

# Experiences in Software Product Families: Problems and Issues during Product Derivation

Sybren Deelstra, Marco Sinnema, Jan Bosch

University of Groningen, PO Box 800, 9700 AV Groningen, The Netherlands,  
[s.deelstra, m.sinnema, j.bosch]@cs.rug.nl, <http://segroup.cs.rug.nl>

**Abstract.** A fundamental reason for investing in product families is to minimize the application engineering costs. Several organizations that employ product families, however, are becoming increasingly aware of the fact that, despite the efforts in domain engineering, deriving individual products from their shared software assets is a time and effort consuming activity. In this paper we present a collection of product derivation problems that we identified during a case study at two large and mature industrial organizations. These problems are attributed to the lack of methodological support for application engineering, and to underlying causes of complexity and implicit properties. For each problem, we provide a description and an example, while for each cause we present a description, consequences, solutions and research issues. The discussions in this paper are relevant outside the context of the two companies, as the challenges they face do or eventually will arise in, for example, comparable or less mature organizations.

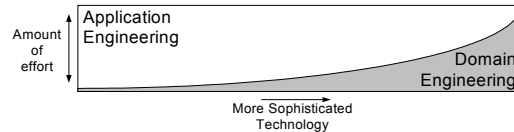
## 1 Introduction

Software product families have received substantial attention from the software engineering community since the 1990s [5][10][24]. The basic philosophy of software product families is intra-organizational reuse through the explicitly planned exploitation of commonalities of related products. This philosophy has been adopted by a wide variety of organizations and has proved to substantially decrease costs and time-to-market, and increase the quality of their software products.

The ability to derive different products from the product family is referred to as variability. Variability in software systems is realized through variation points that identify locations at which variation will occur [13]. Managing the ability to handle the differences between products at the various phases of the lifecycle, i.e. variability management, is a key success factor in software product families [7].

Product development in software product families is organized into two stages [19]. The first stage, domain engineering, refers to the activities that involve, amongst others, identifying commonalities and differences between product family members, and implementing a set of shared software artifacts (e.g. components or classes). The shared software artifacts are constructed in such a way that the commonalities can be exploited economically, while at the same time the variability is preserved. The second stage, application engineering, refers to the activities that involve, amongst

others, product derivation, i.e. constructing individual products using a subset of the shared software artifacts.



**Fig. 1.** This figure presents the fundamental reason for researching and investing in more sophisticated technology such as product families, i.e. decreasing the proportion of application engineering costs

The idea behind this approach to product engineering is that the investments required to develop the reusable artifacts during domain engineering, are outweighed by the benefits in deriving the individual products during application engineering. A fundamental reason for researching and investing in sophisticated technologies for product families is to obtain the maximum benefit out of this upfront investment, in other words, to minimize the proportion of application engineering costs (see Fig. 1).

Over the past few years, domain engineering has received substantial attention from the software engineering community. Most of those research efforts are focused on methodological support for designing and implementing shared software artifacts in such a way that application engineers should be able to derive applications more easily. Most of the approaches, however, fail to provide substantial supportive evidence. The result is a lack of methodological support for application engineering and, consequently, organizations fail to exploit the full benefits of software product families.

Rather than adopting the same top-down approach, where solutions that are focused on methodological support for domain engineering imply benefits during application engineering, we adopt a bottom-up approach in our research. By studying product derivation issues we believe we will be better able to provide and validate industrially practicable solutions for application engineering. This paper is a first step of the bottom-up approach: it provides an overview of problems and issues we identified at two industrial case studies, Robert Bosch GmbH and Thales Nederland B.V.

The case studies were part of first phase of ConIPF (Configuration in Industrial Product Families), a research project sponsored by the IST-programme [11]. Robert Bosch GmbH and Thales Nederland B.V. are industrial partners in this project. Both companies are large and relatively mature organizations that develop complex software systems. They face challenges during product derivation that do or eventually will arise in other, comparable or less mature organizations. The identified issues are therefore relevant outside the context of the respective companies.

The main contribution of this paper is that we show that, although software product families have been successfully applied in industry, there is room for improvement in the current practice. Solving product derivation problems will help organizations in exploiting the full benefits of software product families.

The remainder of this paper is organized as follows. The next section provides a description of a generic product derivation process. This description acts as a basis for our discussion in subsequent parts of the paper. In section 3, we briefly describe the

research method of the case study. We discuss the case study organizations in section 4 and the identified problems and cause analysis in section 5. Finally, we present related work and conclusions in sections 6 and 7 respectively.

## 2 Product Derivation Process

In earlier work [12], we have generalized the derivation processes we encountered in practice to a generic process. In this section, we discuss the aspects of this process that are relevant for discussions in subsequent parts of this paper. For a more elaborate description we refer to [12].

The generic product derivation process consists of two phases, i.e. the initial and the iteration phase. In the initial phase, a first configuration is created from the product family assets. In the iteration phase, the initial configuration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements.

In addition to the phased selection activities described above, typically some code development is required during product derivation. This adaptation aspect can occur in both the iteration phase and the initial phase. Below, we provide a more detailed description of both phases, as well as a separate description of the adaptation aspect.

