Architecture Recovery for Product Families

Martin Pinzger[1], Harald Gall[1], Jean-Francois Girard[2], Jens Knodel[2], Claudio Riva[3], Wim Pasman[4], Chris Broerse[4], and Jan Gerben Wijnstra[5]

[1] Distributed Systems Group, Vienna University of Technology, Vienna, Austria
{pinzger,gall}@infosys.tuwien.ac.at
[2] Institute for Experimental Software Engineering, Fraunhofer, Kaiserslautern, Germany
{girard,knodel}@iese.fhg.de
[3] Software Architecture Group, Nokia Research Center, Helsinki, Finland
claudio.riva@nokia.at
[4] Philips Medical Systems, Eindhoven, The Netherlands
{wim.pasman,chris.broerse}@philips.com
[5] Philips Research Laboratories, Eindhoven, The Netherlands
jangerben.wijnstra@philips.com

Abstract. Software product families are rarely created right away but they emerge when a domain becomes mature enough to sustain their long-term investments. The typical pattern is to start with a small set of products to quickly enter a new market. As soon as the business proves to be successful new investments are directed to consolidating the software assets. The various products are migrated towards a flexible platform where the assets are shared and new products can be derived from. In order to create and maintain the platform, the organization needs to carry out several activities such as recovering the architectures of single products and product families, designing the reference architecture, isolating the variable parts, and generalizing software components. In this paper, we introduce a product family construction process that exploits related systems and product families, and we describe methods and tools used. We also present an approach for classifying platforms according to platform coverage and variation and describe three techniques to handle variability across single products and whole product families.

1 Introduction

Software product families aim at sharing more than the design effort (i.e. code, designs, requirements, and test cases) to reduce development costs and increase the number of product introduction. Typically, families are built on top of existing, related software systems whereas the common artifacts among these systems are integrated to a common asset base. In terms of software architecture these common assets are architectural artifacts used to design the reference architecture of resulting product families. We refer to reference architecture as core architecture for subsequent product families (and products) independently of implementation. A product family architecture follows a reference architecture and is the basis for single products with respect to achieving maximum sharing of parts in the implementation [8].

Constructing a reference architecture out of related products and existing product families basically is a non-trivial task, because knowledge and experiences from previous architectural designs cannot be transferred explicitly. Additionally, architecture descriptions are often not available or insufficient and diverge from implemented concrete architectures. And there is a lack of commonality analysis methodologies to determine the common architectural artifacts that are mandatory for the construction of the reference architecture. As a consequence the amount of manual work is high and the costs of introducing a product family approach increase.

In the context of the European project CAFÉ (from Concepts to Application in system-Family Engineering) [5] we concentrated on these research issues along constructing reference architectures and product family architectures. In this paper we briefly describe our easy-to-use, yet flexible process for reference architecture construction and point out new techniques that aid architects in this context. Our construction process combines techniques and tools that are used to analyze related systems, determine common assets, and integrate these assets along with new ones into the design of a reference architecture.

Variability is a key issue in designing the reference architecture and building the platform. We address these issues and describe an approach for classifying platforms according to two properties: (1) coverage of the platform as indicator of how much additional work is needed to make a specific family member based on the assets in the platform; (2) type of variation mechanism as indicator of how easy it is to produce/develop a specific family member and what might be the impact of unforeseen requirements that must be supported. The outcome of our classification is a guideline that can be used to select the proper variation mechanism for a product family platform. Further, we report on experiences we gained through an investigation of a number of existing product families and in particular describe three approaches to handle changes to platforms.

The remainder of this paper is organized as follows: Section 2 introduces our reference architecture construction process and describes each phase in detail. In Section 3 we describe our approach for classifying platforms according platform coverage and variation. Section 4 shows three approaches to handle changes to platforms and how to adapt the reference architecture. Results are presented in Section 5 and Section 6 draws the conclusions and indicates future work.

2 Reference Architecture Construction by Exploiting Related Prior Systems

Product families are rarely developed on a green meadow; on the contrary, they are most often based on pre-existing systems. An integral part of every product family is the reference architecture that specifies the fundamental architectural concepts in a domain to which resulting product families and products have to conform to [4, 8].

We have developed a reference architecture construction approach that exploits related existing systems and takes into account knowledge gained out of the individual systems and experiences made with already field-tested solutions and reference architecture fragments obtained in prior systems. Figure 1 depicts our construction process. Four major steps are performed:

- Determine architectural views and concepts
- Architecture recovery
- Analysis of recovered architecture
- Design of a reference architecture

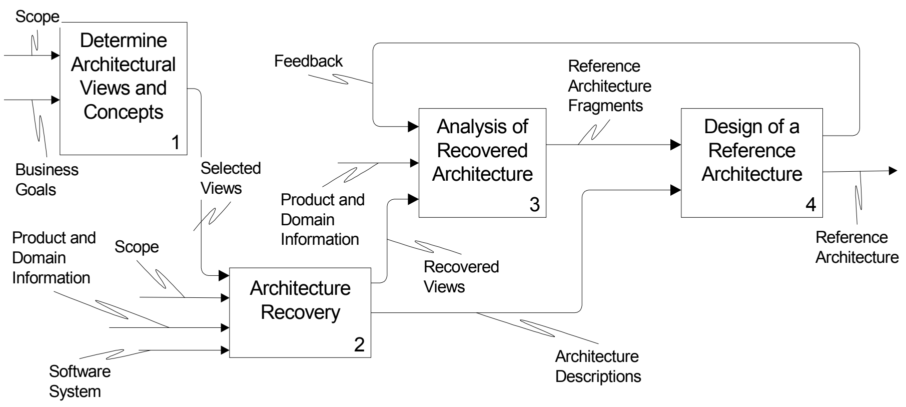
Scope
Determine Architectural Views and Concepts
1
Business Goals
Selected Views
Feedback
Reference Architecture Fragments
Analysis of Recovered Architecture
3
Design of a Reference Architecture
4
Product and Domain Information
Reference Architecture
Product and Domain Information
Scope
Architecture Recovery
2
Recovered Views
Software System
Architecture Descriptions

Fig. 1. Reference architecture construction process.

