Observations from the Recovery of a Software Product Family

Patricia Lago, Hans van Vliet

Vrije Universiteit, Amsterdam, The Netherlands,
[patricia, hans]@cs.vu.nl

Abstract. The problem of managing the evolution of complex and large software systems is well known. Evolution implies the reuse and modification of existing software artifacts, and this means that the related knowledge must be documented and maintained.
This paper focuses on the evolution of software product families, although the same principles apply in other software development environments as well. We describe our experience gained in a case study recovering a family of six software products. We give an overview of the case study, and provide lessons learned, implicit assumptions reconstructed during the case study, and some rules we think are generally applicable. Our experience indicates that organizing architectural knowledge is a difficult task. To properly serve the various uses of this knowledge, it needs to be organized along different dimensions and tools are required. Our experience also indicates that, next to variability explicitly designed into the product family, a "variation creep" is caused by different, and evolving, technical and organizational environments of the products. We propose explicitly modeling invariabilities, next to variabilities, in software product lines to get a better grip on this variation creep.

1 Introduction

This paper reports a list of observations we made about a case study recovering a software product family (PF) from a set of existing software products in the telecommunication domain. A reactive approach has been used [6].

These observations include some lessons learned and a list of assumptions. Lessons learned define experience we gained in applying software engineering techniques to real cases; assumptions formalize example requirements and constraints that quite often exist in our mind and are implicitly applied to production, but that we rationally ignore. If, later on, we reuse existing software artifacts to derive new products, ignoring these assumptions can lead to conflicts or defects. This happens easily if they are not made explicit and formalized in the architecture description. By modeling the invariability of the assumptions next to variability, we may achieve an even better support for software product line evolution.

Even though our observations are derived from a software PF, we think they can be applied in other software production environments as well. Accordingly, we report two general rules that we elaborated from our observations.

The remainder of this introduction gives a flavor of the problems we had with our initial set of products, the chosen solutions, and the actions we took to realize these solutions.

Across a period of about seven years, our research group participated in a series of research projects and contracts in collaboration with industrial partners that involved the development of software-intensive systems in the next generation networks application domain. The software systems had to support a variety of customer-oriented communication services, integrating always newer technologies. The services were adding new requirements, and the service features were increasing in number and complexity.

After the first five years, when the number of products started to grow, we realized we could no longer keep track of all the knowledge necessary to manage all existing products and their associated documentation.

The next developments could potentially reuse existing assets (e.g., software-configuring network resources, components implementing some selected service features) and modify them to derive new products. Unfortunately, the effort required to reconstruct the knowledge about certain reusable assets was so high that it would have been more cost-effective to build them from scratch.

As we were also willing to reconstruct the family of software products in our university laboratory to make it accessible to and usable by students and researchers, we decided to make an inventory of our maintenance and reuse problems. We identified the following problems and their possible solutions:

manage evolution: The PF became unmanageable: it was too complex in terms of cross dependencies among architectural entities and with the underlying technology. For instance, we could not reconstruct which components developed in past projects where using which version of the Java Virtual Machine, or how to properly configure the platform resources (like router or voice gateway) to deploy a certain service. We identified this as a knowledge management problem.
To solve this problem, we needed two things: (1) traceability support from architecture descriptions to implementation, so we could associate design elements to things such as source code, configuration files, and documentation, and (2) a classification of the reusable assets, so we could represent services at a high level, in terms of service features, and then reuse existing design and implementation assets by using the classification.

plan reuse: There was no planned reuse schema: theoretically, features could be freely selected and combined to build new products. In practice, the implications of these combinations where too difficult to reconstruct (e.g., to verify feature compatibility and completeness), hence making reuse more expensive than building from scratch.
To provide better support for reuse, we needed a domain model that (1) offered a formalized feature organization, and (2) mapped domain features to implemented ones.

support communication: The features part of the different products could not be communicated to stakeholders with different skills and background in a straightforward way. The stakeholders involved in our developments included

