programming. In Aspect-oriented programming, features are weaved into the product code [Kiczalez et al.1997]. These features are in the magnitude of a few lines of source code. Feature-oriented programming extends this concept by weaving together entire classes of additional functionality [Prehofer 1997]. Subject-oriented programming [Kaplan et al. 1996] is concerned with merging classes developed in parallel to achieve a combination of the merged classes.

8. CONCLUSIONS

Variability is not trivial to manage. There are several factors that influence the choice of implementation technique, such as identifying the variant features, when the variant feature is to be bound, by which software entities to implement the variant feature and last but not least how and when to bind the variation points related to a particular variant feature.

Moreover, the job is not done just because the variant feature, including the variants of the variant feature and the corresponding variation points, is implemented. It need to be managed during the product's lifecycle, extended during evolution, and used during different stages of the development cycle. This also constrains the choices of how to implement the variability into the software system.

In this paper we present a minimal set of steps by which to introduce variability into a software product line, and what characteristics distinguish the ways in which one can implement variability. We present how these characteristics are used to constrain the number of possible ways to implement the variability, and what needs to be considered for each of these characteristics.

Once the variability has been constrained, the next step is to select a way in which to implement it into the software system. To this end we provide, in this paper, a taxonomy of available variability realization techniques. This taxonomy presents the intent, motivation, solution, lifecycle, consequences and a brief example for each of the realization techniques.

We believe that the contribution of this taxonomy is to provide a toolbox for software developers when designing and implementing a software system, to assist them in selecting the most appropriate means by which to implement a particular variant feature and its corresponding variation points.

The contribution of this paper is, we believe, that by taking into account the steps outlined in this paper, and considering the characteristics we have identified, a more informed, and hopefully more accurate, decision can be taken with respect to the variability realization techniques chosen to implement the variant features during the construction of a product or a software product line.

References

[Bachmann & Bass 2001] F. Bachmann, L. Bass, "Managing variability in software architectures". *proceedings of the ACM Symposium on Software Reusability: Putting Software Reuse in Context*, pp. 126-132, 2001

[Bass et al. 1997] L. Bass, P. Clements, R. Kazman. *"Software Architecture in Practice"*, Addison-Wesley, 1997.

[Batory & O'Malley] D. Batory, S. O'Malley, "The Design and implementation of Hierarchical Software Systems with Reusable Components", in *ACM Transactions on Software Engineering and Methodology*, Vol. 1, No. 4, October 1992, pp. 355-398.

[Becker et al. 2002] M. Becker, L. Geyer, A. Gilbert, K. Becker, "Comprehensive Variability Modelling to Facilitate Efficient Variability Treatment", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[Bosch 1998] J. Bosch, "Product-Line Architectures in Industry: A Case Study", in *Proceedings of the 21st International Conference on Software Engineering*, November 1998.

[Bosch 1999a] J. Bosch, "Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study", in *Proceedings of the First Working IFIP Conference on Software Architecture*, February 1999.

[Bosch 1999b] J. Bosch, "Superimposition: A Component Adaption Technique", in *Information and Software Technology*, (41)5, pp. 257-273, 1999.

[Bosch 2000] Jan Bosch, *"Design & Use of Software Architectures - Adopting and Evolving a Product Line Approach"*, Addison-Wesley, ISBN 020167494-7, 2000.

[Bosch et al. 2002] J. Bosch. G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl, "Variability issues in Software Product Lines", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[Buschman et al. 1996] F. Buschmann, C. Jäkel, R. Meunier, H. Rohnert, M. Stahl, *"Pattern-Oriented Software Architecture - A System of Patterns"*, John Wiley & Sons, 1996.

[Capilla & Dueñas 2002] R. Capilla, J.C. Dueñas, "Modelling Variability with Features in Distributed Architectures", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[Clements & Northrop 2002] P. Clements, L. Northrop, *"Software Product Lines - Practices and Patterns"*, Addison-Wesley, 2002.

[Conradi & Westfechtel 1998] R. Conradi, B. Westfechtel, "Version Models for Software Configuration Management", in *ACM Computing Survey*, 30(2):232-282.

