Blekinge Institute of Technology
Research Report No 02/01

Software Architecture 2001

On the Notion of Variability in Software Product Lines

By
Mikael Svahnberg, Jilles van Gurp,
Jan Bosch

Department of Software Engineering and Computer Science
Blekinge Institute of Technology

On the Notion of Variability in Software Product Lines

Mikael Svahnberg, Jilles van Gurp, Jan Bosch
University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
S-372 25 Ronneby, Sweden
e-mail: [msv|jvg|jbo]@ipd.hk-r.se
www: http://www.ipd.hk-r.se/[msv|jvg|bosch]

Abstract. *Software product lines are used in companies to provide a set of reusable assets for related groups of software products. Generally a software product line provides a common architecture and reusable code for software product developers. In this article we focus on mechanisms that help developers vary the architecture and behavior of a software product line to create individual products. We provide the reader with a framework of terminology and concepts that help understand the concept of variability better. In addition, we present a number of variability mechanisms in the context of this framework. Finally, we provide a method for incorporating variability into software product lines.*

1 Introduction

Over the decades, variability in software assets has become increasingly important in software engineering. Whereas software systems originally were relatively static and it was accepted that any required change would demand, potentially extensive, editing of the existing source code, this is no longer acceptable for contemporary software systems. Instead, although covering a wide variety in suggested solutions, newer approaches to software design share as a common denominator that the point at which design decisions concerning the supported functionality and quality are made is delayed to later stages.

A typical example of delayed design decisions is provided by software product lines. Rather than deciding on what product to build on forehand, in software product lines, a software architecture and set of components is defined and implemented that can be configured to match the requirements of a family of software products. A second example is the emergence of software systems that dynamically can adopt their behaviour at run-time, either by selecting alternatives embedded in the software system or by accepting new code modules during operation, e.g. plug-and-play functionality. These systems are required to contain so-called 'dynamic software architectures' [Oreizy et al. 1999].

The consequence of the developments described above is that whereas earlier decisions concerning the actual functionality provided by the software system were made during requirement specification and had no effect on the software system itself, new software systems are required to employ various variability mechanisms that allow the software architects and engineers to delay the decisions concerning the variants to choose to the point in the development cycle that is optimizes overall business goals. For example, in some cases, this leads to the situation where the decision concerning some variation points is delayed until run-time, resulting in customer- or user-performed configuration of the software system.

Figure 1 illustrates how the variability of a software system is constrained during development. When the development starts, there are no constraints on the system (i.e. any system can be built). During development the number of potential systems decreases until finally at run-time there is exactly one system (i.e. the running and configured system). At each step in the development, design decisions are made. Each decision constrains the number of possible systems. When software product lines are considered, it is beneficial to delay some decisions so that products implemented using the shared product line assets can be varied. We refer to these delayed design decisions as variability points.

1.1 Software Product Lines

The goal of a software product line is to minimize the cost of developing and evolving software products that are part of a product family. A software product line captures commonalities between software products for the prod-

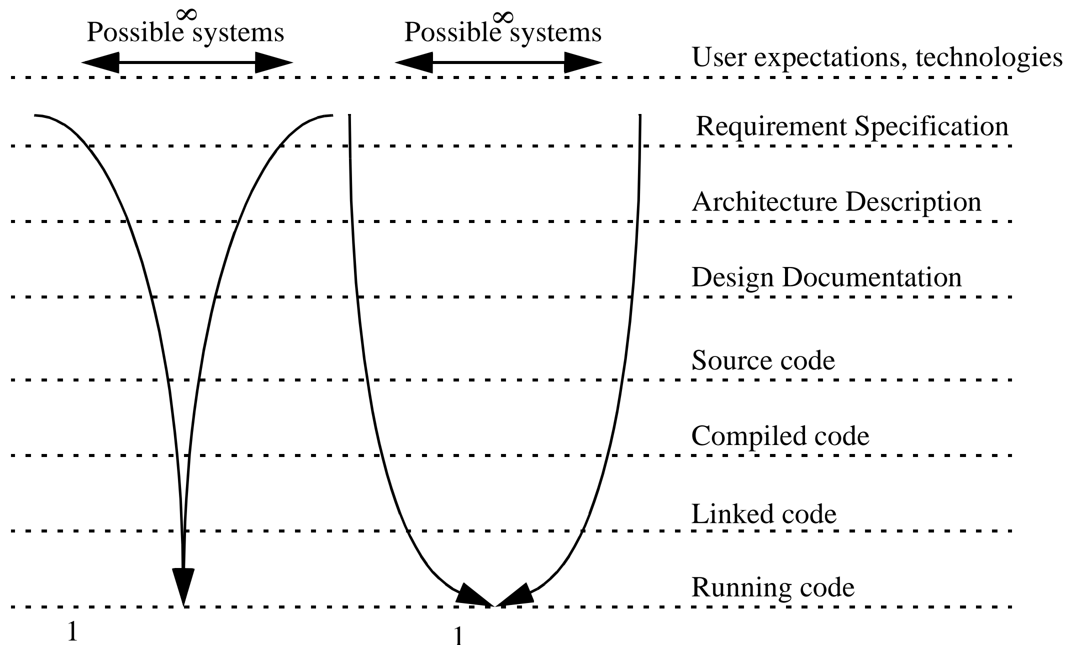

FIGURE 1. The Variability Funnel with early and delayed variability

uct family. By using a software product line, product developers are able to focus on product specific issues rather than issues that are common to all products.

The process of creating a specific software product using a software product line is called product instantiation. Typically there are multiple relatively independent development cycles in companies that use software product lines: one for the software product line itself (often referred to as domain engineering). Software product lines are never finished, rather they evolve during use. And one for each product instantiation.

Instantiating a software product line typically means taking a snapshot of the current software product line and using that as a starting point for developing a product. Basically, there are two steps in the instantiation:

- **Selection.** In this phase the software product line is stripped from all unneeded functionality. Where possible pre-implemented variants are selected for the variability points in the software product line.
- **Extension.** In this phase additional variants are created for the remaining variability points.

From this we can see that there are two conflicting goals for a product line. On one hand a product line has to be flexible in order to allow for diverse product line instantiations. On the other hand a product line has to provide functionality that can be used out of the box in instances.

1.2 Goal of this article

The increased use of variability mechanisms is a trend that has been present in software engineering for a long time, but typically ad-hoc solutions have been proposed and used. To the best of our knowledge, few attempts have been made to organize the existing approaches and mechanisms in a framework or taxonomy, nor suggested design principles for selecting appropriate techniques for achieving variability. The aim and contribution of this paper is to address this problem. In the remainder of this paper, we present a terminology for variability concepts, dimensions and aspects of variability, fundamental mechanisms for achieving variability and instantiations of these mechanisms at different levels, i.e. variability techniques. We present examples from a number of industrial cases in which we have been involved to illustrate the variability techniques that are presented.

The remainder of this article is organized as follows. We introduce features and variability in Section 2 and 3. In Section 4 we introduce our cases. We discuss a number of general patterns that we have found applicable in the development of software product lines in Section 5. In Section 6 we discuss a number of mechanisms based on these patterns. Section 7 discusses guidelines for choosing the right mechanism. We present related work in Section 8 and conclude our paper in Section 9.

2 Features

Products in a product family tend to vary. The differences between the products can be described in terms of features. To better understand variability we need to be able to describe these differences on a high level. We

believe that the feature construct is helpful for making such descriptions. In this section we introduce the concept of a feature and provide a convenient notation for describing systems in terms of features.

