Variation Mechanisms and Multi-view Architecting in Platform-based Product Family Development

Wijnstra, Jan Gerben

Chapter 8
Classifying Product Families using Platform Coverage and Variation Mechanisms

Abstract

The notion of product families is becoming more and more popular, both in research and in industry. Every product family initiative that is started within a company has its own context, such as a particular business strategy and a particular application domain. Each product family has its own specific characteristics that have to fit in with its context. In this paper we will describe two dimensions for classifying product families.

The first dimension deals with the coverage of the product family platform. Platform coverage deals with the proportion of the functionality provided by the platform, and the additional functionality needed to derive a specific product within the product family. The second dimension deals with the variation mechanisms that are used to derive a specific product from the generic platform. The coverage of the platform and the variation mechanisms used are not totally unrelated. We will discuss various types of platform coverage and variation mechanisms, including their characteristics.

These two dimensions are based on experience gained with a number of product families. We will look at four of these in greater detail to illustrate our ideas. We believe that these dimensions will aid the classification of product families. This will both facilitate the selection of a new product family approach for a particular context, and support the evaluation of existing product families.

8.1 Introduction

Over the years we have seen the size and complexity of software within embedded products increase rapidly. This trend is illustrated in [73], for example, where they describe how the increase of the size of the software in TVs has followed Moore's law closely, i.e. doubling in size every 18 months. We can also see that the variation that a system has to support is increasing. For example, in the domain of medical imaging systems, an increasing number of additional features are being provided so that the systems can be tuned towards the various procedures performed by physicians. We see that it becomes more and more important that such medical imaging systems can be optimally integrated in the workflow of the hospital, thus increasing the variation that these products need to support. On the other hand, in markets where there is a lot of competition, business demands a short time-to-market and low development costs, to ensure market share and profitability, and puts high demands on the quality of the software. The authors in [36] describe these trends in the following way: 'products and services that rest upon a software base must get their software faster, better and cheaper'.

The trends are drastically increasing the demands on software development. There are several ways to deal with these increasing demands. One solution is to reuse software across products. Product families[12] are based on the idea of reusing functionality across a range of products within one family.

8.1.1 Product Families

Within a *product family*, software and other artifacts (a generic term for documents, models, components, code, etc. used in the Unified Process [37], among other places) are reused across products that together form a family. Product family engineering is a structured way of working that has an impact on business, architecture, process and organization [116]. So, for a product family decisions must be made dealing with the architectural concepts of the product family, the way variation is handled, the type of reusable artifacts in the platform, the way the family members are developed, etc. We call this set of decisions the *approach* that is applied for a certain product family. When introducing a new product family, first the approach has to be defined.

---

[12] In software engineering literature product families are also referred to as product lines.

A company determines what the scope of the product family will be as part of the business strategy, i.e. which products will be part of the family. These products form a family because they have 'commonality', and share certain functionality. These products will also have certain differences. It is relevant to perform a domain analysis to identify these commonalities and differences. The results of such an analysis can be captured in feature models as for example described in [18]. For one of the product families discussed in this paper, a requirements object model was constructed [1], capturing the common and variable parts. The results of such an analysis are used to define the architecture of the product family. This must be an architecture that, on the one hand, enables reuse while, on the other hand, allows for differentiation between the products. Several techniques can be applied to enable this, like component frameworks with specific plug-in components [113]. A set of reusable artifacts can be defined in the context of the product family architecture. In this paper we use the term *platform* to refer to this set of reusable artifacts. A platform can range from a piece of infrastructure that is reused by the family members as a basis for further development, to a configurable system from which products can be derived by providing configuration parameters.

More and more product family initiatives are being started. We see this trend within Philips, where more and more product groups are applying this way of working. At international workshops and conferences, such as the Product Family Engineering workshop [54], we also see an increasing number of companies working with product families, or planning to do so in the near future. The goal that a company has for such a product family determines important requirements for the product family approach. The goal can be to make the current products more efficiently, or to increase the market in which the company is active. Other factors that influence the decision about the approach include the current way of working, the organization, and the experience with, and maturity of, the domain [116]. No single approach would be suitable for every situation. A product family approach must be tuned to suit the context in which it is to be applied. This explains why there is such a wide range of approaches.

8.1.2 Classifications

We can identify various papers in the literature that classify this 'wide range of approaches'. Examples are the classification into maturity levels [11], classification according to organizational structures [9], classification into architectural styles [81], and classification according to the binding time of variation [100]. Each of these classifications serves a specific goal. For example, the classification according to organizational structures can be used to

see what alternatives are possible and what type of organization would fit best in a certain situation. In section 8.7 we go into more detail on this and other related work.

In this paper we focus on the properties of a product family platform, since a platform forms the basis for the successful development of a product family. When developing a product based on the platform, we identify two dimensions of the platform that we will work on in this paper, namely the *coverage of the platform*, and the *variation mechanisms* that are used. These two properties are also important for the development of the platform itself, and affect its structure. A brief description of these two dimensions is given below:

- **Platform coverage.** By the coverage of a platform we mean how many of the artifacts are already provided by the platform in proportion to the artifacts that still have to be provided by the application engineering to derive a complete product. An example of a platform with a low coverage is one where for the product only a piece of shared infrastructure can be reused. An example of a platform with a high coverage is one that provides a configurable system from which a specific product can be derived via configuration parameters. The coverage that a platform has in proportion to the complete products within the family depends on several factors, such as the commonality between the family members; the more commonality, the higher the coverage can be.
- **Variation mechanisms.** In this paper we focus on behavioral variations that require different pieces of code to implement different behavior. Within a product family, one or more variation mechanisms are applied to realize the family members based on a shared platform. Examples of these mechanisms are configuration parameters that enable or disable certain functionality, or an architecture that specifies interfaces for components that can be optionally added to the system. The type of variation mechanism that is applied within a platform determines the way to make a specific family member, but also what the impact might be if planned changes in the platform have to be carried out, or if unforeseen requirements need to be supported, i.e. during the evolution of the platform. Several factors influence the choice of one or more variation mechanisms, such as the organization, or the maturity of the domain.

8.1.3 Goals of the Article

The dimensions described above can help when selecting a suitable product family approach in a given context, or when evaluating an existing product

8.2 Case Studies

In this paper we use experience gained with four industrial product families in the telecommunication and medical domains. These product families are:

- Telecommunication Switching System (TSS)
- Medical Modality System 1 (MMS1)
- Medical Modality System 2 (MMS2)
- Medical Imaging System (MIS)

The TSS product family deals with telecommunication infrastructure systems. This family is designed for niche markets with a large variety of features. A key requirement is to achieve a reasonable price even with a low sales volume. Products in this family include public telephony exchanges, GSM radio base stations, technical operator service stations, operator-assisted directory services, and combinations of these, such as a public exchange with directory services. When considering the software architecture, the most important concepts are components with clearly defined interfaces, component frameworks with plug-ins, and the layers architectural pattern with four abstraction layers. A product from this family usually consists of approximately 150 software components (excluding the proprietary operating system), grouped in about 25 functional units, resulting in a total of about 400,000 lines of code. The platform itself consists of about 300 software components, containing about 900,000 lines of code. More on TSS can be found in [50].

The other product families come from the medical domain. Philips Medical Systems builds products that are used to make images of the internal parts of the human body for medical diagnostic and/or interventional purposes. The different types of equipment used to obtain images are called ‘modalities’, for example Magnetic Resonance (MR), X-ray, Computed Tomography (CT) and Ultrasound (US). In this paper we use two product families for such medical modalities as case studies, called MMS1 and MMS2. The software architecture of MMS1 identifies about 25 units, each representing a specific area of functionality. Each of these units is realized via one or more software components. An important architectural concept is the use of component frameworks. The generic part contained in the platform comprises about 2.5 million lines of code. A complete product of this family comprises about 3 million lines of code. Further information on the architecture and variation mechanism for MMS1 can be found in [87] and [113]. For the MMS2 case, configuration parameters are used as a mechanism for achieving variation. The MMS2 architecture identifies six software subsystems. The platform contains

family. The dimensions are based on experience gained with a number of product families within Philips. We look at four cases in more detail to illustrate our ideas. Depending on the background and interest of the reader, this classification is relevant in the following ways:

- Various case studies are presented about product families in the literature. Using our classification, it becomes easier to compare and evaluate the different product families; it helps to understand the various interpretations of the product family concept.
- A prerequisite of introducing a product family is understanding what types of product family currently exist. The provided classifications will give you an overview of the range of possible product family approaches. We found that the coverage and variation mechanisms of a platform are very suitable to relate to the overall properties of the platform, both in the technical architecture areas and in the business, process and organization areas. We discuss the properties of different types of platforms in this paper. These properties can be used to see whether the platform matches the requirements of a specific product family.
- The introduction of a product family is usually done in a gradual way. For example, it can take a considerable time to populate the platform with reusable assets. In such a situation, the platform may start small and grow over time. The requirements on the product family may also change over time, possibly requiring changes in the platform. In this paper we also use the two dimensions to illustrate how various types of platforms are related, and which platform evolution paths are possible.