### 2.1 Initial Phase

The input to the initial phase is a (sub)set of the requirements that are managed throughout the entire process of product derivation (see also Fig. 2). These requirements originate from, among others, the customers, legislation, the hardware and the product family organization. In the initial phase, two different approaches towards deriving the initial product configuration exist, i.e. assembly and configuration selection (see Fig. 2). Both approaches conclude with the initial validation step.

**Assembly:** The first approach to initial derivation involves the assembly of a subset of the shared product family assets to the initial software product configuration. Although three types of assembly approaches exist [12], i.e. construction, generation, and composition (a hybrid of the first two), the construction type is primarily relevant to this paper:

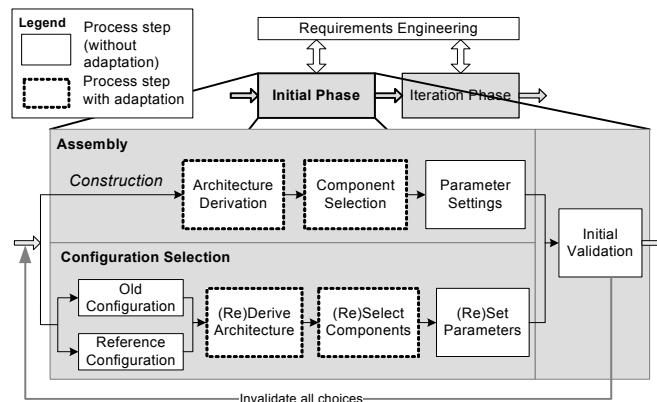
- *Construction:* In the construction approach, the initial configuration is constructed from the product family architecture and shared components. The first step in the construction process, as far as necessary or allowed, is to derive the product architecture from the product family architecture. The next step is, for each architectural component, to select the closest matching component implementation from the collection of shared components. Finally, the parameters for each component are set.

**Configuration Selection:** The second approach to initial derivation involves selecting a closest matching existing configuration. An existing configuration is a consistent set of components, viz. an arrangement of components that, provided with the right

options and settings, are able to function together. Two types of configuration selection are relevant for this paper:

- An *old configuration* is a product implementation that is the result from a previous project.
- A *reference configuration* is a (subset of) an old configuration that is explicitly designated as basis for the development of new products.

Where necessary, the selected configurations are modified by re-deriving the product architecture, adding, re- and deselecting components, and (re)setting parameters.



**Fig. 2.** During the initial phase, a first product configuration is derived by assembly or configuration selection

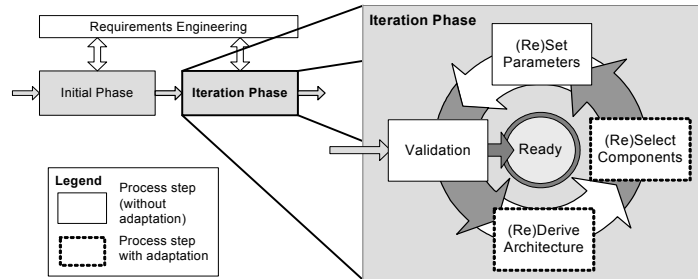
**Initial Validation:** The initial validation step is the first step that is concerned with determining to what extent the initial configuration adheres to the requirements. In the rare case that the initially assembled or selected configuration does not provide a sufficient basis for further development, all choices are invalidated and the process goes back to start all over again. In case the initial configuration sufficiently adheres to the requirements, the product is finished. Otherwise, the product derivation process enters the iteration phase.

## 2.2 Iteration Phase

The initial validation step marks the entrance of the iteration phase (see Fig. 3). In some cases an initial configuration sufficiently implements the desired product. In most cases, however, one or more cycles through the iteration phase are required, for a number of reasons.

First, the requirements set may change or expand during product derivation, for example, if the organization uses a subset of the collected requirements to derive the initial configuration, or if the customer has new wishes for the product. Second, the configuration may not completely provide the required functionality, or some of the selected components simply do not work together at all. This particularly applies to embedded systems, where the initial configuration is often a first approximation. This

is mainly because the exact physics of the controlled mechanics is not always fully known at the start of the project, and because the software performs differently on different hardware, e.g. due to production tolerances and approximated polynomial relationships. Finally, the product family assets used to derive the configuration may have changed during product derivation, for example, due to bug fixes.



**Fig. 3.** During the iteration phase, the product configuration is modified in a number of iterations, until the product is deemed ready by the validation step

During the iteration phase, the product configuration is therefore modified and validated until the product is deemed ready.

**Modification:** A configuration can be modified on three levels of abstraction, i.e. architecture, component and parameter level. Modification is accomplished by selecting different architectural component variants, selecting different component implementation variants, or changing the parameter settings, respectively.

**Validation:** The validation step in this phase concerns validating the system with respect to adherence to requirements and checking the consistency and correctness of the component configuration.

### 2.3 Adaptation

Requirements that are not accounted for in the shared product family artifacts can only be accommodated by adaptation (denoted by the dashed boxes in Fig. 2 and Fig. 3). Adaptation involves adapting the product (family) architecture and adapting or creating component implementations. We identify three levels of artifact adaptation, i.e. product specific adaptation, reactive evolution and proactive evolution.

