Capturing Architectural Knowledge of Software Product Lines

Jose A. Mercado H.

This thesis is presented as part of Degree of European Master in Software Engineering

Blekinge Institute of Technology
December 2011

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden
This thesis is submitted to the School of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 24 weeks of full time studies.

Contact Information:
Author(s):
Jose A. Mercado H. 770203-P357
E-mail: jame09@student.bth.se

University advisor(s):
Dr. Kai Petersen
School of Computing

School of Computing
Blekinge Institute of Technology
SE-371 79 Karlskrona
Sweden
Internet : www.bth.se/com
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57
Abstract

**Context.** In the context of software product lines, architectural decisions determine the design of an architecture that provides the software with the ability to be configured for different product variants and extended to accommodate future requirements. Although, variability models describe the different configurations of current and future products that the product line supports, the knowledge of how the architecture was designed to support variations of a product in space and time exists only in the architects’ mind or remains implicit in architectural models. This thesis argues that the knowledge found in architectural models and design rationale can be used to facilitate the derivation of product variants and the evolution of the product line.

**Objectives.** To support this notion, we propose the AKinSPL method for capturing the architectural knowledge in software product lines.

**Methods.** The method is founded on the factors that architects take into consideration when designing the architecture, and a meta-model that represents the mental models and processes architects follow during the creation of a product line architecture. To validate the concepts of AKinSPL, its guidelines were mapped to activities of the PuLSE-DSSA methodology and new artifacts were created to capture architectural knowledge on the basis of those guidelines. Next, it was applied to capture the architectural knowledge of an embedded software system for automatic control of agricultural equipment within the context of an ongoing case study.

**Results.** The results showed that diagrams augmented with design rationale enable a faster understanding of the purpose of the architectural models. Similarly, the prescriptions of the architecture with respect to the implementation are conveyed more easily.

**Conclusions.** The application of the method permitted modeling design decisions so as to decompose business-level decisions into fine-grained technical decisions and to identify dependencies among them. Similarly, the method permitted to model the relation between the variability described in a feature model and the architecture model.

**Keywords:** Software Architecture, Design, Rationale, Software Product Lines
# Contents

Abstract i

1 Introduction 1
   1.1 Aims and objectives 2
   1.2 Research questions 2
   1.3 Expected outcomes 3
   1.4 Research Methodology 3
   1.5 Risks 4
   1.6 Organization of this thesis 4

2 A systematic literature review of SPL case studies 5
   2.1 Introduction 5
   2.2 Research questions 6
   2.3 Review Protocol 6
      2.3.1 Identification of publications 6
      2.3.2 Selection of publications 7
      2.3.3 Quality assessment of case studies 7
      2.3.4 Data extraction 7
   2.4 Architectural Drivers of Software Product Lines 9
      2.4.1 Business Goals 9
      2.4.2 Quality attributes 11
   2.5 Architecture Rationale 14
   2.6 The influence of business goals on design decisions 14
   2.7 Conclusions 17

3 The AKinSPL method 19
   3.1 Introduction 19
   3.2 Elements of the architectural design of a family of products 19
      3.2.1 Architectural drivers 19
      3.2.2 Variability model 20
      3.2.3 Realization and binding time 21
      3.2.4 Dependencies 21
      3.2.5 Product specific Design Decisions 22
   3.3 A Model of Architectural Knowledge 22
3.3.1 Model selection ........................................ 23
3.3.2 Model adaptation ..................................... 23
3.4 Method ..................................................... 26
  3.4.1 An iterative method .................................. 26
  3.4.2 A general method .................................... 29
  3.4.3 AKinSPL description ................................ 30

4 Application ............................................... 34
  4.1 Introduction ............................................ 34
  4.2 PuLSE-DSSA ............................................. 34
  4.3 Mapping the AKinSPL to PuLSE-DSSA ................. 35
    4.3.1 Specification of business goals ......................... 37
    4.3.2 Document Decisions .................................. 37
    4.3.3 Design Decisions and Architectural Model ............ 39
  4.4 Case Study ............................................ 47
    4.4.1 Introduction ........................................ 47
    4.4.2 Approach ............................................ 48
    4.4.3 Design decisions ..................................... 48
    4.4.4 Architectural Variability Model ...................... 49
    4.4.5 Results ............................................... 51

5 Evaluation ............................................... 53
  5.1 Introduction ............................................ 53
  5.2 Objects of measurement ................................ 53
  5.3 Measurement Goals .................................... 53
  5.4 Measurement questions ................................ 54
    5.4.1 Measurement Metrics ................................. 55
  5.5 Measurement process .................................. 55
    5.5.1 Measurement scope, stakeholders and roles ........ 55
    5.5.2 Organization characteristics and Information needs 56
    5.5.3 Data collection, storage and reporting ............. 58

6 Summary and future work ................................. 60
  6.1 Summary and contributions ............................. 60
  6.2 Future work ............................................ 61

A Data extracted from case studies ....................... 62

References .................................................. 64
Chapter 1

Introduction

Product Line Engineering is an approach to software development that overcomes the continuous increase in size and complexity of software projects by means of managed processes for the systematic identification and application of reusable assets in the production of software. A product line is designed to benefit from the production of a family of products that share common aspects but also include planned varying characteristics. The architecture of a family of products is designed to provide a common structure to all products and is capable of accommodating individual characteristics of variants and future requirements of new products, thus the decisions made during the architectural design phase must be enforced throughout the lifecycle of the product line.

Variability models represent the points in the architecture where variants will occur, however the information concerning the reasons for selecting of an architectural style or design pattern to facilitate variability are not captured in the same model and might be lost due to knowledge vaporization [10]. Additionally, the introduction of new design decisions to evolve the software architecture need to conform with existing design decisions in order to satisfy new or changing requirements and remain consistent with the design of the current product line architecture. Thus it is important to know which design decisions shape the architecture and the architectural drivers that motivated those design decisions [38].

Architectural knowledge (AK) encompasses the design and description (AD) of the different elements that constitute the system’s structure including the properties, roles and relationships of those elements with respect to other elements as well as the design rationale (DR) behind the selection, disposition and utilization of the elements in order to meet the system’s requirements [21]. Architectural knowledge is also important in the context of software product lines because the design decisions that were made to support variability must be well understood either to derive a consistent product-specific variant or to change the current architecture in order to evolve the product line to incorporate new variations for which the architecture was not explicitly designed. These design decisions are neither specified in variability models nor captured in the architectural description of the product line [55].
Chapter 1. Introduction

There are tools that support different approaches for capture and management of architectural knowledge for single-system architecture, each developed with different perspectives [53]. The aim of these tools is to capture the design decisions in order to support the traceability between requirements and design decisions, [14], model the relationship between design decisions, concerns and the resulting architectural elements [38], [54], provide a repository of architectural knowledge as a means to characterize and manage architectural constructs, their attributes and relationships [4], [37].

At the time of writing this thesis proposal none of the aforementioned tools have been applied in the context of product lines to capture and manage the architectural knowledge with respect to neither variability specification nor architecture evolution.

It is the aim of this study to understand the reasons underpinning the design decisions that shape a product line architecture and the implications of these design decisions throughout the lifecycle of the product line. Once this is understood, it would be possible to identify a method to capture the rationale behind the design decisions that would help to reduce the effort to realize variants and evolve the architecture.

1.1 Aims and objectives

The aim of this project is to define a method to capture architectural knowledge so that it can effectively be used in the realization and evolution of software product line architectures.

- To identify the architectural drivers which strongly influence the design decisions that shape the architecture of a software product line.
- To trace the implications of design decisions in the life cycle of the product line.
- To understand how design rationale can be used to reduce the effort to derive variants and evolve the architecture.

1.2 Research questions

1. What architectural drivers have stronger influence on the design decisions that shape a product line architecture?

2. How to capture and manage the Architectural Knowledge of product line architectures?
   - Are there methods for capturing design rationale of software architectures?
• What does a method need in order to be fitted for application in product line architectures?

3. Do design and development activities of product line engineering benefit from using architectural knowledge?

• Is Design Rationale useful when new design decisions are introduced to evolve the SPL

• Is Design Rationale useful to improve the process of product derivation? architecture?

1.3 Expected outcomes

I A categorization of the architectural drivers based on a qualitative comparison. The categorization should reflect the influence of different drivers in comparison with others and show the conflicts among them.

II Guidelines on how to capture and manage architectural knowledge that can be used to improve the process of product derivation by reducing the time to derive variants that require product specific extensions, as well as, to improve the process of product line evolution by facilitating change impact analysis.

1.4 Research Methodology

Different architectural drivers may be of greater influence to the architectural design than others depending on the application domain [55]. For this reason, the identification of the most influential architectural drivers in the design of product line architectures should be performed on the basis of the state-of-the practice information. Thus, scientific literature describing case-study reports from different application domains will be used to identify and categorize the factors that constitute architectural drivers. Next, the members of these categories will be ranked based on the number of case-studies where the same factor was found. An initial interpretation will be performed based on a chart with the results and final conclusions will be drawn based on a qualitative evaluation of the results with the help of experienced software architects at Fraunhofer IESE.

Guidelines will be proposed within the frame of a method founded on a model of architectural knowledge. A model will be selected on the basis of the model’s ability to capture design rationale specific to product line architectures. The proposed method will be evaluated in the following way: An architecture modeling tool will be enhanced to capture architectural knowledge based on the method’s guidelines. It will be used in an established product line within an industrial case
study at an automotive company in Kaiserslautern, Germany. The tool will be integrated with other tools currently used at the organization for variability management of the product family. The ability of the tool to capture architectural knowledge will be assessed by surveying the developers’ opinion on the usefulness of the information for reducing the effort of deriving variants in application engineering and making changes for evolving the architecture in domain engineering.

1.5 Risks

It is very likely that the number of publications describing case-studies will be relatively small. Also, the literature might not describe case studies in several application domains. Both risks represent a thread to conclusion validity of this study, thus the following measures will be undertaken to reduce the risks: a) To use scientific databases to perform an information lookup of literature describing case-studies in industry. The study in [42] reports of at least three common application domains where software product lines have been applied. Thus, b) case studies from at least those three different domains will be selected.

The utilization of the enhanced architecture modeling tool might introduce additional complexity to the existing process at the company. The current process may not include activities to capture design rationale and there may not be a role in the organization in charge of knowledge management. Thus, there is a high risk that the tool will not be actively used and hence the chance of gathering enough data for evaluation will diminish. One way to tackle this threat is to conduct surveys or semi-structured interviews with the personnel who deals with architectural design in order to gather their criteria about the usefulness of the architectural knowledge provided by the tool during the design tasks.

1.6 Organization of this thesis

The contents of this thesis are organized as follows: in chapter 2 we conduct a systematic literature review as described in the research methodology; chapter 3 elaborates the AKinSPL method upon the results of the preceding chapter and a suitable meta-model for describing architectural knowledge; chapter 4 explains how the AKinSPL’s guidelines can be used with the PuLSE-DSSA methodology to capture architectural knowledge with concrete artifacts; chapter 5 define goals that the organization should expect to attain by using the AKinSPL and proposes a measurement process to evaluate the achievement of these goals. Finally, chapter 6 provides a summary of the work, highlights the contributions of this thesis and describes future work.
Chapter 2

A systematic literature review of SPL case studies

2.1 Introduction

The design of a software architecture involves making several design decisions in order to meet the different architectural drives that influence the outcome of the design process. Business goals, functional requirements, quality attributes and constraints are regarded as architectural drivers since they describe not only the functionality to meet business requirements but also non-functional requirements that assure a satisfactory software execution and overcome any existing constraint.