8.1.4 Overview of the Article

The remainder of this paper is structured as follows. Firstly, in section 8.2, we introduce the product families that are used as case studies in this paper. These are mainly cases from the medical imaging domain. In section 8.3 we give some context on what we mean by architecture in the context of product families. Sections 8.4 and 8.5 deal with the two dimensions mentioned earlier. In section 8.4 we present a model for describing the coverage of a platform, while in section 8.5 we discuss a classification of variation mechanisms. Section 8.6 deals with the relationship between platform coverage and variation mechanisms, and also how these classifications can be used when selecting a product family approach. Related work can be found in section 8.7, with our conclusions being presented in section 8.8.

about 3.5 million lines of code, which is the same as for the products derived from it.

These modalities, like the products of the families MMS1 and MMS2, also share certain functionality in the area of imaging. In order to benefit from these similarities and to increase the synergy between them, a family called MIS (Medical Imaging System) was created. MIS focuses on medical imaging functionality. The MMS1 and MMS2 platforms thus contain artifacts that are reused from the MIS platform. Components and interfaces are very important concepts in the platform of the MIS family. Well-defined interfaces support the integration of individual components in a larger system. The functionality of the platform is clustered in five groups containing about 50 functional components, and comprises over 1 million lines of code. The systems in which this platform is used can contain several million lines of code (just as in MMS1 and MMS2). You can find more about MIS in [53].

Table 8-1 gives an overview of some characteristics of the case studies.

	number of elements per product	lines of code in product	lines of code in platform
TSS	25 functional units, 150 software components	400k	900k
MMS1	25 functional units	3,000k	2,500k
MMS2	6 subsystems	3,500k	3,500k
MIS	5 clusters, 50 components	~3,000k	1,000k

Table 8-1 – Overview of the Case Studies

Although these families have different scopes and cover different domains, they share a number of important characteristics:

- complex, software-intensive products (several million lines of code);
- professional products; sold in small numbers, but with a lot of diversity;
- long lifetime and support of products (10-15 years);
- extensible with new features in the field; software updates in the field;
- a customer typically has a number of products from one family;
- relatively mature and stable domains; previous experience with such products.

The fact that they share these important characteristics makes it easier to compare them and to discuss the differences between them. When considering the two dimensions introduced in this paper, these four case studies cover the two-dimensional space well. This makes them valuable both for gaining

experience with these dimensions and for illustrating them. The fact that these cases have important commonalities while also having very different types of platforms (as we shall see later on), was the trigger to carry out the research presented in this paper. When comparing the different platforms for these case studies, we found that the platform coverage and variation mechanisms were the most suitable characteristics to describe the differences between the platforms.

We focus on behavioral variation for which different pieces of code are needed to implement different behavior. Examples for the TSS case study are the different types of signaling that can be used for the trunk lines between the switching systems, or the optional functionality of an automated time service. In the medical case studies, examples of variation are the different applications that the end user can buy, for example analytical functions for medical images or different image acquisition procedures.

You can find more information on the business, process and organizational issues of these case studies (except for MMS2) in [116].

8.3 Product Family and Architecture

In this section we will explain some of the basic concepts concerning product families and architectures that are used in the rest of this paper. Section 8.3.1 explains the meaning of architecture in the context of product families. Section 8.3.2 addresses quality attributes that are especially important for product family architectures. In section 8.3.3 we look at the weight of the product family architecture. Finally, in section 8.3.4 we list the artifacts that form part of a platform and discuss how they can be grouped.

8.3.1 Architecture in the Context of Product Families

We can find several definitions of architecture in the literature; in [4] and [35] for example. In [4], software architecture is defined as ‘the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships between them’. This definition focuses on the components and the relationships between them. However, architecture also involves overarching rules and guidelines for building the components and decisions about how the architecture is to evolve in the future, etc. This is why we think the definition given by the IEEE [35] to be more suitable, namely that an architecture is ‘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’.

Several activities play a role when building products within a product family. In [36], three activities are described, namely *application family engineering* (dealing with the commonality and variability analysis and definition of an overall product family architecture), *component system engineering* (dealing with the realization of reusable artifacts in the platform) and *application system engineering* (dealing with deriving an end product from the platform). We relate these three activities to three types of architecture, as shown in Figure 8-1. We will call the architecture defined in the application family engineering the *product family architecture*. It describes the rules and guidelines that must be applied within the family. This architecture is based on the results of a domain analysis in which the commonalities and variations within the product family are described. The rules of the product family architecture must enable reuse of platform components. For example, the product family architecture describes which variation mechanisms the application engineering can use to derive specific products from the platform. The component system engineering activity uses the family architecture as a basis for its development. Additional architectural rules and guidelines for the development of the reusable artifacts are captured in a *platform architecture*. The platform architecture is thus the architecture of the internal structure of the platform that is used to build the reusable artifacts. Similarly, additional architectural descriptions that are used to build individual products in the application system engineering activity are captured in the *product architecture*. One could say that both the platform and product architecture 'inherit' from the family architecture. The platform and product architecture match with each other in the sense that the result of the platform engineering activity can be used in the product engineering activity.

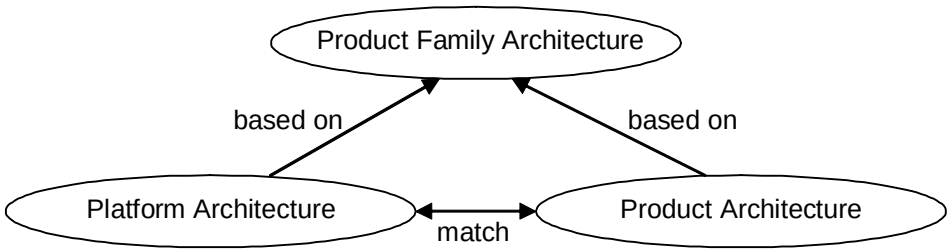

Figure 8-1 – Three Types of Architecture

In the context of this paper, the concept of *subdomains* within an architecture is important. By subdomain we mean a sub-area within a system that requires some specific knowledge, for example the image processing subdomain or the image acquisition subdomain. These subdomains may deal with application knowledge relating to the end user, or with technical knowledge relating to the peripheral hardware or computing infrastructure knowledge [116]. In the

context of design and realization, such a subdomain usually results in one (or more) subsystems/modules/components with one or more interfaces.

8.3.2 Product Family Architecture Qualities

Like any architecture, a product family architecture must support a number of quality attributes. However, when a product family architecture is compared with a 'normal' architecture, certain quality attributes are more important. The product family must provide a stable basis for the development of family members, on the one hand, whilst on the other hand it must allow enough variability now and in the future.

Stability

It is important that the points of variation within the platform, which are realized using variation mechanisms, are based on stable concepts. If this is not the case, the variation mechanisms may cost more than they bring in, since evolution of the platform will probably require many modifications in both the platform software and the product software. A thorough understanding of the domain is required to be able to find stable concepts. An important activity in this context is the modeling of the domain, as described in [1] for example. The why-what-how thinking model (see section 2.6 of [107]) can help when determining the stable concepts. A similar three step approach using three types of objects can be found in [27]; the Enduring Business Themes address *why* something is important, the Business Objects address *what* concepts are important, and the Industrial Objects *how* these concepts can be realized.

Variability

An important property of a product family is that the family members contain both similar and different functionality, both at point in time and over time. There are several quality attributes that support the creation of different products using a shared basis:

- *configurability*, i.e. the system can be configured to meet with specific requirements, e.g. using configuration parameters;
- *extensibility*, i.e. the system can be extended with additional functionality, e.g. using components;
- *composability*, i.e. a system can be built by composing various components with clearly defined interfaces.

In the context of this paper, we need to be able to distinguish between the three qualities as described above. These qualities can be related to different types of

platforms. For some platforms it is clear which products will be derived from them, and the functionality of these products is closely related. It is important that such platforms are configurable for the specific needs. Another group comprises platforms that form a basis for the products in the family, but where the family members require significant product-specific extensions on top of the platform. For such families it is important that the required extensions are enabled. A last group is made up of platforms that must be reusable in the context of a larger number of different family members. For such platforms it is important that the composability of the platform elements in different product contexts is taken into account.

Each variation mechanism focuses on one or more of these qualities, as is discussed in section 8.5. The stable concepts (as discussed in the previous section), variations and future changes must be known when defining the product family architecture together with the points of variation. This will give a stable basis for the product family. Back in 1972 Dijkstra identified the importance of a program structure that is prepared for the future:

> "... program structure should be such as to anticipate its adaptations and modifications. Our program should not only reflect (by structure) our understanding of it, but it should also be clear from its structure what sort of adaptations can be catered for smoothly." [20]

8.3.3 Product Family Architecture Weight

When defining an architecture, the architect must make architectural decisions that will enable a successful realization of the system. This includes decomposing the system into smaller parts that can be specified and implemented. Since each developer of such a part will usually focus on his/her own part, some decisions taken locally may not be beneficial to the rest of the system, and may harm the conceptual integrity (chapter 3 of [4]). This is why system-wide cross-cutting concerns must also be dealt with in the architecture. For example, if there is no single initialization strategy it becomes almost impossible to make a system that starts up correctly.

