Master Thesis

Evolution of Variability in Software Product Families

Sybren Deelstra
March 2003

Supervisors:
- *Prof. P. Sorenson*
- *Prof. dr. ir. J. Bosch*

Preface

The main goal of software engineering has been, and continuous to be, solving the cost, time and quality issues associated with software development. Since the existence of software engineering in the late 1960s, reuse through componentization has been the generally accepted approach to address these goals. Over the last 40 years, several techniques have been proposed that focused on the increase in the scale of reuse. However, primarily during the last decade, it was realized that with the increase in scale, variability was needed to increase the applicability of the components. Therefore, proper variability management is regarded as a key success factor in addressing the main goals of software engineering. In this thesis, we focus on variability management in one of the approaches to reuse, i.e. software product families. In particular, we address the topic of evolution of software product families in relation to variability management. We present a framework of concepts regarding evolution of variability, and discuss a variability assessment technique for architecture improvement.

Keywords: Software product families, variability, product derivation, evolution, variability mismatch, variability assessment.

Acknowledgements

I would like to express my sincerest gratitude towards prof. Paul Sorenson for making my wonderful stay in Canada possible. In addition, I would like to thank prof. Jan Bosch as my internal supervisor. Last, but certainly not least, I would like to thank my girlfriend Nadia Mensink for her patience and support, especially during the last few months.

Table of Contents

Chapter 1 – Introduction

This chapter presents the motivation for our research, as well as the outline of this thesis.

1.1. Motivation

Over the past 40 years, reuse of software elements has evolved from small-grained mathematical routines to large-scale reuse through techniques such as Object-Oriented frameworks or software product families. Each of the adopted reuse approaches has had the common goal of solving the cost, quality and time-to-market issues associated with software development. One of the latest additions, software product families, focuses on intra-organizational reuse through the explicitly planned exploitation of similarities between related products. This approach has proven itself in quite a large number of industrial organizations.

In a product family context, software products are developed in a two-stage process, i.e. a domain engineering stage and a concurrently running application engineering stage. Domain engineering involves, amongst others, identifying commonalities and differences between product family members and implementing a set of shared software artifacts in such a way that the commonalities can be exploited economically, while at the same time the ability to vary the products is preserved. During application engineering individual products are derived from the product family, viz. constructed using a subset of the shared software artifacts.

The ability to derive various products is often referred to as the variability of the software product family. Variability as research topic has grown substantially over the past few years, as the software engineering community started to realize the difficulties of reuse are not associated with capturing the commonalities between various products. Instead, preserving the ability to handle the differences is identified as a major issue. Managing variability properly is therefore considered as the key factor in the success of software product families.

Traditionally, variability management has been associated with two main tasks, i.e. *facilitating differences* and *exploiting variability*. For the sake of simplicity and brevity in this introduction, facilitating differences is mainly a concern during domain engineering, while during product derivation, i.e. the derivation of a product during application engineering, the variability of the software product family is exploited to obtain a unique product. In this thesis, we urge that these two main tasks alone are not enough. We identify evolution as an important cause of problems many organizations face during product derivation and recognize active *variability evolution* as an equally important variability management task.

Literature does address topics such as the creation and instantiation of generic and specific product family assets as well as dealing with variability at the code level [Anastasopoulos 2001]. Although exceptions exist, e.g. [Svahnberg 2000], evolving variability has received, to the best of our knowledge, little attention however. The work presented here, therefore aims to investigate the consequences of evolution on variability and the actions that can be taken to correct and prevent these consequences from occurring during product derivation.

The main contribution of this thesis, we believe, is that we present a framework of concepts regarding evolution of variability, as well as a variability assessment technique for architecture improvement.

1.2. Remainder of this Thesis

The remainder of this thesis is organized as follows. In chapter 2, we provide a short overview of how reuse of software components evolved since the 1960s. Chapter 3 presents background information and related work with respect to variability. In chapter 4, we discuss a case study we performed at the University of Alberta. In chapter 5, we discuss concepts with respect to evolution of variability and detail our problem statement. We present a technique that enables an organization to proactively deal with the consequences of evolution on variability, i.e. variability assessment, in chapter 6. In chapter 7, we conclude our findings.

Chapter 2 – Reuse

In the late 60s of the previous century, the software engineering community faced a major crisis. The languages in which computer programs were written (machine specific string or binary instructions) had evolved to higher level, platform independent languages (e.g. Fortran (1957) and COBOL (1960) [IEEE 2002]). This enabled developers to address problems with higher complexity, but also increased the complexity and costs of development itself. At the same time, the market required shorter time-to-market, demanding more and more products, which should adhere to a higher quality standard, and have a lower price label attached [Jacobson 1997]. (We refer to these phenomena as the bigger-better-cheaper-faster market principle).

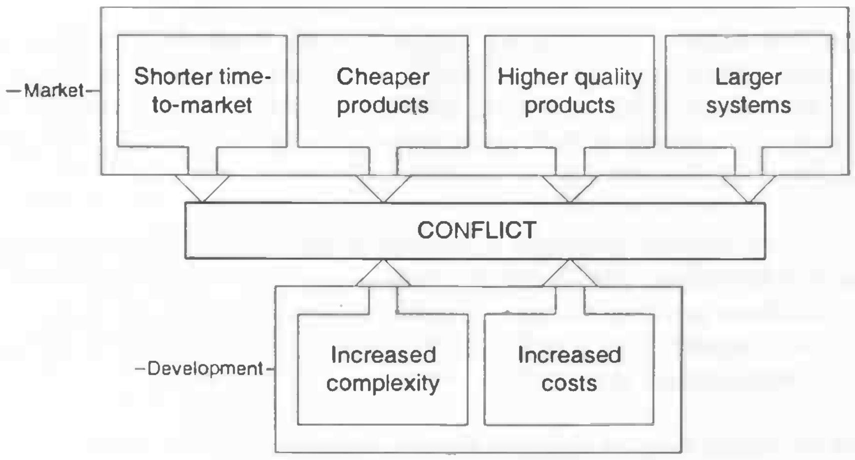

Figure 2-1 – The Software Engineering Crisis

The existing software engineering practice did not sufficiently address the needs as illustrated above (Figure 2-1). As a result, only a small percentage of software development projects were completed on time and within budget. In addition, software products became associated with a form of reliability that would not be acceptable in other engineering disciplines and up to 80% of the total system costs were spent on maintenance.

One of the events that sparked the change in the world of software engineering, was the introduction of Doug McIllroy's view on reuse and software 'components' in [McIllroy 1969]. His ideas on utility libraries and mathematical routines evolved into the conceptual reuse of software elements as a solution for the problems outlined above. Over the past decennia, this concept didn't change, although market demands increased more and more and the crisis is not completely over yet (as exemplified in Example 2.1). What elements are reused however and how they are reused changed considerably. In the next section, we give an overview of how reuse strategies evolved over the years.

Although humankind generally tends to prefer to reflect its success stories, documentation [Flowers 1997] on failed projects does exist. One example is the PROMS project in the late 1980s, which cost $12M, was 10 weeks late on delivery and eventually was suspended as the database system had a response time of over 30 minutes. Other examples include a project from the California Department of Motor Vehicles, which spent over $44M but didn't get a single piece of useful software in return and the CONFIRM project (1988-1994), which required 18 months of additional work as the two main systems didn't work together on beta-testing. For a more detailed overview of these and other examples, see [Flowers 1997]

Example 2.1 – Disaster projects

2.1. *Reuse Strategy Evolution*

To pinpoint when reuse actually began is rather difficult. Reuse of course, was not something that didn't exist before McIllroy wrote about it 1968. In fact, although often creditted the source of the problems in the late 1960s software practice, even code scavenging, or the "cut & paste-strategy", is reuse of software elements. Also, compilers (or high-level languages for that matter) are a form of conceptual reuse; it prevents typing similar pieces of code over and over again by abstraction. However, reuse of software through componentization is a turning point in the history of software engineering and thus provides a good starting point for a historical overview.

2.1.1. Functions and Modules

A large issue associated with reusing pieces of code was the mismatch between the namespace from which the original fragment originated and the namespace of the new application. By separating (at least partially) the namespaces of procedures or functions, imperative languages and procedural based design provided the first step towards component-based, black-box reuse in the 1960s.

In the 1970s, the high-level languages developed further into modular languages. A module basically groups procedures and functions to one instance of a single black-box with its own namespace. Modules as utility libraries, or toolboxes, quickly opened a market for third party development of commonly used functionality, and can thus be seen as an important step towards inter-organizational reuse.

2.1.2. Objects and Object Oriented Frameworks

After being first published in 1960s, the object-oriented programming ideas originating from Simula [Dahl 1966] became more widely accepted in the software engineering community in the 1970s and 1980s. Object-oriented programming is focused around objects, i.e. concepts, abstractions or things with crisp boundaries and meaning for the problem at hand [Rumbaugh 1991]. Objects are quite similar to frames in neural networks [Minsky 1975]. They capture both functionality and state information in a single entity and remove the data dependency problem of conventional programming [Taylor 1992]. The formal representation of this entity without the state information is referred to as a class.

With the increased acceptance of object-oriented programming, Object-Oriented (OO)-frameworks emerged in the late 1980s. An Object Oriented Framework is a set of classes, abstract classes, interfaces and OO-components bundled in a set of modules that partially implement an application in a particular domain [Gurp 2000a]. As such, OO-frameworks not just represent larger reusable software elements, but also capture commonality of applications in design, structure, and behavior.

Although most of the first OO-frameworks were large monolithic frameworks for graphical user interfaces (e.g. the Smalltalk-80 Model View Controller-framework [Goldberg 1989]), OO-frameworks shifted towards other domains and became more fine-grained as well with e.g. the CommonPoint application development environment [Andert 1994]. Over the years, the research community has grown considerably, addressing topics such as framework documentation through patterns [Johnsson 1992] [Gamma 1995] and hooks [Froehlich 1997], framework design documentation [Beck 1989], framework evolution [Roberts 1996] and framework composition [Sparks 1996].

2.1.3. Software Product Families

History

Since the industrial revolution at the time of Ford (1900s), a lot had changed. Technological advances and a society that became increasingly focused on the individual, made that 'any color' is not just 'black' anymore. This desire for variety eventually resulted in products that expressed both manufacturing commonalities and differences. Examples of such *product families,* i.e. products related by common characteristics, or alternatively, products of common ancestry, exist in many areas, ranging from airplanes, cars, televisions and radios to pipes and valves. It was not until the 1990s before product families emerged as a concept in the software industry. However, David Parnas already realized the notion of related products also applied to software in 1976, when he introduced the concept of hierarchical *program families* [Parnas 1976].

An important influence in the emergence of software product lines was the *softwarization* trend, i.e. the increase in the number of different products containing software. As the functionality and underlying hardware of related products contain similarities, consequently the software does too. Combined with the technological advances such as the reuse techniques we described above, and the existence of common platforms (e.g. operating systems, graphical user interfaces and databases), this resulted in more and more organizations simultaneously developing multiple inter-related software systems in different projects, whilst still pressured by quality, time-to-market and financial constraints.

Deja-vu feelings, but also detailed studies, showed that a large amount (up to 90 percent [Cusumano 1991]) of labour in projects of those organizations in any given year, appeared similar to what they had done in other (previous or co-existing) projects. It was realized that similarities in software systems potentially brought an advantage through *economies of scope* [Clements 2001], i.e. an economic benefit by an explicitly planned exploitation of those similarities. The existing technique of arbitrarily connecting building blocks from previous projects (*bottom-up opportunistic* reuse) was not sufficient for achieving these new goals of efficient intra-organizational reuse, however. A new approach would not only require new technical practices, but would also have a large impact on the organization itself.

An important lesson learned by the reuse community over the years, was that reuse had to be explicitly planned by developing components that fit together into a higher-level structure in order to be successful (*top-down structural* reuse) [Jacobson 1997]. Although E. Dijkstra already expressed the importance of structure in his work on operating systems [Dijkstra 1968], it was not until the 1990s, before it was possible to identify an increasing awareness to *explicitly* define software architectures [Bosch 2000]. Of course this lesson could have been learned from other engineering disciplines; You don't build a skyscraper by arbitrary connecting stones, wood and metal, but start with a design and a subsequent construction and selection of new and existing 'building blocks', respectively[1].