In the context of software product lines the architecture of a family of products needs to take into consideration the same drivers as single system software architectures, but certain drivers may pose greater influence than others depending on the scope of the product line, the application domain and the organization type [45]. Since the goal of a product line is to efficiently exploit variability, it has been accepted in the product line community that configurability is a major driver of the architectural design of a product family [45, 58]. There are, other quality attributes like safety, performance, reliability which, although important and need to be considered as well, may nevertheless conflict with configurability. This is particularly important in domains like automotive and avionics where safety and reliability are of paramount importance thus a tradeoff analysis is necessary to find the right balance between different quality attributes. The agreements made after the tradeoff analysis result in new design decisions that lead to architectural transformations. The transformations may include the utilization of particular design patterns to enhance variability support in the architecture or to reduce the flexibility as means to improve other quality attributes. It is the aim of these systematic literature review to identify the different architectural drivers of software product lines and how these drivers influence the architectural design.
2.2 Research questions

RQ1 How do different application domains prioritize architectural drivers?

RQ2 How are design decisions linked to architectural drivers?

RQ3 How is the architectural knowledge captured and managed?

2.3 Review Protocol

The review protocol basically consists of four steps: identification of publications; selection of publications; quality assessment; and data extraction. Software engineering publications will be identified in scientific journals through the web site of editors or collections maintained by scientific associations. The databases maintained by the sources identified in the previous step will be queried for publications that match our selection criteria and to retrieve the respective documents. The publications will be scanned for content that describes the information required to answer the research questions. Finally, the information will be extracted from the publications for analysis and conclusions regarding the research questions.

2.3.1 Identification of publications

The identification of publications will be performed by means of digital libraries and indexing services maintained by scientific associations and publishers of scientific journals on Software Engineering. These organizations provide indexing, online lookup and document retrieval services of scientific publications. Namely, these are: IEEE Xplore [33], ACM Digital Library [1], SpringerLink [51]. Sources like the Web of Knowledge [48], ScienceDirect [13], EI Compendex-Inspeck [34], and Google Scholar [35] are sources that offer solely indexing services yet providing referrals to scientific databases. Two main sources are selected: The first is ScienceDirect, because it indexes publications from IEEE, Springer and Elsevier. The second source is ACM Digital library as its publications are not indexed in the ScienceDirect. In addition to the sources mentioned before, the Software Engineering Institute [36], which is an institution devoted to research and development of product line engineering, publishes case studies and technical reports describing their experience in the field. These publications shall be included as well. The selection of the electronic sources mentioned before is mainly motivated by the possibility of access their online content through the library services at the TU-Kaiserslautern university.
2.3.2 Selection of publications

The publications are selected from database query results that match the keywords and section criteria. Keywords are single words or compound terms commonly used in the topic of interest. Query strings are created from set of keywords. The keywords can be seen in table 2.1. The query strings use logical operators that combine the keywords in order to create a set of expressions to query the database. Depending on the source, one or more query strings have to be used to combine all the keywords in order to perform a full-text database lookup.

![Keywords Table]

Additional selection criteria are: a) The publications should contain any combination of all the keywords. b) The publication is available in full text so that its full contents can be reviewed. c) Duplicates should be removed since two different sources are used and some publications might be found in both sources.

2.3.3 Quality assessment of case studies

The publications are scanned for content that can be used to answer the research questions. The following quality criteria is applied: a) The contents of the publication describe a case study of a product line engineering endeavor in the software industry. b) The case study includes a description of the business goals and the context of the organization. c) The case study describes the software product line architecture.

The list with the final selection of publications can be found in table 2.1.

2.3.4 Data extraction

The form described in this section helped organize the data extracted from publications and also to identify relevant information regarding application domains, design decisions and goals.

Figure 2.2 shows the fields included in the form.
Table 2.1: Selected publications

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Title</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CelciusTech Systems AB: A case study in successful product line development</td>
<td>[12]</td>
</tr>
<tr>
<td>2</td>
<td>Control channel toolkit: A Software Product line that controls satellites</td>
<td>[18]</td>
</tr>
<tr>
<td>3</td>
<td>Successful product line Development and Sustainment: A DoD case study</td>
<td>[20]</td>
</tr>
<tr>
<td>4</td>
<td>The US army’s Common Avionics Architecture Systems CAAS</td>
<td>[17]</td>
</tr>
<tr>
<td>5</td>
<td>Cummins Inc. Embracing the future</td>
<td>[16]</td>
</tr>
<tr>
<td>6</td>
<td>Salion Inc. A Software product line case study</td>
<td>[19]</td>
</tr>
<tr>
<td>7</td>
<td>Market Maker Software A.G.</td>
<td>[46]</td>
</tr>
<tr>
<td>8</td>
<td>AKVAsmart</td>
<td>[47]</td>
</tr>
<tr>
<td>10</td>
<td>DNV Software</td>
<td>[57]</td>
</tr>
<tr>
<td>11</td>
<td>Philips Consumer Electronics</td>
<td>[59]</td>
</tr>
<tr>
<td>12</td>
<td>Philips Medical Systems</td>
<td>[40]</td>
</tr>
<tr>
<td>13</td>
<td>Siemens Medical Solutions</td>
<td>[41]</td>
</tr>
<tr>
<td>14</td>
<td>Telvenet</td>
<td>[9]</td>
</tr>
<tr>
<td>15</td>
<td>The TCS Experience with the Recovery of Family Architecture</td>
<td>[25]</td>
</tr>
<tr>
<td>16</td>
<td>Freeing Product Line Architectures from Execution Dependencies</td>
<td>[22]</td>
</tr>
<tr>
<td>17</td>
<td>Danfos Drives: Minimally invasion migration to Software Product Lines</td>
<td>[39]</td>
</tr>
<tr>
<td>18</td>
<td>Experiences with software product line engineering in Product Oriented Organizations</td>
<td>[52]</td>
</tr>
</tbody>
</table>
Chapter 2. A systematic literature review of SPL case studies

2.4 Architectural Drivers of Software Product Lines

2.4.1 Business Goals

The primary motivation for every organization embracing the software product line approach is to reduce the time-to-market of new products and to reduce the effort necessary for creating similar products. For many of the organizations described in the case studies this motivation constitutes a business goal in itself but for others the goal lies in the inherent benefits of systematic reuse of assets throughout the software development process. Either case, business goals constitute the benefits the organization expects through the successful application of the software system. In the context of product line engineering the benefits concern directly the development organization rather than the customer. It is possible to categorize the business goals with respect to the business area to which the achievement of business goals will directly benefit: economic benefits that the organization seeks through the reduction of development costs, maintenance and adaptation costs, reduction of product time-to-market, and the facilitation of the product portfolio expansion; production will benefit from the application of systematic reuse of software assets, improved planning of variant production, more efficient development process by means of production plans, improved product quality resulting from standard practices and risk reduction, seamless product customization of functionality or behavior and reduced testing effort; and finally organizational benefits like standardization of development practices.

<table>
<thead>
<tr>
<th>Case study</th>
<th>Number</th>
<th>Title</th>
<th>Publication</th>
<th>Author</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Domain</th>
<th>PL Effort</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Business Goals</th>
<th>Key Functional Requirements</th>
<th>Quality Attributes</th>
<th>Constraints</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Design Decisions</th>
<th>Conflicting Drivers</th>
<th>Implications</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Architecture Rationale</th>
</tr>
</thead>
</table>

*Figure 2.2: Extraction form*
Chapter 2. A systematic literature review of SPL case studies

Figure 2.3: Business Goals

Economic Goals All the case studies report the organization’s expectancy of benefits in different economic areas as the major driver for adoption of the product line engineering approach. Cost reduction with respect to development of new products and the adaptation and maintenance of existing products represents the main motivation. It’s also interesting to note that for many organization the motivation of significant economic benefits comes from data reported in other well-known case studies like CelciusTech, and Cummis yet few organizations report the quantification of their own savings.

The reduction of time-to-market of new products is also an appealing reason due to the competitive nature of the market and the organization’s necessity of satisfying customer demands on time. The usual way to deal with future market demands is to develop the software according to a product roadmap and a product portfolio, which is based on a prediction of customer expectations, yet this could be very costly when the product portfolio is extensive and the time between product release is very short.

Production Goals Software reuse is a business goal because of the benefits it offers to enhance software production. That is, the possibility of writing code once for one product feature and reuse it again in different products that have the same feature without duplicating neither the source code nor the effort to write it again. Code duplication avoidance, establishment and utilization of development assets, standardization of development practices and the possibility to automate the production process though model-driven software development are among the reasons that encourage reuse for software production.

Product planning benefits as well because it is easier to estimate the effort required to develop a product when the software elements that constitute the product exist in advance. This also improves development efficiency
because the programming effort is mainly for the integration of different reusable development assets to create a new product.

The organizations subject of the case studies acknowledge the potential benefit of having human resources managed more efficiently due to the possibility of a more flexible allocation of resources to different activities during software development.

For the organizations that expressed product quality improvement as their goal, it involves preserving the quality of products when series of products with several variants are created. This entails an additional complexity in production because the lifecycle of each product in the series is managed separately, thus many different and intricate aspects exists simultaneously.

The customization of product functionality and behavior is in relation with the ability to respond to customer demands on time. It should be possible to adjust the product to answer to the particular needs of individual customers without altering the essence of the product and the production of it, even though new functions are added or qualities are changed.

### 2.4.2 Quality attributes

A software system is characterized by several quality attributes. These quality attributes are properties whereby the behavior of a system can be measured qualitatively and quantitatively in order to determine its fitness for the system context [7]. These properties are commonly specified as non-functional requirements and their presence in a system may be observed both at run-time and non-runtime.

The case studies subject of these systematic literature review describe the functional requirements of the software systems and the relevant quality attributes in the context of their application domain. These attributes can be categorized in five groups based on a taxonomy of quality attributes [7]: performance, availability, modifiability, scalability, portability, security, and safety. The case studies also describe the prioritization of each of these quality attributes as required by their particular business goals. Therefore the quality attributes in the suggested categorization can be further categorized to reflect the relative importance of the attribute in each application domain. The prioritization applies a value from a nominal scale of three priorities: low priority (L), medium priority (M), and high priority (P). Case studies are grouped in application domains and for each case study a value from the scale is assigned to every quality attribute. The determination of the priority value is performed on the basis of the following criteria: a) A explicit mention of the quality attribute in the case study report; b) The number of design decisions that address the specific quality attribute. The categorization can be seen in table 2.4.

As expected, modifiability is the system quality of highest importance for every case study because it determines the ability of the product line to build series
### Figure 2.4: Prioritization of quality attributes by application domain

<table>
<thead>
<tr>
<th>Domain</th>
<th>Case Study</th>
<th>Performance</th>
<th>Availability</th>
<th>Modifiability</th>
<th>Configurability</th>
<th>Extensibility</th>
<th>Scalability</th>
<th>Portability</th>
<th>Security</th>
<th>Safety</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aerospace</td>
<td>3</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>Naval</td>
<td>1</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Avionics</td>
<td>2</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>M</td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td></td>
</tr>
<tr>
<td>Equipment Control system</td>
<td>14</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Automotive</td>
<td>5</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>M</td>
</tr>
<tr>
<td>Train Control System</td>
<td>15</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>H</td>
<td></td>
</tr>
<tr>
<td>Information Systems</td>
<td>8</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>L</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>L</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Consumer Electronics</td>
<td>11</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>14</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16</td>
<td>M</td>
<td>M</td>
<td>H</td>
<td>M</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>19</td>
<td>M</td>
<td>L</td>
<td>M</td>
<td>H</td>
<td>L</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td>Medical systems</td>
<td>12</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td>H</td>
<td>M</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
<tr>
<td></td>
<td>13</td>
<td>M</td>
<td>L</td>
<td>H</td>
<td>M</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>
of applications. The incorporation of modifiability defines the ease with which
the software system can accommodate changes to its structure and behavior, yet
it doesn’t specify the nature of the changes. That is, changes that facilitate
the creation of variants of the software with the similar functional and quality
characteristics, or changes that enable the evolution of the software to accommo-
date new characteristics [55]. In this sense the categorization shows modifiability
decomposed in two sub attributes: configurability and extensibility. Once the
modifiability is decomposed into sub-attributes it is possible to appreciated that
configurability is still the predominant quality attribute with high priority for
every application domain, but extensibility is not equally important.