Should the architecture lay down all the decisions (*heavy-weight architecture*) or leave the developers some freedom (*light-weight architecture*)? In [65] the weight of an architecture is described as the sum of the weight of its rules. The weight of a rule is then determined by its level of enforcement, its scope, its size (text) and the number of dependencies to other rules. In [59] and [65] it is argued that the architecture should only contain rules and decisions that really have to be taken at system level, i.e. those decisions that cannot be deferred to a lower level. All other decisions should be taken at lower levels in order to allow

valuable contributions from the designers and to give them more freedom. This can increase the degree of acceptance of a new architecture.

This holds for architectures in general. But what does this mean in the context of product families and platforms? A platform is based on a product family architecture and provides a number of reusable artifacts such as software components. It might seem that development using a platform is even more restricting than a normal development, since ready-made pieces of code are already provided to the application engineering group. However, the perception of a platform depends entirely on the situation. One extreme is when the platform offers a number of components with basic functionality that can be freely reused for the development of a specific product. In this case, the product family architecture focuses on the rules and guidelines of how to integrate one or more of these components in a product, not on a specific combination of components. Such an architecture is lightweight, since a lot of freedom is given to product development to create the specific product. In fact, the platform allows the product developers to focus on the distinctive part of their product where the added value lies, not having to deal with basic functionality. Another situation is where the platform already prescribes the structure of the system, already containing a lot of components, and where a limited number of specific components are needed to derive a specific product. If the product development group previously created the products from scratch, this means that the freedom of the application engineer is restricted.

8.3.4 Platform Artifacts

When developing a product family, the component system engineering activity works on the platform artifacts, and the application system engineering uses these artifacts to derive products. As already indicated in section 8.1, these artifacts are not restricted to reusable software components. The artifacts provided to the users of a platform include:

- **Product Family Architecture.** A description of the architecture of the product family – relating to the hardware and software – must be provided for the users of the platform. It must also be clear how to use the platform, i.e. what the architectural rules and guidelines are. This architecture is required by the product groups in order to make products based on the platform.
- **Interfaces.** Interfaces are important elements of a platform; they introduce decoupling points in the architecture. Such interfaces can pertain to a subsystem or can even have a wider impact. An example of such an interface concept in the context of the telecommunication

switching system is the 'Information Bus'. This bus is realized in software and allows the addition of features to the switching system. Examples of such features are call forwarding or conference calls. This Information Bus allows the composition of existing features and the successful addition of new features via a standardized interface.

- **Components.** Software components in the platform can already provide the realization of stable and shared concepts. An example in the context of a medical imaging modality is the so-called geometry, which controls the major moving parts in a medical imaging system (e.g. the patient support on which the patient is lying), and determines the part of the patient to be examined and its projection on the image. Movements and positions are important items for the geometry. Among other things, the platform of the MMS1 case study provides a generic implementation for managing these movements and positions. The platform can also contain hardware components in addition to software components.
- **Other realization support and documentation.** In addition to the architecture, interfaces and components, other supporting means can also be provided by the platform. This support can be in the form of abstract base classes that need to be implemented, or tool support that generates code. Furthermore, as well as the architecture documentation, the platform can also provide documentation relating to areas such as the functional requirements specification.

The artifacts of a platform or product can be grouped in different ways. In this paper we use the five architectural views of the BAPO/CAFCR [70] reasoning framework for this grouping. This framework places architecture in the context of business, process and organization (forming the abbreviation BAPO). The architecture is further refined into the following five views CAFCR [70]:

- **C**ustomer (the customer's world)
- **A**pplication (applications important to the customer)
- **F**unctional (functional and non-functional requirements for a system used in a customer application)
- **C**onceptual (architectural concepts of the system)
- **R**ealization (realization technologies for building the system)

The first three views are also referred to as 'commercial views', since they deal with the *what* and *how* of the customer; the latter three views are also referred to as 'technical views', since they deal with the *what* and *how* of the product (see Figure 8-2). In addition to this model, other view models can also be found in

the literature, for example Kruchten's 4+1 view model [46]. We use the CAFCR view model because it has a broader scope than the 4+1 model. This is useful in the discussion of the platform coverage.

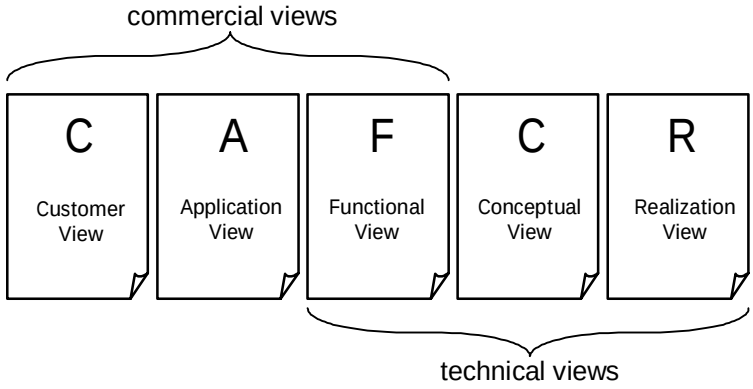

Figure 8-2 – Five Architectural Views

We will use these five views to group the artifacts of a platform. These views cover a broad range of issues, ranging from the customer's key drivers to realized components. Our focus is on the artifacts that are produced in relation to the various views, including the code. The two views on the left have to explain *why* the product is needed. Topics that are addressed here are the world of the customer, what is important for him/her, and the future trends. The applications that the end user needs and performs also have to be described. This type of information is written down in a Commercial (or Customer) Requirements Specification (CRS), for example. The Functional view describes the *what* of the product; what functions should be provided and what are the requirements concerning performance, reliability, etc. Such information can be written down in a System Requirements Specification (SRS) for the system as a whole, and in Functional Requirements Specifications (FRS) for certain functional areas. The Conceptual view describes how the system should be realized. For example, it contains a System Design Specification (SDS) in which the main concepts are described. Furthermore, Module Requirements and Design Specifications (MRS and MDS) have to be provided for the various modules in the system. In the Realization view, software modules written in programming language actually realize the system.

8.4 Coverage of the Platform

In this section we discuss the coverage of a platform. In section 8.4.1 we first discuss platform coverage and illustrate it using the four case studies. Section 8.4.2 then presents a classification scheme in which the coverage of a platform can be expressed.

8.4.1 Properties of Platform Coverage

In this section we explain and illustrate what we mean by platform coverage and how platforms can be classified according to this concept. For the classification we use the five architectural views of the BAPO/CAFCR reasoning framework as introduced in section 8.3.4. Each of the five views contains artifacts, which can be in the form of documentation, code, regression test scripts, etc. A platform that is used within a product family covers part of the CAFCR views. The product development must complete these views to obtain a product.

To quantify the platform coverage, we define the coverage of a platform as the size of the reused artifacts provided by the platform in proportion to the total artifacts needed to build an end product of the product family. This definition implies that it is only possible to describe the coverage in a platform in relation to an end product derived from the platform. For the case studies in this paper, the coverage of the platform related to a family member does not differ too much within a product family. For some product families it may be the case that the quantification of the platform coverage is considerably different for the family members in the family. This may, for example, be caused by differences in the functionality that is reused in the various products; for one product half of the provided platform functionality may be relevant, while for another product 75 percent of the functionality is relevant. It furthermore depends on how much functionality is added to build a specific product. Also for these cases, platform coverage is still relevant, since we will refine this concept to subdomains later on. This allows us to consider coverage properties for smaller parts of the platform.

The artifacts mentioned in the explanation of the views in section 8.3.4 are also used in the MMS1 case. Figure 8-3 shows, schematically, which artifacts are used in the various architectural views. We have also distinguished between artifacts that are made as part of the platform (gray) and additional artifacts that are needed to make the products (white). The lines between the platform and product artifacts indicate that the product artifacts build upon the platform artifacts. For this product family it was decided to make one CRS to cover all family members.

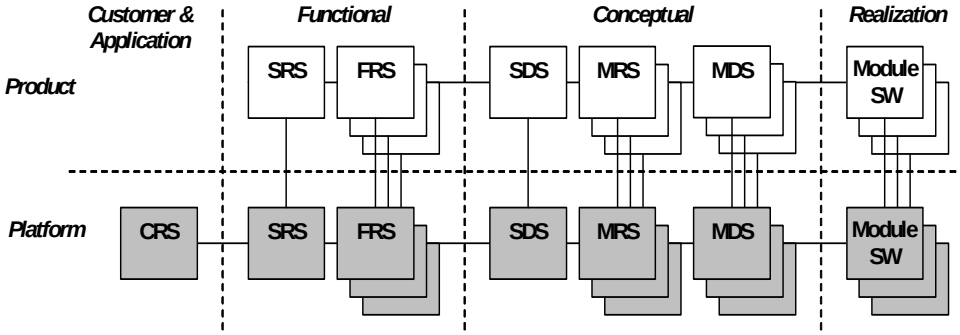

Figure 8-3 – Artifacts of the MMS1 Case Study

The figure above indicates that some of the artifacts are provided by the platform, and that the others are added to build a specific product. To get a better idea of what is provided by the platform and what has to be added to build a product, we also use the concept of subdomains, as introduced in section 8.3.1. By using this concept in combination with platforms, we can make a better distinction between the various types of platforms. Subdomains are identified in a family architecture. The architect together with other stakeholders can decide that the platform of the product family has to provide artifacts for all subdomains. However, to leave room for specific extensions, the subdomains do not have to be covered completely. On the other hand, the architect can also decide that the platform should only deal with a subset of the subdomains. The idea of these two directions of coverage is illustrated in Figure 8-4. This figure illustrates a product family in which six subdomains are covered or partly covered by the platform. The coverage of a platform in relation to a family member can be expressed as a single value, but to really understand the properties of the coverage these two directions of coverage must also be described.

