Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
The Journal of Systems and Software 74 (2005) 173–194
The Journal of Systems and Software
www.elsevier.com/locate/jss

Product derivation in software product families: a case study

Sybren Deelstra, Marco Sinnema *, Jan Bosch

Department of Mathematics and Computer Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands

Received 25 July 2003; received in revised form 27 October 2003; accepted 15 November 2003
Available online 21 January 2004

Abstract

From our experience with several organizations that employ software product families, we have learned that, contrary to popular belief, deriving individual products from shared software assets is a time-consuming and expensive activity. In this paper we therefore present a study that investigated the source of those problems. We provide the reader with a framework of terminology and concepts regarding product derivation. In addition, we present several problems and issues we identified during a case study at two large industrial organizations that are relevant to other, for example, comparable or less mature organizations.
© 2003 Elsevier Inc. All rights reserved.

Keywords: Case study; Software product family; Product derivation; Variability management

1. Introduction

Since the 1960s, reuse has been the long standing notion to solve the cost, quality and time-to-market issues associated with development of software applications. A major addition to existing reuse approaches since the 1990s are software product families (Bosch, 2000; Clements and Northrop, 2001; Jazayeri et al., 2000; Weiss and Lai, 1999). The basic philosophy of software product families is intra-organizational reuse through the explicitly planned exploitation of similarities between 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 (Linden, 2002).

In a software product family context, software products are developed in a two-stage process, i.e. a domain engineering stage and a concurrently running application engineering stage (Linden, 2002). Domain engineering involves, amongst others, identifying commonalities and differences between product family members and implementing a set of shared software artifacts (e.g. components or classes) in such a way that the commonalities can be exploited economically, while at the same time the ability to vary the products is preserved. During application engineering individual products are derived from the product family, viz. constructed using a subset of the shared software artifacts.

The 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.

* Corresponding author. Tel.: +31-50-363-7125; fax: +31-50-363-3800.

E-mail addresses: s.deelstra@cs.rug.nl (S. Deelstra), m.sinnema@cs.rug.nl (M. Sinnema), j.bosch@cs.rug.nl (J. Bosch).

0164-1212/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2003.11.012

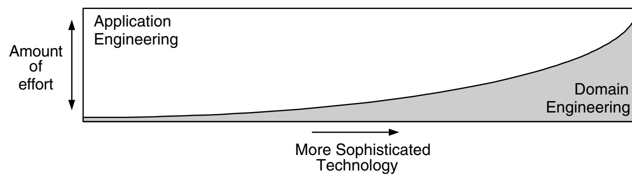

Fig. 1. Domain vs. application engineering. This figure presents the fundamental reason for researching and investing in more sophisticated technology such as product families, i.e. decreasing the proportion of application engineering costs.

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 the 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 (ConIPF, 2003). 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, e.g. comparable or less mature organizations. The identified issues are therefore relevant outside the context of the respective companies.

The main contributions of this paper are a framework of terminology and concepts regarding product derivation as presented in Section 2, and a set of identified problems and issues associated with product derivation as presented in Section 4.

The remainder of this paper is organized as follows. In Section 2, we describe the product derivation framework as a basis for our discussion. In Section 3, we provide a description of the industrial case studies. In Section 4, we discuss the identified problems and issues. Related work is presented in Section 5 and the paper is concluded in Section 6.

2. Product derivation framework

In this section, we present a product derivation framework that is based on the results of case studies of the aforementioned and other organizations. To avoid confusion, we start with a subsection containing definitions of a number of terms used throughout this paper. We continue by presenting a classification for product families, as well as a generic software derivation process. We conclude this section by discussing the relation between product family classification and several aspects of product derivation. Combined, these subsections build up the product derivation framework that is used to discuss a number of product derivation problems in subsequent parts of this paper.

2.1. Terminology

We use the following terminology in this document.

Product derivation. A product is said to be derived from a product family if it is developed using shared product family artifacts. The term product derivation therefore refers to the complete process of constructing a product from product family software assets.

Architecture. A product family architecture is the higher level structure that is shared by the product family members. It denotes the "*fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution*" (IEEE1471, 2000). Each product family member derives its architecture from this overall structure.

Component. A unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes (Bosch, 2000).

Variation point. Places in the design or implementation that identify locations at which variation will occur (Jacobson et al., 1997). Two important aspects related to variation points are binding time and realization mechanism. The term 'binding time' refers to the point in a product's lifecycle at which a particular alternative for a variation point is bound to the system, e.g. pre- or post-deployment. The term 'mechanism' refers to the technique that is used to realize the variation point (from an implementation point of view). Several of these realization techniques have been identified in the recent years, such as aggregation, inheritance, parameterization, conditional compilation (see e.g. Jacobson et al., 1997; Anastasopoulos and Gacek, 2001).

Configuration. A configuration is an arrangement of components and associated options and settings that partially or completely implements a software product. A partial configuration partially implements a software product in the sense that not all variants are selected yet or some variation points are not yet (completely) dealt with. Likewise, a complete configuration is able to fully implement the product requirements, i.e. all necessary variants are selected. In a complete configuration not all variation points have to be bound yet, however. There may still be variation points that are bound at runtime for example.

Knowledge types. We distinct three types of knowledge that are used during product derivation, i.e. tacit, documented, and formalized knowledge. Tacit knowledge (Nonaka and Takeuchi, 1995) is implicit know-

ledge that only exists in expert minds. Documented knowledge is explicit knowledge that is expressed in informal models and descriptions. Formalized knowledge is explicit knowledge that is written down in a formal language, such as the UML (UML, 2000), and can be used and interpreted by computer applications.

2.2. Product family classification

As illustrated in the introduction, product families are a successful form of intra-organizational reuse that is based on exploiting common characteristics of related products. In this section, we present a classification of the different types of product families that captures most product families we encountered in practice. This classification consists of two dimensions of scope, i.e. scope of reuse and domain scope.

The first dimension, *scope of reuse*, denotes to which extent the commonalities between related products are exploited. We identify four levels of scope of reuse, ranging from standardized infrastructure to configurable product base.

- *Standardized infrastructure*. Starting from independent development of each product, the first step to exploit commonalities between products is to reuse the way products are built. Reuse of development methodologies is achieved by standardizing the infrastructure with which the individual applications are built. The infrastructure consists of typical aspects such as the operating system, components such as database management and graphical user interface, as well as other aspects of the development environment, such as the use of specific development tools.
- *Platform*. With a standardized infrastructure in place, the next increase in scope of reuse is when the organization maintains a platform on top of which the products are built. A platform consists of the infrastructure discussed above, as well as artifacts that capture the domain specific functionality that is common to all products. These artifacts are usually constructed during domain engineering. Any other functionality is implemented in product specific artifacts during application engineering. Typically, a platform is treated as if it was an externally bought infrastructure.
- *Software product line*. The next scope of reuse is when not only the functionality common to all products is reusable, but also the functionality that is shared by a sufficiently large subset of product family members. As a consequence, individual products may sacrifice aspects such as resource efficiency or development effort in order to benefit from being part of the product family, or in order to provide benefits to others. Functionality specific to one or a few products is still developed in product specific artifacts. All other functionality is designed and implemented in such a way that it may be used in more than one product. Variation points are added to accommodate the different needs of the various products.
- *Configurable product family*. Finally, the configurable product family is the situation where the organization possesses a collection of shared artifacts that captures almost all common and different characteristics of the product family members, i.e. a configurable asset base. In general, new products are constructed from a subset of those artifacts and require no product specific deviations. Therefore, product derivation is typically automated once this level is reached (i.e. application engineers specify a configuration of the shared assets, which is subsequently transformed into an application).

In addition to the scope of reuse as described above, we identify a second dimension, *domain scope*. The domain scope denotes the extent of the domain or domains in which the product family is applied.

- *Single product family*. The first domain scope is the individual product family, where a single product family is used to derive several related products. In this paper we focus on this domain scope.
- *Programme of product families*. In case of a programme of product families, several product families together form a complete system. A shared architecture defines the overall structure of the software systems. The individual components of the system are developed according to an individual product family approach as described above. The programme of product families is especially applicable for very large software systems.
- *Hierarchical product families*. An hierarchical product family consists of several layers of product families. The top-level product family denotes functionality that is shared by all products, while the lower-level product families specialize the upper levels. This scope is particularly applicable in situations where the number and variability of the involved products is large or very large and a considerable number of staff members is involved in producing those products (Bosch, 2000).
- *Product population*. The product population approach is concerned with reuse of functionality across several domains. Each product family has its own architecture and domain specific components for the required domain specific functionality. Functionality that is shared between the domains is developed in shared components that can be used in the domain specific architectures. For more detailed information on product populations, see Ommering (2002).

2.3. A generic product derivation process

Focusing on the scope of reuse dimension with a single product family as domain scope, we have generalized the derivation processes we encountered in practice to a generic process as illustrated in Fig. 2.

This generic 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. During this phase, the application engineer has substantial freedom in choosing alternative 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. The freedom of choice of the application engineer is much more limited during the iteration phase as all decisions have to be made within the context of the product configuration at hand.

In addition to the phased selection activities described above, in all product families, except for product families with the largest scope of reuse, typically some code development is required during product derivation. This adaptation aspect is not strictly bound to one phase in the derivation process. We therefore provide a more detailed description of both phases, as well as a separate description of the adaptation aspect, below.

2.3.1. 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 Fig. 2). These requirements originate from, among others, the customers, legislation, the hardware and the product family organization. In the initial phase, three alternative approaches towards deriving the initial product configuration exist, i.e. assembly, configuration selection, and a hybrid of the former two approaches (see Figs. 3 and 4). The alternative 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. We identify three types of assembly approaches.

