Product Instantiation in Software Product Lines: A Case Study

Jan Bosch[1]

University of Groningen
Department of Computing Science
P:O. Box 800, 9700 AV Groningen
e-mail: Jan.Bosch@cs.rug.nl
www: http://www.cs.rug.nl/~bosch

Mattias Högström

University of Karlskrona/Ronneby
Department of Software Engineering and Computer Science
S-372 25 Ronneby, Sweden

Abstract. Product instantiation is one of the less frequently studied activities in the domain of software product lines. In this paper, we present the results of a case study at Axis Communication AB on product instantiation in an industrial product line, i.e. five problems and three issues. The problems are concerned the insufficiency of functional commonality, features spanning multiple components, the exclusion of unwanted features, the evolution of product line components and the handling of initialisation code. The issues discuss architectural compliance versus product instantiation effort, quick-fixes versus properly engineered extensions and component instantiation support versus product instantiation effort. The identified problems and issues are based on the case study, but have been generalized to apply to a wider context.

1. Introduction

The development and evolution of software is expensive and complex, causing few software development and maintenance projects to deliver on schedule, within budget and at the required quality level. For decades, the reuse of existing software has been proposed as the most promising approach to attack these problems. In the earliest proposals, software components represented functions that could be reused [11], whereas later proposals suggested object-oriented programming, i.e. the use of classes and inheritance, to achieve reuse of software [8]. During the second half of the 1980s, it was identified that for software reuse to be beneficial, the components need to be of larger granularity than functions and classes, which lead to the introduction of object-oriented frameworks [8,6] and, later, component-oriented programming [14].
All the approaches discussed above aimed at achieving community-wide reuse of software assets. Although successful attempts exist, in general one has to conclude that the promised benefits of software reuse have not been achieved. One explanation is that the software reuse community tried to overcome two dimensions of complexity

[1] This research was performed while the first author was at the University of Karlskrona/Ronneby, Sweden.

at once, i.e. reuse between software products and reuse over organizational boundaries. In response to this, the notion of software product lines, aiming at reuse between software products, but within an organization has achieved considerable amounts of attention during recent years, e.g. [9, 7, 16, 4]. A software product line exploits the commonalities within a family of products by developing and maintaining a set of core assets, i.e. a product-line architecture and a set of components implementing the architecture. Software products are derived from these core assets by deriving a product architecture from the product line architecture and, subsequently, instantiating and configuring the software components and extending these with product specific code where necessary. Ideally, the core assets are large and the product specific application small. To achieve this, the product requirements should, where possible, be harmonized with the product family. As a consequence, products at the boundary of the family, e.g. high-end and low-end products, may need to sacrifice certain requirements, e.g. feature richness or resource efficiency, respectively.

In this paper we focus on the instantiation of products within a software product line. In particular, we present the problems and issues that we identified in a case study that we performed at Axis Communications AB. Since product line based development differs significantly from the traditional approach to developing software, it is reasonable to assume that new problems or new variants of known problems exist. Although we earlier have published case studies discussing software product lines in general [3], product line components [2] and product evolution [15], to the best of our knowledge, few research results are available that address product instantiation in software product lines.

The primary goal of our case study was to identify the problems and issues associated with the instantiation of software products in software product lines. The reasons for choosing Axis Communications for the case study are that we have had cooperation with this company for several years, it has a few well-established software product lines that have been operational for several years and the company employs a philosophy of openness. Finally, we believe that the company is representative for 'normal' software development organizations, e.g. development departments of 10 to 100 engineers and developing products sold to industry or consumers.

The most appropriate method to achieve our goals with this case study, we concluded, was through detailed analysis of a number of software products, discussions with system architects and studying component and product documentation. For practical reasons, we chose to narrow the scope of the study to products within the same product family. Although the analyzed products belong to the same product family, the variety of configurations concerning both software and hardware gives rise to many interesting problems that can be found in other product families as well.

The contribution of the paper, we believe, is that it identifies a set of important problems and issues associated with the instantiation of products within the context of software product lines. Awareness of these problems and issues allows practitioners to minimize the negative effects and researchers may use these topics in their research.

The remainder of this paper is organized as follows. In the next section, we present the case study company and the software product line that is the focus of our case study. Section 3 discusses the process of product instantiation within the product line.