**Product specific adaptation:** The first level of evolution is where, during product derivation, new functionality is implemented in product specific artifacts (e.g. product architecture and product specific component implementations). To this purpose, application engineers can use the shared artifacts as basis for further development, or develop new artifacts from scratch. As functionality implemented through product specific adaptation is not incorporated in the shared artifacts, it cannot be reused in subsequent products unless an old configuration is selected for those products.

**Reactive evolution:** Reactive evolution involves adapting shared artifacts in such a way that they are able to handle the requirements that emerge during product derivation, and can also be shared with other product family members. As reactively

evolving shared artifacts has consequences with respect to the other family members, those effects have to be analyzed prior to making any changes.

**Proactive evolution:** The third level, proactive evolution, is actually not a product derivation activity, but a domain engineering activity. It involves adapting the shared artifacts in such a way that the product family is capable of accommodating the needs of the various family members in the future, as opposed to evolution as a reaction to requirements that emerge during product derivation. Proactive evolution requires both analysis of the effects with respect to current product family members, as well as analysis of the predicted future of the domain and the product family scope. Domain and scope prediction is accomplished in combination with technology roadmapping [16].

Product specific adaptations can be fed back in the product family reuse infrastructure. This approach is often applied when the implications of changes are not well understood, e.g. in case of supporting new technology. Integrating product specific changes at a later stage, however, does require analyzing the impact on the other product family assets, and often implies retesting and additional development effort.

We have now established a framework of concepts regarding product derivation that consists of three main aspects, i.e. the initial phase, iteration phase and adaptation. We use this framework as basis for understanding our discussion on the case study organizations in section 4, and the identified product derivation problems in section 5.

### 3 Research Methodology

The case study was part of the first phase of ConIPF (Configuration in Industrial Product Families), a research project sponsored by the IST-programme [11]. Robert Bosch GmbH and Thales Nederland B.V. are industrial partners in this project. Both companies are large and mature industrial organizations that mark two ends of a spectrum of product derivation; Robert Bosch produces thousands of medium-sized products per year, while Thales Nederland produces a small number of very large products.

The main goals of the case study were to gain an understanding of product derivation processes at large organizations that employ software product families, and to determine the underlying problems and causes of challenges faced during this process. We achieved these goals through a number of interview sessions with key personnel involved in product derivation, amongst others, system architects, software engineers, and requirement engineers.

Although questionnaires guided the interviews, there was enough room for open, unstructured questions and discussions. We recorded the interviews for further analysis afterwards, and used documentation provided by the companies to complement the interviews.

## 4 Case Study Organizations

In this section, we present the business units of the organizations that were examined as part of the case study. We provide a description of product derivation at these business units in terms of the product derivation process presented in section 2.

### **Case 1: Thales Nederland B.V., The Netherlands**

Thales Nederland B.V. is a subsidiary of Thales S.A. in France, and operates in the Ground Based, Naval and Services areas of defense technologies. Thales Naval Netherlands (TNNL), the Dutch division of the business group Naval, is organized in four business units, i.e. Radars & Sensors, Combat Systems, Integration & Logistic Support, and Operations. Our case study focused on software parts of the SEWACO-FD (SEnsor WEapon Control – Fully Distributed) naval combat systems family produced by the business unit Combat Systems.

A Tacticos (TACTical Information and COMmand System) Combat Management System is the prime subsystem of SEWACO-FD Naval Combat Systems. Its main purpose is to integrate all weapons and sensors on naval vessels that range from fast patrol boats to frigates.

### **Derivation Process at business unit Combat Systems**

*Initial Phase.* Combat Systems uses configuration selection to derive the initial product configurations. To this purpose, the collected requirements are mapped onto an old configuration, whose characteristics best resemble the requirements at hand. When in subsequent steps all components and parameters are selected, adapted and set, the system is packaged and installed in a complete environment for the initial validation. If the configuration does not pass the initial validation, the derivation process enters the iteration phase.

*Iteration Phase.* The initial configuration is modified in a number of iterations by re- and de-selecting components, adapting components and changing existing parameter settings, until the product sufficiently adheres to the requirements.

*Adaptation.* Combat Systems applies both reactive evolution and product specific changes when components need to be adapted (see section 2.2.3). Components are also adapted through proactive evolution during domain engineering. Whether requirements are handled by reactive evolution or product specific adaptation, is determined by several Change Control Boards, i.e. groups of experts (such as architects) that synchronize change requests within and between different projects.

### **Case 2: Robert Bosch GmbH, Germany**

Robert Bosch GmbH is a worldwide operating company that is active in the Automotive, Industrial, Consumer Electronics and Building Technology areas. Our case study focused on two business units, which, for reasons of confidentiality, we refer to as business unit A and B. The systems produced by these business units consist of both hardware, i.e. the sensors and actuators, and software.