Another important insight was that in order to reuse through explicit planning, software engineers could not rely on technology alone, but that other factors such as management commitment, focus on long term benefits and investment were very important as well. Software product families are therefore not just a technological means to group related products within a specific domain, but also in the process, business and organizational dimensions, see e.g. [Jacobson 1997] and [Bosch 2000].

Product Family Assets

A software product family consists of the following two primary assets that are used to develop the related products (also referred to as the product family members), i.e. a product family architecture and the shared components.

- **Product Family Architecture:** The product family architecture is the higher level structure that is shared by the product family members. It denotes the "*fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution*" [IEEE1471 2000].
- **Shared Components:** We use the definition of [Szyperski 1997] for the components: "*A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*". Shared components are therefore the reusable assets that fit into the overall product family architecture.

Development

Development within a product family can be viewed as process consisting of two continuously and concurrently running stages. The first stage, domain engineering, is concerned with the analysis, design and implementation of the product family architecture as higher-level structure and software components as building blocks. The second stage, application engineering, is concerned with using these shared artifacts to build individual products.

Figure 2-2 – Domain and Application Engineering

[1] Note that this approach could be a combination of top-down and bottom-up: a decision to use existing components may influence the design.

Domain engineering: A software *domain* consists of products that share "a well-defined set of characteristics that accurately, narrowly, and completely describe a family of problems for which computer application solutions are being, and will be sought" [Berard 1992]. Domain engineering therefore refers to all activities associated with constructing artifacts that together implement (a subset of) those characteristics in a well-engineered manner. These artifacts typically consist of a product family architecture, and a set of software components that capture the commonalities in design concepts and implementation amongst the related product family members.

Application Engineering: Application engineering is concerned with developing individual products. These products are constructed by deriving the product architecture from the product family architecture and subsequently selecting, configuring and instantiating shared software components that fit into the product architecture. Where necessary, the shared components are modified or extended with product specific code.

2.2. Summary

The overview of the various reuse approaches above, shows that the scale of the reusable software elements has grown considerably over the past 40 years. In fact, the trend that one can deduce from this overview, is that the emerged elements encapsulate each other. Modules for example, achieve their scale by encapsulating a number of functions. Similarly, components can encapsulate one or more modules or an entire OO-framework. OO-frameworks in turn, encapsulate several objects. Finally, software product families achieve an even larger scale by grouping multiple reusable components.

The shift from opportunistic to planned reuse opened up the possibility for inter-organizational reuse such as module toolboxes, OO-Graphical User Interfaces and other commercial off the shelve (COTS) components. As product families particularly focus on intra-organizational reuse, inter-organizational reuse through product families is not yet a wide spread phenomenon. This may very well change in the future however, as the growth of system scale will constantly keep the software engineering community occupied with the search for new techniques that also increase the scale of reuse. In addition, inter-organizational reuse through software product families may be pushed by the open-source community. An example [Gurp 2000a] can already be found in the Mozilla community [Mozilla 2002].

Chapter 3 – Handling Differences

Software product families (SPFs) are an important topic in the research community, considering the considerable number of books [Weiss 1999] [Bosch 2000] [Clements 2001], papers [Parnas 1976] [Bosch 2001a] [Bayer 2002] and conferences and workshops. As stated in the previous chapter, software product families are used as a means to capture various commonalities between related products. The real difficulties however, are associated with handling the *differences* between those products, which the software engineering community is starting to realize as the attention with respect to product families is shifting towards variability and variability management in particular, e.g. [Anastasopoulos 2001] [Bachmann 2001] [Batory 1992] [Clauβ 2001] [Griss 1998].

Before we state our concrete problem in chapter 5, this chapter presents the concepts of variability in software product families as background. We start by defining variability in the next section. In the following two sections, we describe the two main activities associated with handling variability. In the last section, we discuss variability in relation to the two stages of product family development.

3.1. Product Lifecycle and Variability

The lifecycle of an independent product is often represented using the phases of the waterfall development model. Besides being well known, the waterfall model is also the first published model of a software development process [Royce 1970]. The waterfall model is a phased model, which, in each phase, consists of transformations from one level of abstraction to the next, starting with the architecture design to detailed design, to implementation, to compile, to link, to run-time. The model is uncomplicated, represents engineering practice, and has been widely used in quite a number of organizations [Sommerville 1982]. Another advantage is that, although the exact nature of software development is technology specific, the model can be tailored easily. If, for example, an interpretive language is used for product development, run-time applies to compiling, linking and running code, whereas in a descriptive language like C, run-time applies to running code (when disregarding dynamic linking).

In our opinion, three phases are missing however, i.e. a *distribution*, *installation* and a *start-up* phase. During these phases important decision may still be taken, such as the decision to ship an email client with or without a built-in editor (distribution phase) or installing the 'minimum' instead of the 'full' edition of a product (installation phase). In this thesis, we therefore use the slightly adapted lifecycle model, as illustrated in Figure 3-1 .

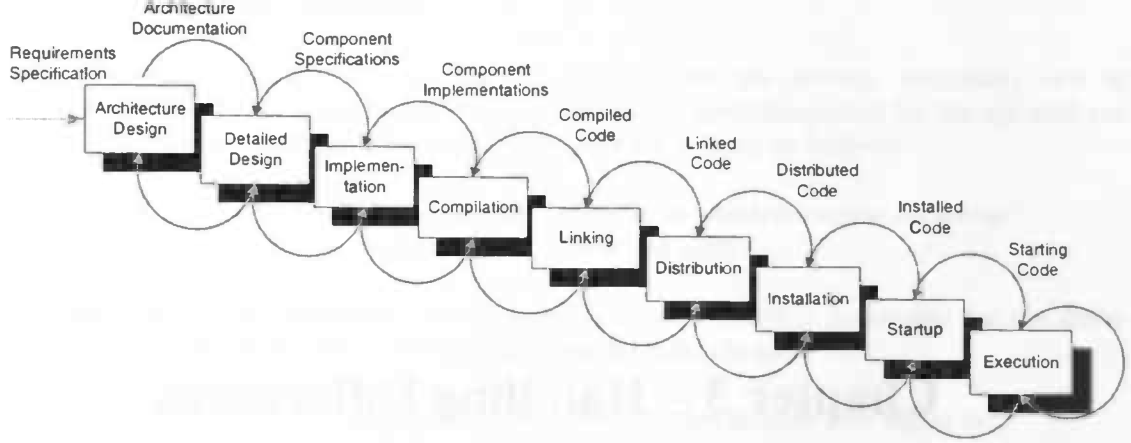

Figure 3-1 – The Expanded Lifecycle Model

Each product family member has its own, unique lifecycle, but expresses some form of commonality with the related members on one or more levels of abstraction. The purpose of the shared product family artifacts is to capture and exploit these commonalities by providing a product family architecture, component designs and component implementations. In other words, shared artifacts are implemented on three abstraction levels, to accommodate the common characteristics of the product family members on nine levels of abstraction.

A product's value however, is determined by its unique characteristics. As a result, products not only express commonalities on different levels of abstraction, but also differ on one or more abstraction levels[2]. Product family members may for example share the same architecture, but include different component implementations. Shared artifacts therefore not only have to be constructed in such a way that the commonality can be exploited economically, but at the same time have to preserve the ability to vary the products. This ability to facilitate known differences between products on different levels of abstraction is referred to as **variability**, which literally means "the ability to be subject to change" [Webster 2002].

Managing the process of taking decisions regarding variability properly is regarded as the key factor in the success of software product families. In literature, there are two main activities associated with variability management, i.e. *facilitating differences* and *exploiting variability*. For the sake of simplicity, we take the viewpoint of facilitating differences in domain engineering and exploiting this variability during application engineering in the next two sections. For a more detailed discussion on this viewpoint, see section 3.4.

3.2. Facilitating Differences

The first step in facilitating differences is to identify the known differences between the products. Once these differences are determined, they can be accommodated by introducing variability in the shared artifacts. We give a more detailed description of these steps below.

3.2.1. Identifying differences

Identifying the variable characteristics of related problems is a field that is studied extensively (e.g. FODA [Kang 1990], FORM [Kang 1998], FeatuRSEB [Griss 1998]). For a large part the research community agrees on two things however, i.e. 1. the concept of *features* can be

[2] Although one may be able to prove that different designs transform into the same implementation, it is save to say that in a product family, products that differ in one stage typically also differ in later stages.

used in order to focus on the essential characteristics in domain model descriptions and 2. in using those concepts, [differences] can be identified more easily [Gurp 2002].

In [Bosch 2000] the definition of a feature is specialized for software systems: "[a feature is] *a logical unit of behavior that is specified by a set of functional and quality requirement*". The book goes on by stating, "... *there should at least be an order of magnitude difference between the number of features and the number of requirements for a product (...)*". In other words, features are used to group and abstract from requirements.

This abstraction from features to requirements however, is not a transformation to an independent space, in other words [Bosch 2000] features are not independent entities. Instead, features abstract from requirements in an n-to-m relation; each feature possibly implements many requirements and a single requirement can apply to several features. This so-called *feature interaction* [Zave 1997] [Gibson 1997] [Griss 2000] prevents straight forwarded design from features to components as interdependencies between features result in situations where [Griss 2000] "*a carefully coordinated and complicated mixture of parts of different components are involved* [in the implementation of a single feature]". This particularly applies for features that have an influence throughout the entire software system (*crosscutting* features, see also [Kiczalez 1997]).

For a structured description of the essence of a domain, features can be categorized into *mandatory*, *optional*, *variant* and *external* features as listed in [Griss 1998] (the first three) and expanded in [Gurp 2001] (the last).

- **Mandatory features** are the features that identify a product or product family, e.g. the ability to view a webpage in a browser. Note that related products typically have different, but not disjoint sets of mandatory features.
- **Optional features** are features that correspond to characteristics that are not essential to all products in a domain, but add value to the core features of the product, such as an alarm clock in a cell phone. In some cases, optional features evolve into mandatory features as product markets proceed. For example, where a few years ago, a cell phone was characterized by the ability to make and receive phone calls and almost anything else was optional, nowadays a cell phone is also identified by its ability to send and receive short text messages and play games.
- A **variant feature** is an abstraction for a set of related features and corresponds to alternative ways to configure mandatory or optional features. Variant features either exclude each other or may be used both. Examples of variant features are the ability to support different communication protocols, or screen resolutions on mobile phones.
- **External features** correspond to features offered by the target platform of a system, such as I/O functionality. External features are typically needed to be able to reason about the context in which a system operates.

Due to feature interaction, features from one set can in- or exclude features from a different set. A mandatory feature may for example require the existence of at least one particular feature variant, or one optional feature may exclude another optional feature.

Over the past few years, several feature notations have been proposed. Most of them are basic extensions of the original FODA modeling. RSEB, for example, extends the notation, while [Gurp 2000a], introduces external features and the possibility to express the stage of the lifecycle in which the feature should be bound (i.e. the binding time) to the model. [Clauβ 2001] extends the standard UML [UML 2000] class diagram by using stereotypes. This extension supports mandatory, optional, alternative and external features, binding time, rationale, and constraints such as mutex, and requires.

3.2.2. Introducing variability

Once the differences between product family members are known, variability can be introduced in the product family artifacts by constructing *variation points* for the optional and variant features. In [Jacobson 1997] variation points are defined as follows:

Variation points identify one or more locations at which variation will occur within a class, type or use case.

The latter part of this definition is very specific, which makes it unsuitable for the entire lifecycle of a product. In [Gurp 2000a] a different definition is used:

A variation point is an element of the representation at hand that refers to a delayed design decision.

Although this definition is applicable to the entire lifecycle, it implies a design decision is postponed, while actually a different, careful and educated design decision is made; the decision to postpone the selection of alternatives. It furthermore suggests differences between products (modeled with variant features) can be represented with a single point, while the feature interaction problem suggests this would not be possible for all features. We therefore adapt the definition used in [Svahnberg 2002]:

Variation points are places in the design or implementation that together 1. provide the mechanisms necessary to make a feature variable [and 2. refer to the delayed selection of the alternatives].