In the following sections we describe each process step in more detail and point out techniques and tools we used (the subsection numbers correspond to the steps in Figure 1).

2.1 Determine Architectural Views and Concepts

For the recovery of the architecture of each individual system, engineers have to agree upon the architectural aspects to be extracted and upon which architectural description language to use for representing them. Basically, the description language should be the same among all individual systems to ease and facilitate the application of commonality analysis (tools).

In the context of our architecture recovery approach we refer to architectural aspects as *architectural concepts* and *architectural views*. Regarding architectural concepts we particularly concentrate on the architectural style by which a software system is built. The style defines the types of building blocks that can be used to compose the system (e.g. components, classes, applications) and the communication infrastructure that enables the components to interact at runtime (e.g. software busses, remote procedure calls, function calls) [12]. Such concepts represent the way developers think of a system, and they are first class entities the terminology of the reconstruction process.

Architectural views such as described by Hofmeister et al. [5] and Kruchten [10] are abstractions of underlying entities (e.g. source model entities) and each view focuses on certain properties of the software architecture such as, for example, structure, control, or communication.

In order to select the proper architectural concepts the business goals and the scope of the emerging reference architecture of planned product families are taken into account. In terms of our architecture recovery approach we focus on the extraction of the following set of architectural views:

- *Component view* to show the high-level dependencies among the logical packages (groups of components).
- *Development view* to show the source code organization and include dependencies.

- *Task view* to show the grouping of components in different OS tasks and the inter-task communication due to the exchange of messages.
- *Management view* to show the organization of the component factories and their dependencies (caused by component usage among different factories).
- *Organizational view* to show the geographical distribution of the components in the development sites.
- *Feature view* to show the implementation of a set of features at the component level.

The decision, which views should be selected for reconstruction, is important to the next steps of our approach, since the selected views form the foundation of the following activities.

2.2 Architecture Recovery

The architecture of each related prior individual software system has to be recovered next. The architecture will be documented in terms of the selected architectural views and elicited concepts. The architecture recovery step is strongly influenced by product and domain information (e.g. domain expert input, requirements specification, user manuals, (mostly outdated) design documentation, etc).

Figure 2 depicts the process we use for the extraction of architectural concepts and views. It consist of four major phases whereas we start with the definition of architectural concepts based on existing design documentation and expert knowledge from designers and developers of the system.

The second step is concerned with creating a raw model of the system containing the basic facts about the system at a low level of abstraction. The facts can be extracted with a variety of methods: lexical or parser-based tools for analyzing the

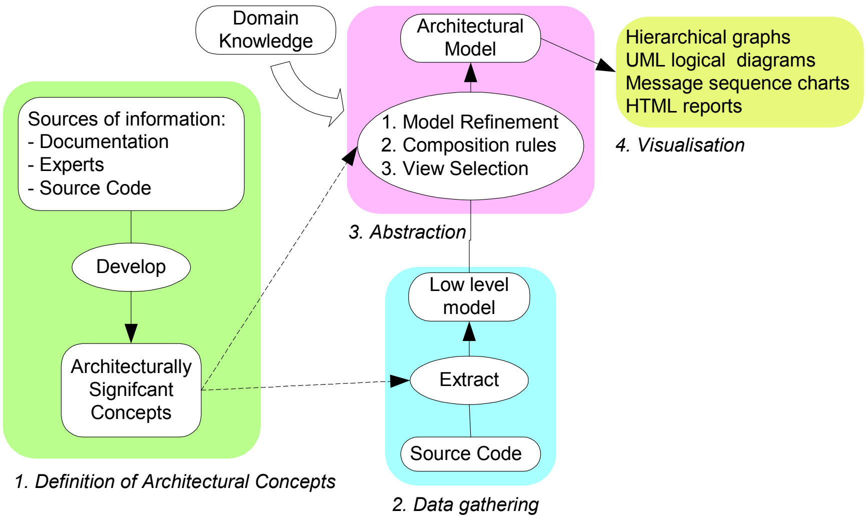

Fig. 2. Architecture Reconstruction Process.

source code, profiling tools for the dynamic information, manual analysis of design documents, and interviews with the developers. The raw model is typically a large and unstructured data set.

In the third step we enrich the model by classifying and structuring the model elements in hierarchies and by removing architecturally irrelevant information. This task requires the human input of the domain experts and combines the raw facts with the domain-specific knowledge of the system. The result is the selected set of architectural views that describe specific concerns of the architecture.

The fourth step is about visualization of the results. Basically, we rely on three visualization formats: *hierarchical relational graphs* for navigating and manipulating the graphs; *hyperlinked web documents* for publishing the architectural models on the web; and *UML* diagrams that can be imported in traditional CASE tools (e.g. Rational Rose or Together).

Apparently, the process step of abstracting meaningful higher-level views from the low-level model or source code is the most complex task. A lack of appropriate (semi)automatic tools increases manual work. To reduce this manual effort we have developed techniques along with tools that were integrated into our reconstruction process. In following paragraphs we briefly describe three techniques addressing the abstraction issue and point out reference architecture construction relevance. For a detailed description we refer the reader to related publications [2, 16, 17].

Semi-automatic Component Recovery

Software components represent basic building blocks (i.e. assets) of software systems that can be reused in single products and products of product families. For the construction of a common asset base, the product family architecture, and the more abstract reference architecture knowledge about components realized by each product family candidate system is mandatory. A first step towards this is the extraction of potential software components and the relationships between them.

We developed a semi-automatically approach that extracts components from source code [2]. Our approach is composed of three stages: initialization, reconstruction, and analysis. First, the initialization stage involves identifying the desired results and collecting information about the context of the architecture. The context poses demands and constraints on the architecture description to be recovered. Then, the reconstruction stage involves semi-automatically extracting one or more architectural views and reviewing them with the architecture expert. The goal of the extraction step is to obtain an as accurate description of the elements of the system and their interrelations as needed, and to build abstracted views on top of them. The last stage involves analyzing the results. The produced views are reviewed and refined, if necessary, with a group of experts.

