On the Role of Architectural Design Decisions in Software Product Line Engineering

Rafael Capilla [1], Muhammad Ali Babar[2]

[1]Universidad Rey Juan Carlos, Spain [2]LERO, UL, Ireland
[1]rafael.capilla@urjc.es, [2]malibaba@lero.ie

Abstract. An increased attention to documenting architectural design decisions and their rationale has resulted in several approaches and prototype tools for capturing and managing architectural knowledge. However, most of them are focused on architecting single products and little attention has been paid to include design decisions in the context of product line architectures. This paper studies two similar research tools that capture architecture design decisions and how these can be extended to include product line specific features. We report the improvements needed by the data models of the two tools and we provide a unified data model as a research agenda to support both the relationships between design decisions and variability models in a product line context.

1. Introduction

Early in the nineties, Perry and Wolf emphasized the importance of Design Rationale (DR) in Software Architecture (SA) [22]. To date, the traditional approaches to documenting software architectures have been mainly based on the description of architecture views that reflect the interest of different stakeholders [10] [18] [23], but little attention has been paid to capturing and managing the rationale for key design decisions. The need to reduce the maintenance effort and to avoid architecture erosion because decisions are never recorded requires them to be captured.

Many claims have been made about the problems caused when design decisions are not explicitly documented [27], as they constitute a clear way to mitigate the effort required in understanding the architecture of a system when the experts or the creators of the architecture are no longer available. Bosch pinpoints [4] that "*we do not view a software architecture as a set of components and connectors, but rather as the composition of a set of architectural design decisions*". This idea, also stated in [10], claims for methods and techniques to enable the representation and capture of architectural design decisions in parallel with their architectures. In order to bridge the traditional gap between requirements and designs, a "new" architecture view so-called the "decision view", is proposed in [11], which considers design decisions as a cross-cutting information with respect to the other traditional architecture views that have to be documented explicitly.

Other research works rely on the definition of specific templates for capturing and representing the knowledge that should be part of the description of a design decision. Some authors like [19] [30] propose extensive list of attributes for characterizing he design decisions whiles other [8] advocate the use of more flexible approaches based on list of mandatory and optional attributes that can be tailored for different organizations and user’s needs in order to reduce the effort spent during the capturing activity. The authors in [19] consider that Architectural Knowledge (AK) = Design Decisions + Design, and they use ontologies to describe both the decisions and the relationships between them. Since design decisions can bridge the gap between requirements and architectures and code, recording these relationships [31] can benefit maintenance and evolution processes, and help to understand the root causes of changes or to estimate change impact analysis. To date, only a few works have partially addressed using design decisions in a product line environment, and from our view more work is required to support the specificity of product line development.

The remainder of this work is structured as follows. Section 2 describes those features specific to product line development as well as some approaches that try to introduce design decisions in product line practices. Section 3 discusses our approach to relate architecture design decisions to product line features. Section 4 outlines two similar prototype tools for supporting and managing design decisions and how these could support those issues attained to product lines. Section 5 describes a unified data model as a future research agenda for a new tool aimed to capture, manage, and represent design decisions both for single architectures and product lines. Section 6 provides some related work, and in section 7 we provide the main conclusions.

2. Product Line Engineering Features

Software Product Line Engineering (SPLE) has emerged as a successful mode of developing software. SPLE aims to create a family of related products based on a single architecture that can be tailored to meet the requirements of different products. This is achieved through the identification of commonalities and variations among the different systems of a given family which are represented in the product line architecture (PLA). The required variability can be realized by means of different variability realization techniques, like those reported in [26]. A PLA has several unique features that allow the creation of multiple products by means of derivation techniques [5]. In this section we discuss some of key characteristics that make product lines different from other software architecture practices. There some other issues not addressed here but we will focus only on those closely related to the architecture development practice.

- **Variability modelling**: Product lines rely on the description of a set of common and variable characteristics of systems, and variability modelling deals with the representation of the common and variable aspects through a set of interrelated variation points and variants. Variability modelling is a challenge activity that needs suitable tooling support for managing the hundreds of variation points in complex systems. A major weaknesses of one of the most widely used modelling languages like UML, is that all variability concerns are difficult to be represented in a UML model. For instance, the relationships and constraints between variation points and variants that can be represented in a FODA tree [17] are hardly to be described in a UML diagram. Only the OCL (Object Constraint Language) combined with UML provides better support. In practice, a variability model acts as a decision model in which the variation points and their variants have to be selected and instantiated for configuring a particular product.