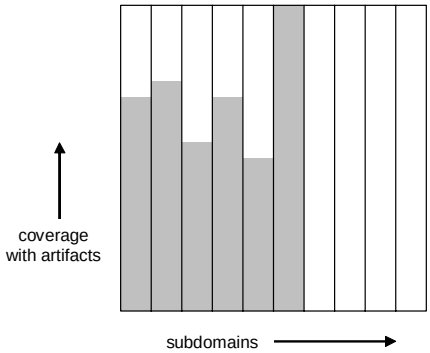

Figure 8-4 – Coverage of Subdomains in a Platform

When quantifying the coverage of a subdomain in relation to the various family members using this subdomain, the numbers will be more equal than when considering the coverage of the platform as a whole. The reason for this is twofold. First of all, when focusing on one subdomain, situations like a platform that contains subdomains that are not being reused by a specific product or a product that adds a lot of specific subdomains have no impact on the coverage number. Furthermore, subdomains are designed with a specific form of reuse in mind. This will in practice lead to more similar numbers for the size of the artifacts that are being reused in relation to the specific artifacts that have to be added for a specific product.

When the coverage of a platform is low, it is difficult for the platform to provide artifacts that deal with the end products as a whole. An example of such an artifact is a CRS that deals with the complete product family. This is especially the case when only a subset of the subdomains is dealt with. The integration of the artifacts will also be lower, because the contexts in which these artifacts are used may vary considerably. If the coverage is higher, the platform group can fill the commercial views more effectively. The integration between the artifacts can also be higher.

The coverage of the subdomains in two directions is related to the weight of an architecture, as discussed in section 8.3.3. A product family architecture describes which subdomains will be included in the platform. The platform provides partial or complete implementations for the subdomains. These subdomains can then be reused when developing a product within the family, so that less product-specific work needs to be done. The more artifacts are provided by the platform, the more the freedom of the product development is limited. So, a higher platform coverage results in a higher architectural weight of the product family architecture.

In addition to the coverage of a platform, the level of enforcement is also an important factor that influences the weight of the product family architecture. Although a certain subdomain is dealt with in the platform, it may not be mandatory to use this in an end product. An *optional coverage* indicates the artifacts in the platform that may be reused for a specific family member, but that are not mandatory. A *mandatory coverage* indicates that certain artifacts have to be used when building a specific family member. As discussed in section 8.3.3, the level of enforcement influences the weight of the architecture; a higher enforcement means a higher weight. As a consequence, a mandatory coverage results in a higher weight than an optional coverage.

The following sections illustrate these ideas on platform coverage with examples from the case studies. The information regarding the coverage of these case studies was collected from domain experts and the available

documentation. This information includes the subsystems that are identified in the product family architecture, the documents that are provided by the platform and additionally created in the product development, the lines of code of the software components in the platform, and the size of the additional code that is needed to derive a product.

MMS1 Case Study

The MMS1 product family architecture identifies subdomains of the family members; the platform deals (fully or partially) with each of them. Examples of subdomains in this product family are the image processing subdomain or the image acquisition subdomain.

The platform in the MMS1 case study already contains a lot of artifacts; the development groups can derive the specific products via identified variation points (both in the documentation and in the code). The development groups complete the Functional, Conceptual and Realization views to make the products by creating additional artifacts. For example, the generic SRS documentation contains an overview of the features of the various family members; the product-specific SRS selects the supported features and can add product-specific descriptions. In the system realization, the product groups add product-specific components within a larger family framework to realize the specific behavior. The platform already realizes about 75 % of the functionality, leaving 25 % to make a specific product[13]. The CRS document is written as one artifact as part of the platform. Such a shared CRS document for the product family provides a common direction for the product family development. The platform coverage for MMS1 is shown schematically in Figure 8-5. This is, of course, a simplified representation, and only gives us a global impression; the figure does not show the fact that some subdomains are completely covered and that others are only covered in part within the various views.

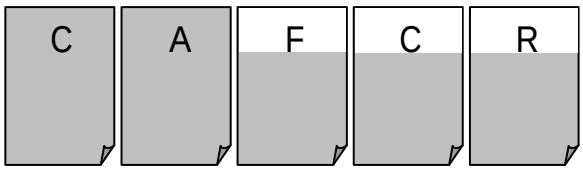

Figure 8-5 – Platform Coverage for MMS1

[13] We based out estimate on the lines of code, as this is relatively easy to measure. We see a similar relationship between the generic and specific documentation.

MMS2 Case Study

A different coverage holds for the MMS2 case. Here, the variation is realized in such a way that each end product contains the same software. The correct functionality for a product family member is activated via configuration parameters. In this situation, a development group develops the family as a whole (in fact, there is no separation between platform and product development groups). This leads to a 100 % platform coverage.

MIS Case Study

The third medical case study, the MIS, is quite different from the previous two. In this product family the focus was initially more on infrastructure functionality, and providing software components for this. The focus is currently shifting more and more towards application functionality. The MIS family is based on experience with existing systems; initially little attention was explicitly paid to the commercial views in CAFCR. The MIS platform only deals with a limited part of the products in the family, namely the imaging infrastructure. This means that this platform only covers a few subdomains of the products in which it is used.

In addition to the fact that not all subsystems are dealt with within MIS, another difference between the MIS case study and the first two case studies is the degree to which it is mandatory to use the platform artifacts to make a specific product. In the MMS1 case study the platform provides the basic system, which can be extended with additional specific functionality. In principle, each product uses the same basis. This is definitely the case for MMS2, where each product has the same software. One can say that in terms of section 8.3.3, the product family architecture is heavyweight. This is not the case for MIS, however. This platform is organized as reusable blocks of functionality, which can be reused if needed. To use such a component into a product, some configuration usually has to be done, and some additional code has to be written to fit it in the rest of the system. Only a small product family architecture needs to be prescribed (some rules and guidelines) to be able to use these components. This product family architecture is therefore lightweight.

TSS Case Study

It is important to realize that the coverage of the platform depends on the context in which it is used, i.e. for which set of products the platform is intended. If a platform is used in different ways, it is important to identify the different usages. This will help us to reason about the platform and the requirements that need to be fulfilled. The TSS platform has to support two types of usage, based on the two main groups of products made with it, namely public telephony exchanges and GSM radio base stations. These two groups can

each be considered as a product family; both product families are then based on the same platform. The TSS platform covers all[14] subdomains of the public exchanges; the development group can build new products by selecting the correct elements from the platform and adding some specific parts. For the public telephony exchange product family, the coverage is therefore very high. In this context we will refer to this product family as TSSpte. The radio base stations only use the part related to the switching functionality and not the application functionality. This switching functionality is delivered to the base station development group, which can add specific components to complete the switching functionality. They also build their application functionality. The platform coverage for the radio base stations product family is therefore low. In this context we will refer to this product family as TSSrbs. As a consequence, this platform has different faces, depending on the product family in which it is used.

8.4.2 Classification of the Various Platform Coverages

In the previous section we discussed two directions of coverage in relation to subdomains, as illustrated in Figure 8-4. We now use these two directions as a basis for a classification into four groups of platform coverage. We have also discussed the level of integration and the level of enforcement. We do not use these two properties explicitly in this classification, since it adds to the complexity of the discussion. Of course, when comparing different product families, we do take these properties into account.

Using the two directions of the subdomain coverage as illustrated in Figure 8-4, we can identify four main types of coverage, as illustrated in Figure 8-6 (the axes are running from complete to partial coverage, so that the traditional product development can be placed at the origin). These four types are as follows:

- **Complete coverage.** An example of this situation is the MMS2 case study. This platform completely covers all subdomains in the products. This means that all documentation and code is present to build a family member; the specific behavior is realized via configuration settings.

---

[14] It should be noted that several subdomains have been added throughout the lifetime of the TSS platform, e.g. dealing with automated speech generation. These extensions were integrated into the platform, with the result that the platform now covers all subsystems.

Such a platform coverage is possible when the family members have a lot of functionality in common. The specific parts can be specialized using configuration parameters. The advantage is that only one set of software components is needed for all product family members. Such a high integration of the generic and specific functionality increases the danger of the two not being optimally separated, making it difficult to extend the family with new functionality. In this situation there may be one group doing the complete development, or there may be a number of groups, each dealing with several subdomains.