In section 4, the problems that we identified are discussed, whereas section 5 discusses the issues that we recognized. Related work and our conclusions are presented in section 6.

2. Axis Communication AB

Axis Communication was incorporated in 1984 and developed a printer server product that allowed IBM mainframes to print on non-IBM printers. Up to then, IBM maintained a monopoly on printers for their computers, with consequent price settings. The first product was a major success that established the base of the company. In 1987, the company developed the first version of its proprietary RISC CPU that allowed for better performance and cost-efficiency than standard processors for their data-communication oriented products. Today, the company develops and introduces new products on a regular basis. Since the beginning of the '90s, object-oriented frameworks were introduced into the company, and since then, a base of reusable assets is maintained, from which most products are developed.

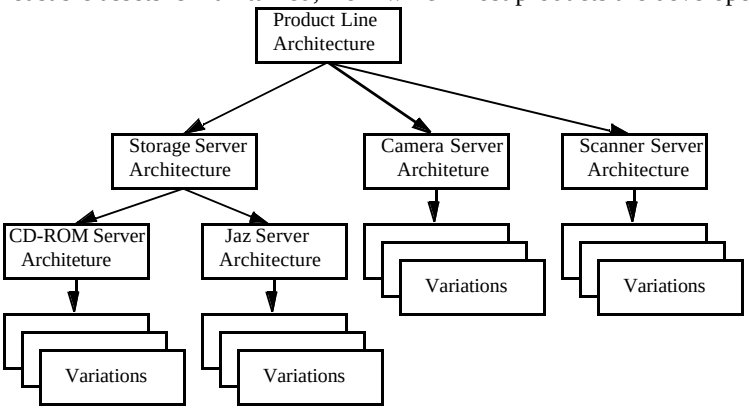


Figure 1. The Axis software product line

Axis develops IBM-specific and general printer servers, CD-ROM and storage servers, network cameras and scanner servers. Especially the latter three product types, i.e. the network devices, are developed and evolved using a common product line architecture and reusable components. In figure 1, an overview of the product-line and the product architectures is shown. The organization is more complicated than the standard case, with one product-line architecture (PLA) and several products below this product-line. In the Axis case, the software product line is hierarchically organized, i.e. there is a top product-line architecture and the product-group architectures, e.g. the storage-server architecture. Below these, there are product architectures, but since generally several product variations exist, each variation has its own adapted product architecture, because of which the product architecture could be called a product line architecture.

Orthogonal to the products, Axis maintains a product-line architecture and a set of reusable assets that are used for product construction. The main assets are a

framework providing file-system functionality and a framework providing a common interface to a considerable set of network protocols, but also smaller frameworks are used such as a data chunk framework, a smart pointer framework, a 'toolkit' framework providing domain-independent classes and a kernel system for the proprietary processor providing, among others, memory management and a job scheduler. In figure 2, the organization of the main frameworks and a simplified representation of the product-line architecture is shown.

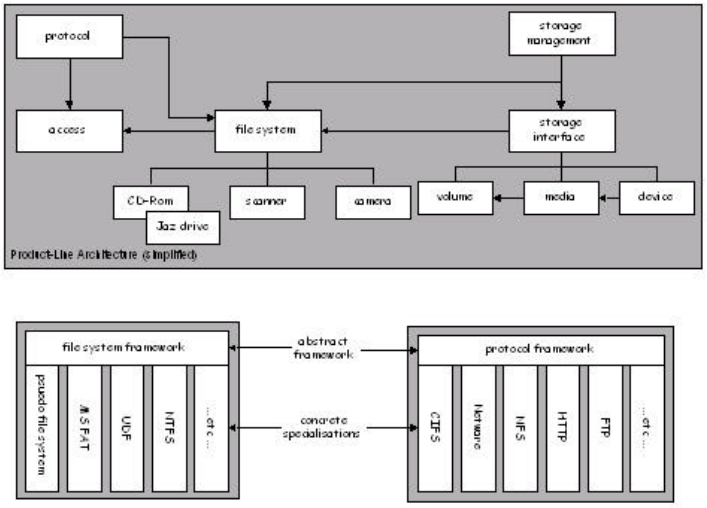


Figure 2. Overview of the main frameworks used in Axis' products

3. Product Instantiation

In this section, we describe the software architecture and the process of product instantiation for the products in the network device product line. We focus on the StorPointCDE100 product, which is a caching CD-ROM server. It is a member of the storage product family, which consists of harddisk, cd-rom, jaz-drive servers. The process of product instantiation consists of three steps, i.e. task scheduler configuration, service configuration and component parametrization. Finally, we discuss the variability mechanisms used in the software product line.