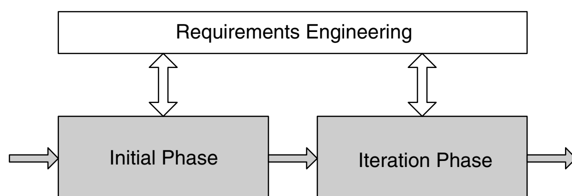

Fig. 2. The generic two-phased product derivation process. The shaded boxes denote the two phases of the generic product derivation process. Requirements engineering manages the requirements throughout the entire process.

- 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 component base. Finally, the parameters for each component are set.
- In case of *generation*, shared artifacts are modeled in a modeling language rather then implemented in source code. From these modeled artifacts, a subset is selected to construct an overall model. From this overall model an initial implementation is generated.
- The *composition* type (see also Fig. 4) is a composite of the types described above, where the initial configuration consists of both generated and implemented components, as well as components that are partially generated from the model and extended with source code.

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.

- An *old configuration* is a complete product implementation that is the result from a previous project. Often, the selected old configuration is the product developed during the latest project as it contains the most recent bug-fixes and functionality.
- A *reference configuration* is (a subset of) an old configuration that is explicitly designated as basis for the development of new products. A reference configuration may be a partial configuration, for example if almost all product specific parameter settings are excluded, or a complete configuration, i.e. the old configuration including all parameter settings.
- A *base configuration* is a partial configuration that forms the core of a certain group of products. A base configuration is not necessarily a result from a previous product. In general, a base configuration is not an executable application as many options and settings on all levels of abstraction (e.g. architecture or component level) are left open. In contrast to a reference and old configuration, where the focus during product derivation is on reselecting components, the focus of product derivation with a base configuration is on adding components to the set of components in the base configuration.

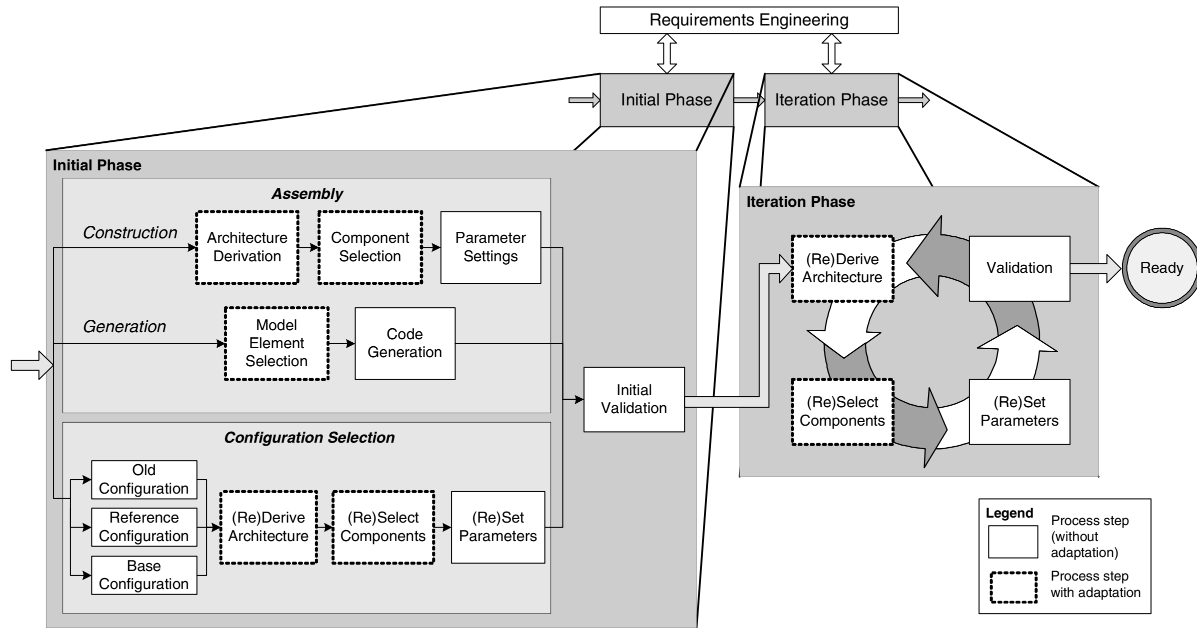

Fig. 3. The generic two-phased product derivation process in detail. During the initial phase of the process, a first product configuration is derived from the product family assets. Until the product is finished, the initial configuration is modified in a number of subsequent iterations during the iteration phase. Requirements that cannot be accommodated by existing assets are handled by product specific adaptation or reactive evolution (denoted by the dashed boxes).

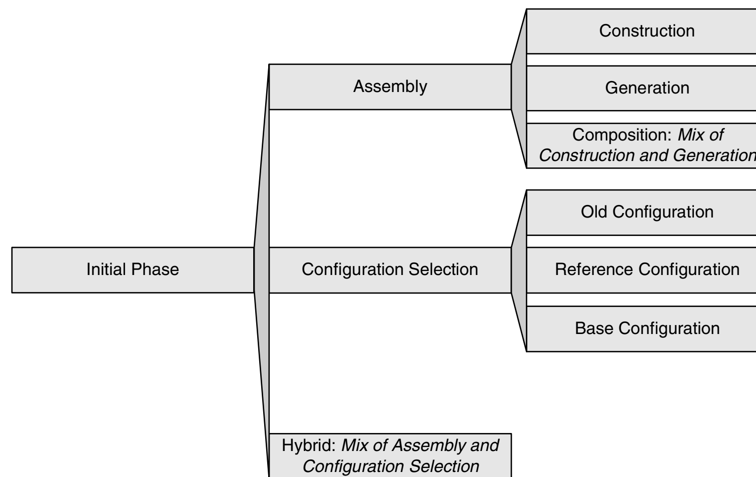

Fig. 4. Alternatives during the initial phase. This figure portraits the alternative ways to derive an initial configuration. The composite approach mixes construction and generation, while the hybrid approach mixes assembly and configuration selection.

The selected configurations are subsequently modified by rederiving the product architecture, adding, re- and deselecting components and (re)setting parameters.

The effectiveness of configuration selection in comparison to assembly is a function of the benefits in terms of effort saved in selection and testing, and the costs in terms of effort required for changing invalidated choices as a result of new requirements. Configuration selection is especially viable in case a large system is developed for repeat customers, i.e. customers who have purchased a similar type of system before. Typically, repeat customers desire new functionality on top of the functionality they ordered for a previous product. In that respect, configuration selection is basically reuse of choices.

Hybrid. Similar to the composite assembly approach, a hybrid approach to configuration exists that mixes assembly and configuration selection. This hybrid approach involves selecting a number of partial configurations (pre-assembled components or subsystems) that are integrated to a larger assembly.

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.

2.3.2. Iteration phase

The initial validation step marks the entrance of the iteration phase (illustrated in 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 'guess'. 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.

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.

Orthogonal to the two phases of the derivation process is the aspect of accommodating requirements that cannot be handled by as-is reuse, i.e. reuse without adaptation, of existing assets. We refer to this activity as adaptation.

2.3.3. Adaptation

Although Macala et al. (1996) suggested a five year prediction window for functionality in software product families, in general, products do not precisely adhere to any designed or planned path (Svahnberg and Bosch, 1999). As a result, new product family members may present requirements during product derivation that are not accounted for in the shared product family artifacts. Rather then selecting different artifact variants during the phases described above, these unsupported requirements can only be accommodated by adaptation (denoted by the dashed boxes in 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 (see Fig. 5).

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

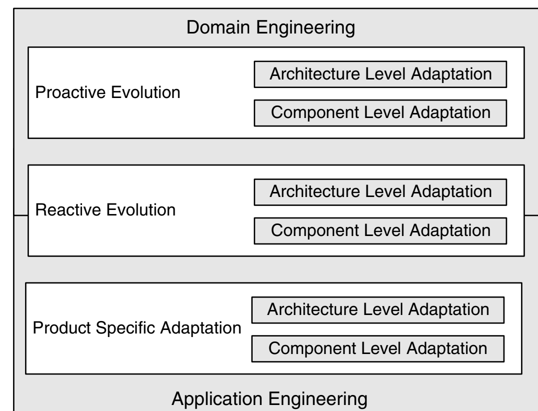

Fig. 5. Three types of artifact evolution. Reactive evolution and product specific adaptation are product derivation activities, while proactive evolution is a 'pure' domain engineering activity.

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 (Kostoff and Schaller, 2001).

Independent of the evolution type chosen, the scope of adjustment required on architecture or component level varies in four different ways.