The performance and availability qualities are highly ranked in the Aerospace,
Naval, Avionics, Automotive, Rail and Automatic Control systems domains. The
nature of these domains mandates highly reliable and robust systems that can
operate efficiently in constrained environments. The case studies show the main
reason the software has these quality attributes is that the systems include elec-
tronic components that pose time- and space-critical requirements. Safety is also
a highly regarded quality attribute due to the possibility of endangering people
or the environment if the system does not function adequately. An important
tradeoff point exists between safety and configurability. Safety mechanisms typ-
ically require the design of failsafe mechanisms that impose restrictions on the
software behavior thus leaving less space for a design in which variations may be
introduced [24].

In the case of software product lines for information systems, the highest
ranked quality attribute besides modifiability is scalability. Scalability is con-
cerned with the ability of the system to change as means to handle large amounts
of work. Commonly, the capacity of the system is a variation among members of
a product family. The architect of a product family with scalability require-
ments has to make design decisions in favor of late-binding variability mechanism that
bound variants with the required capacity. It is possible to argue that scalabil-
ity is related only to performance but it is also a property that applies to other
quality attributes like availability because a scalable system, which has spare
servers to handle high workloads, may also improve the overall system availabil-
ity. Usually, scalability influences positively modifiability, but tradeoff points may
be found in the design decision that choose a specific technology to implement
scalability. Similarly, the application of frameworks to allow for extensibility may
hamper configurability, because of the impossibility of changing the framework’s
behavior, for example web-based and cellphone frameworks define a workflow for
handling events, answer request and send responses that every application must
follow.

Consumer electronics fall in the category of mass-customization product lines
[45]. The architecture of a family or products of this type has as its main concern
the modifiability of the software in order to facilitate the production of several
product variants. Namely, the architects seek the configurability property as
means to build software with different features for similar products as well as a
design that provides for future changes i.e. extensibility. A common challenge
in this domain is hardware variation. The design must make possible the adap-
tation of software to different hardware platforms, and form factor, therefore a
layered design can be applied to separate hardware functions from application
functions. A tradeoff point of this design decision exists between modifiability
and performance. The encapsulation of hardware functions into components and
the separation of functions into low-level and application-level layers that fa-
vors modifiability may suffer from performance degradation due to excessive API
wrapping [28].

The case studies describe how the software architecture dealt successfully with
conflicts between quality attributes, yet the design decision they made in each
case to reach a satisfactory trade-off, depends highly on the context of the product
line and the influence of business goals.

2.5 Architecture Rationale

The descriptions of the case studies were also inspected for the identification
of practices in relation to the management of architectural knowledge. The re-
results in table 2.5 show three groups: case studies that don’t describe any active
management of the architecture knowledge; those that describe activities that
partially record architectural knowledge; and those that manage actively archi-
tectural knowledge. In the first case, the AK is not actively managed thus it
remains implicit in design artifacts. The case studies in the second group re-
port that AK is partially recorded: a Core asset modification guide that provides
detailed steps for changing the design or implementation of core assets; it is de-
scribed in the language used for describing the architecture (ADL); a person has
the knowledge management role and his task is to share the architecture rationale
among the developers; The third group of case studies report the following ways
of AK management: A reuse plan describes the process of architectural instanti-
ation; the architecture vision and philosophy is comprehensively documented and
communicated to software developers; the organization has a stablished knowl-
edge management practice whereby design rationale and experience is structured
and saved in a knowledge base; the architecture rationale is built collaboratively
and saved unstructured in web pages.

2.6 The influence of business goals on design de-
cisions

The case studies describe the decisions architects made during the architecture
design as means to achieve the envisioned benefits of a software product line.
Chapter 2. A systematic literature review of SPL case studies

<table>
<thead>
<tr>
<th>Domain</th>
<th>Case Study</th>
<th>Reported notion of Architectural</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aerospace</td>
<td>1</td>
<td>Yes</td>
<td>Reuse plan describes architectural instantiation process</td>
</tr>
<tr>
<td>Naval</td>
<td>2</td>
<td>No</td>
<td>Implicit in APIs and architectural design views</td>
</tr>
<tr>
<td>Avionics</td>
<td>3</td>
<td>Yes</td>
<td>Production plan describes architectural instantiation process</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>No</td>
<td>Implicit in artifacts from architectural design</td>
</tr>
<tr>
<td>Equipment Control system</td>
<td>16</td>
<td>No</td>
<td>Implicit in artifacts from architectural design</td>
</tr>
<tr>
<td></td>
<td>17</td>
<td>Partial</td>
<td>A Core Asset Modification guide provides detailed steps for changing the core assets' design and implementation</td>
</tr>
<tr>
<td>Automotive</td>
<td>4</td>
<td>Yes</td>
<td>Comprehensive information about architectural vision and philosophy is broadly communicated</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
<td>A development handbook is continuously updated with the component's configuration reference for the realization of variants</td>
</tr>
<tr>
<td>Train Control System</td>
<td>14</td>
<td>Partial</td>
<td>Implicit in the ADL language</td>
</tr>
<tr>
<td>Information Systems</td>
<td>5</td>
<td>Yes</td>
<td>Knowledge management captures design rationale and experience in structured repositories.</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Partial</td>
<td>AK is implicit in scenarios, DR is managed by the Architecture Mgmt role.</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>Partial</td>
<td>Skilled developers share knowledge throughout the development process</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>No</td>
<td>Implicit in artifacts from architectural design</td>
</tr>
<tr>
<td>Consumer Electronics</td>
<td>10</td>
<td>Partial</td>
<td>Implicit in the ADL language</td>
</tr>
<tr>
<td></td>
<td>13</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Medical systems</td>
<td>11</td>
<td>Yes</td>
<td>Architectural Knowledge is published on a web site.</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>No</td>
<td>Implicit in artifacts from architectural design</td>
</tr>
</tbody>
</table>

*Figure 2.5: Architectural Knowledge*
These design decisions could be gathered from text describing the architecture design and also from graphical representations of architectural views. The collection of design decisions found in each case study can be seen in the extraction forms in appendix A.

The analysis of these design decisions shows that business goals influence design decisions so that the conceived strategies not only address quality attributes and functional requirements but also pursue an economic or production goal i.e. architectural concerns. Similarly there are dependencies among decisions since the mechanism conceived in one design decision may require making other design decisions in order to make them feasible. From the previous analysis it is possible to define a template for describing the elements of a design decisions that could help understand the rationale behind it. See figure 2.6

<table>
<thead>
<tr>
<th>Design Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>Description</td>
</tr>
<tr>
<td>Influenced by</td>
</tr>
<tr>
<td>Business Goals</td>
</tr>
<tr>
<td>Addresses</td>
</tr>
<tr>
<td>Requirements</td>
</tr>
</tbody>
</table>

*Figure 2.6: Template for describing design decisions*

The following paragraphs provide a summary of the design decisions that are common among case studies.

Evidence of the influence of business goals found in the case studies suggested that economic objectives influence the selection of strategies (architectural design that follows styles and patterns) that rely on Enterprise Architecture Frameworks. Goals aiming at reducing development and maintenance costs will result in decisions to design an architecture that can be realized using standard software frameworks in which COTS can be easily adapted. Similarly, maintenance efforts can be simplified through the selection of architectural designs with qualities that facilitate maintenance, for example a layered architecture that limits communications between modules in different layers through well-defined interfaces permits changing modules as long as they keep using the same interface.

Production business goals influence how the architecture will prescribe the software detailed design and construction. The aim of reusing software assets influences the decisions of creating a modular design through the application of principles that dictate the encapsulation of functions and data into separate modules with well-defined interfaces that enable the creation of a highly cohesive structures. Similarly the architecture design applies the principle of separation of
concerns that prescribes grouping modules based on the type of service they provide for the realization of individual functionalities. The advantage of a modular design is the partition of functionality into discrete scalable, modules, consisting of isolated, self-contained functional elements that can be reused in different systems to drive multiple functionalities [49]. A component-based software development, which relies on modular design, is the preferred mechanism adopted in almost every case study for accomplishing software reuse. Also a modular design can be separated into software modules that are built independently. Even more, the motivation of software reuse will give priority to the construction of modules that are meant to be reused as building blocks for several products. That is, the construction of software components for reuse and the construction of modules with reuse can be performed by a separate group of developers. The advantage of applying a component-based approach to software development is that it loosens dependencies between modules through the utilization of interfaces for inter-module communication so that developers don’t need to wait for other developers to complete their work.

2.7 Conclusions

In this section we provide answers to the research questions stated at the beginning of the chapter.

RQ1: How do different application domains prioritize architectural drivers?

The information found in the case studies show that the main motivations of an organization for engaging in a product line endeavor are the potential economic benefits. Nevertheless, further analysis showed that business goals can be separated in two groups: economic goals and productivity goals. Organizations that prioritize economic goals may select Enterprise Architecture Frameworks as a means to reduce production cost and product time-to-market, whereas organizations that prioritize production goals aim at changing their current software development practice as a way to attain economic benefit. Modifiability is an intrinsic property of software product lines. In the industry of consumer electronics where mass customization of products is necessary, the ability of configuring the software to adapt it to different products is very important. Case studies in distinct industries prioritize quality attributes differently. The automotive, avionics, rail and automatic control industry, in which the software is tightly coupled with hardware, prioritize performance, availability and safety over all other attributes. In the case of the information systems industry, the ability to scale the software easily in order to handle bigger workload is regarded as important. This ability may also require portability so that the software can be run in more powerful hardware.
RQ2: **What is the link between design decision and architectural drivers?**

The inspection of design decisions of different case studies showed a relation between business goals, design decisions and even with the variation mechanisms chosen to realize the design. The architect makes design decisions to address quality attributes by choosing different strategies. In the case of modifiability, the strategies are translated to variation mechanisms. The collaboration between related design decisions and the selection of appropriate mechanisms permits the accomplishment of business goals.

RQ3: **How is the architecture knowledge captured and managed?** Only few case studies report established practices for capture and management architectural knowledge. A reuse plan provides the application engineers with a detailed description of the architecture instantiation process as well as the architecture vision, philosophy and the design decisions rationale whereby variation mechanisms were selected.
Chapter 3

The AKinSPL method

3.1 Introduction

The systematic literature review in the previous chapter showed that the software architecture has an important role in the implementation of software product line. Software architecture is the first artifact in the solution domain that shows how the requirements will be satisfied through the software so that business goals are met. It is not only the primary enabler of systematic software reuse, but the link between product configuration and product realization. This chapter proposes a method for capturing the architectural knowledge in software product lines (AKinSPL) upon which those links are created. First, we discuss the factors in product line engineering that architects consider during architectural design and next we present a model of architectural knowledge underpinning our method.

3.2 Elements of the architectural design of a family of products

The following subsections describe the elements of architectural design of product families. Each section describes the role of the elements in architectural design and the specific issues of concern to product line engineering.

3.2.1 Architectural drivers

Architectural drivers are the combination of functional, quality attribute and business requirements [7]. Additionally there are organizational, technological and product factors that pose a significant challenge to architectural design [32]. Particularly, the architectural drivers of a product family take into consideration business requirements that include: the scope of the market to be addressed because it defines the product portfolio and hence the scope of the product family; process goals that require the systematic utilization of reusable software assets for the development of products; and organizational constraints that require the separation of asset development and product development. Similarly, the quality
attributes are not only the properties shared by all members of the family, but the properties of the system whereby individual products are derived. That is, the joint work of product configuration, the instantiation of the product line architecture and the variation mechanisms that enable the realization of a product.

Architectural drivers are prioritized and documented in order to establish their influence on design decisions through traceability links between them. Also, the product, technological and organizational factors of architectural drivers should be clearly described in order to identify the problems and architectural challenges which design decisions need to deal with.