2.1 Definition of 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 the book on software product lines, written by co-author of this paper Jan Bosch [Bosch 2000], this definition is specialized for software systems: "*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 a way to abstract from requirements. It is important to realize there is a n-to-n relation between features and requirements. This means that a particular requirement (e.g. a performance requirement) may apply to several features in the feature set and that a particular feature may meet more than one requirement.
To make reasoning about features a little easier, we provide the following categorization:

- **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. 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 from platform agnostic requirements we need external features to link requirements to 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.

The last three categories of features are also listed in [Griss et al. 1998]. The reason we added the category of external features is that we need to be able to reason about the context in which a system operates.

2.2 Feature Interaction

Features are not independent entities [Bosch 2000]. If they were, there would be no good reason to bundle them into a product. When bundling features, the sum of the parts is larger than the individual parts. E.g. the highly controversial browser integration in the windows 98 operating system is more valuable than the individual products (windows 95 and internet explorer 4.0).

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"*. Gibson defines features as *"requirements modules and the units of incrementation as systems evolve"*. During each incremental evolution step of the system, features are added. Because of feature interaction, other, already implemented features may be affected by the changes. As a consequence, some features cannot be considered independently of the system.

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).

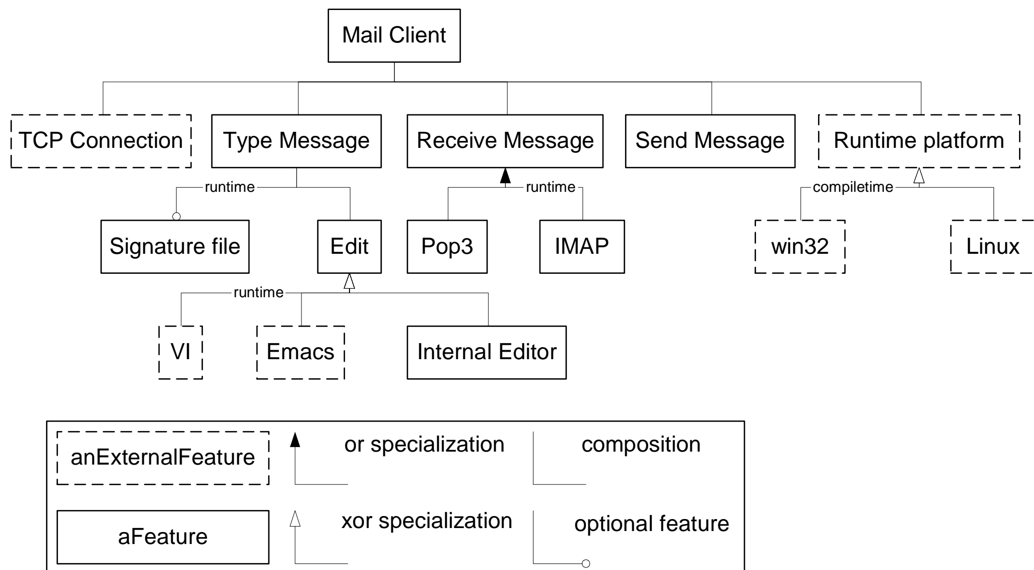

FIGURE 2. Example feature graph

2.3 Notation

The way features interact, can be modelled by specifying the relations between them. In [Griss et al. 1998] a UML based notation is introduced for creating feature graphs. We use an extended notation (see example in Figure 2) that supports the following constructs:

- Composition. This construct is used to group related features.
- Optional feature. This construct is used to indicate that a particular feature is optional.
- Feature specialization (OR and XOR).
- External feature (not in the notation of [Griss et al. 1998]).

Apart from the novel external feature construct, we have added an indication of the moment of binding the variability point to a specific variant (also see Section 3.1). E.g. the mail client supports two run-time platforms (an external feature). The decision as to which platform is going to be used has to be made at compile-time. In the case of the signature file option, the indication is very relevant. Here the developer has the option of either compiling this feature into the product or use a runtime plugin mechanism. The indication runtime on this feature indicates that the latter mechanism should be used.

In Figure 2 we have provided an example of how this notation can be used to model a fictive mail client. Even in this high level description it is clear where variability is needed. We believe a notation like this is useful for recognizing and modelling variability in a system.

3 Variability in Software Product Lines

In this section we introduce the concepts of software product lines and variability in more detail. Related work (e.g. [Griss 2000]) suggests that modelling variability in software product lines is essential for building a flexible architecture. Yet, the concept of variability is generally not defined in great detail. We aim to address this by providing a conceptual framework for reasoning about variability.

3.1 Variability

Variability is the ability to change or customize a system. Improving variability in a system implies making it easier to do certain kinds of changes. It is possible to anticipate some types of variability and construct a system in such a way that it facilitates this type of variability. Unfortunately there always is a certain amount of variability that cannot be anticipated.

Reusability and flexibility have been the driving forces behind the development of such techniques as object orientation, object oriented frameworks and software product lines. Consequently these techniques allow us to delay certain design decisions to a later point in the development. With software product lines, the architecture of a system is fixed early but the details of an actual product implementation are delayed until product implementation. We refer to these delayed design decisions as variability points.

Variability points can be introduced at various levels of abstraction:

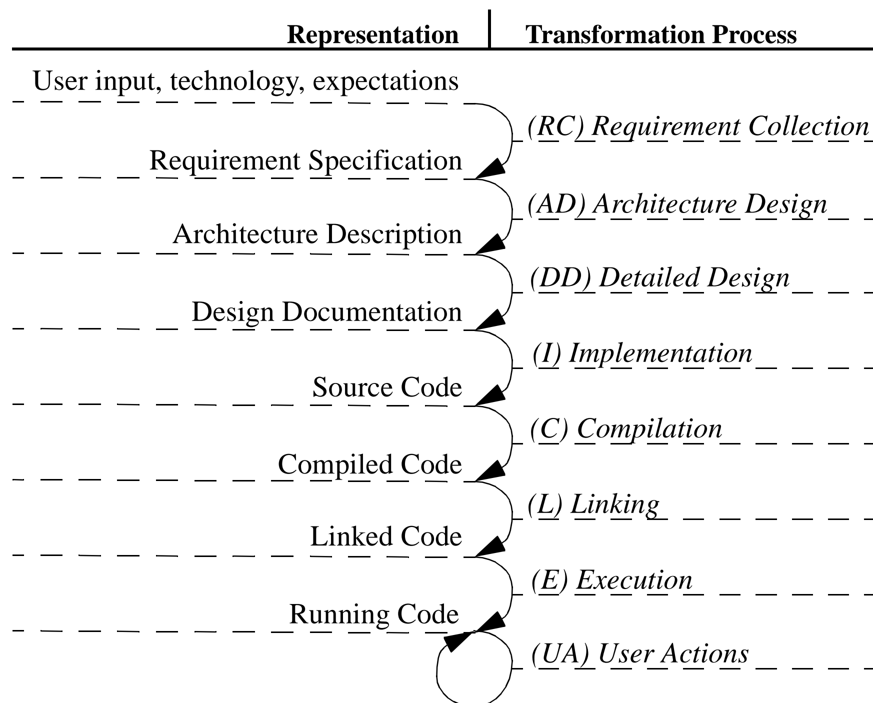

FIGURE 3. Representation & transformation processes

- **Architecture Description.** Typically the system is described using a combination of high-level design documents, architecture description languages and textual documentation.
- **Design Documentation.** At this level the system can be described using the various UML notations. In addition textual documentation is also important.
- **Source Code.** At this level, a complete description in the form of source code is created.
- **Compiled Code.** Source code is converted to compiled code using a compiler. The results of this compilation can be influenced by using pre-processor directives. The result of compilation is a set of machine dependent object files (in the case of C++).
- **Linked Code.** During the linking phase the results of the compilation phase are combined. This can be done statically (at compile time) or dynamically (at run-time).
- **Running Code.** During execution, the linked system is started and configured. Unlike the previous representations, the running system is dynamic and changes all the time.