The Architecture

The underlying hardware platform is a proprietary system-on-a-chip solution. The ETRAX 100 chip is a highly specialized and optimized chip designed to boost network performance. It provides built-in SCSI/IDE ATA-2 and 10/100Mbit ethernet interface. It is also equipped with a flash memory that allows remote updating of the software system. Like most embedded systems, the software runs on top of a micro

kernel, in this particular product the OSYS real-time kernel. OSYS is an in-house developed kernel extended with a task-scheduler and a mail-communication package.

Product template

Axis maintains a generic architecture template from which individual products are derived. The template is a skeleton of integration code connecting the kernel, the mailbox-communication package, the job scheduler, the frameworks and the components. The high degree of similarity between the product architectures allows all product configurations to be combined in one single code base. The product specific code is attached to the reference-product through a C-function interface, thereby allowing the use of product-specific, but type-equivalent components. The 'completeness' of the reference product results in that many product-specific variation points have become distributed over the reference product, cross-cutting the architecture. In order to support all configuration subsets, preprocessor directives are used as a mechanism to attach respectively detach components at compilation. This is performed through a special header file, specific for each product configuration. At compilation, the source code of the reference product first passes through a filter (preprocessor), which is used as an adaptation mechanism to obtain a product-specific configuration. Typical filter tasks include the substitution of parameter values, definition of constants and attaching/detaching components and services.
An overview of the architecture, the interaction of the mailbox-communication, tasks and the jobs is presented graphically in figure 3.

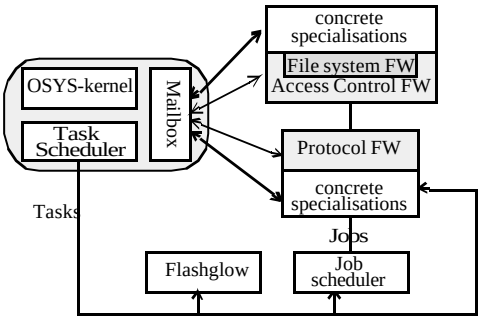


Figure 3. A simplified view of the core product architecture

Since the reference product specifies the component connections, the remaining configuration activities include the definition of the task-list for the scheduler and the parameterization of the mailbox handler.
Task-scheduler configuration. The task scheduler is configured via a globally accessible C-struct. Each product variant supplies its own configuration by explicitly linking it into the final product. Each entry (task) is a function pointer referring to a startup routine in a 'active' component (typically a framework loop), which is scheduled using a round-robin algorithm.

```
struct tasks [ ] = {
          idle_main,
          job_scheduler,
          flashglow,
          protocol_main
};
```

The ‘flashglow’ handles run-time updates of the executable-binary, and the ‘protocol_main’ handles incoming network requests (jobs) and inserts them into the queue of the ‘job_scheduler’.

Service configuration. Services are typically component plug-ins of a framework. Inside each component resides a start-up routine used for initialisation and registration in the framework. When the system boots up, the scheduler posts ‘public_create’ and ‘public_init’ messages via the mailbox. The mailbox itself contains hard-coded function calls referring to each start-up routine for all available components. When the messages arrive, first the frameworks are initialized. Subsequently, the components are initialized and registers themselves at the frameworks. The initialization via the mailbox and component start-up routines provides a uniform way to add and remove services. At compilation a specific configuration can choose whether or not to include a service. From the reference product’s perspective the components and frameworks are detached from each other, but in reality implicit dependencies exist. This is discussed in the next section.