- **Binding time**: This mechanism is used to delay design decisions as late as possible. The realization of the variability can also be achieved through different binding times, such as; compilation, integration, deployment, or runtime. Different techniques can be used to resolve the binding time [14] of a particular software product. In general, binding times are not described in UML architecture models and they are documented separately from architectural designs.
- **Variability dependencies and product constraints**: Variation points and variants may depend on other variation points. Therefore, the selection of a certain variation point or variant may depend on a previous selection. These dependencies are usually represented in the variability model [16] [21]. Dependency models have a great impact on traceability as they provide viable paths to traverse from feature models to products and vice-versa. Frequently, dependencies are used to delimit the scope and the number of products during product configuration and mostly driven by economic and business factors. For instance, more complex rules like *if-then-else* conditions can be defined using a logic formula to specify product constraints. Such rules often crosscut variability models. Because product lines are very market-drive, the domain scoping activity aims to define the number and type of product to deliver through specific product constraints. Hence, when making design decisions to produce a particular product, all the constraints should be resolved to avoid the selection of incompatible variation points that may lead to incompatible products.
- **Common and specific requirements**: Traditional software development use different types of requirements to motivate the design decisions made in the design phase. In product lines, a subset of these requirements are common to the entire product line whilst product specific requirements motivate the decisions to deliver each single product through the variation points defined in the modelling phase.

2.1 Reasoning models in product lines approaches

The majority of the approaches that try to capture design decisions alongside architectures don’t consider the specificities of product lines. Only some few proposals attempt to do this and they try to introducing reasoning in variability models. For instance, in [4] the authors describe a tool for supporting the analysis of feature models using an automatic reasoning mechanism to deal with extra functional features, like for instance quality aspects. This reasoning mechanism can be used to ask questions, such as: which is the number of potential products in a given feature model? Dhungana et al. [12] state the difficulty to transfer general architectural knowledge from tacit to an explicit form understandable by the users, and map this concept to product line variability models in which features have to be linked to architectural artifacts. More specifically, Dhungana et al. [13] perceive decisions as variation points for asset composition. These decisions are organized hierarchically and they become relevant if they are made in a certain order. The consequences of a decisions are expressed a logical dependencies. Decisions are represented in a decision model which only describes the need, the scope, and the constraints, but the same as in [4] no additional information about the rationale or the impact of the decision is supported. In [23], the authors address how to represent and document design decisions in product lines that follow a compositional approach to derive the final products. This composition is supported by the AHEAD tool suite [3], where product line features are used as building blocks of systems. In [23], decisions are documented as XML artifacts during the synthesis of the architecture. In their approach, text descriptions for understanding the decisions are included with product line assets. Finally, none of the tools analyzed in [9] for modeling and managing product line variability models don't consider the description of architectural design decisions as first class entities and in most cases, they only focus on how to deal with feature models or in product derivation tasks.

3. Design Decisions for Product Line Architectures

Because almost all the approaches described in section 2 suffer the lack to incorporate explicit description for the decisions and its underlying reasons that accomplish the selection of variation points and variants in a product line, this section presents attempts to link both concepts as a way to include the rationale of the architectural decisions that affect the selection and the realization of variability models. Hence, we propose to associate the concept of design decision with variability models to enrich the reasoning activity that take place in SPLE practices, and we come up with a general model that can be instantiated for building tool support.

- **Design decisions and variability modeling**: A variability model constitutes a decision model, in which variation points and variants have to be resolved to achieve a specific product configuration. From our view, the definition of these variations should be considered fine grained decisions (as opposed to coarse grained decisions based on design patterns and architectural styles). Different levels of granularity in the decisions can be considered, but from an architecture point of view, the main decisions are precisely made in the early stages of the design phase whereas variation points are introduced later to refine the classes of a product line architecture. In our approach we propose a mapping between the design decisions that affect the definition or the selection of a variation point and variant in order to explain the why of the reasons by which we arrived to a particular feature model.

- **Binding time**: The binding time should be understood in the traditional use of product lines, but the selection of a particular binding time should be also explained as a design decision. Hence, we propose that the binding time of a particular variation point has associated a decision that explains the realization of such variation point in the architecture. This binding time should not be confused with the time in which the decision is made, as all of them are defined at design time. We assume that the binding time of a set of related variation point should happen at the same moment. Otherwise, the realization of the variability will not be feasible.
- **Product constraints**: In the same way we connect different decisions with specific links, we need to specify the links between variation points. Such links are usually defined using logical operators such as AND, OR, XOR, and NONE. More complex formulas can be also defined to support crosscutting relationships to the feature model or to include functional dependencies that usually happen during the execution of the system. These rules that delimit the scope of the products and may affect to previous decisions. The same as described before, a new design decision should document each product constraint, and the dependencies between decisions can be used to define the links between a product constraint decisions to those previous and related decisions made before. Therefore, we could be able to know which decisions related to the definition of variation points depend from decisions that define a product constraint.
- **Common and specific requirements**: In order to discriminate between requirements for a PLA and for a single product, we introduce a new attribute, to discriminate between common and specific requirements.