The various abstraction levels are also linked to different points in the development. However these points in time tend to be technology specific. If for instance an interpreted language is used, run-time applies to compiled, linked and running code whereas in a traditional language like C run-time is associated with running code and linking code (assuming dynamic linking is used). Compilation happens before delivery, in that case. Typically a system is developed using the phases from the waterfall model. When considering variability, some phases of this model are not so relevant (testing, maintenance) while others need to be considered in more detail. In Figure 3 we have outlined the different transformations a system goes through during development. During each of these transformations, variability can be applied on the representation subject to the transformation. Also note that we have two additional levels of representation compared with the ones listed above. However we don't consider these representations concrete enough to consider them when discussing variability points and techniques.

Rather than an iterative process this is a continuing, concurrent process in the case of software product lines (i.e. each of the representations is subject to evolution which triggers new transformations). A software product line does not stop developing until it is obsolete (and is not used for new products anymore). Until that time, new requirements are put on and consequently designed and implemented into the software product line. In a case we observed in a Swedish company, each product was developed with the version of the software product line that was available at that time meaning that it was rare that two products were developed with the same version of the product line. Typically, at the end of a product development cycle, the product line would have changed also (due to new requirements that were applied to both the product and the product line).

If we recall Figure 1, we see that early in the development all possible systems can be built. Each step in the development constrains the set of possible products until finally at run-time there is exactly one system. Variability points help delay this constraint, thus making it possible to have greater variability in the later stages of devel-

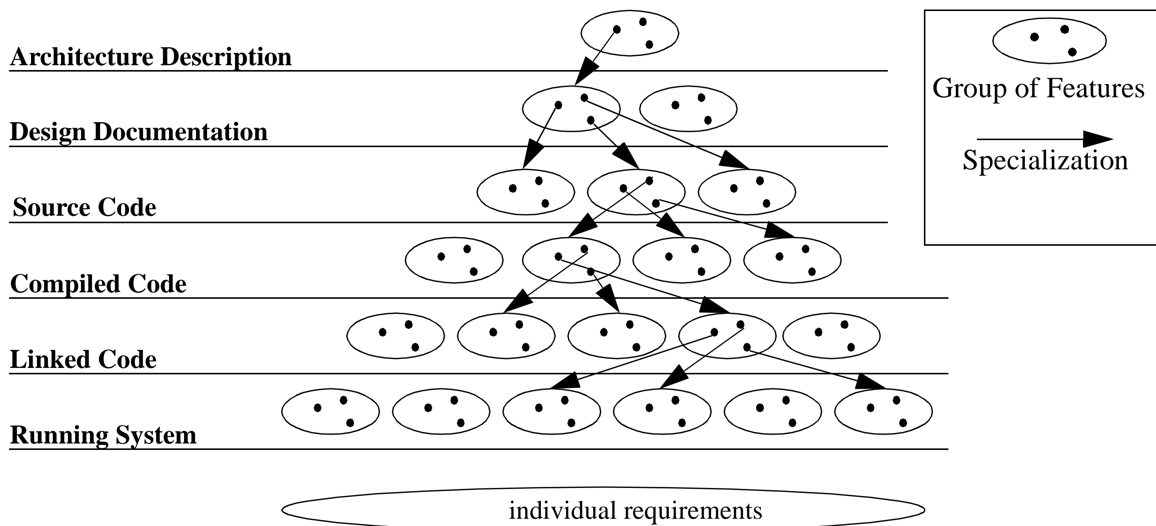

FIGURE 4. The Feature Tree: features on one level decompose into multiple features in lower levels

opment. Variability can be introduced at various levels of abstraction. We distinguish the following three states for a variability point in a system:

- **Implicit.** If variability is introduced at a particular level of abstraction that means that at higher levels of abstraction this variability is also present. We call this implicit variability.
- **Designed.** As soon as the variability point is made explicit it is denoted as designed. Variability points can be designed as early as the architecture design.
- **Bound.** The purpose of designing a variability point is to be able to later bind this variability point to a particular variant. When this happens the variability point is bound.

In addition we use the terms open and closed in relation to the abstraction levels. An open variability point means that it is still possible to add new variants to the system. A closed variability point on the other side means that it is no longer possible to add variants. E.g. if we consider a system where modules conforming to a certain interface can be compiled into the system, the variability is designed into the system during detailed design (where the interface is specified). The variability point is bound at link time when a compiled module is linked to the variability point. Up to the linking phase the variability point is considered to be open (before the detailed design it is implicit however). After the linking phase it is no longer possible to introduce new modules into the system, so the variability point is closed after linking (i.e. in order to introduce new variants the system will have to be linked again).

It is also possible to have a variability point that is closed before it is bound. This means that, for instance, at link time the number of variants is fixed but the variant that is going to be used is not bound until run-time. In the extreme case the variability point is bound when it is designed into the system. I.e. the variants are already known when the variability point is introduced.

3.2 Features and Variability

As we have seen earlier there are different abstraction levels in a software product line: Architecture Description, Design Documents, Source code, Compiled code, Linked code, Running system. These abstraction levels are also applicable for the organization of features. Variability at each abstraction level can be thought of as a change in the corresponding feature set.

In Figure 4 (we left out the top two representations from Figure 3 since they are not very explicit) the relations between features at different abstraction levels is illustrated. At each level there are groups of features (e.g. a feature graph such as in Figure 2).

The general principle is that a single feature at a particular level of abstraction is specialized into a group of less abstract features in the lower level. In the worst case this leads to a feature explosion as in Figure 4. Strictly spoken, the decomposition as presented in Figure 4 is incorrect, since there will always be some overlap in features. The reason for this is feature interaction (also see Section 2.2).

Apart from an abstraction dimension, there also is a time dimension. Over time the feature tree changes and evolves. Features are added, changed or even removed at different abstraction levels. Changes at higher abstraction levels are conceptually easier to understand but are also harder because they generally cause a lot of changes

at lower abstraction levels. Changes at lower levels of abstraction require more knowledge of the system but are also cheaper because there are less side effects.

Another thing that changes over time is the representation of the system. During the development process different representations are used for the system. During architecture design, both ADLs and written text are used to describe the system. During this phase, developers don't worry too much about less abstract things such as algorithms and low-level implementation details. Probably the lower half of the feature tree has not even been established. Later in the development phase, the attention shifts to lower abstraction levels. Since high-level changes are expensive, few things are changed in the more abstract parts of the system.

A software product line can be seen as a partial implementation of a feature tree such as presented in Figure 4. The open spots in the tree can be thought of as variability points where product specific variants can be added. The conceptual model in Figure 4 allows us to reason about a few common problems:

Representation mismatch. During development attention focus shifts from abstract to more concrete things. The representations used to model the abstract part are different from those used later on and consequently there are synchronization problems between the different representations when there are changes. In many organizations the code is the most accurate documentation of the system. All more abstract representations are either out dated or even non-existent. Variability on a more abstract level is still possible (if it was designed into the system) but now requires that the abstract parts of the system are reverse engineered from the code base.

Feature interaction. Feature interaction means that feature changes can have unexpected results on other features in the system. Feature interaction in the model in Figure 4 would mean that two independent features on one abstraction level are specialized into two overlapping sets of features on the abstraction level below. Since it is a very natural thing to do, because of reuse opportunities, this leads to feature interaction for nearly every feature. Therefore features that appear to be conceptually independent on a high level of abstraction are not necessarily independent on lower levels of abstraction.