- developers with general software implementation skills, but no experience in the telecommunication domain. Further, as they usually were not involved in

previous developments, they had to learn everything about the existing software.
- managers working for our industrial partners, with poor or no technical skills
- university colleagues, to whom we had to explain the various research issues (e.g., the extension of general-purpose modeling notations to represent domain-specific properties)

To better support the communication of service features toward and among these stakeholders, we needed three things: (1) visualization/abstraction mechanisms generating views tailored to the type of stakeholder they address, (2) traceability from features to architectural and implementation assets, so that knowledge could be navigated easily, and (3) tool automation aiding the generation of views reflecting selectable stakeholders' criteria.

These problems motivated us in studying and applying techniques devised for software product lines (or families[1]) and trying to enhance them to fulfill our requirements. We defined a notation and developed associated tool support [16] fulfilling the requirements in the following way:

classification of features: We characterized the application domain by defining the types of service features that describe it. We articulated them via a mechanism merging the feature graph used in software product line engineering to define design decisions and their relationships [4,13], and the utility tree used (e.g., to define software quality requirements [2]).

Next, we defined features on two abstraction levels: (1) at the PF Level, where features reflect generic functionality not necessarily mapped to concrete solutions currently implemented, and (2) at the Product Level, where features are concrete assets implemented by one or multiple products, each providing its own design and implementation solution.

Concrete features at the Product Level are linked to the generic features (at the PF Level) for which they provide a product-specific solution. The types of features and relationships between them, as well as visualization conventions are inspired by existing notations [20]. Further details are given by Lago, Niemelä, and van Vliet [15].

traceability: We defined the whole chain of links between the representation of features in the PF and family members, their architectural solution, and the associated documentation and implementation artifacts. Generic features (at the PF Level) keep the traces to the concrete features (at the Product Level) implementing them. Further, at the Product Level, the structure of each software system is modeled by a component diagram showing which structural elements (objects, interfaces, components) realize which concrete feature(s). Next, structural elements are linked to implementation files and documentation, realizing traceability down to code. This link chain supports traceability in a smooth way, a requirement put forth by Weiler [24]. In the literature, no other comprehensive method covers the representation of the complete life cycle of a product family. Instead, the common approach is to focus on a single [9,21], or a few development phases [1,10,23].

[1] In the context of this paper, we use the terms *product family* and *product line* as synonyms.

view generation and tool support: As described by Lago, Niemelä, and van Vliet [16], we developed tool support for representing the knowledge about a software PF and supporting traceability from features to code. Further, we implemented an initial solution to view generation: to aid the stakeholder in defining his/her personalized views on the feature representations, we implemented filtering mechanisms that allow to select the feature- and association types to be visualized.

After defining the notation and associated tool support, we carried out a case study (summarized in Section 2) that covered the whole series of projects and had a twofold objective: (1) to put our PF under control for future evolution, and (2) to assess whether our method was really meeting our needs. In doing this, we drew our list of observations.

The remainder of this paper is structured as follows. Section 2 reports the starting point of the case study and summarizes the process we followed. Section 3 contains the core of the paper, describing the observations we made after the case study was done. It includes some lessons learned and the assumptions that were implicitly made about our software systems. Section 4 draws some general rules derived from our observations that we think are widely applicable. Section 5 concludes the paper.