Constructing a variation point requires several steps [Bosch 2001a]. First, the stable behavior has to be separated from the variable behavior. Second, an interface has to be defined between the stable and variant behavior. Third, a variability handling mechanism has to be designed. Finally, one or more alternatives have to be implemented as variants
.

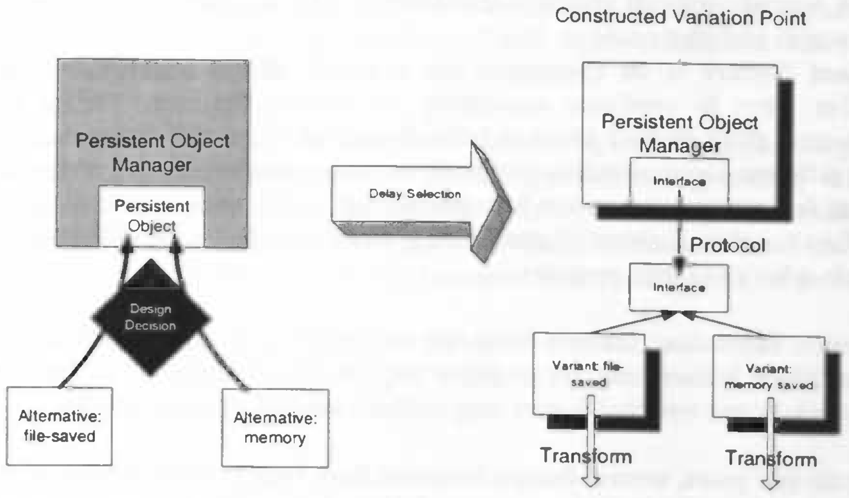

Figure 3-2 – Variation Point Construction

We illustrate the basic mechanism for constructing variation points with a slightly adapted example from the Prothos product family (see Figure 3-2). In this example, we are at the detailed design level where two product family members need different mechanisms for storing persistent objects, i.e. in text files and in memory. To accommodate this difference, a software architect has to delay the selection of either one of the alternatives to a later stage, e.g. to installation time. This can be accomplished by constructing a variation point. Up until this stage that variation point was implicit, the construction itself will make it explicit.

In the transformation of one level of abstraction to the next, the constructed variation point is transformed as well, possibly to a set of variation points on a lower level. In addition, at a lower level new variation points can be introduced for features that were not visible at the previous level, as not all requirements are architecturally significant [Jacobson 1997]. In addition, if the set of variants for variation point was open for adding new variants in the previous stage, it may be closed when transformed to the next, i.e. to a set for which it is not possible to add new variants. Furthermore, during the transformation dependencies may be introduced to other parts of the variant due to implementation details.

Over the past few years, several variability realization techniques have been identified for the various abstraction levels. Examples of these mechanisms include architectural design patterns, aggregation, inheritance, parameterization, overloading, macros, conditional compilation, and dynamic link libraries (see also, e.g. [Jacobson 1997] [Anastasopoulos 2001]). [Svahnberg 2002] presents a taxonomy of realization techniques.

3.3. *Exploiting Variability*

Once the product family architecture and components are in place, individual products can be constructed by using these artifacts. The process of constructing product family members from shared product family artifacts is referred to as *product derivation*. Where during design and construction of the shared software artifacts variability is facilitated through variation points, product derivation is typically the moment when the ability to handle differences between products can be exploited by using those variation points.

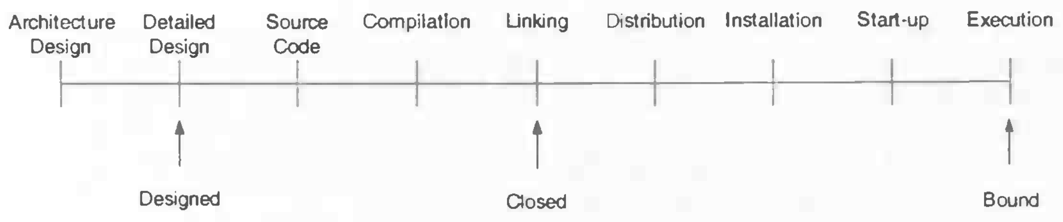

Figure 3-3 – Variation Points and Binding

3.3.1. Collecting & selecting variants

Each explicit variation point is associated with a set of variants and can be bound to one or more different variants from that set. As there may exist dependencies between variants of different variation points, selection of variants requires careful analysis of dependencies and constraints with respect to other variants from the collection.

3.3.2. Extending and constructing variants

In case the required variants are not completely designed or implemented yet, new (versions of) variants can be implemented by extending existing or implementing new variants. Also during this step, engineers have to consider the dependencies with other variants, as for example, changes to existing variants should not lead to inconsistencies.

Whether the set of existing variants at a particular level of abstraction can be extended, depends on the fact whether the variation point is *open* or *closed* [Gurp 2000a], i.e. more variants can be added to the system, or no more variants can be added, respectively.

3.3.3. Binding

Finally, the variants have to be bound to the open variation points. We distinct between *internal* and *external* binding [Svahnberg 2002]. In case of external binding, the variation point is bound by a developer, while in case of internal binding, software code handles the binding.

Closure and binding of a variation point can occur at different stages in the lifecycle. An example (Figure 2-1) originates from [Svahnberg 2002]. This example involves an abstract class (a variation point designed at the detail design level), which is open for adding new subclasses until link-time, as it is impossible to add new subclasses without at least re-linking the system. The system is then bound to a particular variant at run-time.

In case multiple variants exist in parallel, the variant selection and binding process is performed each time the variation point is accessed, rather then being bound permanently. This leads to a second distinction, i.e. *permanently bound*, and *rebindable.*

3.4. Variability vs. Domain and Application Engineering

Note that in previous sections, we described variability from the viewpoint of constructing variable artifacts during domain engineering in order to be able to develop different product family members during application engineering. This view is not entirely complete however. First, besides handling known differences between products, an additional reason for introducing variability can be found in the definition of a variation point in [Gurp 2000a]:

A variation point is an element of the representation at hand that refers to a delayed design decision.

In other words, variation points may be needed in case it is not known which of the different alternatives for a collection of requirements will be chosen. In addition, variation points are not only essential in exploiting similarities between product family members, but also in exploiting similarities in functionality within one application. In fact, this situation and the one illustrated in 3.3 are quite the same [Bachman 2001]. In both situations, it is not known at the abstraction level at hand which alternative will be chosen.

Second, the previous sections may have suggested that during domain engineering the number of unbound variation points increases while during application engineering selections and settings for a particular product cause the number of open variation points to decrease. This picture is not complete however, as product specific variant construction and extensions may very well introduce new variation points, for example, if there is a need for product specific run-time variability.

Chapter 4 – Prothos, an AvraSoft Product Family

To confirm the findings we present in this thesis, we conducted a case study of the Prothos product family at AvraSoft, a small spin-off company of the University of Alberta, Canada. In the next sections, we provide a general overview of the AvraSoft business model and philosophies as presented at [AvraSoft 2002]. In the last section, we discuss the Prothos product family.

4.1. AvraSoft and the University of Alberta

Avra Software Lab Inc. (AvraSoft) was started in 1998 to commercialize the technologies discovered and developed by the Software Engineering Research Lab (SERL) at the University of Alberta. SERL's philosophy is to perform research on problems faced by practitioners and to validate the findings resulting from those efforts in the field of practice as well. Consequently, a considerable amount of work at SERL involves industrial contacts, both in contract and grant settings. Acting on this philosophy provides SERL staff with valuable insights for both research and teaching at the University of Alberta. In that respect, the philosophy highly resembles our believes at the research group in Groningen.

Although basic research at SERL is published and promoted through the traditions of academic research, some ideas lead to the development of new and practical technologies. The University of Alberta retains the intellectual property rights on these technologies, which are licensed to AvraSoft when there is an opportunity for commercialization. AvraSoft subsequently undertakes continuing research and development to adapt and adopt these technologies to its clients needs.

4.2. Avrasoft's Business Model

AvraSoft's business model is based on both a *development* company and a *virtual* company (see Figure 4-1). As a development company, AvraSoft provides both the development service as the helpdesk service for commercial production applications to a variety of customers. AvraSoft retains joint intellectual rights with clients, even when the product ownership is transferred to the customer. Intellectual property rights are never licensed exclusively to one customer.

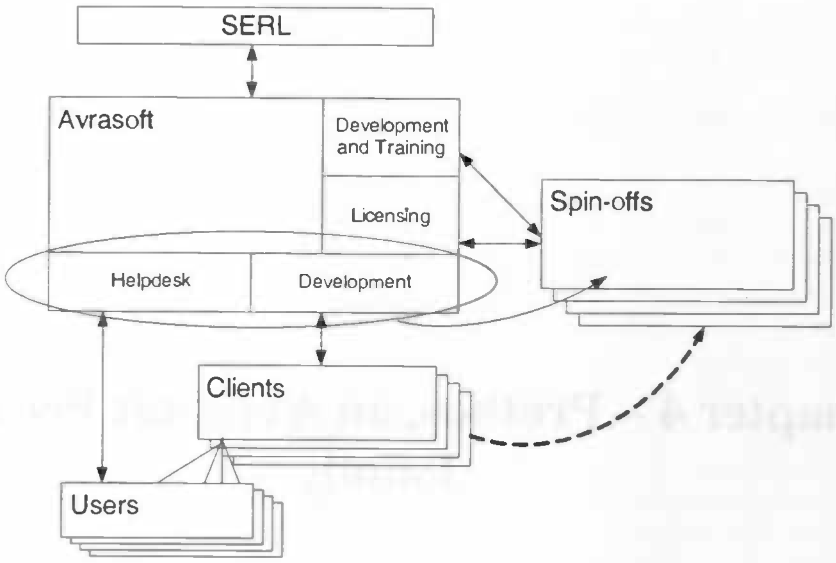

Figure 4-1 – The AvraSoft Business Model

As a commercial product matures, it may be spun-off into a separate development company in order to prevent AvraSoft from growing to an oversized and thus ineffective interface to SERL. The resulting spin-off may provide distribution and support of its products or outsource it. In this setting, AvraSoft can provide development and training services to the spin-offs.

Funds are acquired by billing provided services (development and support) and collecting royalties from the licensed technologies. AvraSoft in turn pays royalties to SERL. In effect, both SERL and AvraSoft benefit from the collected royalties.

4.3. Development

AvraSoft aims at long term customer relationships. To this extent, AvraSoft provides distribution and support services to the customers. This enables customers to focus on their business rather the software their business is running on. Avrasoft furthermore takes an evolutionary product family approach to software development. The evolutionary part is motivated below:

1. Problems are usually better understood after you started to work on the solution.
2. Introducing applications to existing business processes changes them, thus adding new requirements to the delivered applications (due to new demands and possibilities).
3. Rapidly changing technologies (which particularly applies to e-commerce), keep expanding possibilities at a fast rate.
4. The user interface and business workflow for buyers and users of engineered products represent a relatively complex problem that requires evaluating alternatives.

Adopting an evolutionary product family approach not only solved many of the issues associated with the motivation outlined above, but also provided AvraSoft with a competitive advantage: the ability to provide new products and evolve existing products fast and at relatively low costs.

Recent work at AvraSoft has concentrated on markets for engineered product sizing, selection, and ordering software, in particular in the context of web-based e-commerce. In collaboration with SERL, this resulted in the Prothos Product Family. We discuss this product family in further detail below.

4.4. Prothos Product Family

Prothos is a product family that provides a minimalist environment for implementing a web-based client/server RDBMS (Remote Database Management System) application that wraps order processing business classes into a working product. It supports the selection, display, editing, and posting of persistent business data that has been structured into a set of instances of business classes. The business classes implement the application's business rules, which are responsible for the application-specific processing [Hoover 2000].

The Prothos Family is currently in its third generation. The Prothos architecture consists of three OO frameworks, i.e. the Prothos Framework, the User Interface Manager and the Persistent Object Manager). The User Interface Manager is a sub-framework of the Prothos Framework and coordinates the user interface to the set of persistent business classes. These business classes encapsulate access to the business services and obtain their persistence by inheriting from a second Prothos sub-framework, the Persistent Object Manager. The Prothos Framework is mainly a utility framework that provides functionality such as exception handling, file I/O, and logging.