Pattern-Supported Architecture Recovery

Architectural styles and patterns represent widely observed and occurring architectural concepts that aid engineers in designing software architectures. Although styles do not imply their implementation there are certain implementation patterns that are typically used by engineers to realize an architectural style.

In our pattern-supported architecture recovery approach we concentrate on the extraction of such implementation patterns from source code and related files (e.g. meta data information, configuration files) and the lower-level model. Often important architecture related information about the realization of a particular style is specified in configuration files such as, for example, the configuration of which and how components communicate with each other.

Concerning textual data files we apply extended string pattern matching as described in [16, 17] and demonstrated in [9, 15]. Our lexical analysis tool facilitates the specification of simple and complex pattern definitions and allows for the generation of user-defined output formats. Extracted information about matched pattern instances result in "pattern views" that indicate the use of a particular architectural style. Further, the extracted information complements lower-level source models extracted by parsers [9].

For the investigation of lower-level models we use relational algebra tools such as Grok [7]. Based on domain and expert knowledge engineers specify new or load existing pattern definitions from a pattern repository. Similar to the lexical analysis technique described before matched pattern instances are indicators for certain architectural styles.

Combining the results of both analysis techniques, engineers gain information about the architecture and implementation of each product family candidate system that is mandatory for the integration and the construction of a reference architecture.

2.3 Analysis of Recovered Architecture

This task aims at producing useful information for the design of the reference architecture by detailed analysis and comparison of the prior software systems. A comparison of multiple systems is possible and reasonable, because all prior systems are related to each other:

- They implement similar features to a certain degree.
- They operate in the same set of domains.
- They provide similar functionality.
- They were developed within the same organization (most of the time).

In the following two sections we describe the basics and the process of our analysis approach that compares and assesses the software architectures of product family candidate systems.

Principles and Goals of Architecture Comparison

The comparison is based on the architectural descriptions of the individual systems (or partial descriptions containing most relevant information). The goal is to learn about different solutions applied in the same domain, to identify advantages and drawbacks of the solutions, and to rate the solutions with respect to requirements of the product family. By achieving these goals, architecture comparisons contribute to the fulfillment of the given quality goals in the design process of reference architectures.

Product and domain information is used to answer questions when comparing the architecture of different systems and extracting plausible rationales. Existing architecture descriptions are rarely up-to-date or rarely contain all the needed views. However, they offer an anchor point from which the reverse architect produces a description close to the expert mental model of the system. Explicitly recording the concepts,

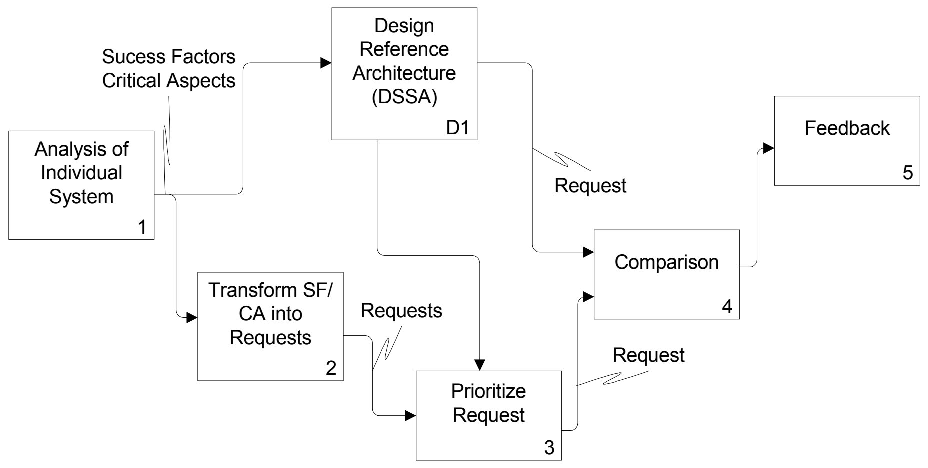
Sucess Factors
Critical Aspects
Design Reference Architecture (DSSA)
D1
Analysis of Individual System
1
Feedback
5
Request
Comparison
4
Transform SF/ CA into Requests
2
Requests
Request
Prioritize Request
3

Fig. 3. The Architecture Comparison Process.

features and functionality expected in the system can guide the reverse architect and help the production of a description expressed in the terms used by experts.

The experts then annotate the architecture descriptions with rationales. When comparing the prior systems' architecture, more detailed information about concepts, solutions or additional views may be needed. This has to be fulfilled by another iteration of architecture recovery. The architecture descriptions are then completed by the newly gained information (e.g., trade-offs involved in the choices of multiple system, integration strategies).

The infrastructure and the architecture style present in the individual systems constitute key inputs to the design of the reference architecture. In addition they offer a good point to anchor initial comparisons. Analyzing the recovered architectures of different systems has the goal to find out how and why they address the requirements in the way they do. It also documents the implementation strategies employed by each system to address a given concern and evaluates their success in their respective context and compares the strategies with respect to criteria relevant to the product family. It captures the conjectured rationales and trade-offs involved in selecting the strategies used in a system (similar to Parnas' idea of a "rational development process and why we should fake it" [12]).

Architecture Comparison Process

According to the principles and goals described in the previous section we developed an iterative architecture comparison process as depicted in Figure 3. The process consists of five major steps:

1. **Analysis of Individual Systems:**
   The process starts with an analysis of all given individual systems (the later phase of architecture recovery). The architecture of each system is recovered with the help of the techniques mentioned above. The goal of this step is to learn about the essential characteristics of each software system. There are two main categories: Success factors (SF) report about means, patterns, and strategies that worked well,

i.e. with the help of these factors advantages were gained. Success factors of different solutions provide essential input to the design process of the reference architecture. The software architect can learn about what worked out well, and why it was a success. Critical aspects (CA) are circumstances (e.g., a pattern, or infrastructure) that leaded to negative, sensitive consequences. For the design process of the reference architecture, it is necessary to handle those aspects with caution (or to avoid them at all if possible).