2 The Communication PF

2.1 Starting Point

Our case study had the following starting conditions:

- The case study had to recover six existing products developed for different projects/contracts with industrial partners. These products belonged to the same application domain–software services running on an integrated Telephony-Internet network [18].
- These products were made of different components (i.e., they were varying in space [5]), and, in some cases, they were also including the same components but in different versions. This was introducing a second dimension to the variability management problem–variability in time [5].
- Each product was made of about 20,000 SLOC (only application code, excluding middleware such as java libraries, basic distributed communication components, and the like). As compared to the many experience papers published in the literature (e.g., [3, 7, 12, 19]), one could claim that our PF is rather small in size. Still, we believe it is representative as some type of "boundary example": in spite of its relative small scale, its evolution became unmanageable.
- The application domain (advanced communication services on converged Internet-Telephony networks) is naturally extensible (e.g., by adding new features) and combinable (by composing features in different ways to build new services).
- To manage its increasing complexity, we first tried to reorganize the software architecture of the existing products according to a common architectural style (three-tiers with peer components). This helped to better organize the implementation files. More specifically

  - Others successfully adopted a similar approach; for example, by defining an ADL tailored for multi-tier architectures [22] or by organizing software product lines in *federated architectures* made of multiple generality layers [8]).
  - Our case is similar to the case study described by Faust and Verhoef [8], in that they also identify at least three layers. In our case, we defined the application layer for client-side interactive components, the middleware layer including the components implementing general-purpose service features, and the platform layer for proprietary network technologies [14].
  - At the implementation level, the three tiers help differentiate the general-purpose components from the service-specific ones. This is a first step to defining commonality and variability.

  A three-tier architecture by itself however is not sufficient for achieving effective reuse.
- We decided to reverse engineer our six products, define the related feature and component models and traceability across them, and validate the support obtained by deriving a new product. To do this, we had to start with a domain engineering step: define a classification of the features in the domain to characterize the problem space.

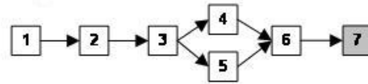
2.2 Description of the Case Study

Our case study started with the reconstruction of the characteristics of our PF, and its evolution, summarized in Figure 1. The graph on the top side of the figure shows the precedence order in which the products were developed. In particular, products 4 and 5 were developed in parallel, starting from the same baseline (product 3). Product 7 was developed as part of our case study, to test the usefulness of our approach.

The table in Figure 1 shows the main aspects of the evolution of these products. Besides project duration and project objective (the latter expressed in terms of type of product and type of development), the table gives an idea of the evolution of the various components. The three central columns characterize the components of the three architectural tiers, whereas the last column shows the technology used to support distributed communication (i.e., the middleware platform). All components are represented by a keyword (e.g., “A” stands for authentication and “Tel-Adpt” stands for “Telephony Adapter”). The columns represent all the components that make up the product. When a component is missing, it means that it has been replaced by some other, new or modified component. When a component is in boldface, it means that some modifications have been applied to it. If a component is repeated as is, it means that it is part of that product with no modification at all.

For example, in project 2, components A and FS are reused as they are, components UP and CC are modified (renamed UP’ and CC’ respectively), and all platform service components have been replaced by three new components.

From this gross picture, we can sense how difficult it is to formally trace the evolution details. For example, project 2 eliminated many middleware service components (DS, PC, and IH). What happened to the functionality provided by these components? Is it completely deleted? Is it included in some other modified components?



<table>
<tr><th>Project</th><th>Duration (months)</th><th>Main development scope, goal</th><th>Application components</th><th>Middleware service components</th><th>Platform service components</th><th>Communication Technology</th></tr>
<tr><td>1</td><td>24</td><td>Infrastructure, integration</td><td>R | P | C | UP | UPadm</td><td>A | UP | DS | FS | PC | IH | CC</td><td>Tel-Adpt | Int-Adpt | Int-I</td><td>CORBA (IIOP)</td></tr>
<tr><td>2</td><td>12</td><td>Infrastructure, porting</td><td>Idem</td><td>A | UP' | FS | CC'</td><td>Tel-S | T2I | I-S</td><td>CORBA (IIOP)</td></tr>
<tr><td>3</td><td>12</td><td>Infrastructure, porting</td><td>Idem</td><td>A | UP^{2} | FS | CC'</td><td>I-S'</td><td>CORBA (IIOP)</td></tr>
<tr><td>4</td><td>08</td><td>Service, development</td><td>R | P | C | UP | SU | L | SM</td><td>A | UP^{3a} | FS' | CC^{2} | S | L</td><td></td><td>CORBA' (IIOP),
EJB and Servlets (RMI/HTTP)</td></tr>
<tr><td>5</td><td>08</td><td>Service, development</td><td>R | P | C | UP | UPadm | GM | GTM</td><td>A | UP^{3b} | FS^{2} | CC^{3} | GP</td><td></td><td>CORBA (IIOP)</td></tr>
<tr><td>6</td><td>12</td><td>Service, integration</td><td>R | P | C | UP' | SU' | L' | SM' | GM' | GTM'</td><td>A | UP^{3a+b} | FS^{3} | CC^{4} | S | L | GP</td><td></td><td>CORBA' (IIOP),
EJB and Servlets (RMI/HTTP)</td></tr>
</table>