3.2.2 Variability model

The variability model is a representation of the association between common and variant elements in a family of products. It defines the different ways in which the elements can be combined in order to derive different products. A Feature Model shows the association of elements from the end-user’s viewpoint, thus a product is described as a combination of features. A feature is a set of software functions working together to provide a particular system functionality thus features are mapped to several architectural elements arranged in a structure that allows for the implementation of the functionality. Equally, the decision points in the feature model are represented as variation points in the architectural model. However, mapping from a feature model to an architectural model is not a straightforward process because of the mismatch between the abstraction levels each model applies to represent the concepts within. For instance, a feature may involve several elements in the design and also span different views of the architectural model. Similarly, a variation point in the feature model may involve the utilization of design patterns that would allow for a flexible architectural model capable of accommodating variations of behavior and quality. Furthermore, there are dependencies among variant and variation points in the variability model that need to be fulfilled by the architectural design so that the structure of the architecture enables the composition of components as dictated by the resolution of the feature model.

One way to deal with these challenges is to make explicit the association between elements in the variability model and the architectural model. Thiel et. al. [55] Propose an architectural variability model wherein architectural variability addresses feature variability. In this model, a variation point specification includes a description of the reasons for the inclusion of the variation in the design and which features are affected by the variation point. The model shows also the inclusion of architectural variation points in different architectural views and the effect of the variation point over the design of elements within the view and across views. The impact of variation points over design elements in multiple views may strongly influence structural properties of the architecture [55].
3.2.3 Realization and binding time

In application engineering the features that constitute the product are selected and a decision model is used to resolve the variability model. The resolution of the variability model is the selection of particular variants in different software development artifacts. The resolution of variability in the product line architecture entails the selection of architectural elements at every point in the architecture where variation points exist. The challenge is to derive an architecture that is consistent with both, the reference architecture and the variability model. The product’s architecture should be consistent with the structure of reference architecture. That is, the elements of the product’s architecture are a subset of those in the reference architecture and the association between elements adhere to the constraints on the possible connections among elements. Additionally, the architecture should enable the achievement of the product’s features as described in the variability model. The decision model is a set of rules describing operations for the derivation of the product’s architecture from the reference architecture. The rules describe the effects of the resolution with respect to the elements that will make up the structure of the architecture. However the resolution rules are high-level descriptions of how to combine coarse grained architectural elements that don’t provide any details with respect to the concrete mechanism that has to be applied for realization and neither any information about the binding time of variants to variation points.

The architecture variability model proposed by Thiel et. al [55] makes explicit the association of a variation point with the resolution rules and binding time. In the model, the resolution rules describe the strategies to be applied for variant resolution and the binding time when a variation needs to be resolved.

3.2.4 Dependencies

The variability model defines dependencies among variation points and variants that constrain the manner these elements can be combined. The most basic dependency is between variants and the variation point from which they are derived. These dependencies constrain the combination and cardinality of variants derived from the same variation point. Additionally, there are dependencies between variants of the same variation point or from different variation points, or more generally a dependency between variation points, that constrain also the selection and combination of the elements that constitute a product. Except for the variation point variant dependency, other dependencies are not usually represented in architectural views, yet the constraints they impose must be observed during architecture realization.

The architectural variability model proposed by Thiel et al. [55] represents dependencies among variation points within one architectural view and between architectural views since the design elements in one view correspond to elements
in other views.

### 3.2.5 Product specific Design Decisions

Application engineers create products through derivation and adaptation of reusable assets. The derivation process is based on the configuration of the features that constitute a product. The decision model ensures that the selection and combination of elements in the variability model be mapped to architectural elements in a structure that is also feasible in terms of the architectural design. However, there are features, which can not be directly derived from the reference architecture, that may entail particular adaptations to reusable assets, thus, product-specific design decisions must be made in order to address these adaptations. In the context of a product line architecture, product-specific features are also architecturally significant requirements that influence the design decisions that shape the product-specific software architecture.

Capilla et. al. [15] proposed a data model comprising the core entities that support architectural knowledge. In the model, an entity represents both, common and product specific architecturally-significant requirements. This entity is associated with another entity that models the design decisions concerning a particular product.

### 3.3 A Model of Architectural Knowledge

The survey of the state of the art of Knowledge Management in Software Architecture [26] shows that several authors have different notions on what architectural knowledge is, and moreover, what this knowledge entails. Nevertheless, the models they use to characterize their own definitions of architectural knowledge show that design decisions and their rationale are common key elements of each model. However, the words they use to describe the same concepts are different thus complicating the creation of a general method for management of architectural knowledge. Furthermore, only the models by Capilla et al. [15] and Thiel et al. [55] include concepts of product line engineering.

The method for capture and management architectural knowledge must rely on a model that is general rather than domain specific and minimal rather than complete so that new concepts could also be represented on the basis of key concepts defined by the same model. Therefore, the model could provide the general concepts of architectural design, which are the same of product line architecture, and extend them to embrace the more particular concepts of product line engineering.
3.3.1 Model selection

The Core Model of Architectural Knowledge proposed by [21] claims to offer maximal expressivity in the architectural knowledge domain so that other models can be expressed in the form of extensions to their reference model. The authors have developed and refined the model through four case studies in industry and validated the core model by mapping the concepts to terms found in accepted literature of frameworks for architectural knowledge. The model proposed by [15] is conceived to represent entities in a data model upon which a software tool could be built. The model by Thiel et al. [55] builds on the IEEE P1471 for extending the artifacts and other documents that define and describe the architecture of a product family. Both models focus on artifacts, yet they lack expressivity as it is not possible to model the activities of architectural design. Although our method won’t use any of these models for its underlying reference model, it will leverage the concepts that are relevant to product line engineering.

For the purpose of the current method, the Core Model of Architectural Knowledge [21] will be used to represent the knowledge within the architectural design process.

3.3.2 Model adaptation

This section describes how the Core Model of Architectural Knowledge accommodates the elements of product family architectural design identified in section 3.2. Figure 3.1 shows a model of Architectural Knowledge adapted from the Core Model of Architectural Knowledge in [21].

Architectural drivers

Business goals and process requirements are stakeholder CONCERNS for which a decision must be made. This process can be clearly described with the model in the following way: a DECISION TOPIC is a concern for which several ALTERNATIVES are proposed; a DECISION is the chosen ALTERNATIVE that addresses best the DECISION TOPIC. Choosing an ALTERNATIVE is the result of a trade-off among several alternatives that address a concern for which the decision topic is the prioritization of specific quality attributes.

Variability concepts

The variety of products characterized by features and decision points in the variability model entail a flexible software architecture design that allows for reuse. Therefore, the outcome of the CONCERN for flexibility is a DECISION that influences the Architectural Design thus, the flexibility of the architectural design is enforced by an OPEN DESIGN DECISION. An architectural design decision that addresses flexibility can be represented in the model as a a sub-class
of the Architectural Design Decision element. Then, the Architectural Design reflects the concepts of variability in different artifacts. See figure 3.1.

**Realization and binding time**

The elements of the Architectural Design can reflect the Architectural Design Decision using distinct Languages for different Artifacts. For instance, it can be represented in the architecture specification with models, as well as impact the source code. Therefore, variation points can be represented with distinct notations in different architectural views; variants and binding time are implemented through the application of a variation mechanism in source code. The Architectural Design uses a Variability Language to represent the variation points, variants and dependencies in the artifacts.

**Dependencies**

The dependencies described in the variability model are also concerns for which architectural design decisions must be made in order to create an architectural design that observes the constraints imposed by those dependencies. The architecture will reflect the dependency constraints with the Variability Language used to represent the Architectural Design in the Artifact.

**Product-Specific design decisions**

Product-specific requirements pose architectural concerns which the reference architecture can not satisfy. These concerns are a special type of Decision Topic which also requires an Architectural Design Decision in order to satisfy the product specific requirements. These design decisions are not, nevertheless completely independent as they must be consistent with design decisions of the reference architecture.
Figure 3.1: Adaptation of the Core model of Architectural Knowledge.
3.4 Method

The AKinSPL method will capture the explicit architectural knowledge found in the models, views and rationale that is conceived during the architectural design and the tacit knowledge found in the business goals and concerns that have influence on the architectural solution. Both types of knowledge are important for the design of product line architectures wherein several applications have the same concerns thus it is necessary to understand all the rationale behind a design that affects multiple products and the solutions that implemented that design.

Although, variability models describe the different products that the architecture supports, the knowledge of how the architecture was designed to support the variation of products in space and time remains implicit in architectural models. This knowledge should be made explicit and saved as an architectural asset in the core asset base of the product line infrastructure.

Software architecture is the mediator between product configuration and product realization, thus to understand how architects designed the architecture for this purpose should facilitate the derivation of products and the evolution of the product line.

Objectives:

The objective of the AK method is to capture the knowledge associated with the activities of architectural design in order to understand the rationale behind the design decisions that shaped the architecture in the way it is. Also, to make explicit the relation among architectural drivers, the design decisions they influence, and the architectural models resulting from the implementation of those decisions.

Aims:

1. To capture architectural knowledge throughout the process of architectural design.
2. To enable understanding of the relation between business goals and design decisions.
3. To describe the relation between architectural model and the design decisions from which it originated.
4. To describe the relation between architectural models and variability models.

3.4.1 An iterative method

Architectural design is inherently an incremental and iterative process [31]. High-level decisions, like choosing and architectural style will naturally lead to a succes-
sion of subsequent design decisions. The AKvar method will capture the architectural knowledge associated with every decision in the succession, from high-level architectural design to detailed realization specification. Also, the method will capture the essential information for each design decisions together with the resulting architectural design.

A business-driven design decision (High Level) that addresses variability, describes a resolution with respect to the selection of strategies for attaining flexibility (configurability and modifiability) of the software. It also describes the variability technique applied to the architectural design for describing variation points. The strategies include the application of principles, the selection of architectural styles or architectural patterns, middleware or frameworks.

A productivity-driven design decisions (lower level) describes a resolution regarding the selection of variation mechanism by which variants are realized at the variation point. So, whenever the variability decision model is resolved for the derivation of a particular variant, the rationale will describe the variation mechanism to apply for the realization of the variant at the respective variation point in the architecture and the binding time so that the process could be performed efficiently.

**Design Decision View**

Design decisions can be represented hierarchically in a design decision view wherein the high-level design decisions are at the top level of the hierarchy and technical design decisions at lower levels [38]. The view shows the decisions that refine a high-level decisions and the dependencies that exist among them. The method captures the knowledge in a top-down fashion in order to acquire the knowledge from both, high-level design decisions and more refined design decisions. The aim is to establish the genealogy of these knowledge. The design decision view extends the notation found in [38] in order to represent business goals and their influence on design decisions. Additionally, it represents a design decision refines another design decision and its conception might depend on other design decisions too. See figure 3.2.

The hierarchical representation of design decisions is orthogonal to other architectural views. The rationale behind a design decision need not be attached to every artifact, but it must be traceable so that it’s clear the purpose of the model described in the artifact through the explicit association with its design decision.

**Design Decision specification**

The method elicits the rationale for every design decision in the hierarchy as well as the influences that contribute to each decision. The Architecture Decision Template described by [56] lists the essential information of a design decision in a structured manner. The fields of the template are described in table 3.1:
Table 3.1: Architecture Decision Template adapted from [56]

<table>
<thead>
<tr>
<th>Issue</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decision</td>
<td>A clear description of the alternative that has been selected.</td>
</tr>
<tr>
<td>Status</td>
<td>The decision’s status, such as pending, decided, approved</td>
</tr>
<tr>
<td>Group</td>
<td>Group decisions based on a common decision topic</td>
</tr>
<tr>
<td>Assumptions</td>
<td>Describe the context in which the decision is made. (Cost, schedule, technology. Also environmental constraints that may limit the alternatives under consideration</td>
</tr>
<tr>
<td>Constraints</td>
<td>Describe any additional constraints to the environment that the chosen alternative might impose</td>
</tr>
<tr>
<td>Positions</td>
<td>List the other alternatives or options considered before the selection of the actual decision.</td>
</tr>
<tr>
<td>Argument</td>
<td>The reasons behind the selection of one alternative above other alternatives. This includes TTM, ROI, and required development resources’ availability</td>
</tr>
<tr>
<td>Implications</td>
<td>The decision might imply organizational requirements (staff training), technical challenges (introduction of new requirements) and project planning alterations (renegotiation of scope and schedule with customers)</td>
</tr>
<tr>
<td>Related decisions</td>
<td>A decisions might be refined by subsequent decisions; decision might also depend on the existence of other decisions.</td>
</tr>
<tr>
<td>Related requirements</td>
<td>The decision addresses one ore more functional requirements or quality attributes.</td>
</tr>
<tr>
<td>Related artifacts</td>
<td>List the architectural models or scope documents that has impact on this decision</td>
</tr>
<tr>
<td>Related principles</td>
<td>The decision should be consistent with design principles or patterns upon which the architecture is built.</td>
</tr>
<tr>
<td>Notes</td>
<td>Capture issues and observations to the</td>
</tr>
</tbody>
</table>
The template encompasses the concepts represented by elements of the Core Model of Architectural Knowledge presented in section 3.1. The template’s fields can capture the concepts of architectural design as it is described by the core model, including the extension that were proposed to include concepts of product line engineering.