A related problem to feature interaction is code tangling. Because features interact and therefore depend on each other, it is often difficult to consider feature implementations separately (also see Section 2.2). This is a problem when features need to be changed, removed or added to a system. In the cases we observed it was very common that over time all sorts of dependencies were created between the different modules in the system. We believe that these dependencies are a reflection of the feature interaction problem.

Separation of concern. During the development process, the system is organized into packages, classes and components. This organization helps to separate concerns and thus makes it easier to understand the system. Unfortunately, there is no optimal separation of concerns, which means that some concerns are badly separated in the system. Some features, for instance, involve more than one class (crosscutting feature). Consequently maintenance on such a feature will affect more than one class. Another problem is that the organization is static. This means that it is hard to change the structure of the system in unplanned ways.

The main reason software product lines are used is that they somehow reduce the cost of developing new products in a certain domain. For this to be possible a software product line has to be able to do three things:

- It has to be flexible enough to easily support the diverse products in the software product line domain.
- It has to provide reusable implementation for parts that are the same in each product.
- It has to be able to absorb new features and functionality from individual product implementations if they are found useful for other products.

The before mentioned problems (representation mismatch, feature interaction and separation of concern) need to be addressed to fully ensure that these goals are fulfilled. Existing literature on feature modelling [Griss et al. 1998], suggests that it is not worthwhile to attempt to create complete full feature graphs of a system. Rather they suggest that the modellers focus on modelling the features that are subject to change. This also seems like a good approach for software product lines. By modelling the points in the system where change is needed, the system can be structured in such a way that change is facilitated. This leads to a better separation of concern and helps to avoid feature interaction. The identified spots where changeability is needed, translate to variability points in the system.

4 Cases/Examples

In Section 5 and 6 we present patterns in how variability is solved and the actual mechanisms available for introducing variability. These observations are based on a number of industry cases, and the mechanisms are also exemplified with descriptions from these industry cases. In this section, we briefly present the cases used in this paper. The cases used are:

- The EPOC Operating System
- Axis Communications and their Product Line
- Ericsson Software Technology, and their Billing Gateway product
- The Mozilla Web browser

Of these cases, we have hands-on experience with the first three, and reasonable knowledge of the fourth. The EPOC and Mozilla cases are two relatively new product lines, so there have, as yet, no evolution history. Axis and Ericsson Software Technology, on the other hand, have used a product line approach for nearly a decade.

4.1 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. Variation 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.

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

4.2 Axis Communications

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.

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].

4.3 Billing Gateway

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.

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

4.4 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

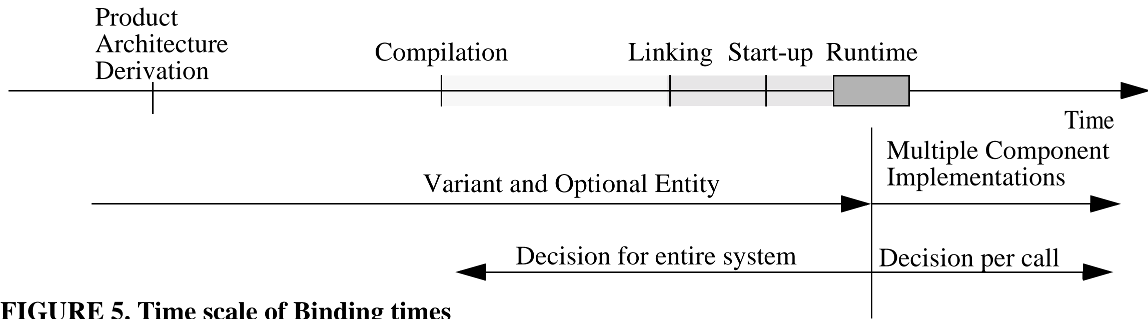

FIGURE 5. Time scale of Binding times

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.

For further information regarding Mozilla, see [Mozilla] and [Oeschger 2000].

5 Variability Patterns

There exist a number of mechanisms to introduce variability into a system. In Section 6, we describe these in further detail. What can be seen is that these mechanisms work on different levels, and strive to achieve variability, i.e. bind the system to one out of many variations, at different times in a system's lifecycle from requirements to runtime. Another, perhaps more important property of these mechanisms is that a few recurring patterns can be seen throughout most of them with respect to how variability is introduced, managed and bound.

Levels. Basically, any entity used in a system can be made to vary. Therefore, we have variation on all phases in a system's lifecycle, from architectural design, to detailed design, implementation, compilation and linking and even at post-delivery. Depending on what entities are in focus on each of these levels, the variation mechanisms work with these different entities. However, many times the actual mechanisms used are very similar.

Binding Times. The main purpose of introducing a variation point is to delay a decision, but at some time there must be a choice between the variants and a single variation will be selected and executed. We call this that the system is bound to a particular variation. However, it is not relevant to bind variants at all levels. The places where one can expect variations to be bound are illustrated in Figure 5, and are further described below. The additional information in the figure is explained in subsequent sections.

Pre-Delivery:

- **Product Architecture Derivation.** The product line architecture contains many open 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 on the level of 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 behaviour (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 relink at will. How long linking is available determines mainly how late new variants can be added to the system.

Post-Delivery:

- **Start-up-time and Runtime (Customisation).** Some decisions must be taken at the customer's site, but can be seen as a delayed step of generating a release binary. These decisions (variations) are thus decided using start-up parameters, often in the form of configuration files that can, for instance, load particular dynamic libraries. These decisions can also be rebound during runtime. How late new variants can be added depends on how advanced the runtime environment is. This binding time is just a special case of linking, with the exception that functionality provided enables linking to be done after delivery.

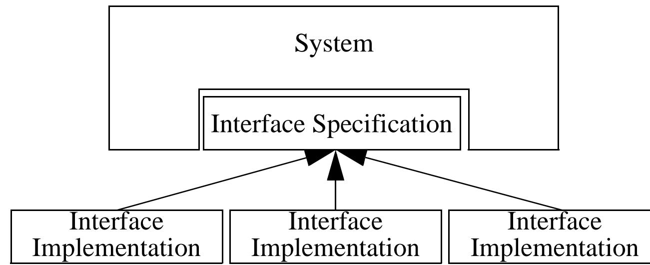

FIGURE 6. Abstraction and Concretization

- **Runtime, Per Call (Adaptation to Runtime Environment).** This is the variability that renders an application interactive. Typically this variability is dealt with using any standard object-oriented language. The set of variations can be closed at runtime, i.e. it is not possible to add new variations, but it can also be open, in which case it is possible to extend the system with new variations at runtime. Typically, these are referred to as Plug-ins, and these can normally be developed by third party vendors.

5.1 Recurring Patterns

The mechanisms presented in Section 6, which, to the best of our knowledge, comprise a complete set, tend to fall into one of the categories below. The entities dealt with by the mechanisms differ (Components, Classes, Code), but the patterns with respect to how and when they are bound are similar.

- **Variant Entity.** Variant entities maps to the XOR-relation in a feature graph, in that there exist many entities, but one, and only one, is active in the system at any given moment.
- **Optional Entity.** An optional entity is in many ways similar to a variant entity, with the exception that there is only one variant available, and the decision is instead whether to include it or not into the system. This maps to optional features in a feature graph.
- **Multiple Coexisting Entities.** The last category consists of the mechanisms where the running system contains several alternate entities, and the decision of which to use is decided at runtime, before each call, or at least for each job, to the entity. This maps to the OR-relation in a feature graph.