[Czarnecki & Eisenecker 2000] K. Czarnecki, U. W. Eisenecker, *"Generative Programming - Methods, Tools and Applications"*, Addison-Wesley, 2000.

[Dikel et al 1997] D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, "Applying Software Product-Line Architecture", IEEE Computer, August 1997.

[Donohoe 2000] P. Donohoe, *"Proceedings of the First Software Product Line Conference"* (SPLC1), Kluwer, 2000.

[Feare 2001] T. Feare, "A roller-coaster ride for AGVs", in *Modern Materials Handling* **56**(1):55-63, january 2001.

[Gamma et al. 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Addison-Wesley Publishing Co., Reading MA, 1995.

[Gibson 1997] J.P. Gibson, "Feature Requirements Models: Understanding Interactions", in *Feature Interactions in Telecommunications IV*, IOS Press, 1997.

[Griss et al. 1998] M. L. Griss, J. Favaro, M. d'Alessandro, "Integrating feature modelling with the RSEB", *Proceedings. Fifth International Conference on Software Reuse (Cat. No.98TB100203)*. IEEE Comput. Soc, Los Alamitos, CA, USA, 1998, xiii+388 pp. p.76-85.

[Griss 2000] M.L. Griss, "Implementing Product Line Features with Component Reuse", in *Proceedings of 6th International Conference on Software Reuse*, 2000.

[Jacobson et al. 1997] I. Jacobson, M. Griss, P. Johnson, *"Software Reuse: Architecture, Process and Organization for Business success"*, Addison-Wesley, 1997.

[Jaring & Bosch 2002] M. Jaring, J. Bosch, "Representing Variability in Software Product Lines: A Case Study", to appear in the Second Product Line Conference (SPLC-2), San Diego CA, August 19-22, 2002.

[Jazayeri et al. 2000] M. Jazayeri, A. Ran, F. Van Der Linden, *"Software Architecture for Product Families: Principles and Practice"*, Addison-Wesley, 2000.

[Kang et al. 1990] K. C. Kang, S. G. Cohen, J. A. Hess, W.E. Novak, A.S. Peterson, *"Feature Oriented Domain Analysis (FODA) Feasibility Study"*, Technical report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

[Kang 1998] K. C. Kang, "FORM: a feature-oriented reuse method with domain specific architectures", in *Annals of Software Engineering* volume 5, pp. 345-355, 1998.

[Kaplan et al. 1996] M. Kaplan, H. Ossher, W. Harrisson, V. Kruskal, "Subject-Oriented Design and the Watson Subject Compiler", position paper for OOPSLA'96 Subjectivity Workshop, 1996.

[Kiczalez et al.1997] G. Kiczalez, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, J. Irwin, "Aspect Oriented Programming", in *Proceedings of 11th European Conference on Object-Oriented Programming*, pp. 220-242, Springer Verlag, Berlin Germany, 1997.

[Krueger 2002] C.W. Krueger, "Easing the Transition to Software Mass Customization", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[Lieberman 1986] H. Lieberman, "Using Prototypical Objects to Implement Shared Behavior", in Proceedings on Object-Oriented Programming systems, Languages and Applications, pp. 214-223, 1986.

[Mattsson 2000] M. Mattsson, *"Evolution and Composition of Object-Oriented Frameworks"*, Phd Thesis defended at Blekinge Institute of Technology, Sweden, 2000.

[Mattsson & Bosch 1999a] M. Mattsson, J. Bosch, "Evolution Observations of an Industry Object-Oriented Framework", in *Proceedings International Conference on Software Maintenance*, IEEE Computer Society Press: Los Alamitos CA, pp. 139-145, 1999.

[Mattsson & Bosch 1999b] M. Mattsson, J. Bosch, "Characterizing Stability in Evolving Frameworks", in *Proceedings TOOLS Europe 1999*, IEEE Computer Society Press: Los Alamitos CA, pp. 118-130, 1999.

[Mozilla] Mozilla website, *http://www.mozilla.org/*.