2. **Transform Success Factors and Critical Aspects into Requests:**
   In the second step, the success factors and the critical aspects are transformed into scenarios, if possible. These scenarios extend the scenario list built when designing a reference architecture. A scenario captures functional and quality requirements of the product family the architecture is designed for.
3. **Prioritize Requests:**
   This optional step orders the scenarios with regard to their PuLSE.DSSA priorities (see Section 2.4), so that potentially important ones are processed first. As the order in which scenarios are addressed is very important, those scenarios that are considered to have the highest significance for the architecture should be selected first.
4. **Comparison of Different Systems:**
   For a comparison, three activities have to be performed one after the other:
   a. *Select Systems:*
      To start, the software systems to be compared against each other have to be selected. The elicitation of these comparison candidates is an activity, which has to be performed carefully for two reasons. On the one hand, the effort for comparing a sound set of comparison candidates is significantly lower than comparing all systems. On the other hand, choosing too few systems leads to more or less useful results in the context of a product family
   b. *Detailed Analysis:*
      In the second activity, the selected systems underlie a detailed analysis focusing on issues concerning the requests. For each system, it will be analyzed how it addresses the requirements reflected in the request. Several questions will be answered after this activity: why was it done this way, what were the trade-offs, and how does the solution fulfill the requirements.
   c. *Conditioning:*
      Each system will contribute to the concluding results, but the more systems are compared, the more data about these systems is produced. To reduce the amount of information, the results are conditioned so that the most important data and the essential gains in experience are returned.
5. **Feedback:**
   In the final step, feedback is integrated into the process. A new iteration may be started because of the feedback. Or the information gained in several feedbacks may lead to new insights about the product family context, so that more information about some solutions is required. Due to the learning effects during the detailed analysis, new scenarios refine the design process of the reference architecture.

The resulting reference architecture should be of a higher quality when using recovered information gained from prior systems than when designing a new one from scratch. Exploiting the pre-existing, individual software systems is worthwhile, since it helps to understand success factors and critical aspects of applied solutions in individual systems. Furthermore, it can avoid bottlenecks by already knowing about consequences, and it promotes the learning about applied solutions. The detailed analysis of recovered architectural descriptions and the comparison of different but related systems are crucial in order to benefit from the existing systems by reusing implemented knowledge, as well as field-tested architectural means (e.g., patterns, strategies and infrastructure, etc.). Hence, we are able to draw conclusions about the applied solutions of the individual systems within their specific context and we learn about the consequences of specific solutions, and therefore this step contributes substantially to the design process of the reference architecture.

2.4 Design of Reference Architecture

PuLSE-DSSA [1] is an iterative and scenario-based method for the design of reference architectures. The basic ideas of PuLSE-DSSA are to develop a reference architecture incrementally by applying scenarios in decreasing order of architectural significance and to integrate evaluation into architecture creation. The quality of the architecture is monitored and assured in each development iteration. Figure 4 shows the main inputs to the PuLSE-DSSA method. In order to design a reference architecture the architect has to consider information obtained through architecture recovery as well as a prioritized list of business goals, given functional and quality requirements, and architectural means and patterns. The inputs are used to produce a reference architecture that satisfies the given goals and requirements of the product family and that is documented using a number of previously selected or defined architectural views. The PuLSE-DSSA method guides software architects in systematically making design decisions based on identified and documented business and quality goals and in assessing whether the architecture under design really satisfies the requirements. The design process might request to deepen the analysis and comparison of the architecture of prior systems, to refine views, or to produce other specialized views.

The following architecture recovery information is gained during the analysis of recovered architectures and fed into the design process of the reference architecture:

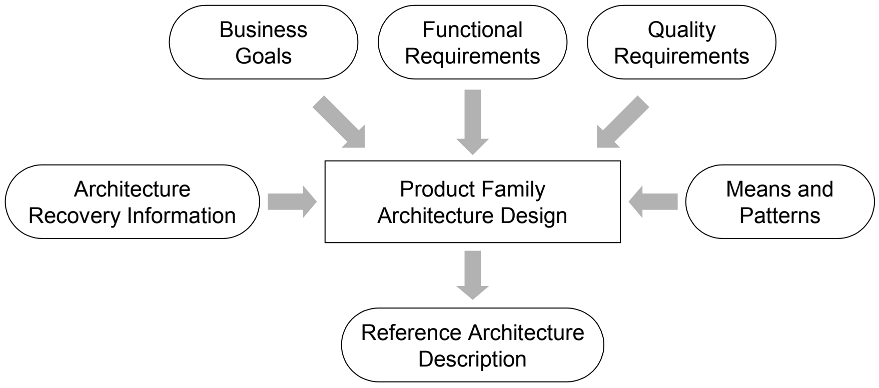
Business Goals
Functional Requirements
Quality Requirements
Architecture Recovery Information
Product Family Architecture Design
Means and Patterns
Reference Architecture Description

Fig. 4. Inputs and Outputs of the Reference Architecture Design Process.

- **Rationales:**
  Solutions in individual systems have been applied in certain contexts to fulfill requirements. Learning about the rationales behind these solutions can help when facing similar or the same requirements for the whole product family.
- **Means and Patterns:**
  Architectural means and patterns used in existing systems and identified in architecture recovery can extend the collection from which the architect can choose in the design process of the reference architecture. The catalogue of means and patterns is expected to come from three different information sources, collected from literature, gathered from other architecture design projects, or recovered from existing systems.
- **Consequences and Pattern Instantiations:**
  Similar to the previous step, information about patterns and their instantiation obtained during architecture analysis of recovered systems can be used as an input when designing reference architectures. Consequences resulting from the instantiation of a specific pattern should be considered when designing the reference architecture. A consequence of a pattern instantiation for instance may be an impact on a quality goal.
- **Evaluation of the Architecture:**
  The evaluation of the reference architecture with respect to functional and quality requirements and the achievement of business goals can benefit from existing individual systems, since the decision whether or not a requirement or goal is fulfilled can not always be answered on the architectural level but by experimentation with a prior system that fulfills the goal with the same underlying concept.
- **Documentation:**
  To document the reference architecture of a product family, recovered architecture descriptions of individual systems can be reused, when the reference architecture and the individual systems have an architectural fragment in common (e.g., when one of the prior systems forms the basis for the product family, or when distinguishable fragments of a single system contribute to the product family).