- **Complete coverage of subdomains, but partial coverage inside the subdomains.** Examples of this situation are TSSpte and MMS1. In this situation the platform deals with all (or most) subdomains of a system, but they are not completely filled in. This type of coverage also enables the platform to provide artifacts for the commercial views of CAFCR.

  Such a platform coverage requires high commonality between the family members. The difference with the previous coverage alternative is that it is now easier to add specific functionality for a family member. In this situation a thorough analysis is needed of the commonality and variation, as stable interfaces (see section 8.3.2) must be defined between the generic part in the platform and the specific extensions. It is possible that a few subdomains are not covered explicitly by the platform, but the main idea is that the subdomains provided by the platform must be extended. This type of platform coverage allows the development of generic and specific functionality to be separated. However, it is also possible to develop the platform and products in one group, as in the TSSpte case, for example.

- **Partial coverage of the subdomains, but complete coverage within the subdomains.** In this situation the platform does not deal with all subdomains of a system, but the subsystems that are dealt with are covered completely. We have no case in our study that matches this situation.

  Such a platform coverage can be applied if a product family contains a number of subdomains that do not need to be tuned for specific family members. One can think of infrastructure-like functionality, dealing with logging, communication and other operating system-like functionality. This functionality is the same for the various family members.

- **Partial coverage of the subdomains, and also partial coverage within the subdomains.** Examples of this situation are MIS and

TSSrbs. In this situation the platform does not deal with all subdomains of a system, and the subdomains are also not completely covered.

Such a platform coverage is useful when the family scope is quite diverse and the family shares a limited set of subdomains. The platform can then provide artifacts for these shared subdomains. The product groups can make these subdomains specific for the various family members. Since only a subset of the subdomains is covered, it becomes hard to provide an integrated solution here. For example, it becomes difficult to make a good description of the commercial CAFCR views because the end products that are based on the platform can vary a lot. When a product has been derived, there is still a lot of work that needs to be done on the architecture. The product family architecture is usually limited to rules for combining the various subdomains into a product.

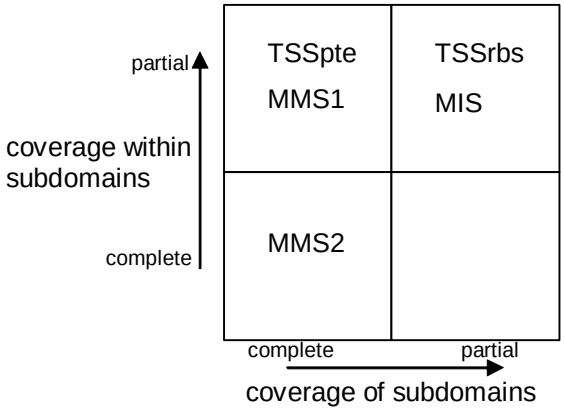

Figure 8-6 – Classification of Platform Coverage

8.5 Variation Mechanisms

In this section we deal with the various variation mechanisms that can be found in platform-based product families. In section 8.5.1 we give an overview of the variation mechanisms that we encountered. In section 8.5.2 we introduce two dimensions along which variation mechanisms can be classified.

8.5.1 Variation Mechanisms and their Properties

The variation mechanisms that we encountered in our case studies range from configuration parameters to composable components. We call this a range

because when moving from configurable components to composable components, less and less functionality is contained in the platform, and more functionality has to be provided via software components to make a complete product. For example, when only configuration parameters have to be set to make a product, no additional software components are needed. For a composable system, on the other hand, the software components needed to build the product depend on the product itself. In this paper we do not consider this as a single range. Instead, we make a separation between variation below the architectural level and at the architectural level:

- **Below the architectural level.** A non-configurable component, a configurable component or a component framework, can all be considered to be components at the architectural level. The architect must be aware of the variation that can be realized within such components, and must provide rules and guidelines on how to do this; the developers of the components realize the variation.
- **At the architectural level.** At the architectural level, components and interfaces are important entities, along with rules and guidelines. The product family architect can identify the components from which the system should be built. If this is too strict, it is possible to define a basic platform to which the development groups can add their own components, preferably via predefined interfaces. A third possibility is to capture important architectural concepts in interfaces. The interfaces should then allow new components to work with each other, even if they were not identified at the beginning. This allows the addition of new components in the system.

In section 8.5.2 we will return to the separation into these two levels of variation mechanisms. In the remainder of this section we describe the variation mechanisms that we encountered on the two levels. This description is an extension of the variation mechanism overview presented in [53]. For the variation within the components, we have:

- **Fixed component.** Such a component does not support any variation. Other components are needed within a product family to realize the variation.
- **Configurable component.** In this situation the variation has already been realized within the component. The desired variation is obtained by selecting the appropriate values for the configuration parameters. The configuration parameters can be used to enable/disable parts of code. The parameters can also be used to influence the behavior of certain algorithms.

The configuration mechanism allows the same set of software components to be used for different family members. This set of software components can be configured in different way, e.g. via license keys. When new functionality has to be supported, the components affected must be modified and recompiled. It is important to point out here that the designers should try to clearly identify which part of the code within the components relates to which functionality/features; this makes maintenance easier if changes or extensions are needed.

- **Extendible component.** An extendible component consists of a base that can be extended with additional subcomponents. One such base component is a component framework. In this situation, parts of functionality are identified as being variable and are factored out into separate components (plug-in components). This idea is similar to the configurable components, except that the variable code is put in separate subcomponents that can be added or removed; the generic and specific functionality are separated by means of a clear separation of stable and variable code. Another example of such a base component is a class framework, which can be specialized using inheritance, for example.

  In the case of a component framework with plug-ins, it is possible that the plug-in components are already provided by the platform. In other situations, the application engineering activity must develop the plug-ins itself. One important cause of this difference is the question of how specific a certain plug-in is; if it can be used for several family members, it is more likely to become part of the platform.

  This mechanism allows a more flexible extension of the functionality than the previous one. Separate product groups can develop the additional subcomponents. An important point for consideration here is the definition of the interface between the component framework and its plug-in components. If this interface is not suitable, this mechanism can give rise to a lot of additional work. Such a mechanism should therefore be applied for subdomains that are already well understood. The concepts used in this interface should be stable, even when looking towards the future (see section 8.3.2).

- **Abstract component.** In this situation only the interface of a component is specified; the product groups must provide the implementation. This mechanism offers the most flexibility of the four. The disadvantage is, of course, that the product groups cannot reuse an implementation provided by the platform. When changing such an

interface, the impact must be considered by taking the current component implementations into account. This variation mechanism is appropriate when the implementations for the various family members are too different; in such a situation a generic implementation would have more disadvantages than advantages.

When comparing these four variation techniques, the first one is of course the simplest, since there is no variation to deal with. When all variation is known, configuration parameters can be used. However, if not all variation is known or will not be implemented in the platform, mechanisms like component frameworks or abstract components are possible. Of these two, the abstract components are the easiest for the platform group to provide. An external interface must be defined here. When a component framework is defined, an internal interface must also be defined between the component framework and its plug-ins.

For variation within the architecture we distinguish:

- **Fixed components.** In this case, the components for the family members are fixed and a (partial) realization is provided for them. The variation must come from the variation mechanisms mentioned above. This way of handling variation is only possible if all the family members have the same subdomains. When more variation is needed, the following variation mechanisms at architectural level come into play.
- **Additional components.** In this case, the structure of the architecture is also reasonably fixed. The majority of the components are identified in the product family architecture and the platform contains a (partial) realization for them. However, it is still possible for the product development groups to add extra components to the system. The product family architecture usually describes the interfaces to which these extensions must adhere. Such an architecture affords greater freedom to the developers of the family members than with fixed components. It can be applied when a large number of subdomains are shared, allowing the definition of the base architecture. Additional components can be used for the specific subdomains of the family members.
- **Composable components.** In this case, the family architect defines the components in such a way that they can be composed with other components, possibly from outside the platform, to form a product. Compared with the previous mechanism, more flexibility is provided. Instead of using a fixed set of components that can be extended, the application engineering activity builds its products by composing the

relevant components. This mechanism is useful when the members of the family share a limited group of subdomains, leading to a group of products that is sometimes referred to as a product population [72]. With this way of working it is especially important to focus on the interfaces of the components, since that is the main concept for building a product family member.

8.5.2 Variation Classification

In the previous section, we introduced two levels at which variation mechanisms can be applied: at the architectural level or below the architectural level. These levels are based on the separation between components that are identified by the architect and laid down in the product family architecture (architectural components), and components below this level. These latter components are part of the architectural components. One might argue that these two levels are similar, except for the different level of granularity of the components. However, we chose to distinguish between these two levels for several reasons.