How these categories are implemented is also similar on all levels of design and implementation, namely by use of abstraction and concretisation. At one level, an abstract interface is included, and this abstract interface is made concrete in a number of variations at the subsequent level or, as the case often is during detailed design, at the same level. The difference between variant and optional entities as opposed to multiple coexisting entities is then how the rest of the system manages the variation point. Figure 6 illustrates the principle of abstraction and concretisation.

The difference between the patterns lies mainly between variant and optional entities on one side, and multiple coexisting entities on the other. In the variant and optional entity patterns, the management of the variation point is done separate from any calls to the entity, whereas with multiple coexisting entities, the management is done within the frame of one call. Moreover, the decision taken is, in the case of the variant and optional entity patterns, on a per system basis, i.e. the variant chosen is valid for all calls, be they concurrent or not, in the system. With the multiple coexisting entities pattern, the decision is taken on a per call basis, i.e. the decision of which variant to use is taken for each call. Note that we use a very wide definition of “call”. By “call” we mean any form of interaction with an entity to complete a task, which can be anything between a single call, a series of calls or a dialogue.

It should also be noted that even a variation point adhering to the multiple coexisting entities pattern, which is, in many cases, mapped to an OR-branch in a feature graph, will be transformed to an XOR-branch at some time, since the system will at one point bind itself to one, and only one entity. Where this transformation is done is more a matter of philosophical interest; it can be said to happen as the mechanism is implemented, but it can also be said to happen during runtime, as the system gets ready to accept calls.

5.2 Management of Variability

The management of variability consists of two main tasks: (a) collect the variants, and (b) bind the system to one variant. There are a number of sub-tasks involved as well, such as loading the variant chosen into memory, but these tasks are typically programming language or operating system specific.

The collection of variants can either be implicit or explicit. If the collection is implicit, there is no first class representation of the collection, which means that the system relies on the knowledge of the developers or users to

provide a suitable variant when so prompted. An explicit collection, on the other hand, implies that the system can, by itself, decide which variant to use. The collection can be closed, which means that no new variants can be added, or it can remain open. Note that even if the collection is closed, it can also be implicit, which is the case with, for instance, a switch-case statement.

Likewise, binding can be done internally, or externally, from the systems perspective. An internal binding implies that the system contains the functionality to bind to a particular variant, whereas if the binding is performed externally, the system has to rely on other tools, such as configuration management tools to perform the binding. Relating this to the collection, we see that the variability management can either be implicit and external, implicit and internal, or explicit and internal.

Selection of what variant to use involves picking one variant out of the collection of variants. In optional and variant entity, the selection is done by a person, either a programmer or a user that makes a conscious decision about which variant to use. In the case of multiple coexisting entities, the system must possess enough information to select between the variations. The interaction the user in this case provides is, at best, by supplying the system with a particular event for processing.

5.3 Adding new Variants

The time when a mechanism is open or closed for adding new variants is mainly decided by the development and runtime environments, and the type of entity that is represented by the variation point. Typically, mechanisms open for adding variations during detailed design and implementation are closed at compile-time. Mechanisms working with components and component implementations are of a magnitude that makes them interesting to keep open during runtime as well.

An important factor to consider is when linking is performed. If linking can only be done during compilation, before delivery, then this closes all mechanisms at this phase. If the system supports dynamically linked libraries, mechanisms can remain open even during runtime. Then it becomes a question whether the management of the variation point is explicit or not, which decides whether the mechanism will be open during actual runtime, or just at start-up time.

Table 1 presents a comparison between the three patterns with respect to what is discussed in the previous sections.

TABLE 1. Comparison between patterns

Characteristic	Variant Entity	Optional Entity	Multiple Coexisting Entity
Management	Separate from Call	Separate from Call	Performed in Call
Scope of Binding	Valid for Entire System	Valid for Entire System	Valid for one Call
Collection	Implicit or Explicit	Not Applicable	Explicit
Binding	External or Internal	External or Internal	Internal
Open and Closed	Depends on Runtime Environment	Immediately Closed	Depends on Runtime Environment

6 Variability Mechanisms

In Section 5, we present the underlying patterns that are used on all levels of development when variability is to be introduced. In this section, we present the actual mechanisms that can be used during each level of development. We base the layout of this section on how we perceive the development and decision process that leads a developer to choose a particular variability mechanism. In our view, this process starts with the requirements, and the feature graph, where the pattern (variant, optional or multiple coexisting entity) of the variability is identified. The next step is to identify the desired binding time, and lastly, the size of the entity to vary is identified.

It should also be noted that the binding time is a relative time, with respect to the different development phases. If the binding time is, for instance, during Product Architecture Derivation, then the time is fairly straightforward, but if the binding time is at Link-time, this is depending on when linking is performed in the development and runtime environment. In some environments, linking may only be performed before delivery, whereas in other environments, linking can be performed at any time, even in the executing system.

6.1 Variant Entity

The variant and optional entity pattern is not depending on explicit collection and binding, which means that it can be applied to more levels than the multiple coexisting entities pattern. However, because the pattern can make use of explicit representations as well, it also extends into runtime, which results in a very powerful tool for introducing variability. The following are the different mechanisms that can be used to achieve the variant entity pattern, sorted by where the binding takes place. Within parenthesises we present the phases during which the mechanisms are introduced.

Product Architecture Derivation:

- Architecture Reorganization (Architectural Design)
- Variant Architecture Component (Architectural Design)
- Variant Component Specialization (Detailed Design)

Compilation:

- Condition on Constant (Implementation)
- Code Fragment Superimposition (Compilation)

Linking:

- Binary Replacement - Linker Directives (Linking)
- Binary Replacement - Physical (Linking)

Runtime:

- Infrastructure-Centered Architecture (Architectural Design)
- Condition on Variable (Implementation)

Below, we present these mechanisms in further detail, sorted by design phase.

6.1.1 Architectural Design

During architectural design, there are three mechanisms available, of which two are bound during product architecture derivation, and one is bound during runtime. This last mechanism is thus useful to, for instance, implement dynamic architectures. The entities in focus during architectural design are the architecture as such, and the components in the architecture.

6.1.1.1 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 mechanism is an implicit and external mechanism, where there is no first-class representation of the architecture in the system. For an explicit mechanism, see the Infrastructure-Centered Architecture mechanism. In the Architecture Reorganization mechanism, the components are represented as subsystems controlled by configuration management tools or, at best, Architecture Description Languages. The variability lies in the configuration requested by the configuration management tools. The actual architecture is then depending on variability mechanisms on lower levels, for instance the Variant Component Specialization mechanism.

Lifecycle. This mechanism 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 mechanism for achieving dynamic architectures. If this is what is required, see the Infrastructure-Centered Architecture mechanism.

Consequences. The major disadvantage of Architecture Reorganization is that, although there is no first class representation of the architecture on lower levels, they (the lower levels) 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.

6.1.1.2 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 instance, 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 lower layers, where the Variant Component Specialization will be used to call and operate with the different components in the correct way. To summarize, this mechanism has an implicit collection, and the binding functionality is external.

Lifecycle. It is possible to add new variations, 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 variation is situated. Moreover, the handling of the differing interfaces cannot be coped with on the same level as the actual variation, 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.

6.1.1.3 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 this variation, and the actual connections, is 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. Make 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, or it can be an in-house developed standard. The infrastructure may also be a scripting language, in which the connectors are represented as sniplets of code that are responsible for binding the components together in an architecture. These code sniplets 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 [Ousterhout 1998], highly suitable for “gluing” components together. The collection in, in this mechanism, implicit, and the binding functionality is internal, provided by the infrastructure.