The knowledge gained in the analysis of the recovered architectures is integrated into the design process of the reference architecture. The construction of the reference architecture provides feedback on what is important for the design process, and which views should be presented in more details.

3 Reference Architecture and the Platform Type

Based on the reference architecture the platform with reusable assets for the product family will be built. The definition of the reference architecture and the platform is an iterative process. The reference architecture can be considered as a high-level description, whereas the platform contains the actual building blocks from which members of the product family can be constructed.

Every product family will have its own unique context, for example a specific business strategy and a specific application domain. As a consequence, each product family approach has its own specific characteristics that fit in such a context. This means again that the type of platform that is used for a product family must also be tuned to this context. This leads to a vast range of platform types. In order to get a grip on this range, we have analyzed a number of product families and their platforms. This analysis led to two dimensions for classifying platforms, namely:

- coverage of the platform
- type of variation mechanism

These dimensions will aid the classification of product family approaches and will both facilitate the selection of a new platform approach for a particular context and support the evaluation of existing approaches. When considering the four steps for the construction of a reference architecture as discussed above, these dimensions can be used at several places. For example in the fourth step where the reference architecture is designed, the two dimensions provide guidelines on which variation mechanisms to use and which platform coverage to apply. Both dimensions also support the evaluation of a reference architecture. Both dimensions are explained below; more on this topic can be found in [24].

3.1 Platform Coverage

Roughly speaking, the coverage indicates the relation between the size of the platform and the additional artifacts that are required to make a complete product within the family. The decision about the coverage of the platform is influenced by several factors including non-technical ones related to business, process and organization (see also [23]).

To explain the platform coverage in some more detail, the concept of subdomains within an architecture is important. By subdomain we mean a sub area within a system that requires some specific knowledge, for example the image processing subdomain or the image acquisition subdomain in a medical imaging system. These subdomains may deal with application knowledge relating to the end user, or with technical knowledge relating to the peripheral hardware or computing infrastructure knowledge. In the context of design and realization, such a subdomain usually results in one (or a few) modules/components with a well-defined interface. In Figure 5, a schematic reference architecture is shown with different coverages for the subsystems. The two leftmost subsystems are completely covered, indicated by the gray color. The

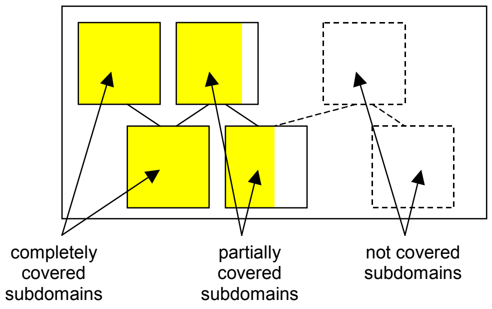

Fig. 5. Reference Architecture and Coverage of Subdomains.

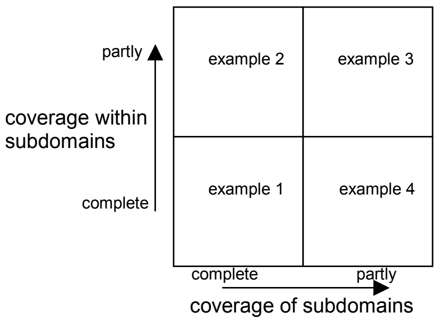

Fig. 6. Classification of Platform Coverage.

two subsystems next to them are partly covered, so that specific functionality has to be added to make a concrete product. The two rightmost subsystems have dotted lines, meaning the no generic functionality is provided for them, but additional subsystems can be added to make concrete product.

Based on the subdomains we can now express the coverage of a platform. In our study we had one product family that was set up in such a way that each delivered family member contained the same software. As a consequence, the platform contained all subsystems of the family, and each of these subsystems was covered completely, resulting in a complete coverage. Another example was a platform that contained all subsystems, but allowed the product development group to add specific functionality to the subsystems. So all subsystems are covered, but the subsystems are internally partially covered. Here, about 75% of the functionality was handled by the platform, leaving 25% to be added by the product groups. A third example was a platform that only dealt with a subset of the subdomains, and these subdomains were only partly covered. A fourth situation is where a part of the subdomains are dealt with, but these subdomains are completely covered. This leads to four areas in which platform coverage can be classified, see Figure 6. It depends on the actual platform, how much of the functionality is already realized inside the platform and how much work still needs to be done by the product groups to make the specific products.

3.2 Variation Mechanisms

In the context of designing reference architectures variability is a key issue. The used variability mechanisms influence the flexibility of the platform, the way specific products can be made with it, the effort needed to make a product, etc. In our investigation we encountered various types of variation mechanisms and classified them in a two-dimensional space. This classification is made from an architectural point-of-view:

- *below architectural level*
  When an architect is defining an architecture, he/she amongst others defines the main components from which the system will be built. These components have interfaces via which they will interact. When the variation of the system stays within the components and does not impact the architecture, we classify the mechanism

as being below architectural level. Examples of mechanisms in this category are configurable components, component frameworks with plug-in components, or even components for which only the interface is defined. The architect must be aware of the variation that can be realized within such components and provides rules and guidelines on how to do so. But, the variation is realized by the developers of the components.

- *at architectural level*
  At the architectural level, components and interfaces are important entities, along with rules and guidelines. The product family architect can identify the components from which the system should be built. If the members of the product family cannot be built from such a fixed set of components, another possibility is to define a basic platform to which the development groups can add their own specific components, preferably via predefined interfaces. A third possibility is to capture important architectural concepts in interfaces. The interfaces should then allow new components to work with each other, even if they were not identified from the beginning. This allows addition of new components to the system.