**Traceability to architectural models**

The architectural model is linked to the set of design decisions from which it has originated and the underlying reasons for its current structure. The different views of the architectural design reflect the architectural design and include the concepts of variability through extensions to the language employed to specify the view. The Core Model of Architectural Knowledge shows how architectural design decision are enforced upon the architectural design i.e. decisions drive views [56].

### 3.4.2 A general method

The General model for Software Architecture Design [31] will provide a process framework for the creation of the AKinSPL method. The model describes the activities and artifacts that are part of the architectural design as well as the workflow in which activities utilize and produce artifacts with the objective of creating the software architecture. Although the model does not make any special consideration with respect to the design of a reference architecture for a family of products, the concepts of product line engineering will be introduced so that assets and activities could be seen in that regard as well.

The following are descriptions of the main aspects of the general model for software architecture design and their related artifacts.
Chapter 3. The AKinSPL method

Architectural concerns  Concerns are requirements on the system and also system properties such as performance, reliability, security, and evolvability [30]. A typical issue of a product family is the trade-off of between flexibility and other system properties so that the software system could be parameterized and extended in order to include the customer’s features.

Context  According to IEEE 1471 the system’s environment or context determines “the setting and circumstances of developmental, operational, political and other influences upon the system” [30]. In the contexts of a product family the organization’s business goals include economic goals and productivity goals. The former are influenced by the current state of technology and the latter by characteristics of the organization.

Architecturally-Significant Requirements  An ASR is “a requirement upon a software system which influences its architecture” [30]. The ASR of a product family concern the reference architecture and the architectures of individual products as well.

Architectural Analysis  This activity identifies problems in the architectural concerns and context that the architecture must solve. The scope of the product family is analyzed in order to determine the scope of the software product line architecture.

Candidate Architectural solutions  Candidate architectures present alternative solution to the ASR. Each candidate architecture includes a description of the design decisions that define the structure of the proposed solution, the rationale behind the decisions and traceability of decisions to requirements.

Architectural synthesis  This activity puts forward the architectural solutions to a set of ASRs.

Validated architecture  The validated architecture is a candidate architectural solution that satisfies consistently the ASR. The set of artifacts in the architectural specification are consistent and complete. That is, functional and quality requirements are meet in every architectural view and the overall architectural specification can enable realization.

Architectural evaluation:  This activity validates whether the design decisions made can consistently satisfy the ASR.

3.4.3 AKinSPL description

The AKinSPL method defines a sequence of activities in which architectural knowledge is acquired in a top-down fashion.
Chapter 3. The AKinSPL method

1. **Assertion business goals** A clear statement of the organization’s business goals helps architects to be tuned with the stakeholder’s expectations of economic benefits and productivity improvements for the organization. This activity identifies the high-level decisions that address business goals.

1.1. **Specify economic goals** The enterprise’s business goals with respect to the economical and organizational factors determine the context in which the software is developed. These may affect the selection of technologies and the production strategy of the project. Economic goals may influence several decisions therefore it's necessary to document the rationale behind the decision and how the organization will benefit from it, and what the consequences are for the architecture. The business goal and the expected outcome should be described. Additionally, the metric to assess the accomplishment of the goals should be specified.

1.2. **Specify productivity goals** The acknowledgment of production goals will help the architects make decisions on the appropriate strategies for the application of principles and mechanisms that will help achieve productivity goals. These are architectural concerns with respect to the flexibility of the design and how to make effective use of it. The productivity goal and the expected outcome should be described. Additionally, the metric to assess the accomplishment of the goals is specified.

Business goals are specified using the template described in section ???. For each business goal, a goal-type node is placed at the top of the graphical representation.

2. **Document decisions** Decision making is the process of selecting the alternative that has the highest rank among several alternatives proposed to address business goals [21]. The alternatives are ranked based on the result of the trade-off analysis between different concerns. Particularly for product families, the most important concerns are the quality attributes that would facilitate product derivation and allow for software reuse.

The template described in section 3.4.1 captures the concerns each decision addresses as well as the influences posed by either, business goals or other decisions. The influences on design decisions are represented graphically, by placing links between a decision node and other decision nodes or the goal-type node that influences it.

3. **Document links between Design Decisions and the Architectural Model.** Key functional requirements and quality attributes are addressed by decomposing the system into functional elements and making decisions
regarding the most convenient way of using these functional elements, either by following an architectural style or an ad-hoc design i.e. a solution strategy.

These decisions are reflected in an architectural model that adheres to the design guidelines of the selected solution strategy, thus the documentation should describe the functional elements included in the solution and the tactics for the application of the strategies with the given functional elements.

For instance in a product line architecture, decisions aim at defining strategies for addressing the modifiability and configurability quality attributes and thus, architectural models incorporate design elements that enable a flexible architecture.

3.1. **Link variability Model with Architectural model** The documentation describes the rationale behind the connection between a variation point in the architectural model to decision points in the variability model. Equally, the rationale makes explicit the link between generic architectural elements to features in the variability model.

The design elements that allow for the configurability quality attribute define points of alteration or extension where variations to the architectural model are permitted. Therefore, the documentation must include how variability is modeled in architectural design. That is how variation points and variants are represented in every view of the architectural design.

3.2. **Describe VP realization and Binding time** For every variation point in the design, describe the variation mechanism and binding time. The rationale documents the reasons for the selection of a particular variation mechanism and how it has to be used to derive variants from the variation point.

The selection of a concrete variation mechanism should be consistent with the results of the trade-off analysis between modifiability, configurability and other quality attributes. The description of the relation between the variability model and the variation mechanism is important when the object of variation are the quality properties of the product and these qualities may only be attained through the selection of a particular variation mechanisms.

4. **Describe product-specific design decisions** The documentation describes the rationale behind the adaptations that are necessary to meet the requirements of products which are not fully covered by the architecture.
Figure 3.3: Workflow
Chapter 4

Application

4.1 Introduction

The previous chapter presents the AKinSPL method for capturing the knowledge associated with the design process of a software product line architecture. The method defines processes that match the common activities of the architectural design process but focuses on the activities and artifacts that concern with product line engineering. AKinSPL is designed to be a general method that can be adapted to any software architecture methodology. It defines abstract procedures that can be adjusted in order to define concrete tasks to capture the knowledge associated with specific activities of architectural design. The aim of this chapter is to apply the concepts defined by the method to adjust the activities of the PuLSE-DSSA methodology so that the architectural knowledge can be captured and managed.

4.2 PuLSE-DSSA

PuLSE is a Product Line Software Engineering methodology developed at Fraunhofer IESE for the purpose of enabling the conception and deployment of software product lines within a large variety of enterprise contexts[8]. PuLSE is articulated around three main elements: the deployment phases that represent the logical stages through which the product line is initiated, established and used; the technical components that provide practical knowledge required to put into operation the product line development; and the support components that guide the appropriate adaptation, evolution and deployment of the product line. See figure 4.1

Domain Specific Software Architecture (DSSA) is a technical component of PuLSE that defines guidelines on how to develop the reference architecture while maintaining traceability to the model that represents characteristics of the products within the scope of the product line. That is, the software architecture that covers current and future applications of the product line as described by the variability model.
PuLSE-DSSA follows an incremental and iterative process of architectural design. See figure 4.2. The main input to the process are the business goals and the economical scope of the product line. The process is guided by generic scenarios and property-related scenarios, which describe respectively functional requirements and domain-independent quality aspects i.e architectural drivers. The generic scenarios are derived from the variability model created in the PuLSE-CDA phase and complemented with quality attributes described in property-related scenarios. The scenarios are ranked with respect to architectural importance so that the highest ranked scenarios be used as the initial set on which the initial architecture is based. The initial architecture is further developed in an iterative process until it supports all generic scenarios. At the end of the process the architecture is deemed as a reference architecture because the architecture of each individual application of the software product family can be derived from it.

### 4.3 Mapping the AKinSPL to PuLSE-DSSA

As stated in the introduction, processes defined in AKinSPL can be mapped to PuLSE-DSSA so that the knowledge associated with the activities of the architecture definition process be captured. The purpose of mapping activities is to identify the areas in which the PuLSE-DSSA can be adjusted to capture the architectural knowledge.

The following paragraphs describe the PuLSE-DSSA from the viewpoint of
the AKinSPL method in order to identify where AK could be captured and how to do it effectively based on AKinSPL’s procedures. Additionally, the different artifacts described in AKinSPL will be instantiated with the tools the company currently uses for architecture development. That is, the design utilities provided with Enterprise Architect (EA) [50] will be used either to extend the architectural diagrams or to create the necessary artifacts for capturing architectural knowledge. Conveniently, the authors of PuLSE-DSSA have customized the Enterprise Architect UML modeling tool to support the concrete architecture definition process [23]. They show how the general principles of architectural views are mapped to UML elements with tool support and how views and their artifacts can be well organized in a view-based documentation. Figure 4.3 shows the structure of folders for the organization of views and artifacts in EA’s Project Browser. Similarly, additional extensions to EA will be performed in order to show how AK can be efficiently managed with tool support.

**Figure 4.2:** Architecture Definition Process

**Figure 4.3:** Organization of Views and Artifacts in EA
4.3.1 Specification of business goals

PuLSE-Eco is a task wherein the economic scope of the product line is identified on the basis of business objectives and evaluation functions. Business objectives are the stakeholders’ main concerns. These concerns are specified through the results of the benefit functions that determine the best characteristics and best products for the product line to cover. The result is a product line plan that supplies information about the products, their characteristics, and constraints. PuLSE-CDA refines the economic scope produced by PuLSE-Eco by establishing boundaries of the product line based on storyboards that describe relevant types of action sequences in the domain.

The business goals with respect to economical benefits are clearly specified by PuLSE-Eco in the scope of the product line. It reflects the stakeholders’ economical expectations from the application of the product line for delivering existing, future and potential products. Additionally, it is possible to define goals with respect to productivity by benefit functions that show how the effort of developing the product line is shared by all products.

4.3.2 Document Decisions

The reference architecture design in the PuLSE-DSSA methodology is guided by generic scenarios, characteristics of external systems and constraints. A scenario is decisive and succinct description of a situation the system is likely to face together with the definition of the response expected of the system [...]. The architect makes decisions in order to create the architecture of a system that satisfies the situations described in different scenarios and deal satisfactorily with the constraints imposed on the system by its environment. A decision is made by selecting the most adequate alternative among different solutions that address to different extent the scenario and the constraints. Naturally, a decision that benefits one scenario may impact adversely the situation described in other scenarios, thus a trade-off among scenarios must be made.

These architectural decisions should be included in the architecture specification in order to enable a clear traceability from the requirements to the architecture and architectural solutions [23]. Also, architectural decisions follow principles of architectural design and the collection of decisions constitutes the philosophy of the architecture.

PuLSE-DSSA defines a meta-model of the scenario and decision view that makes explicit the rationale for making a decision in order to address the scenario. See figure 4.4.