Lifecycle. Depending on what infrastructure is selected, the mechanism 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 concretisised during compile and linking, and is thus open for new additions until then. However, since the additions are in the magnitude of architectural components or component implementations, it becomes unpractical to talk about adding new variations during, for instance, the implementation phase, as components are not in focus during this phase. This mechanism can be seen as open for adding new variants during architectural design, and during runtime. If this perspective is taken, it is closed dur-

ing all other phases, because it is not relevant to model this type of variation in any of the intermediate layers. Another view is that the mechanism 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 mechanism binds the system to a particular variant either during compilation time, when the infastructure is tied to the concrete range of components, or at runtime, if the infrastructure supports dynamical adding of new components.

Consequences. Used correctly, this mechanism 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 infastructure, which then performs more processing on a connection, before it is concretisised as a traditional interface call again. In many ways, this mechanism 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.

6.1.2 Detailed Design

During detailed design, there is only one mechanism available, due to the fact that there is only one element in focus, namely classes.

6.1.2.1 Variant Component Specializations

Intent. Adjust a component implementation to the product architecture.

Motivation. Some variation 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 varying component, needs to be replaceable as well. This mechanism 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 mechanism has an implicit collection, and external binding functionality.

Lifecycle. The available variations are introduced during detailed design, when the interface classes are designed. The mechanism is closed during architectural design, which is unfortunate since it is here that it is decided that the mechanism is needed. This mechanism 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. 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, 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 variation mechanism to work with the cache type present in the system.

6.1.3 Implementation

During implementation, the entity available is the source code, and there are two mechanisms by which the source code can be made to vary. The mechanisms are very similar, and differ mostly on the binding time. Typically, they are used by other, higher level variation mechanisms to implement the connections in the system.

6.1.3.1 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 Specialization, where the variation point is not large enough to be a class in its own right. The reason for using the condition on constant mechanism can be for performance reasons, and to help the compiler remove unused code. In the case where the variation 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 mechanism, 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 structure of the system, for instance by opting to include one file over another, whereas the latter can only work within the frame of one system structure. In both cases, the collection is implicit, but, depending on whether traditional constants or pre-processor directives are used, the binding mechanism is either internal or external, respectively.

Lifecycle. This mechanism 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 mechanism 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 variation.

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 mechanisms on different levels are used.

6.1.3.2 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 mechanism 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, i.e. variation management. This mechanism cannot use any compiler directives, but is rather a purely programming language construct. Unlike the Condition on Constant mechanism, the management of the variation point needs to be internal for this mechanism to work. However, the collection need not be explicit, it is sufficient if the binding is internal.

Lifecycle. This mechanism is open during implementation, where new variations 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 mechanism, and can also be used to achieve variability of the multiple coexisting entity pattern. It is a relatively harmless mechanism, but, as with Condition on Constant, if the variation is spread throughout the code, it becomes difficult to get an overview.

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

6.1.4 Compilation

During compilation the source code is transformed into an executable binary. This is normally not an interactive process, which limits the number of mechanisms available. One new mechanism is, however, introduced.

6.1.4.1 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 behaviour can be introduced using practically any mechanism, 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 mechanism to introduce variations 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 in a generic way, 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, e.g. 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 behaviour is wrapped around existing behaviour. The collection is, in this case, also implicit, and the binding is performed external of the system.

Lifecycle. This mechanism is open during the compilation phase, where the system is also bound to a particular variation. However, the superimposition can also provide support for 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 mechanisms, 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 mechanism. This is not surprising, considering that most techniques for this mechanism are at a research and prototyping stage.

6.1.5 Linking

During linking, what can be made varying, is the files that are included in the system. As mentioned earlier, the duration of the linking phase is very much depending on the runtime environment. It can end directly after compilation, and it can also extend until start-up-time, or even until runtime. There are two mechanisms that concern linking, and these differ mainly in whether the management is internal or external.

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

Consequences. This is a fairly well developed variation mechanism, and the consequences of using it are relatively harmless.

Examples. The web browsing component of Internet Explorer can be replaced with the web browsing component of Mozilla in this fashion.

6.1.5.2 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

mechanisms at variation points already existing in the system, but in others, the system does not currently support variation at the place 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 instance, a new variation point using a pre-delivery variability mechanism. Also in this mechanism, is the collection implicit, and the binding external to the system.

Lifecycle. This mechanism is bound after delivery, normally before start-up of the system. In this mechanism the method for binding to a variation is also the one used to add new variations. After delivery, the mechanism is always open for adding new variants.

Consequences. If the new binary does not introduce a "traditional", first class, variation point the same mechanism will have to be used again the next time a variation at this point is detected. However, if a traditional variation point is 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.

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.

6.2 Optional Entity

The mechanisms concerning optional entities are very similar to those concerning variant entities. One reason for this is that an optional entity is in many cases just a special case of variant entity, where one of the variants is empty. A characteristic of mechanisms of the optional entity pattern is that they are closed as soon as they are introduced, since there can be only one variation to choose. The similarities between optional and variant mechanisms make it possible to combine them while implementing the solutions. However, a huge difference lies in the fact that whereas a call to a variant entity is always direct, a call to an optional entity needs to make sure that there actually is something to call. This makes it possible to implement an optional entity mechanism in two ways: Either on the calling side, to simply remove the call, or on the called side, by ignoring the call.

The mechanisms available, adhering to the optional entity pattern, are (sorted by binding time):

Product Architecture Derivation:

- Optional Architecture Component (Architectural Design)
- Optional Component Specialization (Detailed Design)

Compilation:

- Condition on Constant (Implementation)

Runtime:

- Condition on Variable (Implementation)

Below, we present these in further detail. The mechanisms Condition on Constant and Condition on Variable are exactly the same as for the variable entity pattern, because of which we do not present them again. See Section 6.1.3.1 and Section 6.1.3.2 for further information about these patterns.

6.2.1 Architectural Design

During architectural design, one mechanism is available for introducing an optional entity into the system.

6.2.1.1 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 instance, 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 downwards to other optional entity mechanisms. 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 collection is in an optional entity pattern not relevant, which, in general, results in it being implicit. The binding for this mechanism is done external to the system.

Lifecycle. This mechanism is open when a particular product architecture is designed based on the product line architecture, but, for the lack of architecture representation in later stages, 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 mechanism is that the components depending on the optional component must either have mechanisms to support its not being there, or have mechanisms 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 a variant architecture component point, the functionality is already in place, and all that needs to be done is to open the collection of variations in order to add more variants.

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.

6.2.2 Detailed Design

Detailed design introduces an additional mechanism for adding an optional entity variation point, as described below.

6.2.2.1 Optional Component Specializations

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

Motivation. A particular component implementation may be customized in various ways by adding or removing parts of its behaviour. For instance, 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 behaviour into a separate class, and create a "null" class that can act as a placeholder when the behaviour is to be excluded. Let the configuration management tools decide which of these two classes to include in the system. Alternatively, surround the optional behaviour with compile-time flags to exclude it from the compiled binary. Binding is in this mechanism done externally.

Lifecycle. This mechanism 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.

Consequences. It may not be easy to separate the optional behaviour into a separate class. The behaviour 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.

6.3 Multiple Coexisting Entities

Mechanisms adhering to the multiple coexisting entities pattern occur slightly later during the lifecycle of the system, since these mechanisms are concerned with how to respond to a particular call, which means that the system naturally must be executing as the call occurs. It also implies that these mechanisms require a first class representation in the system, even if they need not be explicit per se.

The mechanisms available are:

Runtime:

- Multiple Coexisting Component Implementations (Detailed Design)
- Multiple Coexisting Component Specializations (Detailed Design)
- Condition on Variable (Implementation)

Below, these mechanisms are presented in further detail.