More information on this classification and the variation mechanisms can be found in [24].

4 Component Generalization in Product Families

Technology for platforms is changing rapidly and requires adaptation of reference architectures to make them resilient to these changes. The solution can be found by making the whole or certain areas of the architecture of the product families more explicit and especially pay attention to the variation mechanisms. Summarized, in CAFÉ we considered the following approaches:

- Trace features to components to support migrating shareable features to the platforms.
- Add a technology abstraction layer in the platform.
- Compare similar products families, to identify commonalities and extend the framework of a product family to serve different families.

4.1 Feature-Oriented Reverse Engineering

The definition of the term “feature” is often domain dependent and in the literature there are several interpretations. We reference to the work of Tuner et al. [25] for an extensive discussion of this definition. In this article, we use the term “feature” to mean a “coherent and identifiable bundle of system functionality” that is visible to the user via the user interface.

We distinguish between the *problem domain* and the *solution domain*. The problem domain focuses on the user’s perspective of the system and describes the requirements (functional and non-functional) that the system has to satisfy. The solution domain is centered on the developer’s perspective and concerns with the implementation of the system. The problem domain specifies *what* the system is supposed to do.

The solution domain specifies *how* the system achieves what is promised. The features represent the contact point between the problem and the solution domain. At this level, marketing people and developers can speak a common language and they can understand each other.

Features (typically used to advertise the product) are implemented by particular architectural elements (such as components, classes, functions). They also represent the highest elements of abstraction we can decompose a system in the solution domain. This leads to the concept of *"feature oriented reverse engineering"* proposed by Turner et al. [25]. From a *feature engineering* perspective, the goal of reverse engineering is to discover how the features have been implemented, what are they interactions, or what are the necessary components for their execution. In the mobile phone example, reverse engineering could be used to identify how the feature *"make call"* has been implemented in a GSM and a 3G phone, or to discover how the feature "receive call" interferes with game playing, or to recover the procedure that has been used to set up a WAP connection with GPRS.

Understanding the implementation of the features is a crucial activity in the context of a product family. Features represent reusable assets that are available in the platform and are shared among different products. Each product development project can select the platform features, configure them and integrate them in the products. Features are combined together to create a particular product and they are presented in the user interface (UI) in a simple and coherent way. This approach puts a strong pressure on the integration phase and in the design of the UI. In particular, the problems of feature interaction (functional or logical dependency between features) are often unavoidable and difficult to control.

To avoid costly delays in the integration phase, we require

- to identify and specify the possible feature interactions as early as possible during the design phase and
- to analyze the implementation of the features (and their interaction) during the development of the product.

To tackle the first point Lorentsen et al. [11] have proposed a method for modeling the feature interactions in mobile phones with explicit behavioral models of the features. The approach is based on the Colored Petri Nets and allows the UI designers to simulate particular features and analyze their behavior with automatically generated Message Sequence Charts (MSC).

To support the second point, our reverse engineering technique is based on the extraction of static and dynamic information from the implementation of the system. The static information is extracted from the source code following the process described in Section 2.2. The dynamic information is extracted by instrumenting the system and tracing the execution of the features. The result is combined in the *feature view* that describes the implementation of a set of features at the architectural level. Our approach emphasizes the correct choice of architecturally significant concepts as described in Section 2.1, the abstraction as the key activity for creating high-level sequence diagrams, and relies on combining static and dynamic information in the same architectural views [21].

Our method is based on the following steps:

1. Create a use case that covers the set of features under analysis.
2. Execute the use case on the instrumented system and trace all the relevant information (such as function calls, messages, inter task communication, memory accesses).
3. Create the feature view by combining the extracted traces with a high level static view (typically, the component view or the task view).
4. Abstract the feature view by detecting interaction patterns to reduce its size.
5. Navigate the feature view by expanding/collapsing the participants and messages of the MSC, by filtering un-relevant information and by slicing the static views with particular dynamic scenarios.

This method allows us to derive insights about the implementation of the features and to navigate them in a high level abstracted form.

4.2 Adding Abstractions

An Operating System (OS) offers functionality towards all important system resources such as memory, timers, synchronization and communication mechanisms, I/O and underlying hardware, etc. Typically applications depend on this functionality and build upon it. In small systems not many applications use the offered functionality of the OS and if they do, designers and architects focus on a dependency that is as small and thin as possible.

In larger systems however more applications depend on the OS functionality. This increases the use of the OS and calls for a well-defined use of the offered functionality. Systems may even grow to a point where more than one computer is used and maybe even more than one OS. Rather than having software engineers to cope with the differences between OS's, an Operating System Abstraction Layer (OSAL) offers standardized functionality to all software engineers running on different OS platforms.

Thus an OSAL helps in standardizing usage of OS functionality, one can focus on solving domain problems rather then having to dig in OS differences. It even helps when changing from one OS to another (although this a job you do not want to do often). If an OSAL is build upon two different OS's, one will certainly run into compromises. Compromises will most certainly be found in the following fields: processes and threads, support for asynchronicity and real time behavior.

The easiest way to start an OSAL is to look at functionality widely used by all applications in the entire system. OS functionality that meets this requirement is usually: file I/O, memory management, synchronization principles, serial I/O and sockets. By focusing on these aspects, one has a quick win: in a short period of time, a large amount of functionality is abstracted which helps a lot of software engineers.