### **Derivation process at business unit A**

*Initial Phase.* Starting from requirements engineering, business unit A uses two approaches in deriving an initial configuration of the product: one for lead products and one for secondary products. For lead products, the initial configuration is derived by construction, viz. by deriving the architecture, selecting the appropriate components and setting the component parameters. For secondary products, i.e. in the case a similar product has been built before, reference configurations are used to derive an initial configuration. Where necessary, components from the reference configuration are replaced with ones that are more appropriate.

*Iteration Phase.* In the iteration phase, the initial configuration is modified in a number of iterations by reselecting components, adapting components, or changing parameters.

*Adaptation.* If the inconsistencies or new requirements cannot be solved by selecting a different component implementation, a new component implementation is developed through reactive evolution, by copying and adapting an existing implementation, or developing one from scratch.

### **Derivation Process at business unit B**

*Initial Phase.* Each time a product needs to be derived from the product family of business unit B, a project team is formed. This project team derives the product by construction. It copies the latest version of the architecture and the shared components, and selects the appropriate components from this copy. Finally, all parameters of the components are set to their initial values.

*Iteration Phase.* When the set of requirements changes, or when inconsistencies arise during the validation process, components and parameters are reselected and changed until the product is deemed finished.

*Adaptation.* When during the initial and iteration phase the requirements for a product configuration cannot be handled by the existing product family assets, copies of the selected components are adapted by product specific adaptation.

## **5 Case Analysis**

In the previous section, we presented the derivation processes of business units at Robert Bosch GmbH and Thales Nederland B.V. Based on the case study described in section 3, we identified a number of problems that are relevant outside the context of these organizations. We discuss these problems in this section.

The outline of our discussion is illustrated in Fig. 4. We start the discussion with the high-level challenges that served as motivation for investigating product derivation problems. In the observed problems subsection, we discuss the main problems that were identified during the case study and that underpin those challenges. The core issues are the common aspects of these problems. On the one hand, product derivation problems can be alleviated by process guidance and methodological support that finds a suitable way to deal with these issues. On the other hand, product derivation problems can be alleviated by addressing the underlying causes of the core issues. The core issues and related methodological

support issues are discussed in subsection 5.3. The underlying causes of the core issues are addressed in the last two subsections (5.4 and 5.5).

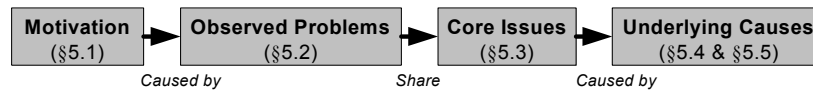


Fig. 4. The outline of the discussion in section 5

## 5.1 Motivation

Two main observations prompted the need to investigate product derivation.

**Time, effort and costs:** The first observation involved the classic software engineering problems, such as high costs associated with deriving individual products, as well as the time and effort consumption because of difficulties in individual product derivation steps and the large number of cycles through the iteration phase.

**Expert Dependency:** The second observation involved the fact that the product derivation process depends very much on expert involvement. This not only resulted in a very high workload on experts, and as such in the lack of accessibility of these experts, but also made the organizations very vulnerable to the loss of important derivation knowledge.

## 5.2 Observed Problems

During the data collection phase of the case study, we have identified several underlying problems for the observations discussed in the previous subsection. In the remainder of this subsection, we provide a description for each underlying problem, followed by an example.

### False positives of compatibility check during component selection

**Description:** During product derivation, components are selected for addition to or replacement of components in the (partial) configuration at hand. Whether a component fits in the configuration depends on whether the component correctly interacts with the other components in the configuration and whether no dependencies or constraints are violated. The interaction of a component with its environment is contractually specified through the provided and required component interface. Thus, the consistency of the cooperation with the other components can be validated by checking the correctness of the use of the provided interfaces, the satisfaction of the required interfaces, the constraints, and the dependencies of the selected component. The problem we identified is the situation in which the (manual or automated) consistency check does not find any violations, while during testing it turns out some of the selected components are not compatible. As a result, extra iterations and expert involvement are necessary in order to correct the inconsistent configuration.

**Example:** Part of the compatibility check between components during component selection is performed by tooling at business unit A. This automated check is based on

the syntactical interface specification, i.e. function calls with their parameter types. One of the causes for iterations is that although the interface check indicates a positive match for components in the configuration during earlier phases, in some cases, components turn out to be incompatible during validation on the test bench.

#### **Large number of human errors**

**Description:** Due to the large amount of variation points, i.e. ranging up to tens of thousands, and the possibly even larger number of (implicit) dependencies between these variation points, product derivation at both organizations has become an error-prone task. As a result, a substantial amount of effort in the derivation process of the interviewed business units is associated with correcting these human errors in the iteration phase.

**Example:** Parameter settings at TNNL Combat Systems account for approximately 50 percent of product derivation costs. The engineers indicated that most of the effort for this activity results from correcting unintended side-effects introduced by previous parameter setting steps.

#### **Consequences of variant selection unclear**

**Description:** Software engineers often do not know all consequences of the choices they make in the derivation process. At times, in later phases of product derivation, earlier selected variants proved to complicate development as the earlier choice had negative consequences.