First of all, the variation mechanisms that are used on the architectural level have a major impact on the process and organization that are used to build the products. When all variation is handled within the components that are identified within the architecture, the basic structure of each of the products is fixed. Variation can 'only' be achieved by using variation mechanisms within the components. At the other end of the scale, we find the products that are built from composable components. The architecture of the family leaves much more freedom on which components are used and which products are built. One can image that this requires other processes and a different organization than the case in which all components are prescribed by the architecture. More on this topic can be found in [116]. Furthermore, by using these two levels, the characteristics of the various case studies can be expressed more clearly.

In section 8.3.2 we presented three qualities related to variation. The two levels are also related to the distinction between configurability, extensibility and composability. Configurability, and part of the extensibility, can be realized within the components defined within the architecture. Composability and extensibility deal with components as a whole that are relevant at architectural level.

The two dimensions are shown in Figure 8-7; the vertical axis concerns the variation within components, and the horizontal axis concerns the variation of components as a whole at architectural level, as discussed in the previous section.

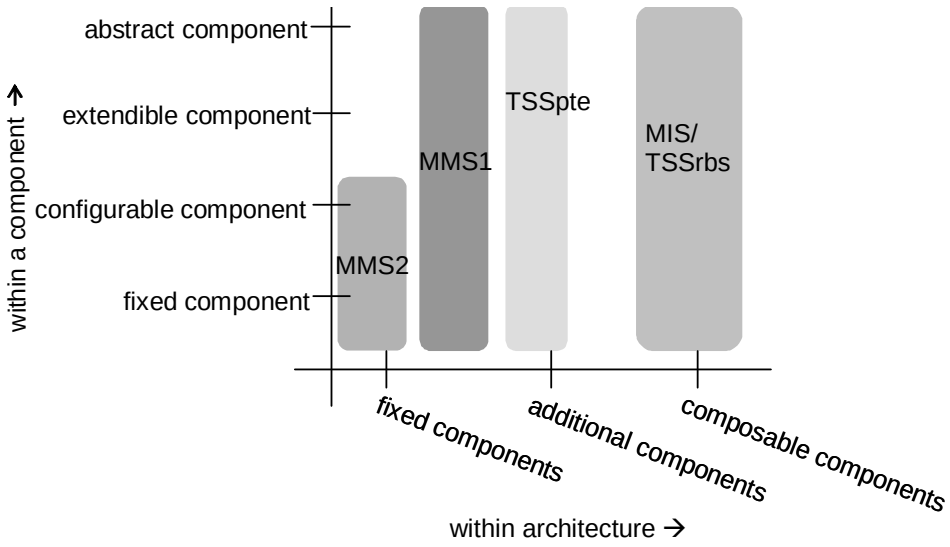

Figure 8-7 – Variation at Two Levels

In the following paragraphs we give an overview of the variation mechanisms used in the various cases. They are illustrated in Figure 8-7. Information on the variation mechanisms can be found in the documentation of the product family architecture or by consulting the architects.

The MMS2 case study has an architecture that defines all components in the system. Some of these components are fixed and others can be configured with parameters. In the MMS1 case, both configuration parameters and component frameworks are used at component level in addition to fixed components. There are also a few components for which only the interfaces are specified. In principle, all components are already identified in the architecture, and all family members contain all these components.

The TSSpte case (related to the public switches) is similar to the MMS1 case. Here, configuration parameters and component frameworks are once again applied. The relevant components can be selected from the 'construction set', a set of all components in the platform. Newly developed components are integrated into the construction set. Since the new components are also integrated into the construction set, not all product family members will contain all architectural components. This means that some components are removed from the platform to construct a family member. This is why this family is located somewhat more to the right, towards the additional (and removable) components.

The MIS case study focuses on the right-hand part of Figure 8-7. The MIS platform can be considered to be a ‘component kit’, from which the product groups can select the relevant components. It therefore allows the addition of new components and the omission of unusable components. Various variability mechanisms are applied within these components. When components from the component kit are used, only a few infrastructure-related interfaces have to be realized in the system, i.e. replaceable components with a fixed interface. The TSS platform for the GSM base station, where the TSS platform only provided switching functionality, can be compared to the MIS case study.

Figure 8-7 shows us that different types of variation mechanisms can be applied within a product family. The case studies presented in this figure use several variation mechanisms within the components of the architecture. For example, in the MMS1 case study, component frameworks with plug-ins, configurable components and abstract components are used to reach the required variability. On the architecture axis, one approach is usually chosen for dealing with variation. This is an architectural style of the product family architecture. For each of the four cases, we identified one approach on the architectural level for dealing with variation. This will probably also apply to most other cases. However, it is possible that different architectural styles are applied for different parts of a system. Distinguishing between the two levels of variation helps to compare the various cases in such a figure.

8.6 Platform Coverage and Variation Mechanisms

In section 8.6.1 we discuss the relationship between the platform coverage and variation mechanisms. We then look, in section 8.6.2, at how coverage and the variation mechanisms are two classifications that can be used when selecting a suitable product family approach. In section 8.6.3 we discuss introduction strategies, based on the classifications introduced.

8.6.1 Relationship between Platform Coverage and Variation Mechanisms

When comparing the coverages of the various platforms and the variation mechanisms used, we identified dependencies between these two dimensions. The highest coverage was provided by the MMS2 platform. The variation mechanism used there – configuration parameters – is very close to the fixed components. The coverage of the MMS1 platform is smaller. The mechanisms used here are mainly component frameworks and abstract components, in addition to configuration parameters. The coverage of the TSSpte platform is

also smaller and, compared with the MMS1 platform, additional components in the architecture are also used. The MIS platform and the TSSrbs platform had the smallest coverage. They mainly use the composable components.

In the coverage of a platform we distinguished between the coverage of the subdomains and the coverage within the subdomains. If a platform covers all or most of the subsystems, this is likely to result in a platform architecture that prescribes how the various subdomain functionalities will be connected together, leaving less room for adding or removing components (i.e. located on the left-hand side of the architecture axis). So, for a platform that covers all subdomains, the variation mechanisms shown on the vertical axis of Figure 8-8 become more important. On the other hand, the product family architect of a platform with a low coverage of the subdomains must ensure that the components related to the subdomains can be integrated easily into the various products. This is where the addition and replacement of components comes into play. However, it is probably still necessary to customize the components for the specific products, using configuration parameters and component frameworks. The degree of coverage is therefore influenced both by the component and the architecture variation mechanisms. This is illustrated graphically by the diagonal line in Figure 8-8. This line does not imply that all product family approaches must be located on this diagonal; it is merely an indication of the overall coverage trend.

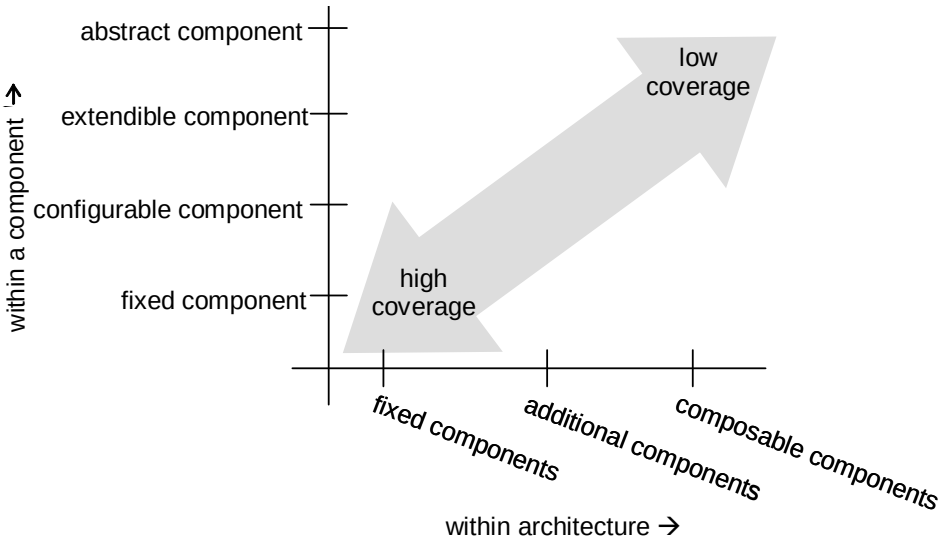

Figure 8-8 – Platform Coverage and Variation Mechanisms

8.6.2 Selection of a Platform Approach

When initiating a product family based on a platform with reusable artifacts, it is important to first determine what products will be covered by the family, i.e. what is the scope? The family architect must determine the contents of the platform. This must be based on a commonality analysis, where the common and different items are identified. Both the functional and non-functional commonalities must be considered. For example, all family members may need image processing, but are the performance requirements similar enough to develop it as part of the platform? The scope of the family and the contents of the platform together determine the coverage of the platform, as discussed in section 8.4.1.

If the various family members have a lot of specific functionality compared to the generic functionality, the platform will have a low coverage. This is especially the case where the family members only share a limited set of subdomains, so that the platform only covers a subset of the subdomains needed to make a specific product (the platform can, of course, support subdomains that are relevant for a subset of the family). In such a situation, the platform will be located to the right in Figure 8-8. The various subdomains then relate to the composable components, making it easy to select the relevant subdomains from the platform and to add specific subdomains.