4.3 Generalization of a Framework

A product family is usually based on a platform, a set of common components and interfaces to be shared amongst the various members of the product line. At the beginning each of these components fits perfectly within the architecture of the product line, adhering to its set of architectural rules and designed according to its architectural paradigm(s). A common reuse problem occurs however when either the scope of the product line (and with that its architecture) is significantly changed (usually extended) or a component from it is extracted to be reused in the scope of another (possibly also a product line) architecture. Now probably there will appear architectural mismatches between the component and its new environment(s), which can range from differences in architectural styles, interface mechanisms, technology, etc. The component probably has to be redesigned to deal with two different environments, a process in which also the evolution of the existing architectures and possible compatibility issues play an important role. In this section we describe a case from the medical imaging domain at Philips Medical Systems in which a component developed in the context of single product family architecture is reused in a much wider scope. It addresses both the architectural changes involved in this generalization as well as aspects concerning backward compatibility.

Within the hospitals, more and more different imaging modalities with new application areas are used. Examples of such modalities are X-ray, Computed Tomography, Magnetic Resonance Imaging, and Ultra Sound. Furthermore, to give the best treatment to the patient, it is important that the various modalities are integrated in the hospital. This also poses more requirements on the servicing of this equipment.

The component for remote access as described in this section was originally developed within the context of the product family architecture for a single modality. This component provides important field service functionality to improve customer support (shorten reaction times) and lower overall service costs. It has been realized as a component framework with plug-ins, allowing field service procedures for specific hardware to be supported by specific plug-ins. After introduction of this component it appeared very attractive to enlarge the scope of it to multiple modalities, which needed the same type of functionality. However all of these modalities have of course their own (product family) architectures.

A product family architecture effort was already under way aiming at the definition of components for very generic (and non modality specific) functionality, see [22]. The main requirement for this architecture is that its components can be reused over multiple modalities that each maintains their own private architectures. It was therefore decided to lift the remote access level component to this level.

Moving the remote access component from its dedicated modality restricted product family architecture to a broader scope introduced four main groups of problems, which are discussed below.

- *requirements matching*
  Historically, there exists a plethora of detailed differences between the field service requirements for the different modalities, although the global service requirements are of course very similar over all medical imaging equipment. Fortunately the original set-up of the remote access component with its plug-ins already allows a great differentiation since all the actual hardware related tests are implemented in the plug-ins.
- *compatibility requirements*
  The remote access component was already deployed in its original product family architecture before it was elected for promotion to a much broader scope. Furthermore extensive investments were made in the development of the remote access plug-in components for this system. It is clear that the generalization of the remote access component was not allowed to nullify these investments.

- *architectural mismatches*
  The architectural differences between the various modalities are quite large. Decomposition, interfacing, general mechanisms, but also low level issues like database access are very different from modality to modality. It is clear that the introduction of a generic remote access component is only feasible when it does not enforce a major turnover of the systems that want to incorporate it. Another issue encountered in the analysis of differences is a large number of more or less hidden dependencies of the framework on the original product family it was designed for. The main dependencies are on the operating system, the database, and mechanisms such as installation, registration and licensing.
- *technological differences*
  Several major technological differences exist between the various product lines in which the remote access component is deployed. The most important differences are in technologies such as operating systems, languages and middleware that must be supported by the remote access component.

So, there are many aspects to be taken into account in the migration of the remote access component to a wider scope. This is probably quite usual for this kind of migration. Below, some choices to deal with these aspects are discussed.

- *operating system, middleware and language*
  For the time being the generalized component will be realized on the Windows platform, since this platform was used in most products. For the remote access component, also the COM/.NET framework is used. It must be noted here that this lock-in is circumvented for some of the generic components deployed in medical systems. Any explicit reference to a single middleware model (such as COM) in the core components is avoided. Separate wrapper layers are developed to map the components to a certain required interface mechanism like COM.
- *dependencies to infrastructure mechanisms*
  The remote access framework was dependent on the infrastructure of the original product family it was developed for, e.g. for installation. To break this link a pragmatic solution has been chosen: the relevant parts of the infrastructure have been copied to the context of the remote access component and are maintained separately from the original product line they were developed for initially.
- *component interfacing*
  For the generic components that are reused within several modalities within medical systems, a set of generic interfaces and so-called information models have been defined. For example, interfaces exist for accessing patient or image data, to using logging of configuration data. Also, an interface is defined for the execution of a "job". The interfaces themselves are small and limited with all further information stored in data structures. The interface of a component in this case consists of its methods and a detailed description of the data model (called the information model) that goes with it. These principles are described in more detail in [22].

What are now the consequences of this approach for the remote access component? The original remote access component uses a strongly typed interface style with a extensive set of interfaces and interface methods for all the different stages in a field service procedure. This enforces a fixed order of activities, which is not desirable for all modalities. Furthermore these broad interfaces impose a certain maintenance risk.

Finally they restrict the flexibility for other modalities to build user interfaces and procedures according to their specification. Therefore a transfer to the much more generic interface and information models described above is made.

When considering the four steps as discussed in Section 2, the focus has been on the third and fourth step: the analysis of the different product families in which the generalized component will be used and the design that has to make the reuse of this component across multiple modalities possible. When considering the classifications as mentioned in Section 3, the modality from which the remote access component was taken has a high coverage. It uses amongst others component frameworks and plug-ins to realize variation (just as for the remote access component). The product family to which the remote access component has been transferred has a much broader scope. As a consequence, the platform coverage is much lower, relatively speaking. In this product family, generic interfaces are an important means for realizing variation across the different modalities. But also the component framework mechanism is still used for the remote access component. More on this generalization work can be found in [18].

5 Results

In the CAFÉ project we developed methods, techniques and tools for architecture recovery, product family and reference architecture construction, and component generalization in the context of product families. In particular, we:

- introduced a reference architecture construction process (Section 2);
- extended and devised architecture recovery methods and techniques (Section 2.2);
- described platform classification mechanisms and created guidelines that aid in platform selection (Section 3);
- described methods for handling platform variation and adaptation of reference architecture (Section 4).

To evaluate our methods, techniques and tools, we carried out a number of case studies with mobile phone systems, medical systems and a stock market system. More details about our achievements can be found in our common task deliverable (to be published), as well as in referenced publications or the CAFÉ website [5].