**Example:** In addition to automated checks, the interviewed business units check component compatibility and appropriateness by manually inspecting the component documentation. The software engineers indicated that despite the fact that the component documentation for individual products comprises up to thousands of pages, during testing, selected components often insufficiently implement requirements or turn out to be incompatible.

#### **Repetition of development**

**Description:** The organizations identified that, despite their reuse efforts, development effort is spent on implementing functionality that highly resembles functionality already implemented in reusable assets or in previous projects.

**Example:** The asset repository of business unit A contains several thousand component-variants and several tens of thousands older versions of these component variants. Product derivation at this business unit comprises selecting several hundreds of components from these component-variants and setting several thousand parameters. Occasionally, this leads to a situation where during component selection, the component variants implementing certain requirements are not found, while they are present.

#### **Different provided and required interfaces complicate component selection**

**Description:** Variants of variation points communicate with other parts of the system through the provided and required interfaces. The problem we address here involves the situation where several variants of a particular variation point have different provided and required interfaces. This complicates component selection as in some cases multiple versions of the same variant provide the same required functionality,

but differ in provided or required interface, or two variants of different variation points can not be selected both, as they provide incompatible interfaces to each other. As a result, selecting a required variant can invalidate the selection of earlier selected variants or require adaptation.

**Example:** This problem was primarily identified at business unit A. Complications in component selection occur frequently as a result of different provided and required interfaces.

### 5.3 Core Issues

Upon closer inspection, the problems listed above share the combination of two core issues as a common aspect.

**Complexity.** The first core issue identified is the complexity of the product family in terms of number of variation points and variants. This complexity causes the process of setting and selecting variants to become almost unmanageable by individuals.

**Implicit properties.** The second core issue involves the large number of implicit properties (e.g. dependencies) of variation points and variants, i.e. properties that are undocumented and either unknown or only known by experts.

**Coping with complexity and implicit properties:** In itself, complexity and implicit properties are not problematic. It is due to the failure to effectively deal with them that the issues result in problems discussed in the previous section. On the one hand, product derivation problems can therefore be alleviated by process guidance and methodological support that provides a suitable way to deal with these issues, for example:

**Hierarchical organization.** One of the reasons why the large number of variation points and variants is a problem, is the fact that none of the product families of the case study explicitly organized variation points in a hierarchical fashion (as for example employed in feature diagrams [23]). As a consequence, during product derivation, software engineers are required to deal with many variation points that are either not at all relevant for the product that is currently being derived, or that refer to the selection of the same higher level variant.

**First-class and formal representation of variation points and dependencies.** The lack of first-class representation of variation points and dependencies makes it hard to assess the impact of selections during product derivation, and changes during evolution, as it is unclear how variations in requirements are realized in the implementation [7]. If formalized, first-class representations of variation points and dependencies enable tool support that may significantly reduce costs and increase efficiency of product derivation and evolution.

**Reactive documentation of properties.** The problems as a result of implicit dependencies are not only associated with the one time occurrence, but also with the fact that mistakes keep reoccurring in several projects. This is due to the lack of documenting implicit properties that frequently result in mistakes. One approach to

dealing with implicit properties is therefore through a reactive documentation process that ensures documentation of important implicit dependencies.

On the other hand, rather than focusing on dealing with the issues, product derivation problems can be alleviated by addressing the source of the core issues. During cause analysis, we identified that both core issues have a number of underlying causes. We discuss the underlying causes of complexity and implicit properties in the following two subsections, respectively. For each of these causes, we provide a description and consequences, as well as solutions and topics that need to be addressed by further research.

#### **5.4 Underlying causes of complexity**

Complexity in product families is both due to the inherent complexity of the problem domain (e.g. complex system properties, number of product family members), as well as the design decisions regarding variability that are made during the evolution of the shared software assets. In the following cause analysis, we focus on increases in complexity that are the result of evolution of variability. We identified issues related to scoping during product derivation, obsolete variation points, and non-optimal realization of variability as underlying causes.

##### **Scoping during product derivation: product specific adaptation versus reactive evolution**

**Description:** As briefly discussed in section 2.3, changes that are required during product derivation (e.g. new features), can be handled through product specific adaptation or reactive evolution. An important aspect related to these types of evolution is scoping, i.e. determining in which of both types a change should be handled.

Such a scoping decision is optimal if the benefits of handling the change through reactive evolution (possible reuse) outweigh the benefits of product specific adaptation (no unnecessary variation points and variants). At Bosch, we identified two extremes in the balance between product specific adaptation and reactive evolution. On the one extreme, all required changes for the products were handled through product specific adaptation (business unit B), while on the other extreme, all changes were handled through reactive evolution (business unit A). In other words, no explicit asset scoping activity existed that continuously interacted with product derivation.

**Consequences:** The drawback of not individually scoping each feature during product derivation, is that, for possibly a considerable amount of functionality, a non-optimal scoping decision is made, viz. a decision either resulting in repetition of development in different projects, or in more variation points and variants than actually needed. The lack of scoping during product derivation is thus the first underlying cause of complexity.