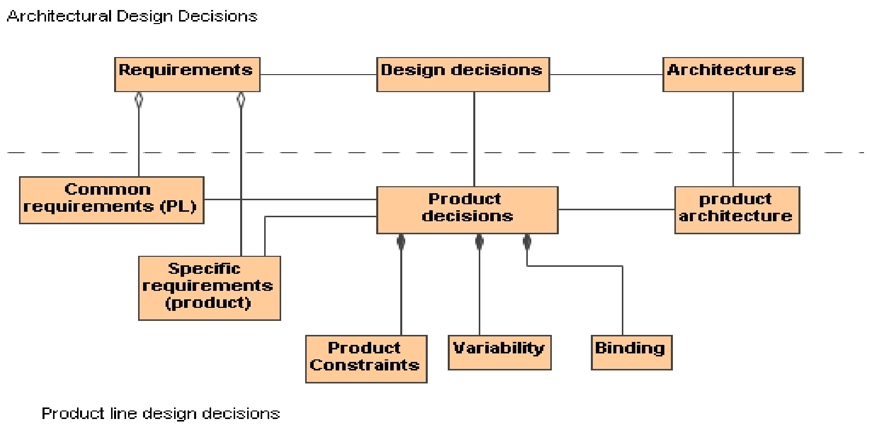

Figure 1: A general model that maps the decisions to requirements and architectures and its corresponding relationships for product line architectures

The top side of Figure 1 show that design decisions are used to bridge the gap between requirements and architectures as requirements motivated the decisions that will produce a particular architecture description. The bottom side of the figure adds the additional entities that belong to product line development. Common and specific product requirements motivate now the decisions that lead to a concrete product architecture. These decisions are also design decisions but enhanced with some distinct features like decisions associates to: product constraints, variability models, and binding time. Similar traces between both sides of the figure provide a complete traceability among the entities. Our improved in this general model is to map significant decisions to the product line variability model which have to be documented explicitly. This general model can be instantiated for building specific tool support.

4. Tool Support for Capturing Architectural Design Decisions

Almost none of the existing tools that manage variability models for product lines offer capabilities to record and document the design decisions and the reasons that led to the selection of any particular product line architecture or to a concrete variability model. In addition, the current research prototype tools that capture and document architectural design decisions (i.e.: Archium [15], AREL [28], ADDSS [7], and PAKME [2]) don't offer support for product line features neither for variability models. Hence, in order to cope with this lack, we have analyzed the similarities and the differences of the data models of two of the tools specifically developed to capture architectural design decisions in order to extend its capabilities for product line engineering. Because of their similarities, we have chosen PAKME and ADDSS as both are web-based tools with similar nature and purpose. In this section we provide a description of both tools and to which degree they are able to include or not product line features.

4.1 Product-line support in ADDSS

ADDSS (Architecture Design Decision Support System [7] is a web-based tool for managing architectural design decisions (http://triana.escet.urjc.es/ADDSS). The tool supports an iterative process in which design decisions are captured along with their rationale and models. ADDSS supports basic dependencies between decisions and traceability between requirements, design decisions, and architectures. The chain of dependencies between decisions is documented explicitly for traceability purposes.

We have analyzed the how the current implementation status of ADDSS could support the features described in section 3. With respect to the inclusion of variation points, variants, and their relationships, the ADDSS data model can be easily extended to store such information and relate this with logical operators (such as AND, OR, XOR) to support product constraints and relationships between the variation points. In addition, ADDSS could associate each single decision to single variation points and variants. Hence, the variability associated to the overall set of decisions will constitute the entire variability model. Relating parts of the variability model to a subset of the architecture is not possible because the current version of

ADDSS cannot relate a set of design decisions to individual architectural parts. A complementary issue that should be addressed is to check the inconsistencies in the variability model to avoid incompatible product configurations. This should be implemented in order to ensure the integrity of the decision model for detecting violations in the decisions when these are added, changed, or removed, but at present this feature is not yet supported. Also, including the binding time as an entity or attribute into the ADDSS data model should not be a problem, and the same stands for discriminating between common and specific requirements that are selected during the reasoning activity in the architecting activity. Finally, we can also improve the documentation generated by the tool if we include the information belonging to the variability model of all product architectures.