The meta-model includes other factors that are also addressed by a decision: the Characteristics of External Systems that the decision takes into account to avoid a mismatch with the architectures of those systems; and Constraints that pose external restrictions that the architecture must deal with by enforcing ap-
appropriate decisions. The semantics of the meta model describe that Scenarios, External System Characteristics and Constrains are associated with a Decision by an \textit{is\_addressed\_by} dependency. Additionally, the \textit{is\_caused\_by} dependency shows the inherent trade-off that exists when a decision is made among scenarios of contrasting system qualities.

The meta-model is implemented in Enterprise Architect with an UML profile from which the elements of the Scenario and Decision view are selected to compose a new diagram. The EA project uses a predefined organization of packages that group the instances of Decisions, Trade-off, Constraint and Ext. Sys. Characteristics in separate packages. Instances of Decision can be further categorized and organized in separate packages according to different criteria that better suits the project.

For the purpose of capturing architecture knowledge, the package organization and UML profile defined by DSSA are extended in the following manner:

\textbf{Traceability matrix} Enterprise Architect shows a package for each group of cohesive scenarios wherein the view-based documentation is developed. The Decision View package within the scenario package contains the decisions addressing the current scenario, thus making the relation among them explicit. Further more, this relation is captured in a traceability table created with a Relationship Matrix template whereby the dependency relationship between a scenario and the the decisions that address it is automatically tracked.
Decision alternatives The alternative solutions to the problem that were re-jected are nevertheless documented in order to indicate the reasons that deemed them to be inadequate. to placed in a separate package. These alternative decisions constitute the Alternatives package. The Rationale artifact describes the underpinnings for the selection of a particular alternative as the architectural decision. It is linked to the decision element through the Linked-Document property of the decision element.

Relationship among decisions Decisions are associated in two ways: a Refine relationship that associates a general decision with a more detailed one; the Depends on defines that one decision depends on another decision in order to exist. Both elements are included as connectors in the UML profile.

4.3.3 Design Decisions and Architectural Model

As shown in the previous section, scenarios represent the link between the quality requirement and the architectural decision that will help accomplish that requirement. The outcome of decision is ultimately reflected in an architectural model that adheres to architectural styles or patterns in order to address the functional requirements and meet the quality attributes. Architectural models are diagrams that show graphical representations of the architecture definition. Architectural views comprise several diagrams showing different aspects of the view at different abstraction levels. Views conform to viewpoints that illustrate how the architecture addresses different stakeholder concerns [30].

In the scenario-driven process of architectural design defined in PuLSE-DSSA, an architectural solution is designed for a set of cohesive scenarios. The design of the architectural solution entails the application of architectural patterns that propose a solution to the problem described in the scenarios. Architectural patterns incarnate architectural design decisions for the realization of quality requirements. A number of patterns embody concrete elements for the application of an architectural style [23].

A pattern may satisfy a quality attribute in different situations and may collaborate with other patterns to satisfy multiple quality attributes. Patterns exist for different problem classes, and are grouped in pattern catalogs. Pattern selection is mainly the process of matching the problem at hand with the problem the pattern solves. If there is a match, the pattern is applied to the design. That is, the pattern is instantiated in the concrete context of the respective architectural elements of the system being designed. See figure 4.5.

Patterns are regarded as a common language that architects use for the communication of known solutions to recurrent problems. However, the use of a pattern in design doesn’t explicitly convey the purpose or the consequences of its application, unless they are documented along with the context in which the pattern is applied. Therefore it’s necessary to specify the utilization of a pattern
Chapter 4. Application

Figure 4.5: Realizing Quality Attributes with Patterns

with at least four sections: context, problem, solution and quality consequences [3]. The sections of the scenario template used in the PuLSE-DSSA provide the necessary information for pattern selection. Nevertheless, the template has to be adapted in order to include a description of pattern application in the design of the architectural solution.

**Context**  The environment, describes the particular situation in which the system is assumed to be applied. The situation described in the scenario should match the situation for which the pattern is recommended.

**Problem**  The problem which the pattern is selected for, can be characterized in terms of the stimulus that triggers the scenario and the expected behavior of the system in the way it is implemented by the pattern in response to the stimulus.

**Solution**  A description of how the pattern will address the problem.

**Quality consequences**  The application of the pattern might involve risks due to undesired side-effects on other quality attributes. Therefore, it is necessary to identify sensitivity points that have an impact on the quality of the system response. This is important since the architect might have to adapt the pattern in order to reduce the impact of pattern application on the overall quality or trade-off the quality attributes between scenarios.

**Application**  The application of a pattern to the design is guided by the roles the pattern defines and the collaboration between roles. These roles are filled by concrete architectural components that collaborate in the way the
Chapter 4. Application

pattern prescribes in order to solve the problem. Thus, pattern application can be described by a component specification model which is a synthesis of components identified by the functional decomposition of the system and components that articulate the roles defined by the patterns [29]. The pattern application is further refined through the decomposition of components into detailed design elements, until concrete software artifacts are identified. Additionally, it is possible to define the most appropriate mechanism for implementing the pattern by taking into consideration the impact on the quality of one mechanism over the other.

The main idea of capturing this information in the scenario-template is to make it explicit and ease the reasoning about design decisions and the impact of later design decision changes. By documenting architectural scenarios with information of both, specification and realization, the rationale for the selection and application of architectural patterns in the architectural solution is made explicit. The extended scenario template is shown in figure 4.6.

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Name of the scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Quality</td>
<td>Related quality attribute</td>
</tr>
<tr>
<td>Environment</td>
<td>Context applying to this scenario</td>
</tr>
<tr>
<td>Stimulus</td>
<td>The event or condition arising from this scenario</td>
</tr>
<tr>
<td>Response</td>
<td>The expected reaction of the system to the scenario event</td>
</tr>
<tr>
<td>Response Measure</td>
<td>The measurable effects showing if the scenario is fulfilled by the architecture</td>
</tr>
<tr>
<td>Steps</td>
<td>Necessary steps for realizing the scenario with the architecture</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Architectural Decisions</th>
<th>Risks</th>
<th>Sensitivity Points</th>
<th>Tradeoffs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name of architectural decision</td>
<td>Risk related to decision</td>
<td>Parameters or factors influencing the architectural decision</td>
<td>Relation to other qualities</td>
</tr>
</tbody>
</table>

**Figure 4.6**: Specification and realization of architectural scenarios

In addition to the explicit association of a scenario to the decisions that address it, Enterprise Architect facilitates the documentation of scenarios by attaching linked documents to scenario elements. The documents are linked document templates created as EA resources and linked to scenario elements in the definition of the UML profile for Scenarios and Decisions. See section 4.3.2. The templates contain a table that resembles the template defined before. The architect documents the scenario by opening the linked document and completing the information on the file defined in the template.
The models that constitute the architectural views of the solution are documented in separate view-type subfolders for the current scenario. That is, the diagrams show the different views for the solutions defined in the decisions that address the scenario. The documentation of architectural models that employ patterns is accomplished through UML Extensions. EA allows the architect to create UML pattern templates so that the solution defined in the pattern can be reused again. As the architect identifies a pattern in the architectural model, the UML diagram can be saved as a new pattern template in the resources view of Enterprise Architect. The creation of a template requires the definition of the actions for the elements contained in the pattern. The actions define how the elements of the pattern will be applied to the model using existing elements already identified during the functional decomposition. That is, pattern elements can: merge with an existing elements so that it takes on the role defined by the selected pattern; be created as an Instance of an existing element; be created directly without modification; and be created as types of an existing element. Figure 4.7 shows the dialog displayed in EA for defining the elements’ actions of the Abstract Factory pattern [50].

The convenience of directly linking the documentation to the diagram is twofold: it allows for the explicit documentation of scenarios in a standardized format; and reduces the effort of maintaining two separate documents using separate tools.

**Link variability model to Architectural model**

PuLSE-DSSA applies generic scenarios to describe a situation wherein the stimulus, environment and response of the software system match the feature variation points of a variability model and the actual values depend on the resolution of the model. Therefore, there are alternative design elements that describe structural and behavioral aspects that can not be bound during architectural modeling because they depend on the configuration of a product on the basis of user preferences. Particularly, when the selection and combination components is guided by the resolution of a decision model that determines the features of a member of the product family, the architectural model must reflect the existence of design elements that support different component specifications. That is, an architectural variability model that shows the parts of the design that allow for variations.

This notion is clearly expressed in the architectural variability extensions proposed by [55] whereby the architectural variability model is described by a set of variation points that identify the parts of the model with variable design elements. These elements might be present in: one design element in one view; multiple design elements in one view; or multiple design elements in multiple views. Thus the impact of the variation point is perceived throughout the architectural specification. See figure 4.8

The architect’s main concern is the creation of an architecture that includes all
Figure 4.7: Actions definition of a pattern’s elements in EA.

Figure 4.8: Architecture Variability Meta-Model [55]
the variation points of the variability model in order to support all the products within the scope of the product family. [58] suggest that there are three basic techniques to introduce variation in an architectural model: adaptation, replacement and extension. Architects would choose among these techniques on the basis of particular requirements of the product line. For instance, defining the behavior of a component that is already present in the system through parameters poses less platform overhead than locating and loading a separate component in order to adapt it to different products. Conversely, the ability to change the software’s behavior through dynamically loaded components allows for an easier adaptation of the system to yet unknown product applications than a pre-established set of behaviors determined by a set of parameters or replacements.

The architectural knowledge associated with the incorporation of variation points in the architectural model can be captured by means of a simple artifact: the Architectural Variability Documentation Form (AVDF). The AVDF includes fields that enable the specification of the design element that are affected by the variation point and the variants that can be bound to it. The form includes a fields to describe the role of each element in the architecture model with respect to the variability model: Variation Point or Variant; and the features that are affected by the particular architectural variation point. Also, the form includes a diagram to show the design elements affected by the variation point in the language of the respective view, typically UML. See figure 4.9

<table>
<thead>
<tr>
<th>Architectural Variability Documentation</th>
<th>Design Element Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>VP</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Role</th>
<th>Features</th>
<th>Model</th>
<th>(View 1) / (View 2) / (View 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element 1</td>
<td>Interface</td>
<td>Variation point/Variant</td>
<td>Feature id+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element 2</td>
<td>Component</td>
<td>Variation point/Variant</td>
<td>Feature id+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Element 3</td>
<td>Component</td>
<td>Variation point/Variant</td>
<td>Feature id+</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Element N</td>
<td>...</td>
<td>Variation point/Variant</td>
<td>Feature id+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A diagram showing Variation Points and Variants in the language of the selected view.

**Figure 4.9:** A form to document the architectural variability model

PuLSE-DSSA extends the Unified Modeling Language (UML) so that variability can be modeled with the conventional elements used in architectural and design diagrams created with existing tools, like Enterprise Architect. The UML extension defines that the variant elements are consistently shown in the different types of models tagged with the _variant_ stereotype. The type of variability the UML element represents is defined in a decision model that contains rules for the resolution of the variation point. Each model is accompanied by a separate decision model containing the decision name, the question, the variation point a
resolution rule and the effect of the resolution on the model. The rationale for
the incorporation of the variation point is also captured by attaching the AVDF
to the model.

The extension to the DSSA's EA profile includes two new artifact elements:
a decision model artifact and a variation point specification artifact. These ar-
tifacts contain a linked document based on a template for the documentation of
the respective aspect of the model. Additionally, the variation point specification
is linked to a Feature Folder in order to make explicit the traceability between
features and the architectural model that is the resolution outcome of the archi-
tecture variability model for the particular feature. See section 4.4.4 for a detailed
description of these extensions.

Describe VP bound and binding time
The derivation of a product architecture through architecture configuration re-
quires an unambiguous derivation of variants at variation points. This requires
the specification of the necessary mechanisms for implementing the variability
of the components affected by the variation point and the point in time when
the variation has to be resolved and bound. Both, the variation mechanism and
bound time, are decisions that must be taken during the design of the reference
architecture as the choice of an adequate variation mechanism and binding time
might affect the qualities of the product for which the architecture is derived.