Scope of adjustment

1. *Add variation points*. A new variation point has to be constructed if functionality needs to be implemented as variant or optional behavior, and no suitable variation point is available. In this, we recognize two distinct situations. In the first situation, the new functionality is needed as option or alternative to already implemented stable behavior. This situation mostly occurs if the need for variance was not recognized before or when a change in market conditions forces the organization to support more than one alternative in parallel. In the second situation, the existing system behavior is only extended with the new functionality. In both the situations that the conceptual variant functionality did and did not exist as stable behavior, an interface has to be defined between the variable behavior and the rest of the system. Furthermore, an appropriate mechanism and associated binding time have to be selected and the mechanisms and variant functionality have to be implemented. In addition, in the situation where existing functionality is involved, the implemented functionality has to be clearly separated from the rest of the system and re-implemented as a variant that adheres to the variation point interface. In case the binding time is in the post-deployment stage, software for managing the variants and binding needs to be constructed.
2. *Change the realization of existing variation points*. Changes to a variation point may be required for a number of reasons. Changes to a variation point interface, for example, may be required to access additional variable behavior. Furthermore, mechanism changes may be required to move the point at which the variant set is closed to a later stage, while a change to the binding time may be required to increase flexibility or decrease resource consumption. In addition, variation point dependencies and constraints may need to be alleviated. In any case, changes to a variation point may affect all existing variants of the variant set in the sense that they have to be changed accordingly in order to be accessible.
3. *Add or change variant*. When the functionality fits within the existing set of variation points, it means that the functionality at a point of variation can be incorporated by adding a variant to the variant set. This can be achieved by extending or changing an existing variant, or developing a new variant from scratch. These new or changed variants have to adhere to the variation point interface, as well as existing dependencies and constraints.
4. *Remove a variant or variation point*. 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, or that functionality becomes obsolete. The need to support different alternatives, and therefore variation points and variants for this functionality, may disappear. As a response, all but one variant can be removed from the asset base, or the variation point can be removed entirely. If in the latter case one variant is still needed, it has to be re-implemented as stable behavior.

Now that we have established a framework of concepts regarding the derivation process with the two phases and the adaptation aspect, the next question is how these concepts relate to the scope of reuse classification discussed in Section 2.2.

2.4. Relating scope of reuse and product derivation

In the previous section, we stated that products were derived according to a generic process, independent of the scope of reuse. However, there are differences for each of the scopes in the realization of several aspects during product derivation. In this section we discuss those differences.

2.4.1. Standardized infrastructure

Although it provides a first step towards sharing software assets, a standardized infrastructure provides relatively generic behavior. The infrastructure is typically very stable, and very little domain engineering effort is required. Except for creating and maintaining proprietary glue code, almost all effort is directed towards application engineering (see Fig. 6). Therefore, adaptation during product derivation in this type of product family usually only concerns product specific adaptation (see Section 2.3.3).

As the infrastructure contains no domain specific functionality, it cannot fully specify a family architecture, let alone fully document and formalize knowledge to derive a product architecture. It furthermore only documents those parts of component interfaces that are concerned with functionality provided by the infrastructure. Although components may contain

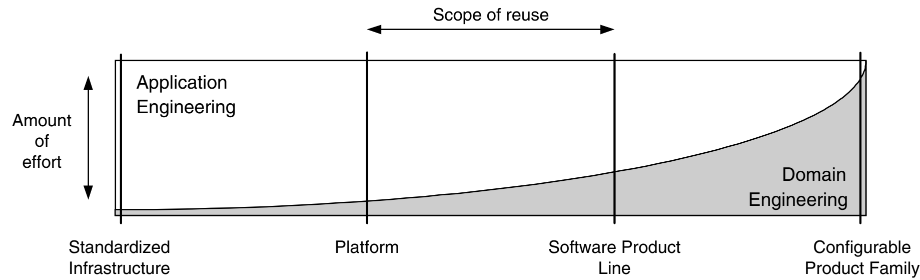

Fig. 6. Proportion of cost in four scopes of reuse. This figure presents the division between effort spent in domain and application engineering for the four scopes of reuse discussed in Section 2.2.

variations, variability can be managed as in traditional software development.

2.4.2. Platform

A platform requires a certain amount of domain engineering to create and maintain the assets that implement the common functionality. The main effort, however, is still assigned to application engineering (see Fig. 6), and adaptation during product derivation is still mainly concerned with product specific adaptation (see Section 2.3.3).

A platform usually lacks the information about specific products constructed on top of the platform. However, products within a platform based product family require more conformance in terms of architectural rules and constraints that have to be followed. These rules and constraints should be explicitly documented. The component interfaces should also be documented, or at least partially.

Since the platform only captures common functionality, the number of variation points is relatively low. The only variation points available are related to variability that cross-cuts all products. Such variations that are common to all products can be captured and managed explicitly.

2.4.3. Software product line

The amount of effort spent in domain and application engineering is roughly equal in the case of a software product line scope of reuse (see Fig. 6). Adaptation during product derivation is concerned with both product specific adaptation and reactive evolution (as described in Section 2.3.3), but the amount of adaptation required highly depends on the stability of the domain and the maturity of the organization.

A software product line provides a product family architecture that specifies both commonalities and differences of the products. For each architectural component, one or more component implementations are provided. For more stable and well understood components, one configurable component implementation exists.

Depending on domain stability and maturity of the organization, for some variation points, the binding time, dependencies and set of variants change frequently, while other variation points are quite stable. Frequent changes make it uneconomical to formalize all knowledge necessary to derive the products. A solution in those situations is to at least partially formalize the specification of the component interfaces for automated consistency checks. The remaining part can remain either documented or tacit.

2.4.4. Configurable product family

A configurable product family typically captures all commonalities and differences in the product base. Most effort in the product family is therefore directed towards proactive evolution. In the occasional event that changes are required they are handled through reactive evolution (see also Section 2.3.3).

The product family architecture is enforced in the sense that no product can or needs to deviate from the commonalities and differences specified in the architecture. The components are often stabilized when this scope of reuse has been reached, and consequently, most components have one configurable component configuration. The return on investment for formalizing knowledge in deriving products using these stable components is therefore substantially higher then in a software product line.

3. Case descriptions

This section details the case studies conducted at Thales Nederland B.V. and Robert Bosch GmbH.

3.1. Thales Nederland B.V.

Thales Nederland B.V., the Netherlands, is a subsidiary of Thales S.A. in France and mainly develops Ground Based and Naval Systems in the defense area. 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 TACTICOS naval combat systems family produced by the Business Unit Combat Systems, more specifically, the Combat Management Systems.

3.1.1. The Combat Management Systems product family

A Combat Management System (CMS) is the prime subsystem of a TACTICOS (TACTical Information and COmmand System) Naval Combat System. Its main purpose is to integrate all weapons and sensors on naval vessels that range from fast patrol boats to frigates. The Combat Management System provides Command and Control and Combat Execution capabilities in the real world, as well as training in simulated worlds.

The asset base that is used to build Combat Management Systems, also referred to as the infrastructure, was first established in the 1990s. It provides a virtual machine (referred to as SigMA) as well as a publish/subscribe communication mechanism through a real time distributed database system (referred to as SPLICE). The common functionality is mainly concerned with handling the real time storage and transportation of message instances, the creation of virtual 'worlds' for training, replay of loggings and tests, as well as handling a duplicated LAN and dynamic reallocation of applications for battle damage resistance.

The asset base is now fairly stable, consists of approximately 1500 kLOC and contains both in-house developed and COTS (Commercial Off The Shelf) components. Each component version consists of the source, the binaries, and a fixed documentation associated with it. The descriptions of the components are stored in a hierarchical component repository, where the top-level denotes the functional component and the leaves the small-sized (10–120 kLOC) individual executables. This component repository and surrounding management tasks play a supportive role within the organization. It mainly provides information and consultancy regarding the use of the components towards several primary processes, handles COTS purchase, component change and maintenance, as well as acceptance of new components to the repository.

Although the Business Unit Combat Systems is in the process of extending the scope of reuse for the Combat Management Systems product family to a software product line, the asset base captures functionality common to all Combat Management Systems and is treated as if it was an externally bought infrastructure. We therefore classify the Combat Management System family as a single product family with a platform as the scope of reuse.

The products built on top of the infrastructure, i.e. the Combat Management Systems, are large and complex technical software systems. An average configuration (including the infrastructure) consists of approximately 37 main components and several MLOC. The infrastructure alone already requires setting several thousand lines of parameters, which often contain multiple parameter settings per line. In the next subsection, we discuss how the process of deriving the products is organized at TNNL Combat Systems.

3.1.2. Derivation process

The derivation process at the Business Unit Combat Systems is highlighted in Fig. 7 and discussed below.

Initial phase. Due to the size of the Combat Management Systems, and the large amount of similarities

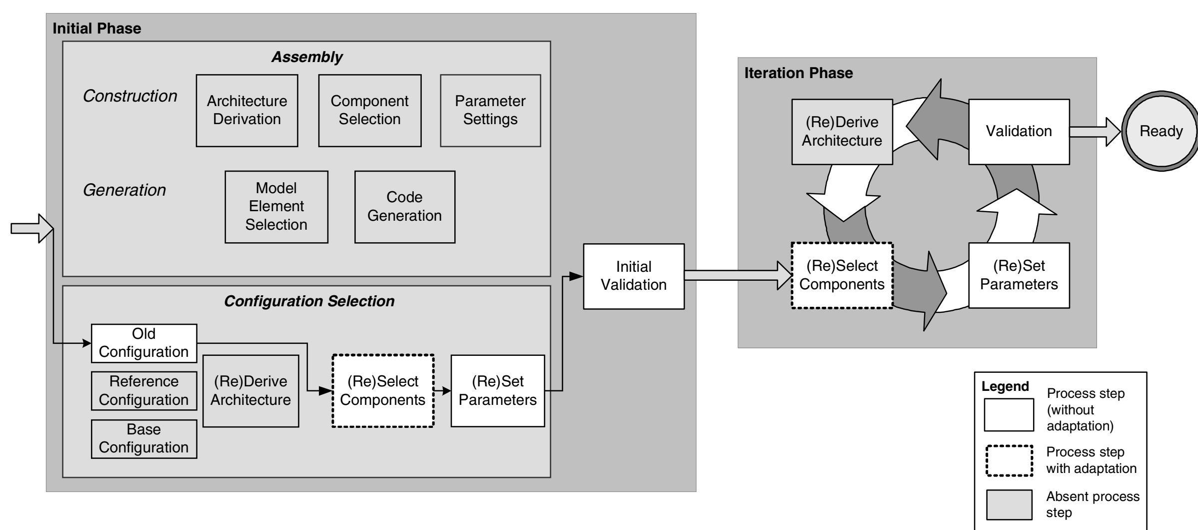

Fig. 7. Product derivation at Combat Systems. The customer requirements are mapped onto a closest matching old configuration, which is modified by re- and deselecting components and parameters. After the initial validation, the configuration is modified in a number of iterations, until the product configuration is deemed ready.

between configurations for similar types of ships, configuration selection is used to derive the initial product configurations (the thin arrow in Fig. 7). To this purpose, the collected requirements are mapped onto an old configuration, whose characteristics best resemble the requirements at hand. In practice, this configuration is usually the most recently completed configuration of the Combat Management System for the same type of ship.