On the other hand, if the family members share a lot of functionality, the platform will have a high coverage. If the subdomains are shared for the various family members, the platform can already provide an integrated solution in which all major subdomain components are already present. This should make it easier to create specific products. Variation can be provided via specific variation mechanisms.

Figure 8-9 plots the four coverage categories, as identified in 8.4.2, roughly into the two-dimensional variation space. It should be noted that a platform that partly covers the subsystems (the top two categories in Figure 8-9) can also contain subsystems that are covered completely. In that sense, such a platform can also contain configurable components or fixed components.

In general, the coverage of a platform is related to the size and diversity of the product family: the larger and more diverse it gets, the harder it is to make a platform with high coverage for it. As a consequence, more specific software needs to be added to make the specific products. The diversity of the product family over time is an important consideration when selecting the variation mechanism. For a large diversity over time, additional components are needed. For a lower diversity over time, the platform can be positioned more in the left-hand part of Figure 8-9.

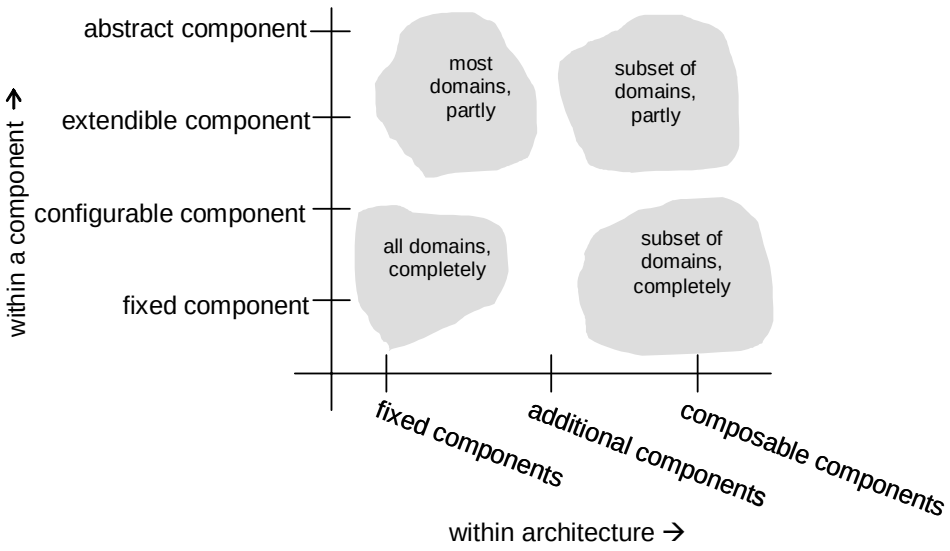

Figure 8-9 - From Coverage to Variation Mechanism

The question of which platform coverage or which variation mechanism to choose for a certain situation is difficult. We have given some indications about the properties of platforms related to product families. The specific case will determine which type of platform is the best.

8.6.3 Introduction Strategies

When introducing a product family, it is not only the technical issues that have to be considered [116]. An important part of the success of the introduction is determined by the acceptance of the people developing and using the platform. This acceptance is influenced by several factors. One is the change that the new way of working introduces compared with the existing situation; the bigger the change, the lower the acceptance is, in general. Another important factor is the amount of freedom that the product developer has; if this decreases, so does the acceptance [59]. Other factors include a high initial investment and the absence of early feedback [3]. Knowledge of the variation in the domain and how this can be realized is also very important for the introduction. During one of the sessions at the 4th Product Family Engineering workshop in Bilbao (October 2001, [3]), attention was paid to lightweight introduction of a product family. The conclusion was that, 'a gradual approach seems to work better'. This means that each change must be limited in size.

Bearing these factors in mind, we propose two ways of introducing a platform. These two introduction strategies tackle the problem from different directions.

They are both based on a specific starting point in the two-dimensional space introduced in Figure 8-7, and follow a probable evolution path. The two introduction strategies are:

- from single product to product family;
- from basic reusable components to product family.

These two strategies are shown schematically in Figure 8-10. We will elaborate on them below. The assumption here is that applying reusable artifacts for the selected domain is relatively new to the company. If there is already a degree of experience, this experience can be used to make a bigger step when introducing a new product family. However, our experience is that one should be careful not to make the step too big, since a lot of (possibly unforeseen) changes already have to be managed.

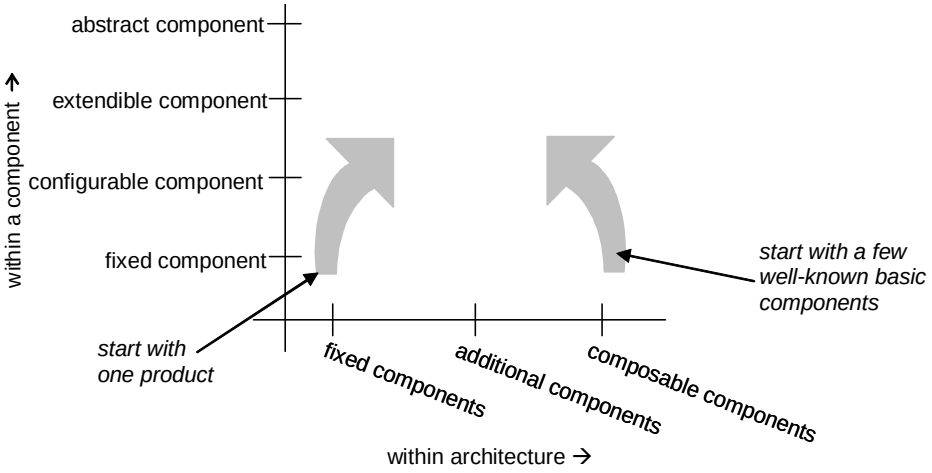

Figure 8-10 – Two Introduction Strategies

The first strategy starts at the bottom left in Figure 8-10. Here, one product is used as a basis for the development of the platform. The reason for starting with a certain product is that the product is already made and there is already knowledge about the domain and the implementation of the product. This knowledge is important to support the variation when the family grows. As discussed with the component frameworks, for example, a thorough knowledge of the domain is required to define stable interfaces that allow the required variation. The direction in which the platform grows depends on the context. This introduction strategy is applicable to platforms with a relatively high coverage.

The factors that influence the acceptance of the product family, mentioned at the beginning of this section, are taken into account as follows. By starting with

a product and then expanding the scope, the development group obtains more knowledge on the variation in the domain and how to realize it as it evolves. Furthermore, changes to the way of working are limited. It is possible for the group that makes the product to also make a platform based on configuration parameters, component frameworks or replaceable components. The change consists mainly of using additional mechanisms and a shift from making one product to several products. This change can be made gradually. For example, if everything works out, a separation can be made between a platform group and one or more product groups. Since the same group can initially make both the platform and the products, the developers do not feel they are restricted in their freedom when making the design. An advantage of this strategy is that it has a clear focus, i.e. a product has to be realized. We noticed in the MMS1 case study that having a concrete product that had to be realized gave the project a relevant focus.

This first introduction strategy can be found in [118], among other places. This is the paper that describes the MMS2 case study. MMS2 started out as a single product. Over time, more and more variation was added. This was realized using configuration parameters. The next step, as described in [118], is to use more components that contain features that can be included in a product or not. In this way, the increasing variation can be better managed, in the sense that separate features are realized in separate components.

The second strategy starts with a few well-known basic subdomains. Such a strategy can be applied when there are several products with a similar functionality, but where there is no sharing of software yet. This was the case for the MIS platform. Based on the domain knowledge for the various products and an analysis of the commonalities, a number of shared components were identified, such as components for printing and archiving of medical images. The first components had fairly straightforward functionality. This enabled a focus not only on the shared components, but also on the new way of working. The various product groups could then reuse these components in their products. As this way of working was developed further over time, more components were considered for the platform. When more is clear about the variation and the members of the family, more variation mechanisms can be applied. This can lead to a greater coverage and an architecture with a higher level of integration.

This strategy also takes into account the factors that influence the success of introduction. Just as for the first strategy, more knowledge on the domain is obtained during the evolution of the platform. The way of working for the various product groups will only change slightly. Instead of each group having to make everything themselves, the platform provides basic functionality that can be reused. In this way, the product groups can put more effort into the

differentiating application functionality. The freedom is therefore initially restricted, in the sense that the basic functionality has already been provided. However, this can also be seen as an opportunity to work on the more challenging issues in the application area. The main change introduced by this strategy is that a platform group is needed to make the shared components (variants exist where the responsibility for these platform components is divided over the product groups).

Both strategies have in common that the introduction impact is limited, and that early feedback and results are obtained. Early feedback is important to learn how product family development works and to get acceptance within the organization. The first strategy starts with a limited number of family members; the second one starts with a limited number of subdomains.

These two introduction strategies both have a starting point from which they will evolve further. A platform for a product family therefore has to evolve over time. When defining such a platform, the direction in which it will probably evolve also has to be taken into account, thus preventing superfluous work. For example, if an abstract component is expected to be the most suitable alternative for a given component, there is little point in defining a component framework for such a component that will only be used for a short while.