PuLSE-DSSA does not explicitly capture this information during the execu-
tion of the architecture design process, yet the following section proposes exten-
sions to the artifacts used in the method in order to register the information
regarding binding variants for variation points.

The architect applies the techniques described in section 4.3.3 for incorpo-
rating variability in the architecture model. These techniques are applied to a
high-level design in order to model the variation points in the reference archi-
tecture. Additionally, the architecture prescribes how to design the architecture
in detail when the reference architecture is instantiated and how to implement
it with the appropriate mechanisms. Also, the reference architecture specifica-
tion describes how the mechanism should be used to realize variants and bound
variation points.

The publications by [27], [43] and [5] categorize distinct variation mechanism
with respect to quality, complexity and technological criteria. The architects can
use these categorizations as a catalog from which to select the mechanism that
suffices their needs. A categorization of mechanisms is shown in figure 4.10.

Nevertheless, the architectural knowledge associated with the selection and
application of the variation mechanism must be captured. The knowledge is
necessary for ensuring that the mechanism is applied correctly in order to derive
variants that are consistent with the reference architecture. The idea is to derive
variants that not only match the architecture configuration, but also fulfill quality
Chapter 4. Application

46

Figure 4.10: SPL Variation Mechanisms presented in [5, 27, 43]

requirements. Also, it is necessary for evolution so that the philosophy of the architecture is kept coherent when modifications are introduced, thus the core concepts are preserved [58].

The AVDF defined in the previous section also describes the variation mechanism that must be used to implement the variation point in the respective components. Additionally, the binding time of the variation point is specified so that the selected variant is bound appropriately. An example of the utilization of the AVDF is shown in figure 4.11
4.4 Case Study

4.4.1 Introduction

The preset case study analyzes a software development organization that belongs to a company devoted to the manufacture of vehicles and machinery for the farming sector. An integral part of the products is the software that enables the precise control of the mechanic components through electronic control units, commonly called “Display.” The software organization develops the DSPx software for the interactive configuration of control units to allow the setup of vehicles and peripheral devices for several applications called precision farming. There are two types of displays for which software is currently being developed, each running software for the same purpose on different hardware platforms: DSP-Low and DSP-High. Although the basic controller software is the same for both displays, the latter runs on a resource-enhanced hardware platform that provides more computational power than the former. Therefore, there is a set of common functions included in both products and a set of features that are included in the software depending on the target display platform, the vehicle model, the peripherals attached to it and the type of precision farming solution chosen by the customer.

The company’s high management has defined strategies that aim at satisfying customer demands by enhancing the software’s current functionalities and including new features in next generation of the displays. The management’s vision of the DSPx products can be summarized in three main aspects: A product that is easy to use, easy to develop and easy to support.

Architects proposed to completely change the software architecture so as to include several technological innovations that would allow the implementation of a software that addresses every aspect of the management’s vision. These innovations include switching from a windows-based to a linux-based development environment, use of the Qt application framework, leverage of COTS, use of industry standard communication protocols and support for multiple application types. The architecture would also be changed from a monolithic structure to a modular design with defined interdependencies among software components.

Once the innovations were approved and accepted the software organization started transitioning to a new development environment for the implementation of the next generation of the products based on the new software architecture. Among the challenges of this transition were the conveyance of the new architectural concepts to the architecture team and software developers. Architects worked separately on evolving the architecture incrementally in order to create detailed designs that developers would use; and developers had to work continuously on the code for including the features described in the architecture. The test-driven software development process (TDD) the organization applied could not admit bottlenecks that would delay software implementation. In fact the
architecture team had to work two springs ahead of the development team in order to produce deliverables on time. For that reason the architecture was used as the mediator for communication and coordination between the two groups of stakeholders.

Notwithstanding the intention of using the software architecture as a mediator, it didn’t fully serve purpose due to the following issues:

- It became evident that the complexity of the software posed a great challenge for the architects to maintain the architecture specification consistent and comprehensible. Frequently, architects couldn’t work independently on different parts of the architecture because they didn’t have enough information about the principal architectural decisions. Similarly, developers didn’t used the architectural specification but rather met with the architects or email them to inquire about the impact of design decisions on the code.

- The complexity that is intrinsic to the architecture of a software intensive system is complicated further by the requirement of extending the architecture to support distinct configurations of the software for different products and product generations. This issue was not addressed explicitly from the conception of the architecture, thus it is not represented in any architectural model nor documented elsewhere.

4.4.2 Approach

In the context of this thesis, the case study focused on the aspects of the architecture that enable the creation of a family of products. The research object of the case study was the architectural specification. The aim was twofold: to analyze the existing architectural documentation in order to understand the knowledge contained within it; and to apply AKinSPL to the architecting process in order to propose improvements that would enhance the architect’s ability to capture, use and share, architectural knowledge. Finally a measurement process focused on goals would be set up in order to measure whether the proposed improvements could alleviate the existing issues.

Due to time limitations, the activities of this study were concentrated upon the documentation of design decisions and design rationale that would allow for the creation of a reference architecture for the product family. The following sections describe the analysis of the existing architecture specification and the suggested improvements on it. The measurement process is described in detail in chapter 5.

4.4.3 Design decisions

The architectural specification of the DSPx software describes the quality requirements of the system as architectural scenarios. Scenarios were defined and
organized into separate packages for each relevant quality requirement. The decisions that address the scenario are placed in the same package together with the scenarios. Figure 4.12 shows the organization of scenarios and decisions made for the extensibility quality requirement.

![Figure 4.12: Scenario and Decision view for the extensibility quality requirement](image)

Although decisions address the situation described in the scenario, a more detailed analysis of the decisions, shows nevertheless they depend on other decisions to accomplish their purpose. For instance, the decision to support a robust set of application types in order to address the scenario of future applications and external development depends on the integration technology, defined in the decision to use Qt as an integration application framework. Also, the decision to use the set of UI, network components and other features provided with Qt, refine the decision to separate applications into collections of components. Figure 4.13 shows the restructuring of the same Scenario and Decision view with the application of the DSSA’s EA profile described in the previous section. In this way, the architects will be able to show the genealogy of the principal decisions in a concise diagram and also to establish traceability links to the diagrams that model the decisions in the architecture.

### 4.4.4 Architectural Variability Model

Different solutions of the precision farming product family have features that aim at supporting the driver by providing detailed information about the equipment operation and its environment of operation. These features are grouped in the documentation domain. Mapping is a feature included in DSPx products that handles geographic data and displays maps to guide driving and provide the
drivers with enhanced visibility of the terrain where the vehicle executes the operations. Table 4.14 shows how the features of the documentation domain are mapped to two models of display.

The product map clearly shows features that are to be present only in the model DSP-High and not in the DSP-Low because of limitations in the hardware platform. Although, the architectural elements that model the mapping functions were grouped in packages and encapsulated in a component, the actual features the component is capable of realizing are not explicitly represented, nor is possible to identify the internal elements of the component required to make up a particular feature.

The architectural models that represent the mapping feature must include variation points in order to indicate the parts of the model where the mapping function can accommodate different sub-features like: Layers, Background images and Labels. The diagrams that constitute the views where the variation point is represented are linked to the Mapping package in the Features folder in the EA project. See figure 4.15.

The architectural model of the mapping feature was created using UML diagrams with the PuLSE-DSSA extensions. There are three classes and one association tagged with the variant stereotype. The DM artifact documents the decision model wherein rules define effect of the resolution on the elements of the diagram. For instance, given the selection of the DSP product, the resolution determines that the mapping feature doesn’t support layers, thus the aggregation association with the IMapLayer class is removed. The rationale of the variation point is captured with the AVDF template in a VP specification artifact. The purpose of this artifact is to show that the variation point employs an extension technique to accommodate variants. Additionally, the rationale indicates that the visualization function can apply zero, one or more layers to the map thus the
design must allow for a flexible selection of layers. The different layer variants are bound to the variation point at compile time in order to comply with the resource-limited platform. See figure 4.15.

### 4.4.5 Results

The application of the method permitted modeling design decisions so as to identify associations among decisions that showed dependencies among them. Also, decisions could be modeled to show the decomposition of complex decisions into simpler fine-grained decisions that can directly be addressed by architectural scenarios. Similarly, it was possible to model the relation between the variability described in a decision model and the variability of the models in the architecture. In this manner it is possible to understand clearly the effect of decisions taken at feature level on the building block elements at architecture level.

This results were assessed qualitatively through feedback from a senior software architect at the company. In his opinion, diagrams augmented with design rationale enable a faster understanding of the purpose of the architectural models. Similarly, the prescriptions of the architecture with respect to the implementation are conveyed more easily. These results shall be further evaluated through a measurement process so as to shed light on the benefits of capturing and using architectural rationale for the organization.
Figure 4.15: Feature view and variants of display map function
Chapter 5

Evaluation

5.1 Introduction

The purpose of Goal-Driven Software Measurement is to illustrate measures and collect data that provide insights into the most important issues. It is possible to trace these measures back to the business goals, by which we can focus on what needs to be understood in the software project [44]. In this chapter we apply the Goal Question Measure (GQM) approach to identify indicators that would enable the understanding of the results of applying the AKinsPL method in the context of the case study.

5.2 Objects of measurement

The method described in previous chapters proposes a mechanism to help the organization deal with the complexity of the software architecture. The Method claims that the enhancement of the architecture specification through the documentation of the architectural knowledge will help the stakeholders to understand the rationale behind the design decisions that shape the architecture in the way it is. Therefore the object of measurement is the software architecture characterized with respect to understandability and its ability to convey information to the respective stakeholders, particularly software architects and software developers for the design and implementation of product variants.

The measurement activities will focus on the parts of the software development process where the architecture specification is used. Namely, the detailed design process and maintenance process, as the architectural knowledge is captured by the activities of the former and used in the activities of the latter.

5.3 Measurement Goals

In Goal Question Metric (GQM) approach, the identification of goals is a critical step as the information needs are derived from those goals. GQM approach assumes that to implement a successful measurement program an organization
must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals [6].

The organization's goal is to improve the software development process through the effective utilization of the architectural specification for the design and implementation of new products on the basis of the existing architectural design. There are two aspects of the previous statement with respect to the software development activities wherein architecture knowledge is captured and used, and the stakeholders whom make use of it. Thus, two measurement subgoals are conceived: to enhance the ability of the architect to capture architectural knowledge in the architectural specification document; to allow developers make effective use of the architecture specification document. In the context of product line engineering, the general goal is to create a complete reference architecture with detailed information regarding the decisions that enable the realization of product architectures and to allow developers derive a product architecture that is consistent with the reference architecture.

5.4 Measurement questions

Once the goals are identified, the next step is to formulate questions that will define the goals in terms of a set of quality issues and a quality model of the product that deals with those issues. Questions regarding Goal 01 and Goal 02 are illustrated in figure 5.1 and figure 5.2 respectively.

![Figure 5.1: Questions and Metrics for Goal 01](image)

The questions that define Goal 1 characterize the architecture specification with respect to the design rationale embedded in it and the ability to capture this knowledge with the tools the architects use to create architectural models and the artifacts they produce.
Figure 5.2: Questions and Metrics for Goal 02

The questions that define Goal 2 characterize the architecture specification with respect to knowledge embedded in it and the ability of developers to employ this knowledge effectively.

5.4.1 Measurement Metrics

The next step in GQM consists in specifying the measures that need to be collected in order to answer the questions. The metrics define methods for measuring the characteristics of the architectural specification with respect to the issues the questions indicate. Figures 5.3 and 5.5 show the specification of metrics for the questions described in the previous section. The definition of metrics include the purpose of using the metric, the scale type and the method of applying the metric so that the appropriate data is used to answer the respective question.

5.5 Measurement process

The measurement process will be planned as stated in the ISO/IEC 15939 Software Measurement Process standard to define the information needs and related measurement infrastructure. The purpose of the measurement plan is to describe the what, who, where, when, how, and why of metrics [6]. That is, what data to collect, who is responsible for collecting those data, how often and where those information is collected, how the data is related to management decisions and how they should be interpreted, and what is the purpose of those metrics.