Using this old configuration, a complete specification of the software product is compiled. This specification is used in the rest of the product derivation process. First, the old configuration is modified to comply with the specification. To this purpose, the configuration is modified by re- and de-selecting components, adapting components and changing existing parameter settings. The architecture is not changed explicitly, but implicitly modified by the selection and adaptation activities.

When 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. Until the product sufficiently adheres to the requirements, the configuration is modified in a number of iterations (the thick arrow in Fig. 7), by re- and de-selecting components, adapting components and changing existing parameter settings.

Adaptation. During the (re)selection of components, both reactive evolution and product specific changes are applied when components are adapted. The decision whether component development or change will result in product-specific code or development that concerns the entire product family, is determined through so called Change Control Boards (CCB). Each product development project (which is an application engineering project) has its own CCB that determines whether a request for development for the product family will go to the component CCB (which is part of domain engineering). The component CCB synchronizes the requests of different projects and decides whether and which of the requests will be honored. The changes are financed and performed by domain engineering. Components are also adapted through proactive evolution.

Although all components were well documented at the moment they were initially developed, some of the documentation was not maintained completely when the components were changed. As a result, the derivation process at Combat Systems strongly depends on tacit knowledge. No formal descriptions are used during the derivation process.

When we relate the description of the product derivation process of TNNL Combat Systems to Section 2.3, evidently, the process at Combat Systems is an instance of the generic derivation process depicted in Fig. 3.

3.2. Robert Bosch GmbH

Robert Bosch GmbH, Germany, was founded in 1886. Currently, it is a world-wide 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, respectively.

3.2.1. The product families

The systems produced by the business units consist of both hardware, i.e. sensors and actuators, and software. The main requirements for originate from regulations at various parts of the world, and different market segments (e.g. low-cost, mid-range or high-end).

Product family A captures both common and variable functionality of the product family members. Product family A is therefore classified as a family with the software product line as scope of reuse. The asset base of the product family B includes functionality that is common to many products in the family. It has a heartbeat of two months, which means every two months the set of shared artifacts is updated with customer specific functionality that is deemed useful for most other product family members. The scope of reuse for the product family of business unit B is therefore classified as a platform (see also Section 2.2).

While TNNL Combat Systems develops a few instances of the large and complex Combat Management Systems per year, the business units derive thousands of instances of systems per year. In the following subsections, we discuss the derivation process of both business units.

3.2.2. Derivation process for business unit A

A system that is derived at business unit A, on average contains approximately 50 components, a few hundreds of compiler-switches and several thousands of runtime parameters. The process that is used for deriving these products is highlighted in Fig. 8 and discussed below.

Initial phase. Starting from requirements engineering, business unit A uses two approaches in deriving an initial configuration of the product, i.e. one for lead products and one for secondary products:

- *Lead products*. For lead products, the first configuration of the product is constructed using the assembly approach (upper thin arrow in Fig. 8). First, the architecture is derived from the product family architecture. In the next step, the closest matching component implementations are selected from the repository and subsequently the parameters are set. If, during the selection of the components, no suitable component implementation can be found in the repository,

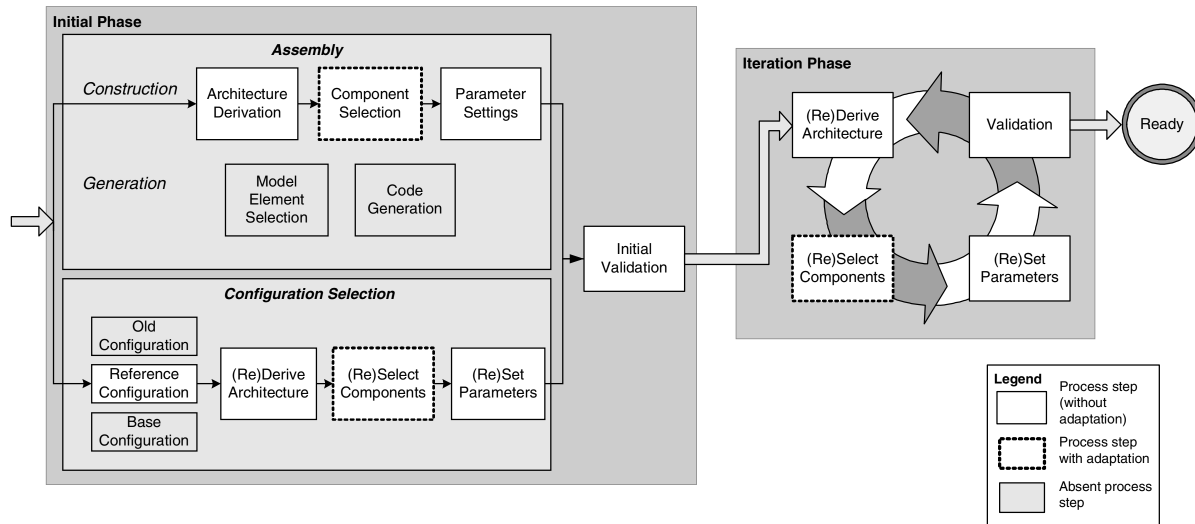

Fig. 8. Product derivation at business unit A. Lead products are derived through assembly, while secondary products are derived using reference configurations. The resulting initial configuration is modified in a number of iterations by reselecting components and resetting parameters. Requirements that cannot be handled using existing components are accommodated by reactively evolving the product family assets during the Component Selection and (Re)Select Components steps.

the most suitable component is copied and adapted where necessary. If the required change involves a completely new concept, the component is developed from scratch (denoted by the dashed box around Component Selection in Fig. 8).

- *Secondary products*. In case a similar product has been built before, the first version of the product is developed by using reference configurations (lower thin arrow in Fig. 8). These reference configurations are lists of components for consistent configurations, and are typically configurations resulting from previous projects that are explicitly designated as reference point. The components from the reference configuration are replaced with more appropriate ones where necessary. As the reference configurations are lists of components, all parameters are still open and have to be set before the iteration step. Knowledge about valid parameter settings is available however, which restricts the range of variability for the parameter settings.

After the initial configuration is ready, the software product is packaged and installed on a test bench for calibration and validating the product with respect to completeness, correctness and consistency of the configuration. If the configuration passes the validation test, the product is deemed ready. If new customer requirements come in, or when the configuration does not pass validation, the derivation process enters the iteration phase.

Iteration phase. In the iteration phase, the initial configuration is modified in a number of iterations by reselecting components, adapting components, or changing parameters (the thick arrow in Fig. 8). At business unit A, up to half of the time spent in deriving a product is consumed by this phase.

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

The component documentation for individual products often comprise thousands of pages, which are quite up-to-date. Furthermore, formal descriptions are available of the component interfaces. All other knowledge about the artifacts is available as tacit knowledge. The engineers indicate this tacit knowledge is still vital for the product derivation process.

When we relate the description of the product derivation process of this business unit to Section 2.3, evidently, the process of unit A is an instance of the generic derivation process depicted in Fig. 3.

3.2.3. Derivation process of business unit B

The product derivation process at business unit B is highlighted in Fig. 9 and discussed below.

Initial phase. Each time a product needs to be derived from the product family, a project team is formed. This project team derives a product by assembly (the thin arrows in Fig. 9). It copies the latest version of the shared components and selects the appropriate components from this copy. Thereafter, all parameters of the components are set to their initial values.

The configuration is then packaged and installed on a test bench for initial validation. If the configuration

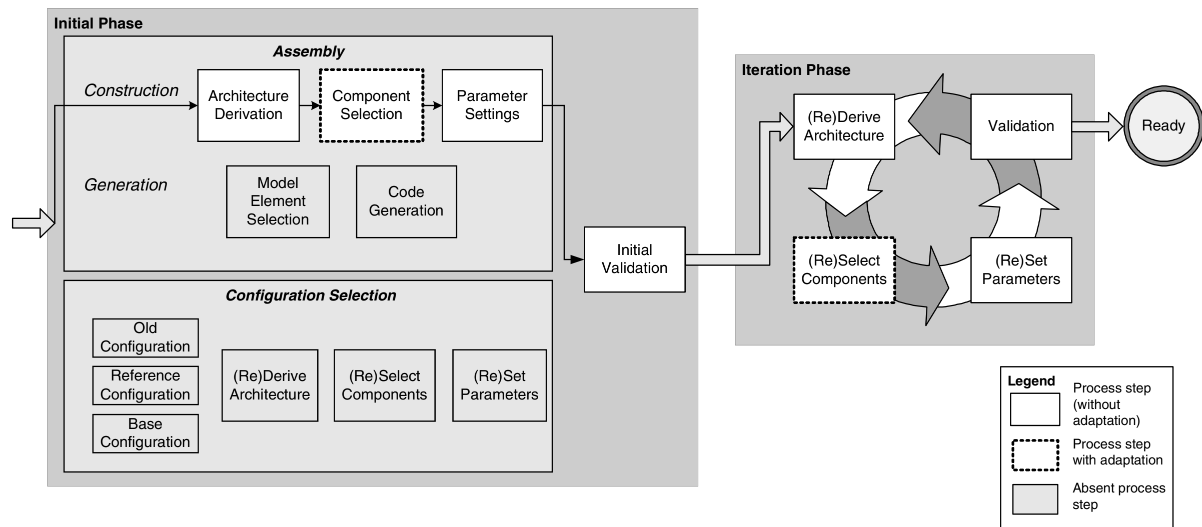

Fig. 9. Product derivation at business unit B. Products are derived by assembling an initial configuration. After the initial validation, the configuration is modified in a number of cycles in the iteration phase. Requirements that cannot be handled by existing product family assets are handled through product specific adaptation during the Component Selection and (Re)Select Components steps.

passes validation, it is deemed ready. Otherwise, the derivation process enters the iteration phase.