The Prothos product portfolio includes an internet helpdesk application for pressure relief valve engineering software (SizeMaster Helpdesk), a groupware and distributed development environment (CafeAvra), a Process Safety Management application (PSM), and a groupware and repository environment for the Westlink innovation network (Westlink).

Chapter 5 – Problem Statement

In our discussion on variability in chapter three, we stated that in order to exploit the similarities economically, known differences between product family members are facilitated through variation points. In this chapter, we discuss the effect of unforeseen differences with respect to problems that emerge during product derivation. The last part of this discussion formulates our concrete problem statement.

5.1. Product Derivation Problems and Evolution

Typically, variation points are designed and implemented during domain engineering and exploited during application engineering. The general belief in the research community is that most benefits on reducing time-to-market and product development costs are gained by spending most effort in domain engineering. However, industrial practice learns that the situation has grown to a point where not always domain engineering is experienced as the most expensive or difficult activity, but where deriving the individual product family members from the variable artifacts poses more and more problems.

One of the problems many organizations face during product derivation is a situation that we refer to as the *inversely proportional product derivation problem* (IPPDP, illustrated in Figure 5-1). This problem exists on both the product family and the individual product level, as explained below.

1. The first level is on the level of the entire product family. In this situation, 60%-70% of the product family members can be built by routine [derivation] [Hein 2000], and are responsible for 10% of the total costs, whereas 10% of products require extensive artifact modification and are responsible for over 60% of the total costs [MacGregor 2002]. In other words, the percentage of products that requires change is inversely proportional to the resulting share in the total costs for deriving product family members.
2. On the second level, i.e. the level of a single product, typically 70% of the functionality can be built up very quickly without modifications, while the last ten percent requires substantial development effort. This results in a situation where the percentage of functionality requiring change is inversely proportional to the resulting share in the total costs for deriving that product.

Apparently, during the derivation of new products there is functionality that is very hard to implement. If we relate this to the concepts of variability presented in chapter 3, the question is, "since the artifacts should contain facilities that ease handling of differences during product derivation, why are extensive modifications required in the first place?"

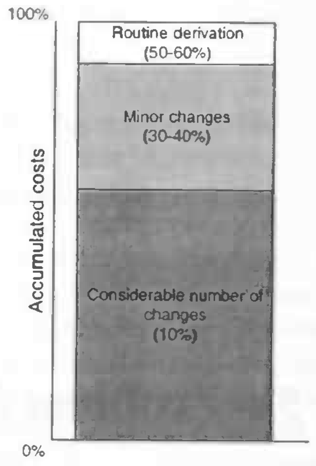

Figure 5-1 – The IPPDP on Product Family Level

Provided that all steps related to handling known differences are carried out carefully, we can find a hint to answer this question in Lehman's law on software evolution. His law states that a useful software system must undergo continual and timely change or it risks losing market share [Lehman 1997]. In that respect, it has a strong influence on software product families; product families typically consist of over three products in order to be economically feasible[3]. Consequently, not only the contemporary differences have to be analyzed and accommodated when facilitating differences, but also the differences of future product family members.

An extreme example of changing markets is the web-browser market, just a few years ago. The intense competition between the products of Microsoft and Netscape resulted in a so called "browser war", with extremely rapidly evolving products. This did not only lead to a large number of new product versions and bug fixes, but also to a wealth of entirely new functionality in just a few years.

Example 5.1 – Evolution to the extreme

Although guidelines suggest a five year prediction window for functionality when designing software product families [Macala 1996], the example above illustrates that it may prove to be very hard to predict the functional evolution of a product for even one year, let alone for all products that will ever be derived from a product family. In addition, even if it could be theoretically proven that a particular product family would be able to accommodate all possible evolution paths, the question is to what cost. Increased variability generally implies increased design and implementation complexity, increased resource consumption and it generally costs more to develop a flexible artifact (see also e.g. [Jacobson 1997]). Providing variability for something that might never be needed is thus not always economically feasible as the chance of return on investment being zero is rather high. Furthermore, if a product family would indeed adhere to one particular planned five-year path, unforeseen requirements will emerge after that period, as product families are often exploited for over a decade.

In any case, once the product family is in place, at some point in the lifecycle, evolution will force it to handle new functionality and thus previously unforeseen or unaccounted differences. This results in situations that we refer to as *variability mismatches*. We detail the concept of a variability mismatch in the following two sections.

[3] A philosophical question that remains is when a system such as one that can be deployed in a home, professional and enterprise flavor becomes a new or different product. If for example the configuration resulting in an enterprise edition was done pre-deployment, would the enterprise edition be a different product then the professional edition? And how about post-deployment (or the runtime) configured editions? We leave this (challenging) discussion outside the scope of this thesis.

5.2. *The Notion of a Variability Mismatch*

From the Webster dictionary [Webster 2002], we learn that to provide and to require is "to supply or make available" and "to demand as necessary or essential", respectively. As we described in chapter 3, variability of a product family is *provided* in terms of variation points with associated interfaces, mechanisms, interdependencies, constraints, binding times and variants. The set of requirements of a product family, i.e. a united set of requirements of the individual product family members, represents commonalities and differences among those members in terms of functionality and quality attributes. The presence of differences *requires* from a product family the ability to handle them, which can also be expressed in terms of variation points with associated interfaces, mechanisms, dependencies, constraints, binding times, and variants.

A good example of well-supported differences between applications in the Prothos product family is the instantiations of a variety of business classes during execution. As soon as an application requires a business class with a rather unique workflow and special processing however, a mismatch occurs between the variability that is provided by the product family and what is required by the new product or product version.

Example 5.2 – Variability Mismatch

During evolution of a product family, the family members may present new requirements to the existing requirement set that translate in both new commonalities and new differences. It may be required to handle these new differences through variation points. In some situations however, these differences cannot (at least not directly) be handled by the variability that is provided by the product family. This is what we refer to as a *variability mismatch* (see Example 5.2). If we summarize the above, we get the following definition:

A variability mismatch is a situation in which the variability supplied by the product family (provided variability), is not suitably associated with the variability that is demanded as necessary by new functional or quality requirements imposed by product family members (required variability).

In the next section we will present a more detailed discussion of what 'not suitably associated', 'provided' and 'requires' mean in the context of variability management and evolution. In particular, we will discuss why the provided variability of a product family may not be capable of handling the required variability and what issues are associated with actions that have to be taken in order to solve the mismatch (in other words, change the variation point).

5.3. *Variability Mismatch in Detail*

When the required functionality fits within the existing set of variation points, viz. can be implemented by selecting, adding, extending or adapting variants, the provided product family variability fits the required variability. Below, we list the situations in which the opposite – a mismatch – occurs in further detail. This discussion is basically an extension of the discussion on evolution patterns presented in [Bosch 2001a]. Here, we relate the discussion to the notion of a variability mismatch, several issues associated with solving the mismatch, as well as examples, amongst others, from our case study. We present these mismatches in a pattern-like fashion (see e.g. [Gamma 1995] and [Buschmann 1996]), with the following topics:

- **Description:** A detailed description of the mismatch.
- **Example:** An example of the mismatch from the case study.
- **Issues:** Issues associated with actions that have to be taken to correct the mismatch.

5.3.1. Interface Mismatch

Description: Variable behavior is accessed through the interface and the associated protocol of a variation point. A mismatch occurs if functionality requires addition or removal of attributes in the provided or required interface or changes to the interface protocols.

Example: Up to version 2.2.0.10, the Persistent Object Manager (POM) supported three variants for Persistent Business Class (PBC) management (Figure 5-2). Objects were saved either to files, memory, or to a database with a Perl Database Interface (DBI). An appropriate variant was selected and bound at start-up, by inserting the right pathname for the object in the appropriate configuration file. The variation point mechanism was inheritance; each variant overrode the necessary methods of the persistent base object (the super class). For performance reasons, the interface was changed in version 2.2.0.11 to allow additional behavior for the file-saved PBC variant.

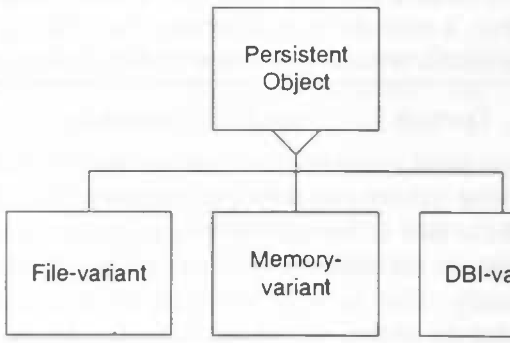

Figure 5-2 – Database Management

Issues: Even minor changes to a variation point's interface may have large implications. A straightforward change to the interface (such as an operation syntax change) leads to a situation where either all existing variants may have to be changed accordingly in order to be accessible from the calling component, or code has to be added to the calling components in order to deal with situations that involve different provided interface. Changing the interface can thus prove to be a quite effort consuming activity, especially for large variant sets.

Frequent changes to the same variation point furthermore lead to a situation where multiple variants exist that express the same behavior, but have different interfaces. Additionally, if not all existing variants have been changed, selecting certain combinations of variants may be restricted as the variants are incompatible. This complicates the variant selection process.

5.3.2. Mismatch due to dependencies or constraints

Description: As we discussed in Chapter 3, variation points may depend on both stable and other variable parts of a software system. In case of a variability mismatch, the selection of a certain variant has undesirable effects on other variable behavior. This includes the situation in which selecting a particular variant restricts the set of legal variants for a variation point in such a way that it excludes other required functionality, or contrary, requires the selection of other variants while this is undesirable or impossible.

Example: An example of a variant requiring another variant is inspired by a case discussion in [Bosch 1999], involving Axis Communications AB and its file system and network frameworks. In this discussion, it was stated that, at some point, implementation details caused a tight coupling between particular file system variants that implemented file system standards, and particular network protocol variants that implemented different networking protocols. A mismatch due to these dependencies would be a situation in which the file system variants should vary independently from the networking variants.

Issues: Although in some cases variation point dependencies are inherent to the problem domain, in case of a variability mismatch the dependencies are the result of implementation details. Removing dependencies is complicated, however, by the fact that, often, not all dependencies are explicitly represented in the software system. After a change to the variation point dependencies the software system may therefore seize to work correctly [Bosch 2001a].

5.3.3. Mechanism Mismatch

Description: A mechanism mismatch occurs when certain changes are required to the variability mechanism. It may for example be that the point at which the variant set is closed needs to be moved to a later stage in the product lifecycle, that the variant set has to be extended, or that certain quality requirements require mechanisms that consume less resources.

Example: In the second generation of the Prothos Family, handling of different HTTP-requests was performed through aggregation, where the selection of the handler was based on the URL extension. This variation point was closed at the implementation level, but bound at run-time. A mismatch would occur both in the situation that a variant should be added, or the time at which the set was closed should move to e.g. start-up time.

Issues: Although some mechanism mismatches can be handled by a few changes and addition of variation management software, changing the entire mechanism may prove to be effort consuming. Not only are variability mechanism often intertwined with stable behavior [Bosch 2001a], also, the variable behavior may have to be reimplemented in order to be accessible through the new mechanism.

5.3.4. Binding time Mismatch

Description: A typical trend in software system development is that the binding of variation points is evolving towards later stages in the product lifecycle, in order to increase the flexibility of the software system. Consequently, variation points previously bound in early phases of the lifecycle, such as compile-time, may now be required to be bound at later stages, such as start-up or run-time. Binding time may also be required to be shifted to earlier phases, for example to decrease resource consumption, or increase predictability. In contrast to the binding time as presented in Chapter 3, the *required* binding time for a feature is not necessarily associated with one abstraction level, but with a window of abstraction levels in which the feature should be bound. We therefore introduce the notion of an *earliest* and *latest* binding time. The earliest binding time denotes the earliest abstraction level at which a feature may be bound, while the latest binding time denotes the latest abstraction level at which a feature must be bound. In Figure 5-3 for example, the variation point may not be bound before linking and at start-up at the latest. A binding time mismatch therefore occurs when the actual binding time is outside this binding window.