Fig. 1. Overview of the PF development projects

There is no way to keep track of such information, unless we explicitly document it. Indeed, the new component CC' merged the functionality of CC together with the generic profile control (PC) functions, invocation handling (IH) became obsolete because of new communication technologies, and component UP' covered the service-dependent PC functions.

Another important issue concerns parallel development: as we developed products 4 and 5 in parallel, starting from product 3, their components are alternatives. This is syntactically represented by the component name (e.g., component UP2 is modified into components UP3a and UP3b): the semantics behind these versions must be described elsewhere.

As a last example, the last column shows that the same CORBA platform was used for the first three projects, and that project 4 modified it. But why? The reason behind this change is that project 4 decided to develop its new components on RMI over HTTP by using an EJB container. Next, some tests revealed that the old CORBA platform was not compatible at the protocol level (i.e., the IIOP - RMI mapping was not working), and we had to move to another compatible CORBA platform. Hence, in this case, the change was not due to direct functional or quality requirements, but to technology immaturity instead.

After the reconstruction of the characteristics and the evolution of our PF, our case study was organized in two main phases shown in Figure 2. The *first phase* focused on the recovery of the PF for the existing products (products 1-6), that is, for each product, it iteratively studied the code, design, and documentation to reconstruct the PF Feature Map, its mapping on the Product Feature Map, and all traceability information down to code. The *second phase* started from the produced Feature Maps and traceability

information to derive a seventh product. This second phase produced a feedback to the first phase, updating the Feature Maps to include the new product in the PF.

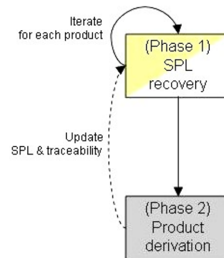


Fig. 2. Process overview: SPL recovery and product derivation process

The details of the two phases are given in Figure 3, which presents the steps concerning the PF recovery in white and those deriving product 7 in gray.

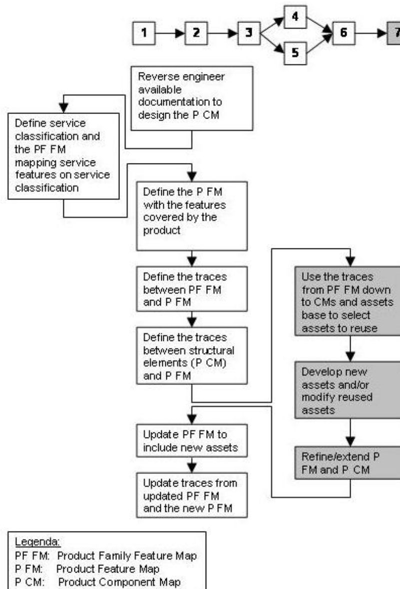


Fig. 3. Process overview

A detailed description of these recovery steps and the differences between the family products are provided by Lago [14].

3 Post-Mortem Observations

When our case study was concluded, we were able to describe some observations in terms of lessons learned and assumptions that were made implicitly about the invariability of the software products but not documented anywhere. By making these invariabilities explicit (as "stability points") next to variabilities (as "variation points"), we can get a better grip on the *variation creep* due to the different dimensions in the evolution of a software product line.