Below, a small code fragment is shown illustrating the use of precompiler directives in the mailbox.

```
MailBox::async_message(<init message>) {
   switch (<message>) {
      #if SMB_INCLUDED
         case SMB_PROG:
            smb_main ( <init> );
            break;
      #endif
      #if APPLETALK_INCLUDED
         case APPLETALK_PROG:
            appletalk_main ( <init> );
            break;
      #endif
      ..etc..
   }
}
```

Component parameterization. Although the frameworks and the framework plug-ins seem independent, in reality there is a hard coupling between the components and the frameworks. The components are not reused at binary level, but at source code level. The source code is a mixture of common code and product-specific code guarded by precompiler directives. Consequently, each product variant needs to specify preprocessor directives that include respectively exclude pieces of code before the product source is compiled into a binary. Both the reference product and the components need to recompiled, when a service is added. Enabling or disabling a

feature frequently requires recompilation of the complete source code tree, since features often have relations to multiple components.
Below, a code fragment from the file system component is shown in which the ISO9660 standard implementation is conditionally included, depending on the F_ISO9660 and FS_9660 precompiler flags.

```
void CDMediumFSPrototypeMap::init( ) {
   #if defined(F_ISO9660) && FS_9660
      this->registerPrototype( new ISO9660Volume( ) );
         //Registers filesystem in the factory
   #endif
   ...etc...
}
```

Summarizing discussion

Axis' product line consist of frameworks, framework plug-ins (components) and a configurable skeleton of integration code. Inheritance and C++ templates are used as means to obtain variability of the core assets. The architecture is characterized by its pervasive use of design patterns, among others Abstract factory, Composite, Observer and Singleton. Axis has taken reuse beyond the availability of core assets by providing a skeleton of generic integration code that, once configured, represents a complete, functional product. This approach has allowed for taking advantage of the similarities within the product family to its full extent.
The products in the product family are functionally similar but technically dissimilar. The hardware differences between products complicates the successful abstraction of common features. Nevertheless, through the extensive use of preprocessing, Axis has managed to do it in practice. Typical types of usage are: definition of constants and parameters, configuration of orthogonal features that span over multiple components, configuration of low-level details hidden behind several abstraction levels, removing calls/references/dependencies between assets and enabling/disabling hard-coded sections of source code. The downside of the focus on the current requirements is that the product line assets are hard to evolve, with consequent effects: increased complexity of the source code, non-trivial impact analysis, decreased reuse in testing and frequent recompilation of the complete source tree. The solution does have several advantages: it maximizes reuse, much more assets can be made "generic", assets can be large without loosing configurability and the size of the executable can be kept small since unnecessary features can be left out. In the next section we have abstracted some of the problems that we found in our case study into more general problems specific to product instantiation in software lines.

4. Problems

Based on the analysis of the products in the software product line, interviews with software architects and engineers and documentation collected at the organizations,

we have identified a number of problems associated with product instantiation in software product line based development that we believe have relevance in a wider context than just Axis Communication AB. Some problems are applicable for product-line development in general, while other problems are specific the embedded systems domain.
The problems presented here share two main underlying causes, i.e. variability and evolution. The variability required from the reusable assets to meet the needs from products in the family creates difficulties because separating general and product specific functionality is generally not trivial. In addition, evolution of the products and the reusable components that define most of their functionality causes solutions and design decisions that were correct at the time they were made to become invalid. The problem is that removing the effects of these solutions and design decisions may be prohibitively expensive.

Functional commonality is not enough

Problem. The reusable components represent the core of software product lines since these capture the commonalities between the products. The more commonalties that can be extracted from the products and captured into reusable components, the smaller the amount of code that has to be maintained. In addition, larger components can be reused more efficient [Szyperski 97].
The problem is that commonalities between products cannot always be captured in generic reusable components, or at least not without considerable effort. A number of instances of this problem exist. First, products may have conflicting quality requirements, which require component implementations that are incompatible with each other. Second, the products may operate on different hardware platforms and it may prove difficult to avoid the effects of these differences in the component. Finally, although all or most products in the family may require a particular domain of functionality represented by a component, different products or subsets of products may require features that cross-cut the component functionality. Since these features need to be optional, including or excluding a feature may have effects at several locations in the component.

Example. The product line of Axis Communications comprises a variety of different products. The products do not share a common hardware platform. They differ in configuration of the hardware chip, among the differences are network connectivity hardware, cache, flash memory and bus architecture (SCSI/ATA-2). The products can however also differ in peripheral configuration, e.g. data media devices (hd/cd/jaz). A product is often released in several configurations focussed on particular market segments. Variants aimed for the high-end market are equipped with expensive hardware peripherals, whereas low-budget variants often lack these. For example, Axis StorPointCDE100 is a caching 100 Mbit cd-rom server while StorPointCD10 is a 10 Mbit non-caching cd-rom server. The two products are functionally almost identical, except for the difference in hardware and the presence of a cache. The commonalties between the product variants do not allow for the development of large reusable assets, because the implementation of the additional features of the high-end

product cross-cuts the component functionality and cannot be handled easily by traditional variability mechanisms.