Iteration phase. When new requirements come in, or when inconsistencies arise during the validation process, components and parameters are reselected and changed in the iteration phase (the thick arrow in Fig. 9), until the product is deemed finished. Similar to the situation at business unit A, the iteration phase at business unit B consumes up to half of the time spent in deriving a product.

Adaptation. When during the initial and the iteration phase the requirements for a product configuration cannot be handled by the existing product family assets, the changes are applied to the selected component copies, in other words, by product specific adaptation. These changes can therefore not be reused unless the new functionality is integrated into the shared artifacts at the heartbeat.

When we relate the description of the product derivation process above to Section 2.3, evidently, the process at business unit B is an instance of the generic derivation process depicted in Fig. 3.

4. Problems and issues

In the previous section, we described the derivation processes of business units at Robert Bosch GmbH and Thales Nederland B.V. In this section, we present several challenges that these business units face in deriving products from their shared product family assets. Based on a series of interviews and documentation provided by the organizations, we have categorized several problems into three areas. These three areas are knowledge externalization (Section 4.1), variability management (Section 4.2), and scoping and evolution (Section 4.3). We discuss the problems in each section by using a structure that consists of a set of observed problems, examples of a number of those problems, and a discussion involving the causes and forces that are related to the problems.

4.1. Knowledge externalization

In Section 2, we discussed several product family classes in terms of scope of reuse. Organizations within those classes use tacit, documented, and formalized knowledge to derive individual product family members. The process of converting tacit knowledge to documented or formalized knowledge is referred to as externalization (Nonaka and Takeuchi, 1995). Below, we discuss the problems, issues and forces that are associated with tacit knowledge and knowledge externalization from a product derivation perspective.

4.1.1. Observed problems

We identified the following problems in the context of knowledge externalization.

Very high workload of experts. As experts are involved in virtually all steps of the product derivation process (see Fig. 3), they tend to experience a very high workload. On the one hand, the pressure on experts is sometimes experienced as negative by the experts themselves. On the other hand, other staff often mentioned the lack of accessibility of these experts.

False positives of compatibility check during component selection. During product derivation, application engineers select components for addition to or replace-

ment of components in the (partial) configuration at hand. Whether a component fits in the configuration depends on the fact 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 of the selected component, the constraints, and the dependencies. The problem we identified is the situation in which the consistency check does not find any violations, while in later stages (e.g. testing) it turns out some of the selected components are not compatible. This is due to the fact that not all compatibility aspects have been externalized.

Large number of errors in parameter settings due to large amount of parameters with implicit dependencies. A substantial amount of effort in the derivation process of the interviewed business units is associated with correcting errors in the iteration phase. Staff indicates that the error-proneness of the stage at which the parameters are set is caused by the large amount of parameters and implicit dependencies between them.

Identical errors in successive projects due to lack of externalizing important implicit dependencies. Related to the previous problem is the occurrence of identical errors in successive projects. One of the reasons indicated for this problem was the apparent lack of externalizing important tacit knowledge, e.g. the fact that a certain error occurred as a result of a particular implicit dependency, during the derivation of previous products.

Over-explicit documentation decreases traceability of relevant information. Problems with documentation are usually associated with the lack of it. However, over-explicit documentation was identified as a problem as well, as large amounts of documentation for a component decrease the traceability of information that is of particular interest for deriving a specific product. In short, besides good structuring, documentation requires an acceptable quality to quantity ratio in order to be effective.

4.1.2. Examples

Business unit CS. The software engineers at Combat Systems indicate that the documentation of some components contain obsolete parts and inconsistencies. The actual properties of these components are partly known by experts that need to be involved in the derivation process each time such a component is used. In addition, the interviewees indicated that as a result of the eroded documentation, variation points existed from which even the experts did not know what their function was and how they should be used. Such a situation was handled by choosing the same variant each time a new product was derived, e.g. by using a default parameter. In those cases the software configuration seemed to work properly, but was it unknown whether the choice was really the optimal choice.

Business unit CS. 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 step results from correcting unintended side-effects introduced by previous parameter setting steps in the iteration phase.

Business unit A. Part of the compatibility check of 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, frequently, some of the components turn out to be incompatible during validation on the test bench.

Business unit A. In addition to the automated check, component compatibility 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 this amount makes it hard to find aspects relevant for a particular configuration. On the other hand, the software engineers also indicate that, at times, even this amount seems not enough to determine the compatibility.

4.1.3. Causes and forces

The desire of many organizations is to shift towards a situation in which all tacit derivation knowledge is externalized to formal knowledge. In this subsection we discuss why this approach is not feasible for all scopes of reuse.

Tacit vs. explicit

A number of drivers for the desire to shift to documented or formalized derivation knowledge have been identified below.

- *Avoiding knowledge starvation*. A frequently expressed concern by organizations is the increasing dependency on experts during product derivation. Drawbacks are that the organization is vulnerable to knowledge starvation, i.e. losing knowledge if experts leave the organization. Although those risks can be minimized by offering adequate conditions of employment and dividing tacit knowledge over multiple experts, these approaches are expensive and/or time consuming (deriving products in a complex technical environment typically requires several years of training).

- *Avoiding known errors as a result of implicit dependencies*. In addition, tacit knowledge increases the chance that unexpected difficulties arise during product derivation, e.g. due to component incompatibility as a result of implicit and forgotten dependencies. Good documentation and description of dependencies and relations can avoid introducing known faults during the derivation process.
- *Enabling tool support*. Besides managing the variability information of a system, the actual selection and binding of variants in a large and complex system can be a tedious and error-prone task. All tasks that can be automated may therefore significantly reduce costs and increase product derivation efficiency.

Externalization also has some disadvantages, however.

- *Externalization process intervenes with every day business*. One problem is that the externalization process requires participation of a number of experts. These experts are also involved in several customer projects. Spending effort on formalizing tasks thus reduces the availability for other 'every day' business.
- *Externalization expensive and time consuming*. In addition, tacit knowledge is described as something not easy visible and expressible, and therefore difficult to communicate and share with others (Nonaka and Takeuchi, 1995). In general, experts tend to be unaware of all dependencies they use. Documenting and formalizing these dependencies therefore requires thorough analysis during the externalization process. As a result, the externalization process is an expensive and time consuming task.
- *Investment in infrastructure*. Specifically related to automating the steps of the derivation process is the upfront investment required for tool development, setting up the tool environment and training of personnel.
- *Maintenance*. Externalized knowledge 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, resulting in situations where documentation is out-dated and rendered almost useless.
- *Semantics*. Formal component interface specifications often consist of a set of provided and required operations in terms of argument and return types. One of the reasons for the false positive problem as discussed in the previous subsection, is the lack of specification of semantics and behavior in such interface specification. This is a problem as different operations can have different meanings (a temperature delta parameter may indicate a positive or negative delta for example) or are not accessible at all times (e.g. in case of fast producers and slow consumers). A drawback of formal specifications is furthermore that they often do not specify the semantics of dependencies or constraints, viz. they specify that components are incompatible rather then 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, then just the fact that certain components cannot be selected as-is. One reason for this is that it helps in assessing the effort required for adapting components. The lack of semantics furthermore makes it much harder to maintain documented and formalized knowledge.

Related to the first four disadvantages is the fact that the externalization process suffers from the law of diminishing returns (Spillman and Lang, 1924), i.e. that the return on investment in externalization effort decreases as the amount of potentially useful externalized knowledge increases. This particularly applies to situations that involve frequent changes, or changes that are handled by product specific adaptation, as the chance of not being needed in other products is almost certain or at least rather high. In other words, especially in scopes of reuse other then a configurable product family, at some point, the process of externalizing knowledge will be unprofitable with respect to the costs and risks of leaving information tacit. In such a situation it is therefore much more viable to externalize the frequently accessed knowledge and externalize knowledge in a more reactive manner, i.e. through a knowledge management process that ensures documentation of implicit dependencies that frequently result in mistakes.

4.2. Variability management

The ability to derive various products from the product family is referred to as variability. Variability in software systems is realized through variation points, which are introduced at various stages of the lifecycle of a software artifact, both during domain engineering (reactive and proactive evolution) and application engineering (product specific adaptation). During each of the product derivation steps (see Fig. 3) variants are selected which are bound to variation points in the code, compilation, linking, or runtime lifecycle phase. Managing the ability to handle the differences between products at the various phases of the lifecycle is regarded as a key success factor in software product

families. In the following, we discuss the problems and issues that are associated with managing variability during product derivation.