3.1 Lessons Learned

Traceability for knowledge communication. Traceability is usually considered a necessary mechanism supporting software development and maintenance [24]. Traceability can also be used as a fundamental means of communication, for transferring the knowledge about an existing system to "outsiders." Traceability allows one to dynamically choose which aspects are more interesting than others, and hence examine them more deeply. In other words, thanks to traceability information, one can tailor the knowledge he/she wants to look at on the fly, and this implicitly creates personalized views. Of course, tool support for traceability is a fundamental factor in successful knowledge communication.

Abstraction for complexity control. We used two different abstraction mechanisms to control the complexity of a PF and its members. The *separation between PF Level and Product Level* draws a fundamental line between the model of our application domain and the population of existing products. The first belongs to the problem space and reflects the core business of the company; the second belongs to the solution space and makes up our reusable assets base. This separation into two levels helps the stakeholder reason, for example, about the market share that the company wants to target, or about the product-specific knowledge acquired (in terms of reusable assets).

The second abstraction mechanism we used is *filtering and view generation*: when we want to examine some specific aspects about one or multiple products, we need support to select these aspects among all the information gathered. Filtering supports the stakeholder in stating the aspects he/she wants to look at, and view generation based on the selected aspects provides visualization focused on these aspects only. If the filter is well defined, the generated view can even be used as the starting point to derive a new product.

A difficult problem to solve concerns the management of consistency among all PF views, especially if generated views must be stored and kept up to date. Let us consider two examples. If we want to create and keep a view showing only those features related to security, we want this view to dynamically reflect all the components implementing security control in all products. If new products are engineered that add new components supporting new security mechanisms, we want to be able to find back these new components in our "security-focused" view. As a second example, if we want to derive a new product and generate a view representing its initial Product Feature Map and linked Component Map, we will probably add/delete some elements and/or modify some others. Here, we need to carefully identify which elements are shared between different products and which instead are created as modified versions of elements. Of

course, all these modifications must be traceable at both the PF and Product Levels. This problem can be partially solved in an automated way (see the next item), but it needs further investigation.

Automation for product derivation. As introduced before, when we want to derive a new product by reusing existing assets, we need automated support at the Product Level for the initial composition of this new product: once its required features are chosen, we look for the components implementing them and add/remove/modify them at needed. This all means that we create a new Product Feature Map for the selected features and a new Component Map for its solution. Both need automation to discover inconsistencies and support careful reuse. Also, their traceability from the PF Level must be kept, and tool support can at least provide a tentative solution; for example, to include a new feature in the PF Feature Map or to add a new component as a new solution for an existing feature. Further investigation is needed in this respect, too.

Integration of architectural information. In our case study, we used an approach that is mainly feature oriented. Architectural information is partially captured at the Product Level by Component Maps, but the overall architecture is only represented by documentation decorating the PF Feature Map. For example, architectural styles and patterns, or reference architectures, play a secondary role in development by providing descriptive documentation that can potentially be inconsistent with design and implementation.

We think that a structural viewpoint similar to the Product Feature Map should be part of the PF Level as well and that the dependencies between architectural and feature-oriented information need to be made explicit as traceability information.

3.2 Assumptions

The post-mortem assessment carried out after the recovery of our PF identified a set of assumptions that were made implicitly during development and made explicit afterwards. The following presents the main assumptions we reconstructed. They are classified in three types: (1) technical, (2) organizational, and (3) management [17]. *Technical* assumptions concern the technical environment in which a system is going to run: programming languages, database systems, operating systems, and middleware software are examples. *Organizational* assumptions reflect the company as a whole and its social settings and principles. These assumptions concern the organizational aspects in the company brought implicitly into software development. Examples include the workflow, the organizational structure reflected in development teams and departments, and the technology adopted as a company standard. Organizational assumptions can refer to the organization developing the software product or the one using it. Lastly, *management* assumptions reflect the decisions taken to achieve business objectives. These assumptions concern the solutions and the operational tasks to achieve organization-level objectives. Examples include management strategies and plans, experience brought into projects, and expansions toward new/different market segments.