Causes. One may argue that this problem is caused by a poor decomposition of the system into its main components, i.e. a badly designed software architecture. We investigated this and found no evidence of this being the case. Instead, we became convinced that even an otherwise optimal decomposition may cause such problems. The system does simply not allow for a decomposition that supports the variability requirements for all aspects, e.g. hardware, features, and domain. As we identified in [2], functionality related to the different aspects becomes intertwined early in the development process and varying one of these aspects independent of the others once developed code exists is very difficult.

Single feature spans over multiple components

Problem. In the previous section, the problem was that multiple features affected a single component. However, the opposite situation, i.e. an optional feature affecting multiple components, creates difficulties as well. Configuring a single component is manageable, but some features require changes in the interaction between components, affecting the software architecture and consequently, the initialisation of the components. In addition, an orthogonal feature may affect the implementation and the variability mechanism of individual components. Features that span over multiple components may require architectural variability to the extent that the benefits associated with software product lines are compromised.

Example. As mentioned earlier, the StorPointCDE100 product is a caching cd-rom server whereas the StorPointCD100 product is a non-caching cd-rom server. The collaborating components behave and interact differently depending on the presence of a cache. If a cache is present, a number of configurational issues needs to be dealt with. First, the system must be configured for handling a harddisk (which caches the content on the CDs). Second, the harddisk requires a read-write file system to be added to the system, in addition to the ISO9660 read-only file system used for the CD-ROMs. Third, the cache module needs to be configured (size of cache, block size, prefetching, etc.). Finally, the cache must be embedded in the software architecture of the product, i.e. the new components have to be integrated with the existing architecture and the connections of the existing components need to be reconfigured to take benefit from the presence of a cache. The presence of a cache affects the parameterization and interaction of several components, and hence sections of several initialisation routines must be maintained to support a single feature.

Causes. One can identify two main causes for this problem. First, the reusable assets of a software product line are not optimized for a specific product, but for a family of products. This means that despite the difficulty of integrating a particular feature in the product software, the current decomposition and implementation of the software product line may still represent the optimal solution. The second cause is that loose coupling is usually favored as a means to decrease dependencies between components

and achieve reuse on a higher level, but not all features can effectively be handled late in the configuration process.

Excluding component features

Problem. In our experience, components in software product lines are typically relatively large entities, e.g. 100 KLOC. Several of the companies that we cooperate with make use of object-oriented frameworks to implement the component functionality. Several of these frameworks, typically black-box, implement the superset of the functionality required by the products in the family. However, since the features implemented in the framework typically have relations to other parts within the framework and to other frameworks, it is normally hard to exclude functionality. Especially in the domain of embedded systems, this conflicts directly with, among others, the resource efficiency quality requirement. Variability can, in several cases, not be supported at the component level without including unused features. Large reusable components have a complex internal architecture with dependencies between its parts, which complicate the exclusion of parts of the component.
An additional complicating factor is the fact that components, during evolution, often have a tendency to incorporate implicit dependencies to other components it uses, see also [Bosch 99b]. These implicit dependencies make it even harder to exclude unused functionality from components.

Example. One of the components in the software product line studied in this paper is a network protocol framework that is reused by all products. It supports a plethora of different protocols. Most network protocols are implemented on top of lower level protocols. The protocols implement different abstraction levels, ranging from details about bit transmission to high-level application logic. For, among others, performance reasons, the layers in a protocol stack are rather tightly coupled, which make it hard to exclude unused protocols. The tight coupling results in relatively large components. The problem is that it generally is very hard to exclude code of a protocol due to the hard coupling in the implementation, whereas especially in embedded systems it is important to keep the system 'footprint' as small as possible.