6.3.1 Detailed Design

During detailed design, two mechanisms are introduced. These do, in fact, address two different levels, one is focused on the interface of the component implementations, and how to vary entire component implementations, whereas the second is focussed on variation inside one component implementation.

6.3.1.1 Multiple Coexisting 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 instance, 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 instance, 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 is patterns like Abstract Factory and Builder. Part of the flexibility of this pattern 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 (i.e. setting up the collection of implementations) is delegated to configuration management tools.

Lifecycle. This mechanism is introduced during architectural design, but is not open for addition of new variant until detailed design. It is not available during any other phases. Binding time of this mechanism is at runtime. 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 mechanism work equally well when all variants are already compiled into the system. However, if the system does support dynamic linking, the mechanism is in fact open for adding new variations even during runtime, provided that the collection of variants is made explicit.

Consequences. Consequences of using this mechanism 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 use this mechanism to, for instance, select between different network communication standards. Ericsson Software Technology use this mechanism 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 an 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.

6.3.2 Multiple Coexisting 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 that are applicable here: Strategy and Template Method. Alternating behaviour is collected into separate classes, and mechanisms are introduced to, at runtime, select between these classes. Using Design Patterns makes the collection implicit, but the binding is still internal to the system.

Lifecycle. This mechanism 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 mechanism 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 behaviour 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.

Examples. A handheld 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 instance, compression differ, but on a lower bandwidth, the system can also decide to retrieve less data, thus reducing the network traffic. This variation need not be in the magnitude of an entire component, but can often be represented as strategies within the concerned components.

6.3.3 Implementation

During the implementation phase, there is one mechanism available, which is also available if the variant or optional entity patterns are used.

6.3.3.1 Condition on Variable

This mechanism is, in all essentiality the same as for the variant entity pattern, and is presented in detail in Section 6.1.3.2. The only difference is that imposed of the pattern itself, in that if the mechanism is used to achieve a variant entity, the current selection is valid for the entire system, whereas for the multiple coexisting entities pattern, the selection is only valid for one call, which also implies that there may be several instances of the same code executing in parallel. If the mechanism is used for the variant entity pattern, the decision of which variant to use is also a more conscious choice of the user than if it is used for a multiple coexisting entities situation.

7 Planning Variability

When developing a software product line, the ultimate goal is to make it flexible enough to meet new requirements the forthcoming years. In our experience, the important variability points need to be anticipated in advance in order to achieve this. It turns out that it is often very hard to adapt an existing architecture to support a certain variability point. In this section we propose a method for identifying and planning variability points.

7.1 Identification of Variability

Object Oriented software development tends to iterate over the well-known phases of the waterfall model:

- Requirement Specification
- Architecture Design
- Detailed Design
- Implementation
- Testing
- Maintenance

It can be argued that developing a software product line follows the same development cycle. It should be noted though that a software product line is never really finished. Rather, stable versions of it are used for product instantiation. Since the software product line itself is continuously under development, it is unlikely that the same version of the software product line will be used twice. Laying out the product line architecture in the right way is important since after the initial development phase it becomes increasingly hard to drastically change the product line.

In the initial phase of software product line development, developers are confronted with requirements for a number of products and requirements that are likely to be incorporated into future products. Their job is to somehow unite these requirements into requirement specification for the software product line. The aim of this process is not to come up with a complete specification of the software product line but rather to identify where the products differ (i.e. what things tend to vary) and what is shared by all products.

The feature graph notation we discussed in Section 2 may help developers to abstract from the requirements. By uniting the feature graphs of the different products a feature graph for the software product line can be constructed. In this merged model all the important features and variability points are present. We have found that

features and feature graphs are an excellent way of modelling variability since features are a basic increment of development (i.e. a change in the system can be expressed in terms of features added/removed/enhanced).

7.2 Planning Variability

Once the variability points have been identified, they need to be planned. This means that developers need to consider how to manage the variability points. To do so, we can use the patterns and mechanisms discussed in Section 5 and 6.

Planning a variability point involves:

- Choosing closing and binding time and a variability pattern.
- Picking a mechanism for implementing the variability point.
- Selecting a technology

It is important that developers find a balance between flexibility and cost. If too much flexibility is incorporated into the architecture, the cost may rise. The desire to create the ultimate, flexible architecture has caused numerous OO projects to fail. If on the other hand too little flexibility is incorporated, developers may find themselves in a situation where their software product line is no longer able to deal with new requirements in a cost effective way.

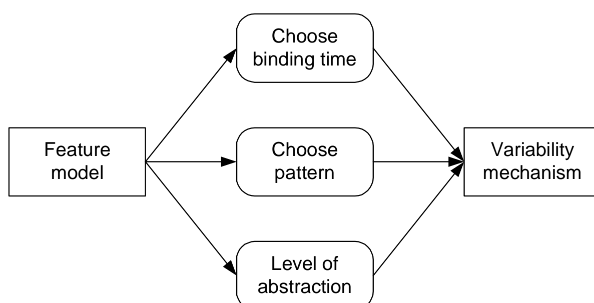

FIGURE 7. The variability mechanism selection method

In Figure 7, a method of selecting a suitable variability mechanism is illustrated. Based on the feature graph, a time for binding the variants is chosen, one of the three variability patterns is selected and a level of abstraction at which the variability is introduced is chosen. These three choices can be used to find a suitable mechanism in Section 6.

8 Related Work

Software Product Lines. Our work was largely inspired by earlier work in our research group. Our co-author Jan Bosch published a book about designing and using software product lines [Bosch 2000]. This book was largely based on case studies and experience reports such as [Bosch 1998][Bosch 1999a][Svahnberg & Bosch 1999a][Svahnberg & Bosch 1999b]. From these reports we learned that evolution in software product lines is a little more complicated than in standalone products because of dependencies between the various products and because of the fact that there may be conflicting requirements between the different products.

Empirical research such as [Rine & Sonnemann 1996], suggests that a software product line approach stimulate 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 such as in Section 7.

Variability Patterns. The mechanisms we present in Section 6 are presented in similar fashion as the patterns in [Gamma et al. 1995] and [Buschman et al. 1996]. Pree elaborated on this work by extracting a set of meta-patterns [Pree 1995]. In a similar way we tried to abstract from the variability mechanisms we found. In Section 5 we list three recurring patterns of variability, which appear to be applicable throughout the development process. They can also be related to the feature graph constructs discussed in Section 2. The three patterns we have found can even be abstracted further to one meta-pattern: specialization.

We were not the first to look for variability patterns. In [Keepence & Mannion 1999], patterns are used to model variability in product families. Unlike us, they limit themselves to the detailed design phase. Instead we try to cover the whole development process, thus gaining the advantage of discovering variability points earlier (as pointed out above). Also by limiting themselves to detailed design they miss many important variability mechanisms such as identified in Section 6.

In [Van Gurp & Bosch 2000], a number of guidelines are presented for building flexible object oriented frameworks. These guidelines bear some resemblance to the variability mechanisms presented in Section 6. Related to this is the work presented in [Van Gurp & Bosch 1999] where a framework for creating finite state machine implementations is discussed. Several mechanisms are used in this framework to achieve variability.

Requirements. Our argument for introducing the external feature in Section 2 is based on [Zave & Jackson 1997]. They argue that a requirement specification should contain nothing but information about the environment. The rationale behind this is that a requirement specification should not be biased by implementation. Since features are an interpretation of the requirements, there is a need to map implementation independent requirements to implementation aware features.