Earlier work investigating the relation between assumptions and software architecture concerns technical assumptions at the component level [11]. This resulted in the notion of architectural mismatch. We look into assumptions from a broader perspective: we consider both nontechnical assumptions and assumptions that result in crosscutting issues.

Technical Assumptions

Tracing base technology versions. A product typically relies on a certain configuration of the software environment in which it can execute. This software configuration includes both special-purpose technology (e.g., a certain software platform for distributed communication) and general-purpose technology (e.g., a certain version of the JVM). We usually keep track of the first technology type, while we implicitly assume that the second type of technology is in place. This, of course, leads to inconsistencies and conflicts when we reuse components originally based on different technologies and hence when we need to reconstruct their original execution environment.

Heterogeneity in base technology. Special- and general-purpose technologies are different for different components within the same product. We usually assume that the execution environment is an atomic unit and that, once in place, it automatically supports all the components. In general, this is true if we consider the product as a whole. When we split up the product to reuse some of its components, we have a problem, because we do not know anymore exactly which technologies are needed for the components we are reusing. This leads to unexpected conflicts if we assume that two components that are functionally compatible will also rely on the same technology type. For example, two CORBA components are compatible provided that they are developed for the same CORBA platform or for two platforms that implement the same standard interfaces and associated behavior.

Information management. Similar to underlying technology, we need to make explicit the assumptions about the data exchanged or shared between different components. Compatibility of the data formats used for information exchange is a known problem, and it is usually described in the specification of component interfaces. Often, components share information stored using a certain external data storage system (e.g., a database), or they manipulate or use information that is partially overlapping or complementary. This implies that we need to make the assumptions about the storage system explicit, as well as those about the overall logical information schema. Otherwise, we might reuse components by selecting versions that do not use the same database or that rely on inconsistent logical information schemes.

Security. The first family product we developed in 1998 implemented secure communication by using SSL and encryption implemented by a US package that we acquired for academic use only. In 2001, we developed the fourth family product. One requirement was to support user authentication as provided by an authentication service implemented in application servers. At that time, the Orion application server became available as free software. It was providing integration with the CORBA/IIOP protocol and a security API supporting both SSL and various encryption algorithms. Unfortunately, the CORBA platform we were using revealed many incompatibilities, and we had to change the CORBA platform too. This had an impact on all the components and required extensive module and integration testing. During the design of this fourth product, we realized that we *implicitly* assumed that all the components where going to be based on CORBA, and we had not foreseen the integration with different middleware platforms as a possible future requirement.

Organizational Assumptions

Modularization versus company expertise. In the design of our first family product, the overall software architecture was organized in components partially reflecting the business role played by the different industrial partners. As partners were geographically dispersed and (most importantly) played well-established business roles in the market, it was politically important to clearly separate service-specific features from service-common ones. Service-specific features provide functionality that is specific to a particular type of application (e.g., video on demand); they are commercialized by service providers. Service-common features provide basic functionality reusable across different applications; they are commercialized by vendors and network providers.

In a similar way, service-specific behavior and communication management are separated. To ensure this separation, additional components were needed to mediate the interaction between the two levels. In both cases, the distribution of functionality on components reflects the business model of the application domain. This can lead to lower quality (e.g., decreased performance) in the final product. It is therefore important to make explicit in the documentation which modularization decisions are driven by the business model.