Causes. The main cause for this problem is evolution. Often, the first version of a component supports the variability requirements on the product line at the time of design. During evolution, typically two types of change scenarios cause the discussed problem. First, functionality that up to now has been included in all products is required to become optional because some new product or a new version of some product demands this. Second, new features are added to the component because a relevant subset of the product family requires this. During implementation, it is discovered that the feature affect the component in multiple locations, complicating the implementation of the optionality of the new feature. A final cause is discussed in the next section.

Managing evolution of product line components

Problem. The core components represent common functionality that is shared by the products. For each product, these components are instantiated and configured to support the product specific requirements. The number of contexts in which a component is used may be substantial. During evolution of a component, it may therefore be difficult to verify compatibility of the new version of the component with all contexts in which the component may be used. This results in the situation, especially during development, where products that successfully compiled and ran one day, suddenly exhibit failures on the next. The problem is a software configuration management problem, i.e. how to handle impact analysis on a practical level. Too much analysis slows down production and delays the introduction of new features in products, whereas too little analysis may require additional work to correct new versions of components that caused errors in products that incorporate it.

Example. Axis Communications has a product release schedule that updates the product hardware once or twice per year and the product software three to four times per year. Customers that have obtained Axis' products can download new versions of the software and install it in their product, typically overwriting a flash memory. Axis constantly enhances its hardware to support more and more functionality. Initially, a SCSI interface was the only interface provided for connecting harddisks and cd-roms. But, the low budget market demanded the ATA interface, since the devices are considerably less expensive than equivalent SCSI devices. There are fundamental differences between the ATA and SCSI interfaces. ATA is synchronous and non-multitasking, whereas SCSI is asynchronous and multitasking. Initially, the software architecture and the components were designed for SCSI, but later versions of these assets had to be adapted to support the ATA interface standard as well, though not necessarily at the same time. It turned out to be challenging to incorporate modifications in core components that are used in all product configurations, especially when the components go through extensive product-specific configuration or are used in several contexts, i.e. the component should be able to cooperate with many different components. Maintenance turned out to be complex, tedious and time consuming, since it was hard to do a thorough up-front impact analysis that covers all contexts in all configurations. Incidental dependencies might generate overlap in functionality or other conflicts. Occasionally, implemented modifications had to be removed and reimplemented differently to solve configuration conflicts.

Causes. The first cause is that the core components are shared entities and all products that incorporate these components are affected by modifications. Whenever the core assets evolve, all products evolve simultaneously. Common features get ideally inserted in the core components, and hence it is desirable that a change is compatible with all affected products. The second cause is the high frequency of incorporating modifications in these core assets. This makes it practically impossible to verify compatibility for each single modification.

Initialisation code is scattered and hidden

Problem. Although the reusable components are shared by all or most products, the components are instantiated and configured for each context in which they are used. One problem that we have identified is the location of initialisation code. A reusable component typically contains two types of initialisation code. First, generic initialisation code that instantiates the internal entities and binds them appropriately. Second, context specific initialisation code that configures the product-specific aspects of the component instantiation. Typically, even generic component functionality needs to be configured for product specific details. The ambition of software product lines is that products in the same product family to share as much common code as possible, but at the same time are not forced to duplicate similar code for all variants. The problem is that, due to this structure, initialisation code is spread out throughout the system. First, this causes potential ordering problems during system start-up. Because the code is spread out through the system, it is hard to predict whether all components will have all required information, e.g. references to other components, available at the time of initialization. Second, it is hard to determine whether the initialisation code is complete. If it is not, uninitialized references and similar problems may cause unpredictable system behavior.

Example. The products in the Axis product line are built around a micro-kernel, a mailbox and a task handler. These parts are relatively tight coupled with the product, since those parts manage most of the initialisation for all products. Initialisation can be viewed as part of the reusable components, since it is shared between all products. The problem is that the initialisation code in a layered architecture of components is usually hard to access and modify. As a consequence, the product specific details are actually configured by code from inside the components. One disadvantage is that it limits the ways in which a product can be tailored and new products can be incorporated, and it is not obvious what code actually is used in a specific product or when the code is obsolete and can be removed.

Causes. The primary cause for this problem is there exist two forces that affect the location of initialisation code. On the one hand, one can argue that initialisation code ought to be associated with the instantiated component because it is a logical part of it. For instance, classes in object-oriented languages have their own constructor method. On the other hand, locating the initialisation code for all components in a single component allows for more control over the ordering of instantiation and binding of components. In addition, product-specific functionality and features that affect multiple components can be implemented and instantiated more easily. The optimal balance between these forces is, in our opinion, specific to each software product line. However, typically, generic component initialisation code should be associated with the component and most product-specific initialisation code should be centralized. This consequently leads to initialisation code being spread out through the system.