[NDC] NDC Automation AB website, *http://www.ndc.se/*.

[Oeschger 2000] I. Oeschger, "XULNotes: A XUL Bestiality", web page: *http://www.mozilla.org/docs/xul/xulnotes/xulnote_beasts.html*, Last Checked: May 2000.

[Ousterhout 1998] J.K. Ousterhout, "Scripting: Higher Level Programming for the 21st Century", in *IEEE Computer*, May 1998.

[Pigoski 1997] T.M. Pigoski, *"Practical Software Maintenance - Best Practices for Managing Your Software Investment"*, John Wiley & Sons, New York NY, 1997.

[Prehofer 1997] C. Prehofer, "Feature-Oriented Programming: A fresh look at objects", in *Proceedings of ECOOP'97*, Lecture Notes in Computer Science 1241, Springer Verlag, Berlin Germany, 1997.

[Rine & Sonnemann 1996] D. C. Rine, R. M. Sonnemann, "Investments in reusable software. A study of software reuse investment success factors", in *The journal of systems and software*, nr. 41, pp 17-32, Elsevier, 1998.

[Rine & Nada 2000] D. C. Rine, N. Nada, "An empirical study of a software reuse reference model", in *Information and Software Technology*, nr 42, pp. 47-65, Elsevier, 2000.

[Roberts & Johnson 1996] D. Roberts, R.E. Johnson, "Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks", in *Pattern Languages of Programming Design 3*, R. Martin, D. Riehe, F. Buschmann (eds), Addison-Wesley Publishing Co, Reading MA, pp. 471-486, 1996.

[Salicki & Farcet 2002] S. Salicki, N. Farcet, "Expression and usage of the Variability in the Software Product Lines", in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[Svahnberg & Bosch 1999a] M. Svahnberg, J. Bosch, "Evolution in Software Product Lines: Two Cases", in *Journal of Software Maintenance - Research and Practice*, 11(6), pp. 391-422, 1999.

[Svahnberg & Bosch 1999b] M. Svahnberg, J. Bosch, “Characterizing Evolution in Product Line Architectures”, in *Proceedings of the 3rd annual IASTED International Conference on Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 92-97, 1999.

[Svahnberg & Mattsson 2002] M. Svahnberg, M. Mattsson, “Conditions and Restrictions for Product Line Generation Migration”, in *Proceedings of the 4th International Workshop on Product Family Engineering*, F. v.d. Linden (ed), Lecture Notes in Computer Science 2290, Springer Verlag, Germany, 2002.

[Swanson 1976] E.B. Swanson, “The dimensions of Maintenance”, in Proceedings of the 2nd International Conference on Software Engineering, pp. 492-497, IEEE Computer Society Press, Los Alamitos CA, 1976.

[Symbian] Symbian Website, *http://www.symbian.com/*.

[Szyperski 1997] C. Szyperski, *“Component Software - Beyond Object-Oriented Programming”*, Pearson Education Limited, Harlow UK, 1997.

[van Gurp et al. 2001] J. van Gurp, J. Bosch, M. Svahnberg, “On the Notion of Variability in Software Product Lines”, in *Proceedings of WICSA 2001*, August 2001.

[van Gurp & Bosch 2002] J. van Gurp, J. Bosch, “Role-Based Component Engineering”, in *Building Reliable Component-Based Software Systems*, editors: Ivica Crnkovic and Magnus Larsson, Artech House publishers, Norwood MA, to be published in 2002.

[Wallnau et al. 2002] K. Wallnau, S. A. Hissam, R. C. Seacord, *“Building Systems from Commercial Components“*, Addison-Wesley, 2002.

[Weiss & Lai 1999] C. T. R. Lai, D. M. Weiss, *“Software Product-Line Engineering: A FamilyBased Software Development Process”*, Addison-Wesley, 1999.

[Zave & Jackson 1997] P. Zave, M. Jackson, “Four Dark Corners of Requirements Engineering“, *ACM Transactions on Software Engineering and Methodology*, Vol. 6. No. 1, January 1997, p. 1-30.