4.2.1. Observed problems

We identified the following problems in the context of variability management in software product families.

Unmanageable number of variation points and variants. One of the problems identified by the organizations is the complexity of the product family in terms of number of variation points and variants. This cognitive complexity causes the process of setting and selecting variants to become almost unmanageable by individuals.

Variation points not organized hierarchically. One of the causes to the first problem is the fact that neither product family explicitly organized variation points. 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.

Consequences of variant selection unclear. At times, in later phases of product derivation, earlier selected variants proved to complicate development as the earlier choice had negative consequences. This problem is slightly different from the compatibility problem discussed in Section 4.1.1, as this problem specifically refers to the qualitative aspects of a consistent configuration.

Limited resources for realizing variability. A typical trend in software system development is for example that the binding of variation points is required to evolve towards later stages in the product lifecycle. Mechanisms that bind late in the lifecycle typically require extra resources, however. This poses a problem in environments with limited resources, such as embedded systems, as this limits the flexibility of the involved software assets.

Variability negatively affects testability. Late binding, combined with the large amount of variation points and variants, makes it virtually impossible to test all combinations during development.

No uniform treatment of variation points over the lifecycle. At the companies, experts dealing with variability were assigned to lifecycle phases, rather than to different parts of the system. As a consequence, related variation points for the same subsystem but bound at different phases were managed by different experts that had limited interaction. Software variability management aims to treat variation points uniformly over the lifecycle. However, as different variability management experts are responsible for different phases in the lifecycle, and those experts have limited interaction, variation points are treated only in the context of the respective lifecycle phases.

4.2.2. Examples

Business unit CS. The instantiation of components consists of setting parameters for each of the contained executables. Not all of these parameters settings in fact apply to a single executable, but to multiple. In other words, there are parameters that should have the exact same value since they indicate the exact same information, such as the number of operator consoles on a ship. Nevertheless, the parameter has to be set for every single executable.

Business unit CS. A Combat Management System is derived by different software engineers in subsequent steps. The interviewees indicated that, although some information is passed between each of the steps, not all rationale behind earlier choices is available in later stages of derivation. This leads to misinterpretation and local optimizations.

Business unit A and B. An example of the limited resource problem was found in business units A and B, which have strict resource requirements. On the one hand, the engineers identify the increasing need to handle more differences between the family members with single flexible component implementations at run-time, but on the other hand suffer from the ever increasing number of component implementations and parameters, as these differences can only be handled by different variants that are bound earlier.

4.2.3. Causes and forces

Many of the problems associated with managing variability are related to the realization of variation points and their usage. In this subsection we describe these forces in more detail and provide the underlying causes.

Variability realization. Over the past few years, several variability realization techniques have been identified (Jacobson et al., 1997; Svahnberg et al., 2002). Variability realization mechanisms typically have a large impact on the performance and flexibility of the 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 variability realization techniques used is often very limited and not always best suited for the problem at hand. We identify a number of causes.

- Especially embedded systems have tight constraints on memory consumption and timing. As variation points typically require extra computational resources such as memory or CPU-cycles, often mechanisms are chosen that minimize the extra overhead.
- The number of variability mechanism used is further limited by the technology, e.g. the programming

language or the infrastructure, used. Programming languages such as C for example, do not include techniques such as inheritance and reflection.

- 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 (Bosch et al., 2001).
- As development time often is limited, variability is implemented in an ad hoc manner, not having considered whether the characteristics of the mechanism used, e.g. resource consumption, meet the required characteristics.
- The 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 run-time binding is prohibited, as the code for certain variants is restricted to be shipped to one customer only.

Although choosing only a few mechanisms simplifies design and implementation of variation points, the drawbacks of disregarding the other mechanisms are not always understood (Bosch et al., 2001). We therefore identify the need for a comparative study of different variability realization mechanisms that investigates the different advantages and disadvantages. In addition, software engineers and software architects should be made aware of the different realization techniques and associated characteristics.

Managing combinatorial explosion. The number of variation points and variants tends to explode in companies that derive either large or many related software systems per year. The ability to manage these vast amounts of variability is key to the success of software product families. Methodologies are therefore needed that not only ensure the minimum number of necessary variation points in the total asset base, but also that minimize the number of variation points that are shown to or needed to be handled by a product developer. One approach in minimizing the choices that have to be made by product developers is by configuration selection as discussed in Section 2. Other approaches include using hierarchy to hide variation points. A more rigorous approach to hide variation points is by moving from a single product family to a hierarchical product family domain scope. A hierarchical product family basically groups variation points and variants to a specific group of products in several layers, thus reducing the number of variation points visible, to those of the group where the product belongs to.

4.3. Scoping and evolution

Several product family approaches, such as described in Clements and Northrop (2001) and Bayer et al. (1999), comprehend periodic scoping, i.e. periodical identification of the assets that should be made reusable, in order to deal with the continuous evolution of a product family. From a product derivation perspective, a distinction between reactive and proactive product family evolution has been identified in Section 2. In the following subsections, we discuss the ramifications of the existence of these types of evolution in relation to product family scoping. In particular, we discuss the resulting problems and issues that were identified in the industrial organizations.

4.3.1. Observed problems

We identified the following problems in the context of scoping and evolution of product family assets:

Repetition of development. In addition to unmanageability as a result of cognitive complexity (Section 4.2.1), the organizations identify that over subsequent derivation processes, development effort is spent on implementing functionality that highly resembles functionality implemented in previous projects.

Scoping of features. During product derivation, it often becomes clear that new features are required for the resulting product. The interviews showed that it frequently turned out difficult to decide whether a feature was product specific or that it should be part of the shared artifacts. In the latter case it was often difficult to decide whether the application or domain engineers should implement the new feature. Practical arguments such as time to market and short term cost frequently caused solutions to be selected that were neither optimal for the product itself nor for the product family as a whole from an engineering perspective.

Road mapping. A problem related to the previous was concerned with the incorporation of new features in the iterative domain engineering process. During the road mapping activity features were under-prioritized and consequently delayed for later releases that turned out to be needed earlier than scheduled. This causes problems as repeatedly common features were implemented as product specific as the domain engineers were unable to quickly enough provide an implementation.

Obsolete variation points not removed. Changes in the solution domain may cause variation points to become obsolete, see also Section 2.3.3. Often these obsolete variation points are not removed and keep being visible within the derivation process. These obsolete variation points contribute to the problem of the cognitive complexity as presented in Section 4.2.1.

Derivation costs higher then expected due to required adaptations. The organizations observed that, frequently, the use of functionality provided by certain shared software assets turns out to require more effort than just setting parameters. Instead, in order to reuse or implement those variants, often some form of adaptation is required. The number of adaptations and its associated costs are hard to estimate beforehand. The

organizations observe that as a result, the derivation process costs often become higher than expected.

Different provided and required interfaces complicate component selection. When changes to the shared artifacts are applied by reactive evolution and the development time is limited, new functionality is often realized by making a new version of a component and later adapting it. This adaptation may influence the dependencies with other components and the provided and required interfaces. When many of the adaptations in the derivation process are applied in a reactive manner, many new versions of components with different interfaces and dependencies are added to the asset base. This hampers the process of product derivation as, even when different versions provide the same functionality, the application engineers have to find out which versions of the component-variants can be connected based on their provided and required interfaces and their dependencies.

Variation point and mechanism considered identical. Software engineers, both domain and application, typically ignored the difference between a variation point and its implementation technique. Rather these were treated as identical, which complicates, among others, evolution as the techniques that are used to implement the conceptual variation point are now intertwined with other functionality provided by the software artifacts. As a result, the variation points in the artifacts are rather rigid with respect to changing their properties. When, for instance, changing the binding time or the phases during which variants can be added for a variation point, the variation point has to be reengineered, the old mechanism has to be removed from the software artifact(s) (remember that it may be spread out over several artifacts), the new mechanism incorporated in the software artifacts and the variants reimplemented to facilitate change. Although this may be acceptable for changes with a low likelihood, we have seen that this is a very typical evolution for a variation point.

4.3.2. Examples

Business unit B. At business unit B, all required changes during product derivation are handled through product specific adaptation. Periodically, the functionality that is deemed useful for the product family is incorporated in the family assets. The drawback of this approach is that during each heartbeat, new functionality cannot be reused across products family members. Combined with the large number of product instances per year, this leads to cases in which, across product groups, the same functionality is implemented multiple times.

Business unit A. In contrast to B, business unit A handles new functionality by reactively evolving the product family assets. In practice, almost each derived product adds one or more component variants or variation points to the asset repository, thereby increasing the cognitive complexity of the product family. To give an example: The asset repository contains several thousand component-variants and several ten-thousands older versions of these component variants. Product derivation comprises selecting several hundreds of components from these component-variants and setting several thousand parameters. This cognitive complexity of the product family, in some cases, leads to repetition of development. An example of this situation is when during component selection, the component variants implementing certain requirements are not found, while they are present. In addition, the large number of parameters frequently lead to mistakes that influence the consistency and correctness of the intermediate product configurations.

Business unit CS. At TNNL Combat Systems, some components are associated with parameter files that contain several thousand lines of parameter settings, with occasionally multiple settings per line. Similar to business unit A, working in an environment with such large numbers easily lead to human mistakes and therefore iterations in the derivation process. The interviewees indicate however, that a number of these parameters are actually not very useful anymore, as they have the same value for each derived product. An example of such a case is the selection of different operating systems for the Combat Management System. Formerly, the Combat Management Systems were designed to be able to run on three different operating systems. Currently, the Combat Management Systems only run on one of these operating systems. It is still possible however, to configure some (especially old) components in such a way that they are able to run on the other types of operating systems as well.