5. Issues

The problems discussed in the previous section present issues that are plain problematic and need to be addressed in software product lines. In this section, we discuss three issues. Different from problems, issues discuss the balancing between two conflicting forces.

Architectural compliance versus product instantiation effort

One of the primary advantages of software product lines is the fact that newly incorporated features automatically become available to all products in the product line. Since the cost of incorporating a new feature is shared among several products, the maintenance cost per product can be decreased drastically.
Software products are developed based on the product line assets by deriving a product architecture from the product line architecture and, subsequently, selecting, instantiating and configuring product line components. When the product requirements force the product architecture to deviate considerably from the product line architecture, fewer product line components can be used for the product and, consequently, more product specific components need to be developed. However, this may still be the most cost-effective development strategy because achieving the product requirements by adding product-specific code to product line components may require even more effort.
However, if product line members are less compliant to the product line architecture, the advantages associated with product line evolution are diminished because new features can only partially be implemented at the product line level and need to be implemented per product for the remaining part.
Thus, a conflict exists between the ease with which a product can be instantiated and the cost of evolution in the software product line. The software architecture team for the product line has to decide where product architectures may deviate from the product line architecture and where the product line concepts need to be enforced.

Determining value of an investment

New requirements on the software product line assets that have architectural impact can generally be implemented in two ways, i.e. using a quick-fix avoiding architecture reorganization or by a properly engineered, but expensive implementation. It is generally difficult to decide whether to accept the drawbacks of a quick-fix or invest in a proper but expensive upgrade. The software product line approach is an architecture-centric approach. That means that common features are implemented once, but may be reused by any product in the product family. This means that the cost of a feature can be divided over multiple products. Unfortunately, architectural changes may cause ripple effects onto existing products, forcing these product to evolve with the architecture. Proper implementation of a requirement with architectural impact is potentially very expensive.

A quick-fix avoids architecture reorganization and is consequently less effort demanding, but degrades the structure and conceptual integrity of the software product line. Multiple quick-fixes will cause architecture erosion and decrease the economical value of product line assets.
In order to make a well-founded decision, an investment analysis of the different alternatives may be necessary. A large up-front investment may prove useless due to unexpected technical issues or market events. A quick-fix requires less effort and allows for short time-to-market, but are harmful for maintainability of the product line. Since typically no investment analysis is done, quick-fixes are often not replaced after the deadline, for instance because few understand the actual implementation and because it requires effort that otherwise could be spent on adding new features or products to the product line.
The question is which approach provides the highest return on investment. It is easy to criticize quick-fixes, but in some cases it is the preferable approach. The lack of economical models make it hard to show the long-term benefits of a software product line and how it is affected by various maintenance alternatives.

Component instantiation support versus product instantiation effort

Components in a software product line are generally large and relatively complex. Instantiating and configuring a component for a new member of the product line (or a new version of an existing member) may therefore be difficult and time consuming. Since this has to be repeated for all product line components used in the product, the effort required for the instantiation of the product may be substantial.
Although a software component may just provide the source code and the documented provided, required and configuration interface, the developers of the component are able to provide more advanced instantiation and configuration support. For instance, in [13] the use of visual programming tools and domain specific languages is discussed to instantiate and configure object-oriented frameworks. The availability of such tools can decrease the instantiation effort considerably.
Although tools supporting component instantiation provide important support for product instantiation, there are two important issues that should be considered. First, the development of component instantiation support is effort demanding and increases the total component development cost substantially. Second, since more software is associated with a component, i.e. the component code and the instantiation support, the incorporation of new features in the component will become more expensive because the instantiation support must evolve as well.
The balance between providing component instantiation support and product instantiation effort is influenced by the number of new features that need to be incorporated and by the number of product instantiations that incorporate the product. Consequently, component instantiation support is more suitable for stable domains where relatively few new features are incorporated.