5.5.1 Measurement scope, stakeholders and roles

This measurement plan is prepared for the same software organization within the scope of the case study described in this project. Thus, the goal of the measurement process is to establish activities for collecting data for a given set
of metrics that characterize the impact of the Method on the development and usage of the architectural specification.

The two main groups of stakeholders are the software architects, who create the architectural specification, and the developers, who use the architecture specification documents to create, change, and update the software system. The influence of both groups in the measurement process is not significant because the measurement falls within the scope of case study.

For the reason described before, the roles of measurement analyst and measurement user fall to the person conducting the case study.

### 5.5.2 Organization characteristics and Information needs

As described before, the company offers software products for the operation of the vehicles and machinery that constitute their primary product. The software is developed in-house by a software development organization that is divided into two groups geographically separated. Development teams at each location apply an agile approach to distribute units of work in terms of user stories. The allocation of modules to development teams is based on the implementation view of the architectural specification.

The software development process is initiated by requirements originated from the strategy director which are mapped to existing features described in the soft-
### Chapter 5. Evaluation

#### Figure 5.4: Metrics specification for Goal 02

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Purpose</th>
<th>Scale Type</th>
<th>Method of Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsystem specification/implementation difference</td>
<td>This metric indicates the difference among the number of modules specified in the architecture and those implemented in the source code</td>
<td>To measure the degree of completeness of the architecture specification for a given subsystem.</td>
<td>Ordinal</td>
<td>Subtract the number of modules in the specification of a subsystem from the number of modules in the implementation of the same subsystem. Use the absolute value of the result.</td>
</tr>
<tr>
<td>Module specification/implementation difference</td>
<td>This metric indicates the difference among the number of classes specified in the architecture and those implemented in the source code</td>
<td>To identify the modules that are poorly designed and require additional implementation details.</td>
<td>Ordinal</td>
<td>Subtract the number of modules in the specification of a subsystem from the number of modules in the implementation of the same subsystem. Use the absolute value of the result.</td>
</tr>
<tr>
<td>Rate of interface discrepancy</td>
<td>This metric indicates the consistency level of an interface implementation with respect to its specification</td>
<td>To identify the interfaces that need redesign</td>
<td>Ordinal</td>
<td>For each interface definition count the number of public methods whose signature does not match the architecture specification.</td>
</tr>
<tr>
<td>Number of design rule violations p/product</td>
<td>This metric indicates the degree of deviation of a product's architecture with respect to the reference architecture</td>
<td>To identify the products with the highest degree of design rule violations.</td>
<td>Ordinal</td>
<td>Count the number of design rule violations per product</td>
</tr>
<tr>
<td>Rating of violated design rules</td>
<td>This metric indicates the number of times a design rule was broken.</td>
<td>To identify the reasons for the violation of a design rule.</td>
<td>Nominal</td>
<td>Categorize the reasons for breaking a design rule and count the number of times a design rule was broken because of that reason.</td>
</tr>
</tbody>
</table>
ware architecture or it is updated to cover the requirements. Development teams would use the software architecture to understand the scope of new requirements, include the requirements in the product backlog, estimate the implementation effort and define sprint backlog.

The information needs were identified applying the GQM approach. A set of metrics characterize the attributes of the information needs in relation to the organization’s goals. See section 5.3. The following template will be use for the documentation of the information needs:

<table>
<thead>
<tr>
<th>Information Request Form</th>
<th>IRF # 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originated On:</td>
<td>Originated By:</td>
</tr>
<tr>
<td>Requested Due:</td>
<td>[ ] Manager</td>
</tr>
<tr>
<td>Related Project:</td>
<td>[ ] Customer</td>
</tr>
<tr>
<td>Description:</td>
<td>The architects need to know whether the design rules defined in the architectural specification are being followed when products instantiate the reference architecture for a product.</td>
</tr>
<tr>
<td>Reporting Format:</td>
<td>For each product, the architects should be able to visualize the design rules and the number of times it was not followed in the product implementation. The products and the rules with the highest values should be highlighted by the chart.</td>
</tr>
<tr>
<td>Frequency of reporting:</td>
<td>Every time a new product is derived from the reference architecture.</td>
</tr>
<tr>
<td>Measurement Rationale:</td>
<td>The requested information product will help to understand the causes of architecture misuse. Namely, the reasons that developers decide not to follow a design rule.</td>
</tr>
<tr>
<td>Decision Criteria:</td>
<td>The goal of allowing developers to use the architectural specification in a more efficient manner will be achieved when the value of this indicator is 0. That is, no design rules are broken.</td>
</tr>
</tbody>
</table>

*Figure 5.5: Documentation template for information needs*

### 5.5.3 Data collection, storage and reporting

The data required for the metrics is collected from three sources: The diagrams created with Enterprise Architect to document the architectural models and any form of textual documentation in relation to the architecture; the source code that realizes modules and the organization of modules in the file system; subjective information surveyed from architects and developers.

The collection of data from the architecture specification document, will be mainly an inspection activity guided by a checklist. A different checklist will be crafted to seek the data required for different metrics. Also, the EA enables exporting diagrams to an machine-readable representation reusing XMI files. This files can be processed with XSLT scripts to retrieve the required information. For instance, to retrieve the signatures of interface definitions and the number of classes implementing that interface.

The measures taken from the source code will be automated with the use of tools for static analysis. Such tool can be easily created with the CDT framework of Eclipse as most of the measurement operations require simple arithmetic rules. For instance, counting the number of classes in a given module and the number of design rule violations.

The Mercurial revision control system will be queried for metrics that require data from the same artifacts at different points in time.
Chapter 5. Evaluation

Once the measurement has been performed, both the metrics and the corresponding data will be recorded with Mercurial in order to track changes in measurements at different points in time.

Finally, the metrics will be compared against indicators so that it is possible to assert how far or close is goal achievement. The measurement process shall be applied before and after the introduction of the method so that it is possible to show whether there is a significant improvement in the way the architecture specification is used when it has been augmented with design rationale.
Chapter 6

Summary and future work

6.1 Summary and contributions

This thesis discusses architectural knowledge of software product lines and proposes mechanisms to capture it. The first chapter is an introduction to the role software architecture plays for the success of a product line and the importance of architectural knowledge for the effective usage of the architecture throughout the life cycle of a product line.

The second chapter describes the process and presents the result of applying desk research in order to obtain the factors of software product line architecture that are most relevant in industry. It suggests a categorization of architectural concerns based on a literature survey of nineteen case studies which give account of the design and implementation of a software product line in different domains.

On the basis of those premises we developed the AKinSPL method for capturing architectural knowledge, which is presented in the third chapter. The method is based on a meta-model that represents the mental models and processes architects follow during the creation of a product line architecture. It defines guidelines for capturing the knowledge implicitly present in the general activities of architectural design and proposes additional activities wherein the knowledge pertaining to design decisions in favor of variability is also captured.

In chapter four, the method’s guidelines were mapped to concrete activities of the PuLSE-DSSA method. Mapping the method required the design of new artifacts for the specific purpose of capturing architectural rationale. The artifacts contribute to the architecture documentation schema defined in PuLSE-DSSA. These artifacts were implemented as extensions for Enterprise Architect to be used as a documentation profile for projects applying the PuLSE-DSSA method.

In order to illustrate the application of the AKinSPL method, PuLSE-DSSA and the EA extensions were used to document the architecture specification of automotive software for the configuration of a family of electronic control units. The application of the method permitted modeling design decisions so as to decompose business-level decisions into fine-grained technical decisions and to identify dependencies among them. Similarly, the method permitted to model the relation between the variability described in a feature model and the architecture model.
Chapter 6. Summary and future work

Finally, chapter five proposes a measurement process for evaluating the application of the AKinSPL method in the context of a running project. The measurement process applies the GQM approach to define the questions and appropriate metrics that would help determine whether the use of the architecture in software development activities is more effective when the architectural specification is enhanced with architectural knowledge.

Notably, the main contributions of this thesis are described in the following paragraphs:

- A categorization of the different factors that architects observe when creating architectural designs for a software product line. Collectively, these are the factors that influence the decisions architects make in order to design an adequate solution.

- A conceptual method that defines guidelines for capturing the knowledge associated with the design of a software product line architecture. The method aims at capturing the rationale behind the design decisions that address the factors specific to product line engineering.

- The development of set of documentation artifacts and guidelines for capturing architectural knowledge, including documentation templates and a project profile for Enterprise Architect.

6.2 Future work

In this thesis, the AKinSPL method has been applied to capture the architectural knowledge of an embedded software for automatic control system for agricultural equipment. Although the results received positive feedback from a senior architect at the development organization, the method was only applied to a small toy example and did not cover the architecture thoroughly. A future work will be the application of the method in the context of a running project where we want to conduct a measurement process to validate the results as proposed in chapter 5.

The ability of viewing associations between an architectural scenario, the decisions that address it and the architectural design that models the solution, opens the possibility of managing these links automatically. It could be possible to assess beforehand the impact of new or updated decisions on the existing architectural model. This would require the definition of certain metric that measures the extend of the ripple effect of a single change throughout the architecture. The metric would show the level of complexity introduced by a single decision.
Appendix A

Data extracted from case studies
1. Control Channel Toolkit

### Domain
- Aerospace: Ground-based spacecraft Command and Control system
- Domain Engineering: Create an Asset Base

### Long-term savings by reducing the production cost of several spacecraft Command and Control systems by sharing the CCT assets among different organizations for current and future projects.

### Business Goals
- Monitoring and Command capabilities
  1. Batch and online processing of spacecraft and ground system commands
  2. Event response simulation
  3. Mission planning in off-line processing
  4. Conversion of data from front-end processing equipment into client-visible format
  5. Rule-based automation of client requests
  6. Archive log of telemetry and command data
- GUI Capabilities
  1. Broker capabilities to distribute data to client processes attached to the user interface
  2. Simulation of state and behavior
  3. CC model on-board processor instructions and data loading

### Key Functional Requirements
- Performance
  - Real-time communications between ground control and spacecraft
  - Execution of processing
  - Efficient real-time processing
  - Availability
  - CC retrieves data more than 1 from front-end processing equipment
  - Modifiability
  - Extensibility and configurability
  - Testability

### Quality Attributes
- Incorporate legacy spacecraft assets
- Reference Architecture compliance
- Limited communication between CC and spacecrafts

### Constraints
- DD1: The separation of the system into two subsystems to deal with time-critical and near-time-critical operations. Addresses: QA, KFR.
- DD2: The application of a non-blocking interface between the execution and planning subsystems. Influenced by DD1. Addresses: QA, KFR.
- DD3: Modularization of functionality into components and organization of components into component categories wherein related components might be integrated together to achieve higher-order functionality. Components that belong to one category can interact with other categories’ components through well-defined ways. Influenced by D2. Addresses: QA4.
- DD4: Component composition. Many system functions would be accomplished through the operating and interaction of components across different categories. Influenced by DD3. Addresses: QA4.
- DD5: Facilities common to all components are provided as services that should be intrinsic to the middleware platform. Influenced by: BG2.
  - Addresses: KF6.
- DD6: Replacement of components at architectural level is performed through the component Broker architecture. New or legacy components are included at architectural level by wrapping the components’ interface with IDL. Influenced by: BG3.
  - Addresses: QA4, C1.
- DOR 7: All component categories contain variation points to enable variation to occur. Influenced by BG2.

### Design Decisions
- Modifiability
  - QA against Legacy spacecraft assets. All legacy code had to be reengineered, reviewed, and finally wrapped to meet interface specifications dictated by the Architecture. Addressed by DD6.
- Modifiability against compliance with Reference Architecture

### Architecture Rationale
- In addition to the documentation of the 4+1 views of the reference architecture, a reuse plan provides detailed information about how to perform the instantiation of the reference architecture.


References


References


[34] Elsevier Inc. Engineering village, 2011.