4.3.3. Causes and forces

The main forces associated with the observed problems are related to the three types of evolution presented in Section 2.3.3. We discuss these forces in more detail below.

Product specific adaptation vs. reactive evolution. During product derivation, changes can be handled through product specific adaptation and reactive evolution. Both types of evolution have their advantages and disadvantages, which we discuss below.

- *Product specific adaptation*. The first approach, i.e. handling changes through product specific adaptation, has the advantage that changes that are very specific to one product do not increase the cognitive complexity of the product family assets. Second, product specific changes are generally cheaper and less time consuming then reactive evolution, at least in the short term. This makes it easier to meet project deadlines and budgets. On the other hand, product

specific changes and variants cannot be reused unless an old configuration is copied and adapted, and do not lead to variation points that could be useful for other product family members.

- *Reactive evolution*. The second approach, i.e. handling changes through reactive evolution, has the advantage that new functionality is made available to other products, thus potentially improving reuse and decreasing overall cost for implementing common functionality. A disadvantage is that the change is potentially more expensive then actually needed in the product under derivation. Moreover, functionality in the product family assets that will never be reused in other products unnecessarily increases the cognitive complexity of the assets.

A scoping decision regarding these types evolution 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, there was no explicit asset scoping activity that considered the benefits and disadvantages each time new functionality emerged during the derivation of the individual products.

The drawback of such an approach is that for possibly a considerable amount of functionality, a non-optimal scoping decision is made. Therefore, the product derivation processes should include continuous interactions with the scoping activities during domain engineering. The Change Control Boards as employed by TNNL Combat Systems (see Section 3) provide a good example in that respect.

Product specific variation points. In addition to the scoping issue involving product specific adaptation and reactive evolution, there are two other issues that involve proactive evolution. The first issue involves the observation that within the derivation process the usage of certain variation points often requires some form of change to shared assets. The underlying problem here is that some variation points handle the selection of alternatives well for a particular set of family members, but are rather inflexible with respect to the others. This problem mainly arises if the evolution of the shared software assets is often realized by reactive evolution rather then proactive evolution. One of the reasons for this is that 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, product derivation projects often have an intense focus on time-to-market and project budgets. This pressure is reflected on the activities during reactive evolution, since the changes have to be ready within the time-frame of the project. With this focus, short term benefits often get a higher priority than long term benefits, which basically means that implementing the requirement for a specific product is prioritized over reusability in other products.

Addition vs. removal. The second issue involves the observation that the cognitive complexity hampers the derivation process. 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. In addition to handling all changes through reactive evolution, we identify the existence of obsolete variation points and the lack of removing these obsolete variation points during 'pure' domain engineering activities, such as proactive evolution, as a second important case in which the cognitive complexity is larger then absolutely necessary.

As we briefly discussed in Section 2.3.3, the need to support different alternatives, and therefore variation points and variants for this functionality, may disappear. In addition, 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. Often, these obsolete variation points are left intact, rather then being removed from the assets. This approach has a number of advantages and disadvantages.

On the one hand, removing unnecessary variation points requires effort in many areas. All variation points related to the previously variable functionality have to be removed, which may require redesign and reimplementation of the variable functionality. There may be dependencies and constraints which have to be taken care of. Also, changing a component may invalidate existing tests and thus requires retesting. On the other hand, removing variation points increases the predictability of the behavior of the software, decreases the cognitive complexity of the software assets, and improves traceability of suitable variants in the asset repository. We also note that if variability provided by artifacts is not used for a long time, or removed from the documentation, engineers may start to forget some facilities are there. The interviewees, for example, indicate the existence of parameters from which no one knows what they are for, or what value they should really have. Furthermore, when incomplete documen-

tation leads to a situation in which changes to artifacts lead to the removal of parts of the facilities but not all, artifacts may suffer from inconsistent behavior.

Concluding, shared software assets have to evolve continuously in order to keep the economic benefits of a product family at an optimal level. However, careful thought has to be given to how assets should actually evolve. For example, the asset base should not only increase continuously in terms of the amount of provided functionality, but unused functionality should also be removed where feasible. In addition, we identify the need for methodologies and guidelines that assist software engineers in making well-founded choices with respect to the types of evolution.

5. Related work

After being introduced by Parnas (1976), the notion of software product families has received substantial attention in the research community since the 1990s. The adoption of software product families has resulted in a number books, amongst others, Clements and Northrop (2001), Jacobson et al. (1997), Jazayeri et al. (2000), Weiss and Lai (1999) and our co-author's book (Bosch, 2000), as well as workshops (PFE 1-4), conferences (SPLC 1 and 2), and several large European projects (ARES, PRAISE, ESAPS and CAFÉ).

Several articles that were published through these channels are related to this paper. Schmid (2000) and Kishi et al. (2002) discuss the scoping activity from the perspective of adopting a product family rather then as something that is a continuously running activity that highly interacts with the product derivation process.

The notion of a variation point was introduced by Jacobson et al. (1997). The notion of variability has also been discussed in earlier work of our research group, amongst others, by presenting three recurring patterns of variability and suggesting a method for managing variability in software product families (van Gurp et al., 2001), discussing variability issues that are related to problems presented in Section 4.2 (Bosch et al., 2001), and presenting a taxonomy of variability realization mechanisms (Svahnberg et al., 2002). Several variability realization techniques have further been identified by Jacobson et al. (1997), Jazayeri et al. (2000) and Anastasopoulos and Gacek (2001), while Bachmann and Bass (2001) specifically discuss design and realization of variability in software architectures.

Geyer and Becker (2002) and Salicki and Farcet (2001) present the influence of expressing variability on the application engineering process. They assume a more naïve unidirectional process flow, however, rather then a phased process model with iterations for re-evaluating the choice for variants. Process models that are more concerned with the development of individual software products have emerged over the years as well. The first known published process model is Royce's waterfall model (Royce, 1970), which is often also used to model the abstraction levels or, alternatively, the lifecycle phases of a software product. Other well known iterative development process models are the spiral model (Boehm, 1988), the Rational Unified Process (Kruchten, 2000), and the incremental model (Mills et al., 1980).

The two-dimensional maturity classification of product families presented in this paper is an extension and refinement of the one-dimensional maturity classification presented in earlier work (Bosch, 2002). This one-dimensional classification consisted of the following levels: standardized infrastructure, platform, software product line, configurable product base, programme of product lines, and product populations. The configurable product family relates to the approaches presented by Weiss and Lai (1999) and Czarnecki (1997), who employ generative techniques to derive individual products, and to the Model Driven Architecture approach proposed by the OMG (OMG, 2003). This relationship between product derivation and MDA has furthermore been identified by, for example, Monestel et al. (2002) and Deelstra et al. (2003).

Several industrial case studies have been performed inside our group, e.g. on product instantiation (Bosch and Högström, 2000) and evolution in software product families (Svahnberg and Bosch, 1999). Also outside our group, several case studies have been presented over the years, such as from the SEI (Brownsword and Clements, 1996; Clements et al., 2001) and others (Ardis et al., 2000). The contribution of this paper is that it specifically identifies and addresses issues and problems in product derivation, whereas earlier case studies primarily focused on domain engineering.

6. Conclusions

The work presented in this paper is motivated by the impression that despite the substantial attention in the software product family research community to designing reusable software assets, deriving individual products from shared software assets is still rather time-consuming and expensive for a large number of organizations. We studied the product derivation process for the product families at two large and relatively mature industrial organizations, i.e. Robert Bosch GmbH, Germany, and Thales Nederland B.V., The Netherlands, to investigate the source of those problems.

The aforementioned and other product families we encountered in practice can be classified according to two dimensions of scope. The first dimension, *scope of reuse*, denotes to which extent the commonalities between related products are exploited. The first scope of

reuse is the *standardized infrastructure,* which involves reusing the way products are built. The *platform* consists of the standardized infrastructure, as well as artifacts that capture the domain specific functionality that is common to all products. In a *software product line* not only the functionality common to all products is reusable, but also the functionality that is shared by a sufficiently large subset of product family members. As a consequence, individual products may sacrifice aspects such as resource efficiency or development effort in order to benefit from being part of the product family, or in order to provide benefits to others. Finally, the *configurable product family* is the situation where the organization possesses a collection of shared artifacts that captures almost all common and different characteristics of the product family members, i.e. a configurable asset base.

The second dimension, *domain scope*, denotes the extent of the domain or domains in which the product family is applied. The first domain scope is the *single product family*, where a single product family is used to derive several related products. In a *programme of product families* several product families together form a complete system. A *hierarchical product family* consists of several layers of product families, and a *product population* approach is concerned with reuse of functionality across several domains.

Focusing on the scope of reuse dimension in a single product family domain scope, the derivation process that we generalized from practice consists of two main phases, i.e. the initial and the iteration phase (see Fig. 3). In the initial phase, a first configuration is created from the product family assets by assembling a subset of shared artifacts or by selecting a closest matching existing configuration. The initial configuration is then validated to determine to what extent the configuration adheres to the requirements imposed by, amongst others, the customer and organization. If the configuration is not deemed finished, the derivation process enters the iteration phase. In the iteration phase, the initial configuration is modified in a number of subsequent iterations until the product sufficiently implements the imposed requirements.

Requirements not accounted for in the shared artifacts are handled by adapting those artifacts. We have identified three levels of adaptation. The first level is *product specific adaptation*, where new functionality is implemented in product specific artifacts. The second level, *reactive evolution*, involves reactively adapting shared artifacts in such a way that they are able to handle the new functionality, and can still be shared with other product family members. The third level, *proactive evolution*, is actually not a product derivation activity, but a domain engineering activity that we added for completeness sake. 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.