6. Conclusions

The notion of software product lines currently provides the most promising approach to increasing reuse of software and, consequently, decreased development cost and time-to-market. Several authors have discussed software product lines. For instance, in [7], the authors discuss an approach to software product lines consisting of application family engineering, component engineering and application engineering phases. Although product instantiation is discussed, the problems and issues discussed in this paper are not addressed. [16] discuss the FAST method, which is a systematic approach creating a software product line with a primary focus on the use of domain-specific languages and code generators for these languages. Other work in the area of software product lines includes [5, 10, 1]. However, none of these publications discusses product instantiation in software product lines in detail. In [4], we discuss the development, deployment and evolution phases of software product lines, but do not discuss the problems and issues presented in this paper.

In this article, we have presented the results of a case study identifying the problems and issues associated with product instantiation. The case study was performed at Axis Communications AB and focussed on their set of CD-ROM products in the networked devices product line. We identified five problems and three issues. The problems we discussed are concerned with the insufficiency of functional commonality, features spanning multiple components, the exclusion of unwanted features, the evolution of product line components and the handling of initialisation code. The issues discuss architectural compliance versus product instantiation effort, quick-fixes versus properly engineered extensions and component instantiation support versus product instantiation effort.

The contribution of the paper, we believe, is that it identifies a set of important problems and issues associated with the instantiation of products within the context of software product lines. Awareness of these problems and issues allows practitioners to minimize the negative effects and researchers may use these topics in their research.

Acknowledgements

We would like to thank Axis Communication AB; in particular Torbjörn Söderberg, Hendrik Ruijter and Mikael Starvik for their valuable comments and encouragements.

References

1. L. Bass, P. Clements, R. Kazman, *Software Architecture In Practice*, Addison Wesley, 1998.
2. Jan Bosch, 'Evolution and Composition of Reusable Assets in Product-Line Architectures: A Case Study', *Proceedings of the First Working IFIP Conference on Software Architecture*, February 1999.
3. Bosch, J., 'Product-Line Architecture in Industry: A Case Study', *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

4. J. Bosch, *Design and Use of Software Architectures : Adopting and Evolving a Product-Line Approach*, Addison-Wesley, ISBN0-201-67494-7, 2000.
5. D. Dikel, D. Kane, S. Ornburn, W. Loftus, J. Wilson, 'Applying Software Product-Line Architecture,' *IEEE Computer*, pp. 49-55, August 1997.
6. M. Fayad, D. Schmidt, R. Johnson, *Building Application Frameworks - Object-Oriented Foundations of Framework Design*, ISBN 0-471-24875-4, Wiley, 1999.
7. L. Jacobson, M. Griss, P. Jonsson, *Software Reuse: Architecture, Process, and Organisation for Business Success*, Addison-Wesley-Longman, May 1997.
8. R. Johnson, B. Foote, 'Designing Reusable Classes', *Journal of Object-Oriented Programming*, vol. 1, no. 2, pp. 22-35, 1988.
9. F. van der Linden (Editor), 'Development and Evolution of Software Architectures for Product Families', *Proceedings of the Second International ESPRIT ARES Workshop*, Las Palmas de Gran Canaria, Spain, LNCS 1429, Springer Verlag, February 1998.
10. R.R. Macala, L.D. Stuckey, D.C. Gross, 'Managing Domain-Specific Product-Line Development,' *IEEE Software*, pp. 57-67, 1996.
11. M. D. McIlroy, 'Mass Produced Software Components,' in 'Software Engineering,' *Report on A Conference Sponsored by the NATO Science Committee*, P. Naur, B. Randell (eds.), Garmisch, Germany, 7th to 11th October, 1968, NATO Science Committee, 1969.
12. Parnas, D., 'On the Criteria to be Used in Decomposing Systems into Modules', *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.
13. D. Roberts, R. Johnson, 'Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks,' *Proceedings of the Third Conference on Pattern Languages and Programming*, Montecillio, Illinois, 1996.
14. C. Szyperski, *Component Software - Beyond Object-Oriented Programming*, Addison-Wesley, 1997.
15. M. Svahnberg, J. Bosch, 'Evolution in Software Product Lines: Two Cases', *Journal of Software Maintenance*, Vol. 11, No. 6, pp. 391-422, 1999.
16. D. Weiss, Robert C. Lai, *Software Product-Line Engineering A Family-Based Software Development Process*, Addison-Wesley-Longman, ISBN 0-201-69438-7, 1999.