Management Assumptions

Company participation in standardization bodies. The second family product was developed for an industrial customer collaborating with third-party industries. In this collaboration, components developed by different companies had to be integrated to test their compatibility. As our customer was acting in the telecommunication sector, integration was crucial for future interoperability, as stated by international regulatory bodies. During integration, we found out that some of these companies had the mandate to be compliant to certain telecommunication interface standards. All the components handling communication were impacted by this decision, and all the interfaces had to be changed to reflect this standard, even if, in some cases, this implied lower performance and tricky solutions.

End-user communication as a service. At the PF feature level communication has been modeled as a feature belonging to the platform level only, that is, it was implicitly assumed that all users were having their own communication application deployed in the terminal. Moreover, it was somehow "obvious" that the PF was about offering communication service features to customers. Therefore, under the "End User Functionality" feature category you cannot find any end-user service feature supporting communication.

4 General Rules

From the observations and assumptions described in the previous sections, we are able to draw the following generally applicable rules.

Documentation rule. Documentation about the external environment should be provided for each reusable asset individually. For example, in a component-based system (like ours), each component should be decorated with all the information describing its

external environment. Instead, current practice usually provides this documentation at the system level, once for the complete system. This finer-grained form of documentation helps to reason about the technological assumptions that we sometimes left implicit and that play a significant role (e.g., when reusing existing software elements).

To apply the documentation rule, we devised the following component template:

```
ID Component
   % Component properties
   Developer IDs'
   Component tier [A|M|P]
   Component assumptions
   Interface ID % for each interface
             Interface assumptions

   % Environment configuration
   % Distributed communication platform
   Middleware ID
             Middleware protocol type
             Middleware protocol standard ID

   % Environment configuration
   % General purpose platform
   Programming language
             Compiler/Interpreter version
   Data storage type/ID/version
   ...

   % Specific platform technologies
   ...
```

Analysis of the organizational/management impact. Once the software architecture of a system is defined, we should always assess it against the structure of the organization and its management issues. This helps make organizational and management assumptions explicit, so their embodiment in the software architecture is analyzed and verified explicitly whenever the system evolves. For example, we observed in the previous section that participation in standardization bodies and business roles played by industrial partners have an impact on the architecture and its quality. This impact should definitely be a deliberate choice coming from careful evaluation of the advantages and drawbacks.

5 Conclusions

This paper describes experience gained in a case study recovering a software PF. It reports a list of lessons learned, a list of implicit assumptions reconstructed after the case study, and some rules that we think are generally applicable, together with a template for the documentation of reusable software components.

The general objective of our case study was twofold: (1) to put our PF under control for future evolution and (2) to assess our method. We verified the achievement of our objective by successfully deriving a seventh product. This derivation phase was carried out by a person not previously involved in any development of the existing products. Hence, as he did not have particular problems, we think that our method can effectively contribute to knowledge management and reuse.

From this case study, we observe that organizing architectural knowledge is a complex issue. It can be done along different dimensions: feature oriented, structure oriented, stakeholder oriented, and so on. Each of these dimensions is relevant for answering certain questions. Flexible tool support, which allowed for the filtering of information and the generation of appropriate views, was found to be of great help.

We also observe that, next to the explicit variability designed into the products, there is kind of a "variation creep." Because of assumptions made regarding the environment in which the products function, assumptions which later turn out to be no longer appropriate, an extra and possibly unwanted type of variation in the PF is introduced. We conjecture that, for PFs, it is helpful to model not only the variation points, but the "stability points" as well. These stability points concern the assumptions we make–things we assume to remain constant over time and space. By modeling invariability next to variability, we may achieve an even better support for software product line evolution.

Acknowledgments

We would like to thank research support from Eurescom (the European Institute for Research and Strategic Studies in Telecommunications), the Cisco Program for Directed Research, the TINA Fellowship Programme, and Telecom Italia Lab. Thanks to their support, we could initiate the industrial collaborations that made our case study possible.