Feature Modelling. Our extended feature graph is based on the work presented in [Griss et al. 1998]. The main difference, aside from graphical differences, between our notation and theirs is the external feature and the addition of binding time. In [Griss 2000] the feature graph notation is used as an important asset in a method for implementing software product lines. Unlike their work we link feature graphs to a set of concrete mechanisms (see Section 6).

Also related is the FODA method discussed in [Kang et al. 1990]. In this domain analysis method, feature graphs play an important role. The FORM method presented in [Kang 1998] can be seen as an elaboration of this method. In this work feature graphs are recognized as a tool for identifying commonality between products. We take the point of view that it is more important to identify the things that vary between architectures than to identify the things that are the same since the goal of developing a software product line is to be able to change the resulting system. The FORM method uses four layers to classify features (capability, operating environment, domain technology and implementation technique). We use a more fine-grained layering by using the different representations (architectural design, detailed design, source code, compiled code, linked code and running system) as abstractions. The advantage of this is that we can the relate variability points to different moments in the development. We consider this to be one of the contributions of our paper.

Our hierarchical feature graph bears some resemblance to the integral hierarchical and diversity model presented in [Van de Hamer et al. 1998]. Unlike their model, we use variation points to model variability. The notion of variation points was first introduced in [Jacobson et al. 1997]. The model uses a similar layering as can be found in [Batory & O'Malley]. In this paper, three distinct granularities of reuse are identified (component, class and algorithm) that correspond to our architecture design, detailed design and implementation levels.

Feature interaction. Feature interaction can be modelled in a feature graph as dependencies between different features [Griss 2000]. Since features can be seen as incremental units of development [Gibson 1997], dependencies make it impossible to link individual features to a single component or class. As a consequence, source code of large systems such as software product lines tends to be tangled. Features that are associated with a lot of other features are called crosscutting features. Variability in such features is very hard to implement and often requires that a system is designed using for example design patterns [Griss 2000].

Methodology. Our method, outlined in Section 7, was inspired by the method outlined in the software product lines book written by our co-author [Bosch 2000]. Rather than replacing it, our aim is to refine the initial steps of this method. Our method also bears some resemblance to the architecture development method outlined in [Kruchten 1995]. The first steps in this method are to select a few cases to find major abstractions. Our method of creating a feature graph based on a number of cases in order to find variability points, can be seen as a refinement of these steps.

Another method that is related to ours is the FAST (Family-Oriented Abstraction, Specification and Translation) method that is discussed in [Coplien et al. 1999]. This empirically tested method uses the SCV (Scope, Commonality and variability) analysis method to identify and document commonality and variability in a system. The result of this analysis is a textual document. A notation modelling variability in terms of features, such as provided in this paper, is not used in their work. An important lesson learned in this paper is that variability points should be bound early in order to save on development cost.

9 Conclusions

In this paper we study the phenomenon of variability in software product lines. We do this by establishing where variability enters the production of a software system, what forms variability can take, and what mechanisms are available to implement variability into a software system, and a software product line in particular.

As is presented, variability is introduced early in the development process, in the form of features. Knowing that the term "feature" has lately become an overly loaded term, we give our definition of features, to alleviate further

reading. We also give our definition of software product lines, variability, and how features and feature graphs are used to model variability and interaction between different software entities.

Based on four cases, presented in Section 4, we discern three major patterns regarding how variability is implemented. Conveniently, these three patterns map fully to the variability forms modelled in feature graphs. Besides these three patterns we present other important characteristics, namely where the management of the variation point is done, when the variation point is bound to a unique variant, and how long the variation point is open for adding new variants. We then present the actual mechanisms available for introducing the variability recognized in feature graphs, presenting for each mechanism how it adheres to the three overall patterns, and giving examples of usage from the four cases.

We summarize by presenting a process by which to plan and introduce variability into a software product line, and some guidelines regarding variability in general.

9.1 Contributions

There are a number of contributions in this paper. We provide an extensive definition of the terms feature and variability in relation to software product lines. In addition, we introduced the notion of variability binding time. To the best of our knowledge this has not been introduced before. The work presented in Section 2, 3 and 5 allowed us to create a taxonomy of variability mechanisms. The mechanisms are organized into three dimensions:

- Abstraction Level.
- Variability Pattern.
- Binding time.

This provides us with an intuitive way to find the appropriate way of translating variability requirements into implementation. And finally we provide guidelines for finding the right variability mechanism.

9.2 Future Work

Future work involves investigating further how to design new systems, possibly into an existing software product line, to support the currently required and future planned variability. We also intend to investigate means by which to introduce new variation points into an already existing software product line, to support evolving requirements. Another viable path, which we intend to investigate, is to move from the mechanisms presented in Section 6 to new programming paradigms, similar to what, for instance, Aspect Oriented Programming and Subject Oriented Programming has already done.

10 Acknowledgements

We would like to thank Axis Communications in Lund, and Ericsson Software and Symbian in Ronneby for their involvement in our research and for providing access to their software products. In addition we would like to thank the developers of the Mozilla project for providing an up to date vision on how to implement variability in software product lines.

11 References

[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.

[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.

[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.

[Coplien et al. 1999] J. Coplien, D. Hoffman, D. Weiss, “Commonality and variability in software engineering“, *IEEE Software*, November/December 1999, pp. 37-45.

[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, Montreal, Canada, June 1997, IOS Press.

[Griss et al. 1998] M. L. Griss, J. Favaro, M. d'Alessandro, “Integrating feature modeling 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“, to appear in *Proceedings of 6th International Conference on Software Reuse*, Vienna, Austria, June 2000

[Van Gurp & Bosch 2000] J. van Gurp, J. Bosch, “Design, implementation and evolution of object oriented frameworks: concepts & guidelines“, technical paper, accepted with minor revisions in the journal *Software-Parctice and Experience*, April 2000.

[Van Gurp & Bosch 1999] J. van Gurp, J. Bosch, “On the Implementation of Finite State Machines“, in *Proceedings of the 3rd Annual IASTED International Conference Software Engineering and Applications*, IASTED/Acta Press, Anaheim, CA, pp. 172-178, 1999.

[Van de Hamer et al. 1998] P. van de Hamer, F.J. van der Linden, A. saunders, H. te Sligte, “N Integral Hierarchy and Diversity Model for Describing Product Family architecture“, in *Proceedings of the 2nd ARES Workshop: Development and evolution of Software Architectures for Product Families*, Springer Verlag, Berlin Germany, 1998.

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

[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, Carnegy Mellon University, Pittsburgh, PA.

[Kang 1998] K.C. Kang, “FORM: a feature-oriented reuse method withdomain-specific architectures“, in *Annals of Software Engineering*, V5, pp. 354-355.

[Keepence & Mannion 1999] B. Keepence, M. Mannion, “Using Patterns to Model Variability in Product Families“,in *IEEE Software*, July/August 1999, pp 102-108.

[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.

[Kruchten 1995] P.B. Kruchten, “The 4+1 View Model of Architecture“, in *IEEE Software*, November 1995, pp. 42-50.

[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/.*

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

[Oreizy et al. 1999] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, “Self-Adaptive Software: An Architecture-based Approach”, in *IEEE Intelligent Systems*, 1999.

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

[Pree 1995] W. Pree, *“Design Patterns for Object-Oriented Software Development“*, Addisson-Wesley, 1995, ISBN 0-201-42294-8.

[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.

[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.

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

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

On the Notion of Variability in Software Product Lines

By: Mikael Svahnberg, Jilles van Gurp, Jan Bosch

ISSN 1103-1581
ISRN BTH-RES--02/01--SE

Copyright © 2001 by the authors
All rights reserved
Printed by Kaserntryckeriet AB, Karlskrona 2001