The derivation process discussed above builds up our product derivation framework. The main distinct characteristics of the product families from our case study are summarized in Table 1. As shown, the derivation processes at both organizations represent a subset of the generic process we discussed above.

Both organizations from our case study face several challenges during product derivation. We have categorized these challenges in knowledge externalization, variability management, and asset scoping and evolution.

Related to knowledge externalization, i.e. the process of converting tacit to documented or formalized knowledge, we identified *the very high workload of experts, false positives of the compatibility check during component selection, the large number of errors in parameter settings due to large amount of parameters with implicit dependencies, the identical errors in successive projects due to lack of externalizing important implicit dependencies*, and the *decreased traceability of relevant information due to over-explicit documentation* as the main problems. In addition, we discussed forces particularly related to problems as a result of implicit knowledge, and the difficulties and economics of the knowledge externalization process.

For variability management, we identified *the unmanageable number of variation points and variants, the lack of hierarchy in the organization of variation points, the unpredictable consequences of variant selection, the limited resources for realizing variability, the negative effect of variability on testability*, and that *variation points are not treated uniformly over the lifecycle* as the main problems. In the forces discussion we discussed the lack of awareness with respect to variability realization techniques and their associated drawbacks and advantages. In addition, we briefly discussed how the

Table 1
Case study characteristics

Product family	Scope of reuse	Initial derivation phase	Adaptation
TNNL Combat Systems	Platform	Old configuration	Reactive and product specific
Bosch A	Product line	Reference configuration and assembly	Reactive
Bosch B	Platform	Assembly	Product specific

The main characteristics of the product families at Combat Systems, Thales Nederland B.V., and two business units at Robert Bosch GmbH.

number of variation points and variants shown to an application engineer can be minimized in order to manage combinatorial explosion.

Related to scoping and evolution we identified *repetition of development, scoping of features, road mapping, lack of removing obsolete variation points, unexpected derivation costs due to required adaptations different provided and required interfaces complicate component selection*, and the fact that *variation points and mechanisms are considered identical by software engineers* as the main problems. In the forces discussion we presented how aspects related to the three types of evolution contribute to many of those problems.

The main contributions of this paper are the product derivation framework presented in Section 2, and the identified problems and issues associated with product derivation as presented in Section 4. The problems and issues are relevant outside the scope of the aforementioned case studies as they do, or eventually will, arise at other, e.g. comparable or less mature organizations. They show that, although software product families have been successfully applied in industry, there is room for improvement in the current practice. Future work of the ConIPF project (ConIPF, 2003) aims to define and validate methodologies that are practicable in industrial application and that address the problems discussed in this paper.

Acknowledgements

This research has been sponsored by the ConIPF project, under contract no. IST-2001-34438. We would like to thank the business unit Combat Systems at Thales Nederland B.V., as well as the business units at Robert Bosch GmbH. In particular, we would like to thank Paul Burghardt, Egbert de Ruyter (Thales), John MacGregor, Andreas Hein and Stefan Ferber (Bosch), for their valuable input.

References

Anastasopoulos, M., Gacek, C., 2001. Implementing product line variabilities. In: Symposium on Software Reusability (SSR'01), Toronto, Canada, Software Engineering Notes 26 (3) 109–117.

Ardis, M., Daley, N., Hoffman, D., Siy, H., Weiss, D., 2000. Software Product Lines: A Case Study, Software—Practice and Experience 30 (7), 825–847.

Bachmann, F., Bass, L., 2001. Managing variability in software architecture. In: Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'01), pp. 126–132.

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. In: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Reusability (SSR'99), pp. 122–131.

Boehm, B.W., 1988. A spiral model of software development and enhancement. IEEE Computer 21 (5), 61–72.

Bosch, J., 2000. Design and use of software architectures: adopting and evolving a product line approach. Pearson Education (Addison-Wesley and ACM Press), ISBN 0-201-67494-7.

Bosch, J., Högström, M., 2000. Product instantiation in software product lines: a case study. In: Second International Symposium on Generative and Component-Based Software Engineering, pp. 147–162.

Bosch, J., Florijn, G., Greefhorst, D., Kuusela, J., Obbink, H., Pohl, K., 2001. Variability issues in software product lines. In: Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 11–19.

Bosch, J., 2002. Maturity and evolution in software product lines; approaches, artefacts and organization. In: Proceedings of the Second Software Product Line Conference, pp. 257–271.

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.

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.

Clements, P., Northrop, L., 2001. Software Product Lines: Practices and Patterns. In: SEI Series in Software Engineering. Addison-Wesley. ISBN: 0-201-70332-7.

ConIPF, 2003. Configuration in Industrial Product Families, contract no. IST- 2001-3448. Available from <http://search.cs.rug.nl/conipf>.

Czarnecki, K., 1997. Leveraging reuse through domain-specific architectures. In: Proceedings of the Eighth Workshop on Institutionalizing Software Reuse.

Deelstra, S., Sinnema, M., van Gurp, J., Bosch, J., 2003. Model driven architecture as approach to manage variability in software product families. In: Proceedings of the Workshop on Model Driven Architectures: Foundations and Applications, pp. 109–114.

Geyer, L., Becker, M., 2002. On the influence of variabilities on the application engineering process of a product family. In: Proceedings of the Second Software Product Line Conference, pp. 1–14.

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

IEEE, 2000. IEEE Standard P1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems.

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

Jazayeri, M., Ran, A., Linden, F. van der, 2000. Software Architecture for Product Families: Principles and Practice. Addison-Wesley.

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.

Kostoff, R.N., Schaller, R.R., 2001. Science and Technology Roadmaps. IEEE Transactions on Engineering Management 48 (2), 132–143.

Kruchten, P., 2000. The Rational Unified Process: An Introduction, 2nd ed., ISBN 0-201-707101.

Linden, F. van der, 2002. Software Product Families in Europe: The Esaps & Café Projects. IEEE Software 19 (4), 41–49.

Macala, R., Stuckey, L., Gross, D., 1996. Managing Domain-Specific, Product-Line Development. IEEE Software, 57–67.

Mills, H.D., O'Neill, D., Linger, R.C., Dyer, M., Quinnan, R.E., 1980. The Management of Software Engineering. IBM Systems Journal 19 (4), 414–477.

Monestel, L., Ziadi, T., Jézéquel, J., 2002. Product line engineering: Product derivation, Workshop on Model Driven Architecture and Product Line Engineering, associated to the SPLC2 conference, San Diego.

Nonaka, I., Takeuchi, H., 1995. The Knowledge-Creating Company: How Japanese companies create the dynasties of innovation. Oxford University Press, New York.

OMG, 2000. OMG Unified Modeling Language Specification, Version 1.3, First ed. Available from <http://www.omg.org>.

OMG, 2003. MDA Guide v1.0.

Ommering, R. van, 2002. Building Product Populations with Software Components, Proceedings of the International Conference on Software Engineering 2002, pp. 255–265.

Parnas, D., 1976. On the Design and Development of Program Families. IEEE Transactions on Software Engineering 2 (1), 1–9.

Royce, W.W., 1970. Managing the Development of Large Software Systems: concepts and techniques, Proceedings of the IEEE WESTCON, pp. 1–9.

Salicki, S., Farcet, N., 2001. Expression and Usage of the Variability in the Software Product Lines, Proceedings of the Fourth International Workshop on Product Family Engineering (PFE-4), pp. 287–297.

Schmid, K., 2000. Scoping Software Product Lines–An Analysis of an Emerging Technology, Proceedings of the First Software Product Line Conference, pp. 513–532.

Spillman, W.J., Lang, E., 1924. The Law of Diminishing Returns.

Svahnberg, M., Bosch, J., 1999. Evolution in Software Product Lines. Journal of Software Maintenance–Research and Practice 11 (6), 391–422.

Svahnberg, M., van Gurp, J., Bosch, J., 2002. A taxonomy of variability realization techniques. Technical Paper ISSN: 1103-1581, Blekinge Institute of Technology, Sweden.

Weiss, D.M., Lai, C.T.R., 1999. Software Product-Line Engineering: A Family Based Software Development Process, Addison-Wesley. ISBN 0-201-694387.

Sybren Deelstra is a PhD student at the University of Groningen. After obtaining an MSc at the University of Groningen, he started work on his PhD under the supervision of Jan Bosch.

Marco Sinnema obtained the MSc degree in computer science at the University of Groningen in August 2001. After one year of working as a software engineer at the University of Groningen, he started as a PhD student in September 2002 under the supervision of Jan Bosch.

Jan Bosch Prof. dr. ir. Jan Bosch is a professor of software engineering at the University of Groningen, The Netherlands, where he heads the software engineering research group. He received a MSc degree from the University of Twente, The Netherlands, and a PhD degree from Lund University, Sweden. His research activities include software architecture design, software product families, software variability management and component-oriented programming. He is the author of a book "Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach" published by Pearson Education (Addison-Wesley and ACM Press) and (co-)editor of three volumes in the Springer LNCS series. He has organized numerous workshops, served on many programme committees, including the ICSR'6, CSMR'2000, ECBS'2000, GCSE, SPLC and TOOLS conferences and is member of the steering groups of the GCSE and WICSA conferences. He was the PC co-chair of the 3rd IFIP (IEEE) Working Conference on Software Architecture (WICSA-3) and is the general chair for WICSA-4. Currently, he is a visiting professor at the University of Alberta, Canada.