8.7 Related Work

The work presented in this paper is related to several topics that can be found in the product family research. In this section we will address scoping, classifications of product families, and introduction scenarios.

In [10], two types of scoping are distinguished: the selection of products that can form a product family, and the selection of the features that will be included in the product family. In the context of that paper, this last type of scoping can be considered as determining which reusable artifacts should be included in the platform. The process of determining both types of scope is an iterative one; based on the identified products the shared features can be identified, and based on the features candidate products can be identified. Increasing the set of products that form a family leads to a smaller platform coverage; the commonalities will decrease and the differences increase. The more common features that can be identified, the larger the coverage of the platform can be. So, both types of scoping have an impact on the coverage of the platform. The results of section 8.4 can therefore be used to determine the properties of both types of scope. The relationship to the variation mechanisms, as discussed in section 8.6, also gives indications of the variations mechanisms that can be

used. This means that the proposed classifications can be used to support the scoping process.

As indicated in section 8.1.2, several classifications are defined that apply to a certain aspect of product families. These classifications include maturity levels [11], organizational structures [9], architectural styles [81], and the binding time of the variation [100]. All these classifications serve specific purposes. The classification presented in this paper deals with two basic properties of a product family. A related point is that the most suitable variation mechanisms can be selected. This classification will both facilitate the selection of a new product family approach for a particular context, and support the evaluation of existing approaches. This, in turn, will allow other classifications to come into play, e.g. which type of architecture should be used, as discussed in section 8.4.1 in relation to the architectural weight.

To illustrate how our method relates to other work, we look at the maturity levels of software product families from Bosch [11]. These levels are: standardized infrastructure, platform, software product line, configurable product base, program of product lines and product populations. These levels can be mapped onto the two-dimensional space of Figure 8-7, as shown in Figure 8-11. The *standardized infrastructure* focuses on the operating system and a few commercial components on top of it; little or no domain engineering is required, and no variability management is needed. The products can be made by composing specific components on top of this infrastructure. This approach can be located in the bottom left in the two dimensions, i.e., no internal variation and a lot of components still have to be composed. The *platform* approach is similar to the previous one, except that more common functionality is included; only a little variation management is needed. Due to the possible variation inside a component, the platform approach is located above the previous approach. In the *software product line* approach, the set of shared artifact becomes larger; functionality that is common to several, but not all, products is also included. Based on the example of a family of servers (scanner, printer, storage, and server) it is assumed that the family architecture describes a basis that can be extended with additional components. Various variation mechanisms can be used internally in the components. The *configurable product base* approach is located to the left; it consists of a shared code based where no functionality has to be added by the application engineering (i.e. fixed components) and uses license keys or automated tooling to configure the software. The *program of product lines* approach defines a number of components that are product families themselves, i.e. can be tuned towards specific needs. These components can be related to subdomains. The architecture thus defines a fairly fixed architecture with components, and the product families provide configurable components to fill it in. This approach is

located in the top left corner, illustrating the fixed set of components, while also indicating that various variation mechanisms are used internally. The *product population* approach is located to the right, since the components can be composed freely to form a system. The main maturity path described in Bosch's paper is indicated by the black arrows in Figure 8-11. More and more (application) functionality is added to the set of reusable assets along this path, i.e. a higher platform coverage. The two additional approaches are useful for an increased number of products (product population) or an increased number of features (program of product lines).

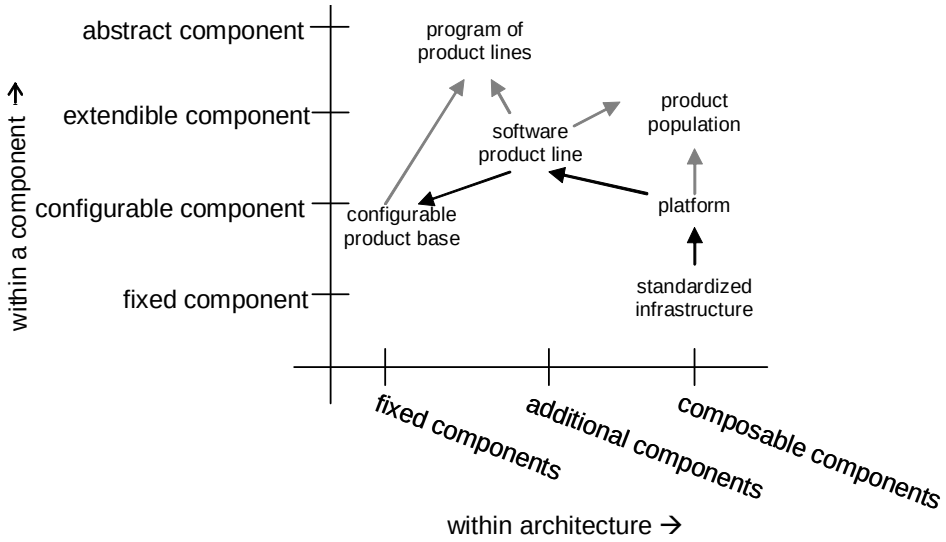

Figure 8-11 – Relation to Another Classification Scheme

In section 8.6.3 we discussed two introduction strategies for product families. These two strategies are roughly as follows: 1) start with a small set of family members and expand this set later on, or 2) start with a small set of subdomains (components) and expand this set later on. Several ideas can be found on this topic in the literature. In general, one can observe the focus on introduction strategies that have a limited initial impact and that are evolutionary. This can be observed in the discussion related to the sessions "Diversity Solutions" and "Light-Weight Processes" at the PFE-4 [3], among other places. An example of a paper discussing how to introduce product families is [47]. The author states that it is very important to lower the adoption barrier in order to obtain successful results. In [11], two dimensions of product family initiation are presented: evolutionary vs. revolutionary, and existing product line vs. new product line. In the case of the evolutionary introduction of an existing product line, components are identified that implement requirements of more than one product. These components are then provided as part of the platform. By

selecting more components over time, the platform grows. This is similar to the second introduction strategy presented in section 8.6.3. The first introduction strategy described in section 8.6.3 is also applicable to the evolutionary approach of an existing product line. In such a case, only a limited number of family members are integrated into the family as a first step, after which more products can be added. The case of the revolutionary introduction of an existing product line differs from the previous one in that the product family replaces all existing products in one large effort. In principle, such a strategy can enter our two dimensional space at any place. If the changes compared to the existing situation are large, this imposes a serious risk for success; this is why we do not propose it in section 8.6.3. When starting a new product family, an evolutionary or revolutionary path can also be taken. The difference between evolutionary and revolutionary, as sketched in [11], is whether only the existing products are taken into account, or whether all possible future family members are also taken into account. The evolutionary way resembles the first introduction strategy described in section 8.6.3, where we described starting with only a few products and then scaling up. The revolutionary way takes much more possible future products into account, and therefore introduces a larger change and risk. In our paper we have focused more on introduction strategies that have an evolutionary character. Of course, if an organization already has more experience, a larger introduction step may be taken. The added value of our paper is that the two discussed introduction strategies are related to domain coverage and variation mechanisms, and that evolution scenarios are sketched for these two classifications.

8.8 Conclusions

Each product family is used within a different context and with different requirements. As a consequence, different product family approaches exist, each with their own properties. In this paper we have discussed two dimensions that can be used to position product family approaches using a platform with reusable artifacts. The two introduced dimensions can help when selecting a suitable product family approach in a given context, or when evaluating an existing approach.

The first dimension is the coverage of the platform. This is an indication of how much the platform already realizes the family members, and how much specific, additional work has to be done by the product groups. Using four case studies from industry, we have given illustrations of the different types of platform coverage. If the domain for a product family is divided into a number of subdomains, then the platform can either deal with all subdomains of the family or only with a subset of them. Another important issue is whether the platform

deals only with the artifacts related to the realization of the system, or whether documents such as system requirement specifications are also dealt with within the platform.

Another important dimension of the product family approach is which variation mechanisms are applied. This can range from a configurable system to a platform from which the relevant components can be selected that will be integrated into a product. We have distinguished between variation that is visible at architectural level and variation that is realized within the components of the architecture. The right variation mechanisms must be selected based on the type and degree of variation within a product family. We have given some properties and guidelines in this area. For example, the component framework mechanism is useful if there is sufficient experience with the subdomain in which it is applied. This is because the interface between the component framework and its plug-in components should be stable, but must also support future plug-in components.

It is still very difficult to select the right platform-oriented product family approach. Many factors, a lot of which are of a non-technical nature, determine whether it will be successful in a particular situation. However, the coverage and variation mechanisms of a platform are very important properties that must be considered well before a platform is set up. We have also considered how the introduction of a family approach can be mapped to the coverage and variation mechanisms. Two introduction strategies have been described: one going from a high coverage and evolving to a lower, and one going from a low coverage and evolving to a higher. Both are based on the idea of lightweight introduction. Lightweight introduction strategies promise a higher success rate because the consequences within the development groups are minimal, and the resistance against the introduction is therefore lower as well.