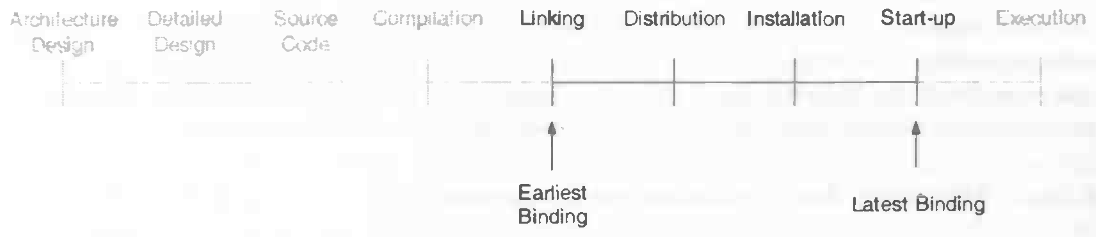

Figure 5-3 – Earliest and Latest Binding

Example: In the third Prothos generation, a variant for database management (see 5.3.1) was bound at start-up time, by specifying the right path name to the variant in a configuration file. A situation in which a binding time mismatch would occur, is when the persistent object manager should be able to rebind to another variant at run-time.

Issues: In fact, the suitability of the variation point binding time is closely related to the suitability of the variation point mechanism, as the binding time is inherently connected to the variability mechanism. On top of the issues associated with a mechanism change, however, moving the binding time to a later moment in the lifecycle typically increases transport, load, and runtime resource consumption [Bosch 2001a]. In addition, binding in stages later then link-time requires management software that handles the selection and (re)binding of the appropriate variants. This management software also has to handle cases in which selection fails due to dependencies and constraints.

5.3.5. Mismatch due to non-existing variation point

Description: An additional situation in which a mismatch will occur is when functionality needs to be implemented as variant or optional behavior, and no suitable variation point is available. In this, we recognize two distinct situations. In the first situation the new functionality is needed as option or alternative to already implemented stable behavior. This situation mostly occurs if the need for variance was not recognized before or when a change in market conditions forces the organization to support more than one alternative in parallel. In the second situation, the existing system behavior is only extended with the new functionality.

Example: An example of a case in the third generation of the Prothos family, where already implemented functionality would be involved in the construction of a variation point and associated optional variant would be optional access control. Although we note it is not very likely to be required, no variation point exists in order to be able to turn access control on or off.

Issues: In both the situations that the conceptual variant functionality did and did not exist as stable behavior, an interface has to be defined between the variable behavior and the rest of the system. Furthermore, an appropriate mechanism and associated binding time have to be selected and the mechanisms and variant functionality have to be implemented (as also discussed in section 3.2.2). In addition, in the situation where existing functionality is involved, the implemented functionality has to be clearly separated from the rest of the system and re-implemented as a variant that adheres to the variation point interface. In case the binding time is in the post-deployment stage, software for managing the variants and binding needs to be constructed.

Issues primarily emerge in the situation where already implemented functionality needs to be re-implemented as variable behavior. First, as variability mechanisms typically require extra resources and hamper testing due to the large number of possible combinations, quality attributes of the system with the re-implemented functionality selected may differ from the previous situation (e.g. predictability, reliability and resource consumption). Second, due to code scattering and tangling [Kiczalez 1997], it may prove to be laborious and hard to find and separate all the existing stable behavior. Third, as dependencies in between software parts are often implicit, the software may seize to function correctly (see also 5.3.2).

5.3.6. Mismatch due to obsolete variation point

Description: A typical trend in software systems is that functionality specific to some products becomes part of the core functionality of all product family members, e.g. due to market dominance. The need to support different alternatives, and therefore variation points and variants for this functionality, may therefore disappear. This phenomenon leads to the existence of obsolete variation points, i.e. variation points for which in each derived product the same variant is chosen, or, in case of optional variants, not used anymore.

Example: An example of the Prothos family where a variation point would become obsolete is related to the Persistent Objects in the Persistent Object Manager (see fig). Currently, the POM supports three variants, although the memory variant is not used anymore. A likely scenario is that the need to support the file variant will eventually disappear, leaving the database variant as the only alternative.

Issues: Obsolete variation points are often left intact, rather then being removed from the assets. Although the general opinion in the research community is that variability improves the ability to select alternative functionality and thus the ease of deriving different product family members, the lack of removing obsolete variation points and variants result in a situation where the cognitive complexity in terms of number of variation points and variants can only grow. This growth in cognitive complexity may hamper the product derivation process.

On the one hand, removing unnecessary variation points requires effort in many areas. All variation points related to the previously variable functionality have to be removed, which may require redesign and reimplementation of the variable functionality. There may be dependencies and constraints that have to be taken care of. Furthermore, changing a component may invalidate existing tests and thus requires retesting. On the other hand, removing variation points increases the predictability of the behavior of the software, decreases the cognitive complexity of the software assets, and improves traceability of suitable variants in the asset repository. We also note that if variability provided by artifacts is not used for a long time, or removed from the documentation, engineers may start to forget some facilities are there. Furthermore, when incomplete documentation leads to a situation in which changes to artifacts result in the removal of parts of the facilities but not all, artifacts may suffer from inconsistent behavior.

Even if it is obvious that a variation point is obsolete, determining whether it should actually be removed may prove to be rather difficult. Although estimating the costs of redesign, reimplementation, and retesting can probably be done rather accurately, predicting the benefits of removing a variation point in terms of effort accurately may turn out to be hard, as the benefits are typically related to cases in which obsolete variation points are problematic. Also, the effects of removing a single variation point on cognitive complexity can be rather small, as it is the collection of obsolete variation points that adds up to the complexity.

5.4. Handling Variability Mismatches

Rather then selecting different variants during product derivation, the variability mismatches presented in the previous section, can only be accommodated by adapting the shared artifacts. An organization can employ three types of adaptation in parallel, i.e. product specific adaptation, reactive evolution, and proactive evolution.

- **Product specific adaptation:** The first level of evolution is where, during product derivation, new functionality is implemented in product specific artifacts. To this purpose, application engineers can use the shared artifacts as basis for further development, or develop new artifacts from scratch. As functionality implemented through product specific evolution is not incorporated in the shared artifacts, it cannot be reused in subsequent products. One could argue that product specific adaptation would not be a concern for variability management, as all changes cannot be reused and therefore be hard-coded in the artifacts. Variability, however, is not only concerned with handling differences between product family members, but also e.g. between differences in behavior at run-time (see also chapter 3). In other words, new requirements can also require changes to variation points in product specific adaptation.

- **Reactive evolution:** The second level of evolution involves adapting shared artifacts in such a way that they are able to handle the new functionality that was imposed by the product at hand, and can still be shared with other product family members. As adapting shared artifacts has consequences with respect to the other family members, those effects have to be analyzed prior to making any changes.
- **Proactive evolution:** The third level is a pure domain engineering activity. It involves adapting the shared artifacts in such a way that the product family is capable of accommodating the various family members in the future. Proactive evolution requires both analysis of the effects with respect to current product family members, as well as analysis of the predicted future of the domain and the product family scope.

As effective reuse in software product families can only be achieved if it continuously facilitates differences between its members, we can formulate an evolution law for product families similar to [Lehman 1997]:

The variability of a product family has to undergo continual and timely change, or it will risk losing the ability to effectively exploit the similarities of its members.

The product specific and reactive evolution types described above both encompass reactions: first, the mismatches emerge, and then they are dealt with. Although reactive evolution of variability does provide a mechanism to enact part of our variability evolution law, i.e. to *continually* change variability, the question is whether these reactions are really that *timely*. Below, we formulate a number of related reasons for why we would want to *prevent* variability mismatches from occurring during product derivation.

- **Decrease time-to-market or increase amount of functionality:** One of the goals of many organizations is to decrease the amount of time spent in between start of development and product delivery. If variability mismatches can be predicted and prevented prior to the actual derivation of a software product, valuable time and effort in the derivation process can be saved, or used elsewhere, for example to implement additional functionality.
- **Remove Overhead:** A well-known fact in any engineering discipline is the fact that reaching a goal in one big step is generally cheaper than in a number of small steps. A typical example of a situation where by preventing mismatches overhead can be saved is a situation in which evolution frequently requires changes to the same variation point.
- **Flexibility:** Related to the previous points is the reusability of the software assets. Product engineers typically have an intense focus on time-to-market. In order to meet deadlines, long-term benefits such as reusability are easily sacrificed. By preventing mismatches from occurring during product derivation, required changes can get the attention that they deserve.

Of course, preventing possible mismatches not only have benefits, but also have an associated risk. Predictions always contain an amount of uncertainty. Spending time and effort in changing predicted mismatches can therefore have two negative effects, i.e. either time and money is spent on something that will never be needed or worse, on something that even makes the mismatch larger. Any method for prediction and preventing mismatches should therefore consider minimizing these risks.

5.5. Concrete Problem

Considering the previous problem investigation, we conclude that evolution of variability should be added as a main variability management task. This brings us to our concrete problem. Literature does address topics such as the creation and instantiation of generic and specific product family assets as well as dealing with variability at the code level [Anastasopoulos 2001]. Although exceptions exist, e.g. [Svahnberg 2000], evolving variability has received, to the best of our knowledge, little attention however. The work presented here, therefore aims to investigate the consequences of evolution on variability and the actions that can be taken to correct and prevent these consequences from occurring during product derivation. This thesis therefore aims to investigate the following research questions:

- R1 How can we prevent that variability mismatches occur during product derivation?

This question immediately results in the following sub questions:

  - R1.1: How can we determine that mismatches occur
  - R1.2: When should we take actions and when not?

[illegible]

Chapter 6 – Variability Assessment

As we stated in the previous chapter, in our opinion, the source of the inversely proportional product derivation problem is related to the new differences and the resulting variability mismatches that emerge during the product family lifecycle. We asked ourselves how we could predict when variability mismatches will occur and whether actions should be taken to prevent them. The answers to these questions, we believe, can be found in techniques for variability assessment, which is the subject of this chapter.

6.1. Introduction

Over the past decade, several prediction methods have emerged for various software system attributes, e.g. architecture assessment for quality attributes such as ATAM [Kazman 1996], SAAM [Kazman 1998], and ALMA [Bengtsson 2002], as well as cost prediction with code metrics [Boehm 1981], and design-level metrics [Briand 1999]. An increasingly number of architecture assessment methods contains the following generic steps:

1. Set assessment goal
2. Construct scenario profile
3. Analyze impact based on the profile
4. Interpret results

As this approach has been successfully applied for assessing modifiability and maintainability [Bengtsson 2002], we considered the use of these steps for variability assessment. In the following sections, we discuss a straightforward application of this approach for variability assessment, and illustrate this approach by applying it to our case study. In the last section, we discuss several unresolved issues associated with such an approach.

6.2. Straightforward Variability Assessment Approach

6.2.1. Assessment Goal

Prior to starting the assessment, it is important to determine the goal of the assessment, as different goals pose different requirements on the results of the assessment. For variability assessment, goals depend on the context in which the assessment is used, i.e. during product derivation or during domain engineering.

- **Product Derivation:** At the product derivation level, variability assessment involves determining how well the product family is able to handle the differences imposed by the product at hand. As such, it can be of assistance in cost prediction, and planning for the project. In addition, variability assessment can be used to determine whether a difference should be handled product specifically, or by reactively evolving the artifacts.
- **Domain Engineering:** At the domain engineering level, variability assessment has a number of purposes. First, variability assessment may be of assistance in architecture improvement activities, for example for organizations that have a platform heartbeat that periodically incorporates functionality shared by a sufficiently large number of family members (reactive improvement). In addition, architecture improvement may be used during the design or later stages in the lifecycle of the product family (candidate selection and proactive improvement, respectively), in order to prevent variability mismatches. Furthermore, variability assessment may be used to assess which future events may pose risks. We note that these forms of predictive assessment closely resemble a needs-driven technique that is generally known as *Technology Roadmapping*. Basically, the technology roadmapping process delivers a road atlas for making intelligent choices related to R&D investments and marketing. Within the technology roadmapping process gaps are identified between the [essential] product, market or corporate needs and the technology available [Garcia 2002].