**Solutions and research issues:** Product derivation processes should include continuous interactions with the scoping activities during domain engineering. In that respect, the way Change Control Boards are employed in TNNL Combat Systems (see section 4) provides a good example for large product families. For smaller

product families, this continuous interaction is the responsibility of the product family architect(s). Currently, however, there is a lack of variability assessment techniques that consider both the benefits and the drawbacks of scoping decisions. We consider these techniques to be crucial in assisting the system engineers and architects during evolution.

#### **Proactive evolution: obsolete variation points**

**Description:** The problem we address here is twofold. First, a typical trend in software systems is that functionality specific to some products becomes part of the core functionality of all product family members, e.g. due to market dominance. The need to support different alternatives, and therefore variation points and variants for this functionality, may disappear. Second, variation points introduced during proactive evolution are often not evaluated with respect to actual use.

Both phenomena lead to the existence of obsolete variation points, i.e. variation points for which in each derived product the same variant is chosen, or, in case of optional variants, not used anymore. At both organizations, these obsolete variation points were often left intact rather than being removed from the assets.

**Consequences:** Removing obsolete variation points increases the predictability of the behavior of the software [7], decreases the cognitive complexity of the software assets, and improves traceability of suitable variants in the asset repository. We also note that if variability provided by artifacts is not used for a long time and removed from the documentation, engineers may start to forget some facilities are there. For example, the interviewees indicate the existence of parameters from which no one knows what they are for, let alone what the optimal value is. In addition to handling all changes through reactive evolution, we therefore identify the existence of obsolete variation points and the lack of removing these obsolete variation points, as the second underlying cause of complexity.

**Solutions and research issues:** The solution to this issue is not as simple as just removing the variation point. Besides the fact that it may prove to be hard to pinpoint the exact moment at which a variation point becomes obsolete (“but it might be used again next month!”), removing obsolete variation points requires effort in many areas. All variation points related to the previously variable functionality have to be removed, which may require redesign and reimplementation of the variable functionality. There may be dependencies and constraints that have to be taken care of. Also, changing a component may invalidate existing tests and thus require retesting. In order to reduce complexity, we identify the need for methodologies regarding road mapping and variability assessment that do not only consider the necessity and feasibility of including new functionality in the shared software assets, but that also regularly consider the removal of unused functionality (and thus variation points and variants).

#### **Non-optimal realization of variability**

**Description:** Over the past few years, several variability realization techniques have been identified [13][23]. Variability realization mechanisms typically have a large impact on the performance and flexibility of a software system. Therefore, a well-considered choice should be made when selecting a mechanism, based on aspects such as the size of the involved software entities, when the variation should be

introduced, when it should be possible to add new variants, and when it needs to be bound to a particular variant. In practice, however, the number of different variability realization techniques used is often very limited and not always best suited for the problem at hand. We identify a number of reasons:

- As some variation mechanisms require extra computational resources such as memory or CPU-cycles, especially organizations that produce embedded systems often choose mechanisms that minimize the overhead.
- The number of variability mechanism used is further limited by the technology used. The programming language C for example, does not allow for techniques such as inheritance.
- Software architects typically lack the awareness with respect to the advantages and disadvantages of certain mechanisms and often only map variation to the mechanisms they know [7].
- Customer requirements may require a certain mechanism to be used, whereas this mechanism is not recommended from a technology point of view. An example of such a situation is when runtime binding is prohibited, as the code is restricted to be shipped to one customer only.
- Reactive evolution is mainly concerned with incorporating the requirements that are specific to a product in relation to existing product family requirements, rather than analyzing how they relate to requirements in future products. In addition, reactive evolution suffers from time-to-market and budget pressure of individual projects. With this focus, implementing variation points and variants for a specific product is prioritized over reusability in other products.

**Consequences:** Although choosing only a few mechanisms simplifies design and implementation, the drawbacks of disregarding the other mechanisms are not always understood [7]. In the interviewed business units, most variation is mapped to variant component implementations and parameterization. The drawback of this approach for business unit A, for example, is that even small changes to the interfaces needed to be implemented through new component implementations. This contributed to the fact that the asset repository of business unit A now contains several thousand component-variants and several ten-thousand older versions of these component variants. Realization of variability is therefore identified as a third underlying cause of complexity.

**Solutions and research issues:** We identify the need for detailed comparative studies of realization techniques in order to understand the qualities and drawbacks of selecting certain mechanisms. Adequate training of software architects with respect to design for variability helps to create awareness of different variability mechanisms.

## 5.5 Underlying causes of implicit properties

The second core issue, implicit properties, is related to the creation and maintenance of documentation. In the following cause analysis, we identify three underlying causes, i.e. causes related to erosion, to insufficient and over-explicit documentation, and to the lack of specifying semantics and behavior.

### **Erosion of documentation**

**Description:** Documentation has to be actively maintained in order to keep it useful. At the original development of software artifacts, the specification can be correct, or at least intended to be correct. If in a later stage an artifact is changed, the specification should also be changed accordingly. Quite often however, investigating all consequences of changes and updating the documentation accordingly is not a priority.