References

1. F. Bachmann and L. Bass. Managing variability in software architectures. In *Proceedings of the Symposium on Software Reusability*, pages 126–132, Toronto, Ontario, Canada, 2001.
2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, second edition, 2003.
3. G. Böckle, J. Muñoz, P. Knauber, C. Krueger, J. Sampaio do Prado Leite, F. van der Linden, L. Northrop, M. Stark, and D. Weiss. Adopting and institutionalizing a product line culture. In S. Verlag, editor, *Proceedings of the Software Product Lines Conference*, volume 2379 of *Lecture Notes in Computer Science*, pages 49–59, San Diego, CA, USA, Aug. 2002.
4. J. Bosch. *Design and Use of Software Architectures – Adopting and evolving a product-line approach*. Addison-Wesley (Pearson Education), 2000.
5. J. Bosch, G. Florijn, D. Greefhorst, J. Kuusela, J. Obbink, and K. Pohl. Variability issues in software product lines. In *Proc. of the fourth Software Product-Family Engineering Workshop (PFE)*, number 2290 in Lecture Notes in Computer Science, pages 13–21, Oct. 2001.
6. P. Clements and C. Krueger. Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, Aug. 2002.

7. P. Clements and L. Northrop. Salion, Inc.: A software product line case study. Technical Report CMU/SEI-2002-TR-038, Software Engineering Institute, CMU, Nov. 2002.
8. D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience, John Wiley & Sons, Ltd.*, 33(10):933–955, Aug. 2003.
9. C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *Proceedings of the Symposium on Software Reusability*, pages 109–117. ACM Press, 2001.
10. A. Garg, M. Critchlow, P. Chen, C. Van der Westhuizen, and A. van der Hoek. An environment for managing evolving product line architectures. In *International Conference on Software Maintenance*, pages 358–367, Sept. 2003.
11. D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, Nov. 1995.
12. M. Jaring and J. Bosch. Representing variability in software product lines: A case study. In S. Verlag, editor, *Proceedings of the Software Product Lines Conference*, volume 2379 of *Lecture Notes in Computer Science*, pages 15–36, San Diego, CA, USA, Aug. 2002.
13. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Nov. 1990.
14. P. Lago. The UNIK project: from product family to product line. Technical Report PDT-DAI-PI-P-001-B0, Politecnico di Torino, Italy, Aug. 2002.
15. P. Lago, E. Niemelä, and H. van Vliet. Integrating features and structural aspects in engineering software product families. Submitted, 2003.
16. P. Lago, E. Niemelä, and H. van Vliet. Tool support for traceable product evolution. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 261–269, Tampere, Finland, Mar. 2004. IEEE Computer Society Press.
17. K. Laudon and J. Laudon. *Management Information Systems – Managing the Digital Firm*. Prentice Hall, eight edition, 2004.
18. A. Modarressi and S. Mohan. Control and management in next-generation networks: challenges and opportunities. *IEEE Communications Magazine*, 38(10):94–102, Oct. 2000.
19. C. Riva and C. del Rosso. Experiences with software product family evolution. In *Proceedings of the International Workshop on Principles of Software Evolution*, pages 161–169, Sept. 2003.
20. S. Robak. Feature modeling notations for system families. In *International Workshop on Software Variability Management*, pages 58–62, 2003.
21. K. Schmid and I. John. A practical approach to full-life cycle variability management. In *International Workshop on Software Variability Management*, pages 41–47, 2003.
22. R. Taylor, N. Medvidovic, K. Anderson, E. Whitehead, J. Robbins, K. Nies, P. Oreizy, and D. Dubrow. A component- and message-based architectural style for GUI software. *IEEE Trans. Software Eng.*, 22(6):390–406, June 1996.
23. D. Webber and H. Gomaa. Modeling variability with the variation point model. In C. Gacek, editor, *Proceedings of the International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 109–122. Springer Verlag, Apr. 2002.
24. T. Weiler. Modeling Architectural Variability for Software Product Lines. In J. van Gurp and J. Bosch, editors, *Proceedings of the Workshop on Software Variability Management*, pages 55–63, Feb. 2003.