From these goals, only predictive variability assessment is directly related to our research question. In the discussion of a variability assessment technique in the (sub)sections below, we will therefore focus on predictive variability assessment for proactive architecture improvement.

6.2.2. Construct Scenario Profiles

A common aspect of many assessment techniques is the use of a *scenario profile* that specifies a set of scenarios [Bosch 2001b]. In case of modifiability assessment, these scenarios are for example descriptions of possible relevant changes to a software system. For a straightforward variability assessment technique, we choose the following definition:

Scenarios are descriptions of predicted essential product needs that emerge at certain moments within a particular timeframe.

As features focus on the essential characteristics of the domain and abstract from requirements they can be used to describe these essential needs (see also Chapter 3). These features emerge from the *time* and *space* dimension of software product families.

Time and Space Dimension

In traditional one-at-a-time software development, variability is regarded to evolve in one dimension, i.e. when a software artifact evolves over time. In software product families however, a second dimension, space emerges, since software artifacts appear in several products and evolve in that direction as well [Bosch 2001a]. To illustrate this, we constructed a product evolution map for the Prothos product family, in which we plot several products and their respective release dates and framework versions in time (see Figure 6-1). In this illustration, the inner squares are colored according to the product name and the outer squares according to the family generation they belong to (Prothos 1.*x*.*y*, 2.*x*.*y*, or 3.*x*.*y*). The colored diamonds represent the respective platform versions.

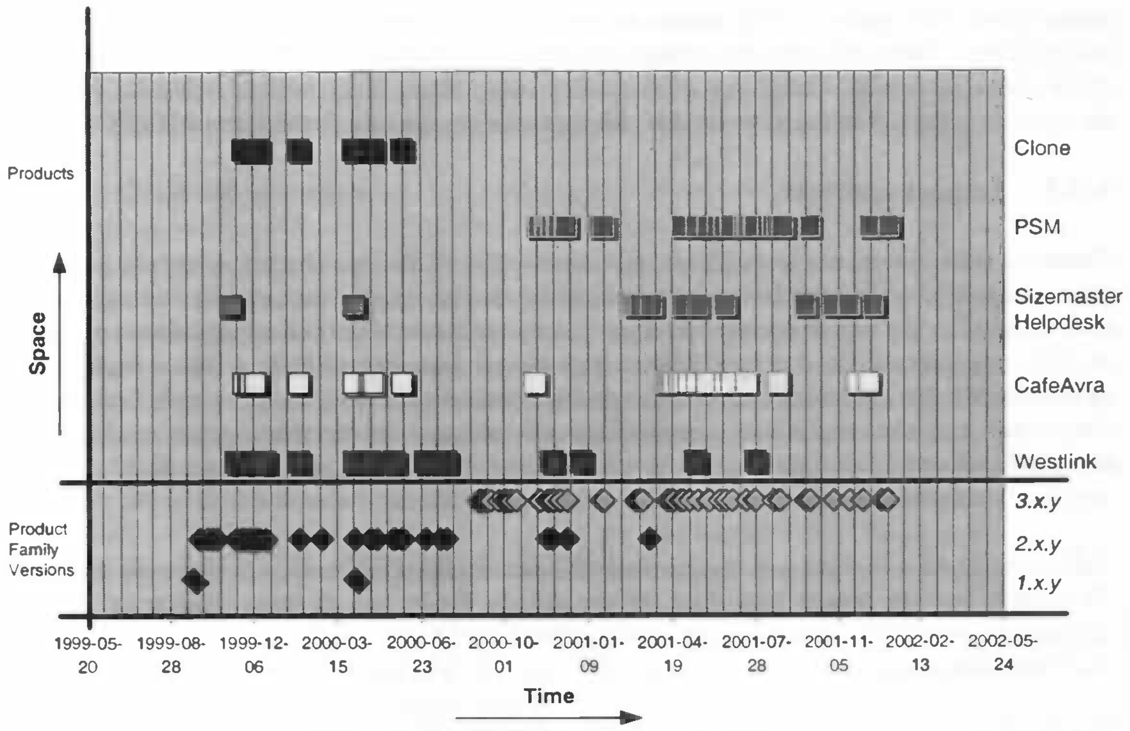

Figure 6-1 – Product Evolution Map for the Prothos Family

Complete vs. Selected Profiles

Scenario profiles can be either complete or selected [Bosch 2001b]. As evolution is concerned with the space and time dimensions, in case of predictive variability assessment, a complete scenario profile would encompass all possible evolution paths for all product family members within a certain timeframe (also known as the horizon [Ommering 2001]). We constructed a simplified representation of the evolution paths for a single product, where points in time are represented as a number of states in which at each moment in time choices lead to a different state (see Figure 6-2a). From this figure, it immediately becomes clear that, especially if we combine all possible evolution paths for all product family members, complete profiles would be far too large and complex.

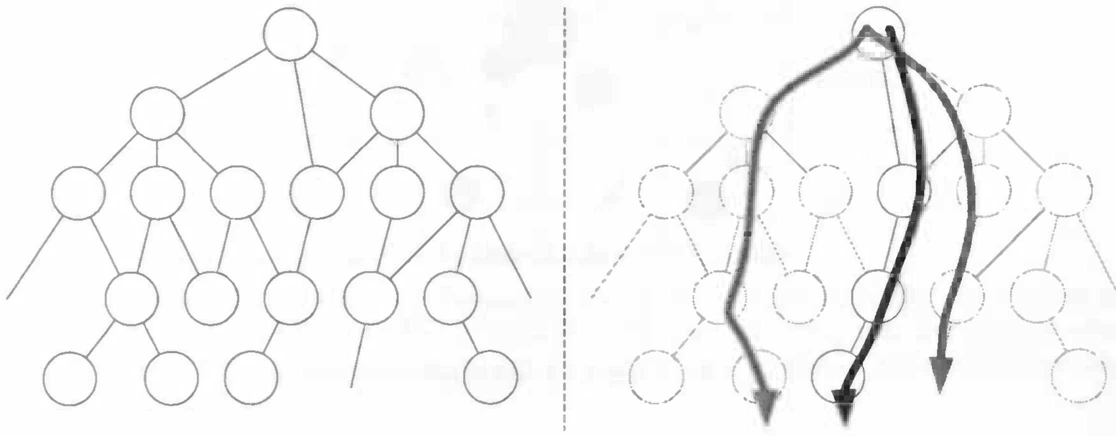

Figure 6-2 – (a) Complete versus (b) Selected Profile of Evolution Paths

The alternative, a selected profile, consists of a representative set of scenarios, e.g. the most likely evolution path of the product family, the most likely evolution paths for each product (as illustrated in Figure 6-2b), or a subset of possible evolution paths. Determining the most likely evolution paths highly depends on the experience and vision of the involved stakeholders. These stakeholders should involve representatives such as from the marketing department, architects, and in some cases even competitors. Competitors will most likely not be actively involved in the assessment, although exceptions do exist [Maccari 2002].

6.2.3. Impact analysis

Once the scenario profile is available, the next step is to analyze the impact of the profile. In case of variability, this boils down to comparing the variability required by the scenarios to the variability provided by the product family architecture. The straightforward impact analysis we present here uses a feature gap model [Deelstra 2002]. A feature gap model presumes that the mismatch between the provided and required variability can be mapped to a five-value scale that represents a normalized classification of the mismatches resulting from the profile. For our straightforward approach, this classification is the normalized effort for reactively solving the mismatch.

Additionally, in a feature gap model, features are weighted according to there classification. This classification ranges from key, to desired, to fancy and is quite similar to the three classes of software-functionality as distinguished by [Kano 1984], i.e. core, additional and visionary functionality.

Each scenario is then represented by a dot in a feature gap diagram and the moment in time is represented by the size of the dots. If a particular scenario occurs at T_x , $T_0 < T_x < T_n$, where T_0 is the start of the horizon and T_n the end of the horizon, the size is calculated as follows:

$$scale \cdot \frac{T_n - T_x}{T_n - T_0}$$

where *scale* is a fixed scalar used to scale all dots to an appropriate size. The result of such an impact analysis in a feature gap diagram would be somewhat similar to Figure 6-3, where the largest dots represent the features that are needed closest in time, and the left most lower corner denotes a scenario that does not represent a mismatch.

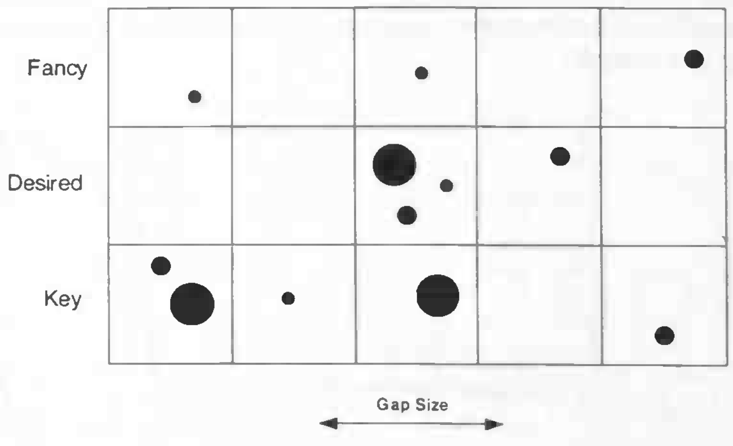

Figure 6-3 – Impact Analysis using an FGD

6.2.4. Result interpretation

Once the impact analysis is finished, we need to interpret the results to take the appropriate actions. This interpretation entirely depends on the initial goals of the assessment. In this case, we were interested in when and whether changes had to be made to the set of variation points in order to prevent possible mismatches. One approach could be, for example, to change the largest mismatches for a key feature that emerges closest in time first, since based on the information we have here, this ensures the highest certainty that the mismatch will actually occur and will save most effort during product derivation.

6.3. *Case Application*

We illustrate the presented approach using an analysis performed on a particular version of the architecture of the second generation of the Prothos Product Family (see Chapter 4). We based our scenario profile on historical data and informal discussions that had already been taken place with the chief architect of the Prothos family. In the scenario elicitation process, we focused on scenarios that actually occurred during six months of the Prothos lifecycle, rather then using predicted scenarios. The reason for this is that it is not our intention here to illustrate the accuracy of predictions, but to illustrate the approach and to determine whether the straightforward approach is really enough to perform sensible changes.

The process resulted in the selected profile that included the following scenarios:

	Description	**Month**
S1	Require additional behavior for the file-based database management variant for performance improvement	3
S2	Handle new 'kinds' of pages	6
S3	Access control for all page-handler variants, uniform for all rather then per variant	6
S4	Support multiple database connections for DBI database management variant	1
S5	Support optional database arguments for database connections in the DBI database management variant	3
S6 - …	Support new persistent business class variants	…

Using this profile, we performed our impact analysis, by classifying the effort needed to solve any resulting mismatches in the architecture version at hand. The results are illustrated in Figure 6-4.

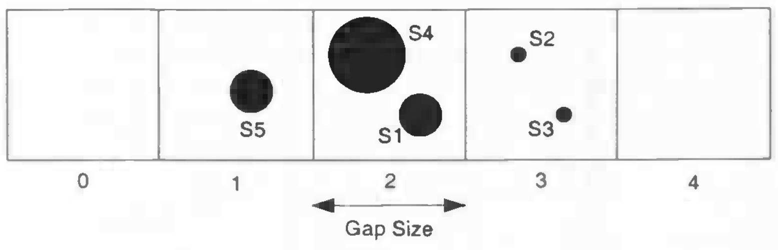

Figure 6-4 – Impact Analysis for Prothos

Note that we disregarded the feature classification, as most of the scenarios that resulted in mismatches were changes in Prothos that were really necessary. We also left out scenarios involving the differences in terms of persistent business class instances as these are typically well supported in the Prothos family.

6.4. *Problems and Issues*

The question that remains, is whether the analysis provides enough information to determine whether and when a mismatch should be solved. In this section we will show that the straightforward approach discussed above is a rather naïve approach. We identify several unresolved issues, which cause that this approach is not only not accurate enough, but also does not provide enough information to make an expert judgment.