**Consequences:** The severity of situations where documentation shows signs of erosion, range from minor inconsistencies that result in errors, and thus iterations, up to the stage where it is out-dated and rendered almost useless. Eroded documentation furthermore contributes to, for example, the fact that it requires several years to train personnel at TNNL Combat Systems.

**Solution and research topics:** The issue discussed here, confirms the documentation problem regarding reuse discussed in earlier work [4]. Solutions and research issues suggested in that article, such as not allowing engineers to proceed without updating documentation, a higher status and larger amount of support from management, and investigating novel approaches to documentation, therefore also apply here. A technique that specifically addressed erosion is literate programming [17], where source code and documentation live in one document. Literate programming is state of practice through, for example, Javadoc [14], and may provide a large and easy step forward for many C oriented systems.

### **Insufficient, irrelevant and voluminous documentation**

**Description:** Software engineers often find it hard to determine what knowledge is relevant and should be documented. Although problems with documentation are usually associated with the lack of it, the large amount and irrelevance of documentation was identified as a problem as well.

**Consequences:** In addition to the automated check, component compatibility at the interviewed business units is performed manually by inspecting the component documentation. The software engineers indicated that the component documentation for individual products often comprise thousands of pages. On the one hand, the volume and irrelevancy of the information contained in the documents made it hard to find aspects (e.g. dependencies) relevant for a particular configuration. On the other hand, the software engineers also indicate that, at times, even the large amounts of relevant documentation seem not enough to determine the provided functionality or compatibility.

**Solution and research topics:** Besides good structuring, documentation requires an acceptable quality to quantity ratio in order to be effective. Relevant research issues have been discussed in the research issues for erosion of documentation.

### **Lack of semantics and behavior**

**Description:** Component interface specifications often consist of a set of provided and required operations in terms of argument and return types. Such specifications do not specify the semantics and behavior of the interface.

**Consequences:** One of the reasons for the false positive problem as discussed in section 5.2 is that syntactically equal operations can have different meanings (a

temperature ‘delta’ parameter may indicate a positive or negative delta for example) or not be accessible at all times (e.g. in case of slow producers and fast consumers).

A second drawback of the lack of semantics is that dependencies and constraints often specify that components are incompatible, rather than why they are incompatible. From the point of view from an engineer that has to consider selecting a closely matching component, however, it is also very important to know why certain components exclude or depend on other components, or whether certain constraints are weak or strong, than just the fact that certain components cannot be selected as-is.

**Solution and research topics:** Relevant research issues, such as novel approaches to documentation, and literal programming, have been discussed in earlier work [4] and the research issues on erosion of documentation.

## 6 Related Work

The notion of software product families has received substantial attention in the research community. The adoption of software product families has resulted in a number of books ([5], [10], and [24]), workshops (PFE 1-4), conferences (SPLC 1, 2), and several large European research projects (e.g. PRAISE, ESAPS and CAFÉ [19]).

The notion of a variation point was introduced by Jacobson et al. [13]. Several variability realization techniques have been identified in e.g. [13], and [23]. The notion of variability has also been discussed in earlier work of our research group, amongst others, by discussing variability issues that are related to problems presented in section 5 [7], and presenting a taxonomy of variability realization mechanisms [23]. The contribution of this paper is that we have identified a number of issues that are specifically related to evolution of variability.

In the context of evolution, [2] and [10] propose periodic scoping during domain engineering as solution for dealing with the continuous evolution of a product family. In this paper, we have proposed the use of a feature scoping activity that continuously interacts with product derivation, to prevent non-optimal scoping decisions. Product family scoping is furthermore discussed in [15] and [21].

Well-known process models that resemble the ideas of the phased product derivation model we presented in section 3, are the Rational Unified Process [18] and Boehm’s spiral model [3]. More related work and an elaborate description of the generic process can be found in [12].

The problems with implicit properties confirm problems regarding documentation in the context of software reuse discussed in earlier work [4]. [20] provides a discussion on the process of externalization, i.e. converting tacit knowledge to documented or formalized knowledge. Literate programming was introduced in [17], and addresses the erosion of documentation problem discussed in section 5.5.

Several industrial case studies have been performed inside our group, e.g. on product instantiation [6] and evolution in software product families [22]. Also outside our group, several case studies have been presented over the years [1][8][9]. The contribution of this paper is that it specifically identifies and analyses product derivation problems, whereas earlier case studies primarily focused on domain engineering.

## 7 Conclusions

In this paper, we have presented the results of a case study on product derivation that involved two industrial organizations, i.e. Robert Bosch GmbH and Thales Nederland B.V. The case study results include a number of identified problems and causes.

The discussions in this paper focused on two core issues. The first issue is complexity. Although the assumption that variability improves the ability to select alternative functionality and thus the ease of deriving different product family members is easily made, the results of our case study suggest otherwise. At a certain point, each additional variation point leads to an increase in the cognitive complexity of the product family, and possibly complicates the derivation process. On the one hand, product derivation problems can be alleviated by product derivation methodologies that effectively deal with complexity, e.g. through hierarchical organization and first-class representation of variation points and dependencies. On the other hand, problems can be alleviated by addressing the main source behind complexity, i.e. evolution of variability. We have identified research issues related to evolution of variability, such as variability assessment techniques that assist software engineers in determining the optimal scoping decision during product derivation, as well as assessment techniques to determine the feasibility of removing obsolete variation points, and selecting a particular variability realization mechanism during domain engineering.