6 Conclusions

In this paper we focused on product family engineering and in particular concentrated on reference architecture construction by exploiting related pre-existing systems and on handling variability across single products and product families. The main contribution of our paper is a reference architecture construction process that integrates architecture recovery, architecture analysis and comparison, and the reference architecture design process. For each process step we pointed out key activities and described new, revised and extended techniques used.

Further, we addressed the handling of variability in platforms and the reference architecture that is a major issue of product family engineering. Particularly, we described two techniques for classifying platforms with respect to platform coverage and variation. The outcome of our classification is a guideline that aids engineers in selecting the proper variation mechanism for the product family platform. In addition to our classification techniques we presented three approaches to handle variability across single products and whole product families. These techniques base on the concept of making the whole or certain areas of the architecture of product families more explicit. Concerning these areas we took into account features, technology abstraction layers, and framework extension by commonalities identified across similar product families.

Future work will be focused on two issues that are concerned with the improvement of presented techniques and tools, as well as the integration of these techniques and tools into a workbench. Consequently, we have to devise a common data format for the data extracted and generated by our tools. We also plan to perform additional case studies to further evaluate and improve our reference architecture construction process and platform classification methods.

Acknowledgments

We are grateful to the national ministries of Austria, Germany, Finland, and The Netherlands for funding our work under EUREKA 2023/ITEA-ip00004 'from Concepts to Application in system-Family Engineering (CAFÉ)'. We further thank our industrial partners Philips, Nokia, and MarketMaker for providing the case studies. Finally, we would like to thank the anonymous reviewers for their feedback.

References

1. Anastasopoulos, M., Bayer, J., Flege, O., Gacek, C.: A Process for Product Line Architecture Creation and Evaluation – PuLSE-DSSA Version 2.0. Technical Report, No. 038.00/E, Fraunhofer IESE. (2000).
2. Bayer, J., Girard, J. F., Schmid, K.: Architecture Recovery of Existing Systems for Product Families. Technical Report, Fraunhofer IESE. (2002).
3. Bayer, J., Ganesan, D., Girard, J. F., Knodel, J., Kolb, R., Schmid, K.: Definition of Reference Architecture Based on Existing Architectures. Technical Report. (2003).
4. Bosch, j.: Design and Use of Software Architectures: Adopting and evolving a product line approach. Addison Wesley, Mass. and London. (2000).
5. CAFÉ (from Concepts to Application in system-Family Engineering): (http://www.esi.es/en/Projects/Cafe/cafe.html).
6. Hofmeiser, C., Nord, R., Soni, D.: Applied Software Architecture. Addison-Wesley, Reading, Mass. and London. (2000).
7. Holt, R. C.: Software Architecture Abstraction and Aggregation as Algebraic Manipulations. In: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON), Toronto, Canada. (1999).
8. Jazayeri, M., Ran, A., van der Linden, F.: Software Architecture for Product Families: Principles and Practice. Addison-Wesley, Mass. and London. (2000).
9. Knodel, J., Pinzger, M.: Improving Fact Extraction of Framework-Based Software Systems. In: Proceedings of the 10th Working Conference on Reverse Engineering (WCRE). (2003) to appear.

10. Kruchten, P. B.: The 4+1 View Model of architecture. IEEE Software, Vol. 12 Issue 6. (1995) 42-50.
11. Lorentsen L., Tuovinen A-P., Xu J.: Modelling Feature Interaction Patterns in Nokia Mobile Phones using Coloured Petri Nets and Design/CPN. In: Proceedings of the 7th Symposium on Programming Languages and Software Tools, Szeged, Hungary. (2001) 15-16.
12. Mehta, N. R., Medvidovic, N., Phadke, S.: Towards a taxonomy of software connectors. In: Proceedings of the 22nd International Conference on Software Engineering (ICSE), Limerick, Ireland. (2000) 178 – 187.
13. Parnas, D.L., Clements, P. C.. A Rational Design Process: How and Why to Fake It. IEEE Transactions on Software Engineering, Vol. 19, Issue 2. (1993) 251-257.
14. Pasman, W.: Platform Coverage and Variation in Product Family Approaches. Technical Report, Philips.
15. Pinzger, M., Gall, H., Jazayeri, M., Riva, C.: Extracting Architectural Views from Large Telecommunications Software: A Case Study. Technical Report TUV-1841-2002-50, Vienna University of Technology. (2002).
16. Pinzger, M., Fischer, M., Gall, H., Jazayeri, M.: Revealer: A Lexical Pattern Matcher for Architecture Recovery, In: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE). (2002) 170-178.
17. Pinzger, M., Gall, H.: Pattern-Supported Architecture Recovery. In: Proceedings of the 10th International Workshop on Program Comprehension (IWPC). (2002) 53-61.
18. Pronk, B. J.: Component generalization in a multi product family context. Technical Report CAFÉ consortium-wide. (2002).
19. Riva, C.: Architecture Reconstruction in Practice. In: Proceedings of the 3rd Working IEEE/IFIP Conference on Software Architecture (WICSA). (2002).
20. Riva, C., Yang, Y.: Generation of Architectural Documentation using XML. In: Proceedings of the 9th Working Conference on Reverse Engineering (WCRE). (2002) 161-169.
21. Riva, C., Rodriguez, J. V.: Combining Static and Dynamic Views for Architecture Reconstruction. In: Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR). (2002) 47-55.
22. Wijnstra, J.G.: Components, Interfaces and Information Models within a Platform Architecture. In: Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering, Erfurt, Springer Verlag LNCS 2186. (2001) 25-35.
23. Wijnstra, J.G.: Critical Factors for a Successful Platform-based Product Family Approach. In: Proceedings of the 2nd Software Product Line Conference, San Diego, Springer Verlag LNCS 2379. (2002) 68-89.
24. Wijnstra, J.G.: Classifying Product Family Approaches using Platform Coverage and Variation. submitted for publication to Software: Practice and Experience.
25. Turner C. R., Fuggetta A., Lavazza L., Wolf A. L.: A conceptual basis for feature engineering. The Journal of Systems and Software, Vol. 49, Elsevier. (1999).