6.4.1. Scenario definition

The first issue, involves the definition of the scenario profile. In the straightforward approach, a scenario was defined as a feature that was most likely needed at a certain moment within the assessment horizon. This definition, however, does not provide sufficient information to determine the mismatch between the provided and required variability, for a number of reasons. First, features typically have associated dependencies and binding times, which are needed to determine whether there will be a variability mismatch in the first place. Second, if we recall the variant feature type (see Chapter 3), variability is not only concerned with the presence or absence of a feature in different product family members, but also with handling multiple features in parallel in a single product. Finally, some optional or variant features are common to a subset of product family members. Being aware of this fact may turn out to be very important for our assessment (see subsection 6.4.2)

As features emerge are grouped in new product versions, consequently, we redefine a scenario as a product feature tree delta to the product family feature tree, including the dependencies and binding times.

6.4.2. Interacting scenarios in space and time

The second unresolved issue, involves interacting scenarios [Bosch 2001b]. The straightforward approach implicitly assumes that solving mismatches for one scenario has no impact on the possible mismatches caused by features from other scenarios. However, products in a product family are related to each other in the sense that different groups of members have different commonalities and differences. Therefore, it is highly unlikely that all of the scenarios from the profile only contain features that are alternative or variant to existing features supported by the product family and not to other features from other scenarios.

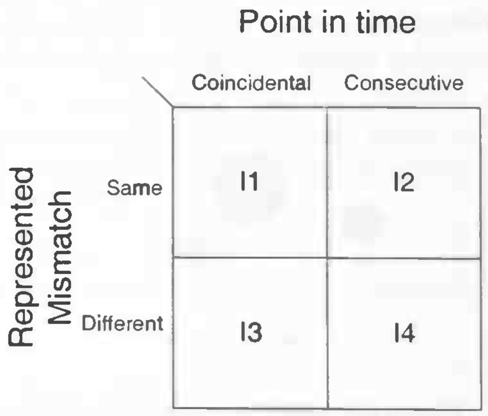

Figure 6-5 – Interacting scenarios

When considering the space and time dimensions, the scenarios may *interact* in the following four ways (see also Figure 6-5) :

(I1) Multiple scenarios require the same variability that should be provided by a particular variation point at (approximately) the same moment. In other words, multiple scenarios represent the same mismatch for a particular variation point at the same point in time.

(I2) Multiple scenarios represent the same mismatch for a particular variation point at consecutive points in time.

(I3) Multiple scenarios represent different mismatches for a particular variation point at (approximately) the same point in time.
(I4) Multiple scenarios represent different mismatches for a particular variation point at consecutive points in time.

Each of these four interactions has different implications on the accuracy and correctness of our straightforward assessment. First, as the impact in our straightforward approach is independently calculated for each scenario, (I1) and (I2) for example, would lead to double counting as the effort required to change a variation point is counted for all scenarios, while in practice the variation point only needs to be changed once (as was actually the case with scenarios S2 and S3).

Second, when multiple features result in the same mismatch, we may wonder if the amount of effort to solve the mismatch is enough to determine which variation point change should be prioritized. Consider, for example, the imaginary situation in which a mismatch costs four person hours, involves five scenarios compared to a mismatch that costs six person hours to solve, and only involves one scenario. Obviously, solving the first mismatch may deserve priority over the other, which calls for some form of weighing mechanism.

The interactions bring us further to a third important issue of our straightforward assessment, i.e. the issue of costs versus benefits.

6.4.3. Costs vs. Benefits

So far, all we have considered are the costs of reactively incorporating either independent scenarios, or scenarios sets that require the same variability. We illustrate that this is insufficient with an example from the Prothos case

In the POM version 2.0.2.07, several interface changes were needed, amongst others, an extension to handle multiple databases. In version 2.0.2.11, this extension was changed to handle optional arguments. (see also section 6.3, scenario S4 and S5)

Example 6.1 – Consecutive Mismatches

From this example, it becomes clear that it is not only a question of whether and which variation point we should change, but also *how* the variation point should be changed. Scenarios such as presented in the example, should lead to the consideration to group these so-called *compatible* scenarios and change a variation point at once rather then in several consecutive changes. In other words, compatible interacting scenarios present a case in which grouping particular changes to a variation point and handling them proactively at once is beneficial with respect to reactively changing the variation point for each scenario.

Product Specific vs. Shared

Rather then being compatible, however, scenarios can also *conflict* with each other. Consider for example two scenarios that have an impact on the same variation point, but the first requires a latest binding time at compile-time, and the second an earliest binding time at run-time. This not only iterates the importance of the timeline (see also 6.4.4), but also illustrates the fact that not all scenarios can always be incorporated in shared artifacts, and therefore some have to be implemented product specifically.

In fact, it cannot only be impossible to incorporate a change, but it can also be cheaper not to incorporate the change if it is not needed by other products anyway. As a rule of thumb, a change should not be incorporated, unless the costs and benefits of handling the change by evolving the shared artifacts outweigh the costs and disadvantages of product specific adaptation.

6.4.4. Timeline and Certainty

Related to compatibility and conflicts, the timeline aspect causes that the costs of solving a mismatch actually depend on the order in which the changes are executed. If two scenarios involve a mismatch to one variation point at consecutive points in time, a change to the first may influence the costs of the second change either positively (making it cheaper) or negatively (making it more expensive), or, in case of conflicting scenarios, determine whether the mismatches can be resolved in the first place.

A further concern is the timeline in relation to the certainty that the predicted scenario will actually occur. So far, we have stated that for scenarios in the near future, we are more certain that they will actually occur then for scenarios that occur in the far future. A good example of the fact that this assumption does not hold in all cases, is related to legislation (see Example 6.2).

> New laws are typically announced abundantly prior to the actual date at which they are enacted. Software systems such as car engine controllers, whose quality attributes depend on emission laws, for example, can be substantially more certain about scenarios involving those laws even if they do not occur in the near future.

Example 6.2 – Time vs. Certainty

Rather then just taking the point in time, a better approach would therefore be to consider the chance that a feature is needed by a product, where the time aspect is an influencing factor in determining the chance. This does not mean that the time aspect should be thrown away, however, since it is still needed to determine which changes should be prioritized.

6.4.5. Assessment Effort vs. Accuracy

A final important issue is the accuracy of an assessment in relation to the amount of effort spent for the assessment. The assessment relies on the expertise and vision of the involved stakeholders, and the criteria for selecting scenarios are based on expert opinion. Therefore, the assessment technique should deal with uncertainty, but prevent too much quantization and double counting of uncertainty margins.

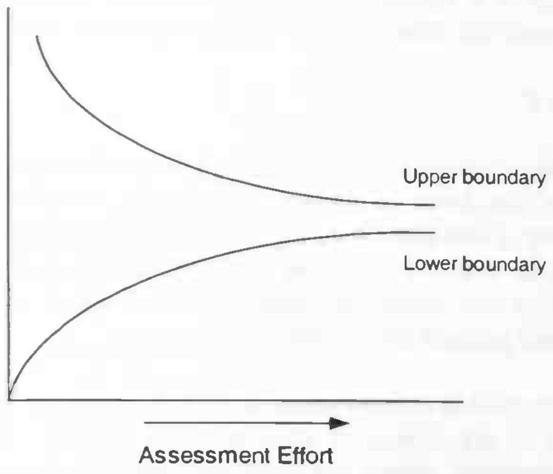

Figure 6-6 – Accuracy versus Assessment Effort

Furthermore, the scenario profile may grow quite large, particularly for assessments with a large horizon, and many product family members. It is most likely that the time available for the assessment is limited to some extent. Thus, a trade-off has to be made between the effort spent for the assessment and the accuracy of the results. Ideally, when more effort is put into the assessment, the accuracy should increase. As the assessment is based on predictions, however, the ideal situation will be an assessment with the ability to determine the upper and lower boundaries, which makes sure that the accuracy boundaries move as illustrated in Figure 6-6.

Chapter 7 – Conclusion and Future Work

On a high level, research in software engineering is focused around effectively dealing with the conflict resulting from the bigger-better-cheaper-faster market principle on the one side, and the increase in development costs and complexity of software systems on the other side (see Chapter 1). In this chapter, we summarize and highlight our contribution to this research goal. Additionally, we provide research directions for future work.

7.1. Summary

We started this thesis with a brief overview of reuse techniques that have emerged over the past 40 years. We continued with a description of concepts regarding variability as background information and related work. In chapter 4, we discussed our case study, i.e. the Prothos product family. In chapter 5, we provided the reader with a conceptual framework of evolution in relation to variability and we stated that, in our opinion, the lack of attention to evolution of variability is one of the main forces in the inversely proportional product derivation problem. We also formulated a number of research questions, which we addressed in chapter 6. In chapter 6, we presented a technique for variability assessment that we applied to the Prothos family, and discussed the major issues associated with such an approach.

7.2. Contribution

The main contributions of this thesis, we believe, are the following main points:

- We have discussed the inversely proportional product derivation problem.
  In section 5.1, we stated that organizations experience, both on individual product and product family level, a situation in which the percentage of required changes is inversely proportional to the resulting share in the total costs.

- We also have identified one important source of the inversely proportional product derivation problem.
  In section 5.2 we introduced the notion of a variability mismatch in terms of provided and required variability. In section 5.3 we provided a more detailed discussion on various mismatches.

- We have identified the importance of evolution as variability management task.
  This is the conclusion (section 5.5.) of our problem investigation in chapter 5.

- We have provided the reader with a framework of concepts for evolution of variability.
  In addition to the concept of a variability mismatch, we formulated an evolution law, which stated that variability should undergo continuous and timely change. We furthermore identified three types of evolution in order to deal with a variability mismatch, and provided several reasons for preventing variability mismatches.

- In addition, we discussed a technique for assessing the variability of a product family.
  We presented a straightforward application of a generic assessment technique to variability in chapter 6. We applied this approach to our case study and identified several problems and issues associated with this approach.

7.3. Further Directions

In addition to our contribution, naturally, still some issues remain. We will summarize the main open issues here.

- In chapter 5, evolution was pinpointed as one cause of the inversely proportional product derivation problem. Additional research is required to determine other important sources, some of which will be discussed in [Deelstra 2003].

- Also in chapter 5, we formulated several reasons for proactively improving the variability of the product family architecture. In addition to variability assessment as technique to assist in decreasing product derivation costs, however, several other techniques can assist as well. Rather then preventing mismatches, techniques could be proposed that decrease the costs of changing a variation point. Good separation of concerns and an explicit description of important dependencies, for example, should ease the process of finding all parts of the implementation of certain behavior to some extent. Especially related to the code scattering and tangling issues, Aspect Oriented Programming [Kiczalez 1997] shows a promising approach [Murphy 2001].

- Most open issues with respect to variability assessment have already been formulated in section 6.4. Of particular interest are aspects such as dealing with uncertainty and determining the accuracy of the assessment, as well as the costs and benefits aspects in relation to the interacting scenarios. Future work should aim to incorporate these issues, and validate this changed technique to a number of case studies.

- One issue related to our straightforward variability assessment has not been discussed so far. In our approach, we focused on proactive architecture improvement as assessment goal. The other assessment goals, however, impose different requirements on an assessment technique. For the assessment for proactive improvement as discussed in this thesis, for example, the impact analysis can be performed on an implemented architecture with existing dependencies. For candidate architecture selection in the design phase of a product family, however, we would have to make assumptions about implementation aspects. In addition, assessing variability for determining the change effort prior to product derivation does not use a predicted set of scenarios, but a very concrete set, i.e. the features that are needed for the product. In other words, more investigation is required to determine the effects of the different goals on the proposed assessment technique.

7.4. Concluding remarks

The work presented here is the result of the research for my Master Science project. Part of this research has been conducted at the Software Engineering and Research Lab of the University of Alberta, Canada. The other part has been performed at the Software Engineering and Architecture group at the Rijks*universiteit* Groningen, the Netherlands. The results presented in this thesis will form a basis for my Ph. D., under supervision of prof. dr. ir. J. Bosch.