The second important issue is the large number of implicit properties (such as dependencies) of software assets. An approach to coping with this issue is to reactively document implicit properties that frequently result in mistakes. Problems can further be alleviated by addressing the sources of implicit properties, such as erosion of documentation, insufficient and over-explicit documentation, and the lack of semantics and behavior in specifications. Parts of these causes behind implicit properties confirm the problems related to documentation and reuse discussed in earlier work [4]. Research issues related to this topic can therefore also be found in that article.

Concluding, software product families have been successfully applied in industry. By studying and identifying product derivation problems, we have shown that there is room for improvement in the current practice. Solving the identified problems will help organizations in exploiting the full benefits of software product families. Future work of the ConIPF project (see section 3), therefore aims to define and validate methodologies that are practicable in industrial application and that address the product derivation problems discussed in this paper.

**Acknowledgments:** This work was sponsored by the CONIPF project, under contract no. IST-2001-34438. We would like to thank the participants of the case study for their valuable input.

## 8 References

1. Ardis, M., Daley, N., Hoffman, D., Siy, H., Weiss, D., 2000. Software Product Lines: A Case Study, *Software – Practice and Experience*, Vol. 30, No. 7, pp. 825–847

2. Bayer, J., Flege, O., Knauber, P., Laqua, R., Muthig, D., Schmid, K., Widen, T., DeBaud, J.-M., 1999. PuLSE™: A Methodology to Develop Software Product Lines, Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), pp. 122-131.
3. Boehm, B. W., 1988. A spiral model of software development and enhancement, IEEE Computer, Vol. 21, No. 5, pp. 61–72
4. Bosch, J., 1999. Product-line architectures in industry: a case study, Proceedings of the 21st International Conference on Software Engineering, pp. 544-554
5. Bosch, J., 2000. Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7
6. Bosch, J., Högström, M., 2000. Product Instantiation in Software Product Lines: A Case Study, Second International Symposium on Generative and Component-Based Software Engineering, pp. 147-162
7. Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., Pohl, K., 2001. Variability Issues in Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19
8. Brownsword, L., Clements, P., 1996. A Case Study in Successful Product Line Development, Technical Report CMU/SEI-96-TR-016, Software Engineering Institute, Carnegie Mellon University
9. Clements, P., Cohen, S., Donohoe, P., Northrop, L., 2001. Control Channel Toolkit: A Software Product Line Case Study, Technical Report CMU/SEI-2001-TR-030, Software Engineering Institute, Carnegie Mellon University
10. Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns, SEI Series in Software Engineering, Addison-Wesley, ISBN: 0-201-70332-7
11. ConIPF project (Configuration of Industrial Product Families), <http://segroup.cs.rug.nl/conipf>
12. Deelstra, S., Sinnema, M., Bosch, J., 2003. A Product Derivation Framework for Software Product Families, accepted for the 5<sup>th</sup> Workshop on Product Family Engineering (PFE-5), November 2003
13. Jacobson, I., Griss, M., Jonsson, P., 1997. Software Reuse. Architecture, Process and Organization for Business Success. Addison-Wesley, ISBN: 0-201-92476-5
14. Javadoc, 2003. <http://java.sun.com/j2se/javadoc/>
15. Kishi, T., Noda, T., Katayama, T., 2002. A Method for Product Line Scoping Based on Decision-Making Framework, Proceedings of the Second Software Product Line Conference, pp. 348–365
16. Kostoff, R. N., Schaller, R. R., 2001. Science and Technology Roadmaps, IEEE Transactions on Engineering Management, Vol. 48, no. 2, pp. 132-143
17. Knuth, D.E., 1984. Literate programming, Computer Journal Vol. 27, pp. 97-111
18. Kruchten, P., 2000. The Rational Unified Process: An Introduction (2<sup>nd</sup> Edition), ISBN 0-201-707101
19. Linden, F. v.d., 2002. Software Product Families in Europe: The Esaps & Café Projects, IEEE Software, Vol. 19, No. 4, pp. 41-49
20. Nonaka, I., Takeuchi, H., 1995. The Knowledge-Creating Company: How Japanese companies create the dynasties of innovation, New York: Oxford University Press
21. Schmid, K., 2000. Scoping Software Product Lines - An Analysis of an Emerging Technology, Proceedings of the First Software Product Line Conference, pp. 513–532
22. Svahnberg, M., Bosch, J., 1999. Evolution in Software Product Lines: Two Cases, Journal of Software Maintenance, Vol. 11, No. 6, pp. 391-422
23. Svahnberg, M., Gulp, J. van, Bosch, J., 2002. A Taxonomy of Variability Realization Techniques, ISSN: 1103-1581, Blekinge Institute of Technology, Sweden.
24. Weiss D. M., Lai, C. T. R., 1999. Software Product-Line Engineering: A Family Based Software Development Process, Addison-Wesley, ISBN 0-201-694387