Bibliography

In this chapter, we present a list of the literature and weblinks referenced in this thesis.

[AvraSoft 2002] Avra Software Lab Inc., http://www.avrasoft.com, March 2002.

[Anastasopoulos 2001] M. Anastasopoulos, C. Gacek, *"Implementing Product Line Variabilities"*, Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes, Vol. 26, No. 3, pp. 109-117, May 2001.

[Andert 1994] G. Andert, "*Object Frameworks in the Taligent OS*", Proceedings of the Compcon, 1994.

[Bachmann 2001] F. Bachmann, L. Bass, *"Managing Variability in Software Architectures"*, Proceedings of the Symposium on Software Reusability: Putting Software Reuse in Context, pp. 126-132, May 2001.

[Batory 1992] D. Batory and S. O'Malley, *"The Design and Implementation of Hierarchical Software Systems with Reusable Components"*, ACM Transactions on Software Engineering and Methodology, 1(4): pp. 355-398, October 1992.

[Bayer 2002] J. Bayer, R. Kolb, "*The Software Product Line Bibliography*", Revision 3.3, http://www.iese.fhg.de/PuLSE/Bibliography, November 2002.

[Beck 1989] K. Beck, W. Cunningham, "*A Laboratory for Teaching Object-Oriented Thinking*", OOPSLA'89 Conference Proceedings, October 1989.

[Bengtsson 2002] P.O. Bengtsson, N. Lassing, J. Bosch, H. van Vliet, "*Architecture-Level Modifiability Analysis (ALMA)*", conditionally accepted for the Jounal of Systems and Software, 2002.

[Berard 1992] E. Berard, *Essays in Object-Oriented Software Engineering*, Prentice Hall, 1992.

[Boehm 1981] B. Boehm, *Software Engineering Economics*, Prentice Hall, 1981.

[Bosch 1999] J. Bosch, "*Product-Line Architectures in Industry: A Case Study*", Proceedings of the 21st International Conference on Software Engineering, November 1999.

[Bosch 2000] J. Bosch, *Design & Use of Software Architectures, Adopting and evolving a product-line approach*, Addison-Wesley, ISBN 0-201-67494-7, 2000.

[Bosch 2001a] J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, H. Obbink, K. Pohl. "*Variability Issues in Software Product Lines*", Proceedings of the Fourth Workshop on Product Family Engineering, Springer LNCS, 2001.

[Bosch 2001b] J. Bosch, P.O. Bengtsson, "*Assessing Optimal Software Architecture Maintainability*", Proceedings of the 5th European Conference on Software Maintenance and Reuse (CSMR 2001), pp. 168-175, November 2001.

[Braind 1999] L. Briand, E. Arisholm, S. Counsell, F. Houdek, P. Thévenod-Foss, "*Empirical studies of Object-Oriented Artifacts, Mehtods, and Processes: State of The Art and Future Directions*", Empirical Software Engineering, Vol 4, no. 4, pp. 387 – 404, 1999.

[Buschmann 1996] F. Busschman, R. Meurier, H. Rohnert, P. Sommerlad, M. Stal, *A System of Patterns*, Wiley, 1996.

[Clauβ 2001] M. Clauβ, "*A proposal for uniform abstract modeling of feature interactions in UML*", accepted at the workshop on Feature Interaction in Composed Systems, ECOOP, 2001.

[Clements 2001] P. Clements, L. Northrop, *Software Product Lines, Practices and Patterns*, Addison-Wesley, ISBN 0-201-70332-7, 2001.

[Cusomano 1991] M.A. Cusomano, *Japan's Software Factories: A Challenge to US Management*, New York: Oxford University Press, 1991.

[Dahl 1966] O. J. Dahl and K. Nygaard, "*SIMULA, An ALGOL-based simulation language*", Communications of the ACM, 9(9): pp. 671-678, September 1966.

[Deelstra 2002] S. Deelstra, P. Sorenson, "*Effort Estimation based on Feature Gap Analysis in Software Product Lines*", accepted for 3rd International Workshop on Software Product Lines: Economics, Architectures, and Implications, ICSE, May 2002.

[Deelstra 2003] S. Deelstra, M. Sinnema, J. Bosch, "*Product Derivation in Software Product Families*", to be published.

[Dijkstra 1968] E. W. Dijkstra. "*The Structure of the THE Multiprogramming System*", Communications of the ACM, Vol. 11, no 5, pp. 341-346, 1968

[Flowers 1997] S. Flowers, *Software Failure: Management Failure, Amazing Stories and Cautionary Tales*, ISBN 0471951137, 1997.

[Froehlich 1997] Froehlich, G., Hoover, H.J., Liu L. and Sorenson, P.G. "*Hooking into Object-Oriented Application Frameworks*", Proceedings of the 19th International Conference on Software Engineering, pp. 491-501, May 1997.

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

[Garcia 2002] M.L. Garcia, Olin H. Bray. "*Fundamentals of technology road mapping*", Sandia National Laboratory, http://www.sandia.gov/Roadmap/home.htm, March 2002.

[Goldberg 1989] A. Goldberg, D. Robson, *Smalltalk – The Language*, Addison-Wesley, 1989.
[Gibson 1997] J.P. Gibson, *"Feature Requirements Models: Understanding Interactions"*, Feature Interactions in Telecommunications IV, IOS Press, 1997.

[Griss 1998] L. Griss, J. Favaro, M. d'Alessandro, *"Integrating feature modeling with the RSEB"*, Proceedings of the Fifth International Conference on Software Reuse *(Cat. No. 98TB100203)*, IEEE Computing Society, xiii+388, pp.76-85, 1998.

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

[Gurp 2000a] J. van Gurp, *"Variability in Software Systems: The Key to Software Reuse"*, Licentiate thesis, October 2000.

[Gurp 2000b] J. van Gurp, J. Bosch, "*Managing Variability in Software Product Lines*", LAC 2000.

[Gurp 2001] J. van Gurp, J. Bosch, M. Svahnberg, "*On the notion of Variability in Software Product Lines*", Proceedings of The Working IEEE/IFIP Conference on Software Architecture (WICSA 2001), pp. 45-55, August 2001.

[Hein 2000] A. Hein, J. MacGregor, S. Thiel, "*Configuring Software Product Line Features*", accepted for the Workshop Feature Interaction in Composed Systems, ECOOP 2001, Budapest, Hungary, June, 2001.

[Hoover 2000] H.J. Hoover, T. Olekshy, G. Froehlich. and P.G. Sorenson, *"Developing Engineered Product Support Applications"*, Proceedings of the 1st Software Product Line Conference, Published as Software Product Lines - Experience and Research Directions, P. Donahoe, ed., pp. 451-476, Kluwer Academic Publishers, 2000.

[IEEE 2002] Events in the History of Computing, http://www.computer.org/history, April 2002.

[IEEE1471 2002] IEEE Standard P1471-2000, "*Recommended Practice for Architectural Description of Software-Intensive Systems*", IEEE, 2000.

[Jacobson 1997] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture, Process and Organization for Business Success*, ISBN: 0-201-92476-5, 1997.

[Johnsson 1992] R.E. Johnsson, "*Documenting Frameworks with Patterns*", Proceedings of the 7th Conference on OO Programming Systems, Languages and Applications, Vancouver, Canada, 1992

[Kang 1990] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. "Feature-oriented domain analysis feasibility study". Technical report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, 1990.

[Kang 1998] K.C. Kang. *"Feature-oriented development of applications for a domain"*. Proceedings of the 5th International Conference on Software Reuse, pp. 354-55. IEE Computer Society Press, 1998.

[Kano 1984] N. Kano, N. Seraku, F. Takahashi, S. Tsuji, "*Attractive Quality And Must-Be Quality*", Quality Vol. 2 (in Japanese), pp. 39-44, 1984.

[Kazman 1996] R. Kazman, G. Abowd, L. Bass and P. Clements, 'Scenario-Based Analysis of Software Architecture'. *IEEE Software*, 13 (6):47-56, 1996.

[Kazman 1998] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson and J. Carriere, 'The Architecture Tradeoff Analysis Method', In *Proceedings of the 4th International Conference on Engineering of Complex Computer Systems (ICECCS98)*, Montery, CA: IEEE CS Press, pages 68-78, 1998.

[Kiczalez 1997] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. *"Aspect-oriented programming"*, Proceedings of the European Conference on Object-Oriented Programming, v. 1241, pp. 220-242, Springer-Verlag, 1997.

[Lehman 1997] M.M. Lehman, J.F. Ramil, P.D. Wernick, D.E. Perry and W.M. Turski, *"Metrics and Laws of Software Evolution — The Nineties View"*, Proceedings of the Fourth International Software Metrics Symposium (Metrics'97), Albuquerque NM, 1997.

[Macala 1996] R. Macala, L. Stuckey, , and D. Gross, *"Managing domain specific product-line development"*, IEEE Software, 13(3): 57-67, 1996.

[Maccari 2002] A. Maccari, *"Experiences in assessing product family software architecture for evolution"*, Proceedings of the 24th International Conference on Software engineering, Orlando, Florida, 2002.

[MacGregor 2002] J. MacGregor, Personal communications, June 2002.

[McIllroy 1969] M.D. McIllroy, *"Mass produced software components"*, Proceedings at the NATO Conference in Software Engineering, pp. 88-89, New York, 1969.

[Minsky 1975] M. Minsky, *"A framework for presenting knowledge"*, The Psychology of Computer Vision, pp. 211-277, P. Winstion (ed.), McGraw-Hill, New York, 1975.

[Mozilla 2002] The Mozilla Organization, http://www.mozilla.org, 2002.

[Murphy 2001] G.C. Murphy, R.J. Walker, E.L.A. Baniassad, M.P. Robillard, A. Lai, M. A. Kersten, *"Does Aspect-Oriented Programming Work?"*, Communications of the ACM, vol. 40, no. 10, October 2001.

[Ommering 2001] R. van Ommering, *"Roadmapping a Product Population Architecture"*, Proceedings of the 4th Int. Workshop on Software Product Family Engineering, LNCS vol. 2290, Springer-Verlag, October 2001

[Parnas 1976] D.L. Parnas, *"On the Design and Development of Program Families"*, IEEE Transactions on Software Engineering, Vol SE-2, no. 1, March 1976.

[Roberts 1996] D. Roberts, R. Johnson, *"Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks"*, Proceedings of the Third Conference on Pattern Languages and Programming, Montecillio, 1996.

[Royce 1976] W.W. Royce, *"Managing the development of large software systems: concepts and techniques"*, Proceedings of the IEEE WESTCON, Los Angelos, 1970.

[Rumbaugh 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-oriented Modelling and Design*, Prentice Hall, Englewood Cliffs, NJ., 1991.

[Sommerville 1982] I. Sommerville, *Software Engineering*, Sixth Edition, Addison-Wesley, ISBN 0-201-39815-X, 2001.

[Sparks 1996] S. Sparks, K. Benner, C. Faris, "*Managing Object Oriented Framework Reuse*", IEEE Computer, pp. 53-61, September 1996.

[Svahnberg 2000] M. Svahnberg, J. Bosch, "*Issues Concerning Variability in Software Product Lines*", in Proceedings of the Third International Workshop on Software Architectures for Product Families, LCNS, Springer Verlag, Berlin Germany, 2000.

[Svahnberg 2002] M. Svahnberg, J. van Gurp, J. Bosch, "*A Taxonomy of Variability Realization Techniques*", technical paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden, 2002.

[Szyperski 1997] C. Szyperski, *Component Software - Beyond Object Oriented Programming,* Addison- Wesley, 1997.

[Taylor 1992] D. Taylor, *Object-Oriented Information Systems,* John Wiley, New York, 1992.

[UML 2000] The OMG Unified Modeling Language Specification, Version 1.3, First Edition, http://www.omg.org, March 2000.

[Weiss 1999] D. Weiss, R. Chi Tau Lai, *Software Product-line Engineering: A Family-based Software Development Process,* Addison-Wesley, ISBN 0-201-694387, 1999.

[Webster 2002] Merriam Webster-Online, http://www.webster.com, June 2002.

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

List of Figures

In this chapter, we present a list of the figures used in this thesis.