4.2 Product-line support in PAKME

Process-centric Architecture Knowledge Management Environment, PAKME, is a web-based tool to support software architecture design, documentation and evaluation activities [2]. PAKME provides a knowledge repository, templates and features to capture, manage, and present architectural knowledge and design rationale. PAKME's knowledge repository is logically divided into two types of knowledge:

- **Generic:** such as general scenarios, quality attributes, design options
- **Project specific**: such as concrete scenarios, contextualized patterns, quality factors, architecture design decisions and rationale underpinning them.

Project-specific AK consists of the artefacts either instantiated from the generic knowledge or newly created during the software architecture process. Access to a repository of generic AK enables designers to use accumulated "wisdom" from different projects when devising or evaluating architecture decisions for projects in the same or similar domains. The project specific part of the repository captures and consolidates other AK artefacts and rationale such as concrete scenarios, design history, and findings of architecture evaluation. A project specific AK repository is also populated with knowledge drawn from an organisational repository, standard work products of the design process, logs of the deliberations and histories of documentation to build organisation's architecture design memory.

Though, PAKME has initially been developed to support the architecting activities for a single product, many of its artifacts (such as general scenarios, design options, and analysis models) can be used to support the activities for designing and evaluating PLAs. However, there needs to be certain changes required in the data model and interface for establishing and maintaining explicit relationships between different artifacts of a product specific architecture and PLAs. PAKME's data model also needs to be modified to accommodate the requirements of the solution proposed in Figure 1. Such changes can easily be accommodated as PAKME has successfully tailored to a specific domain and the experience showed such modifications are easy.

5. Towards a Unified Model to Support Design Rationale in Product Line Architectures

In order to compare PAKME and ADDSS data models, we used an illustrated example from the Intrada Product Family [24] to know how the variability, product constraints, and common and specific product requirements can be supported in both tools. After the case study, we analyzed which entities of both data models perform the same function and which others are completely different or are not supported in one of the tool. As a result, we arrive to Table 1 (see Appendix 1) which shows those entities and concepts that are similar or equal in the tools as well as those not supported by one of them (in Table 1 the rows in blue are entities from PAKME not supported in ADDSS while in light brown one feature of ADDSS not supported by PAKME).

5.1 A unified data model for architectural design decisions

As a result of the comparison, we distilled a minimum number of entities required to support architectural knowledge in both tools and leaving all other complementary aspects like for instance those related to quality attribute architectural evaluation supported by PAKME. The entities we agreed to support as our core data model are described in Table 2 (see Appendix 2) and depicted in the UML class diagram in Figure 2. For the sake of simplicity, we tried to keep simple and small the unified model of Figure 2, and include only those entities we perceived are necessary for supporting the decision-making process in software architecture.

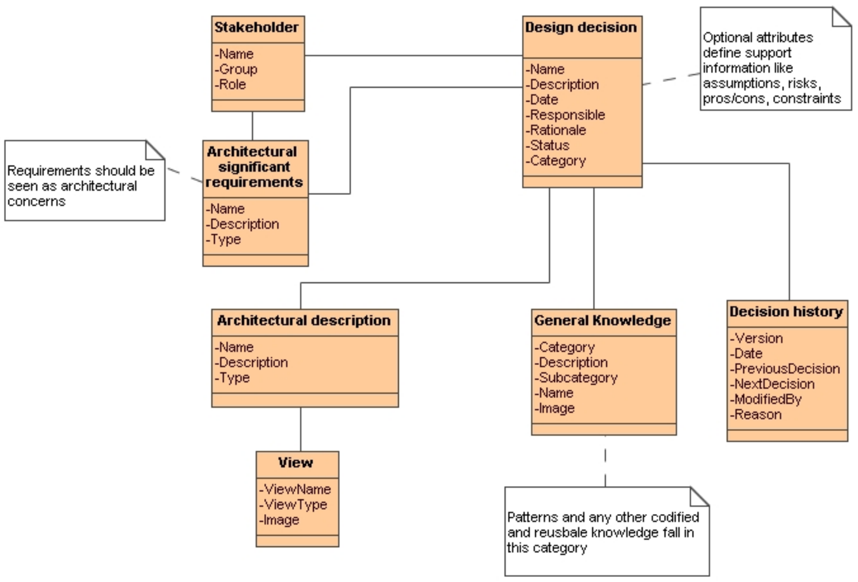

Figure 2: Unified data model for supporting the core entities that support architectural knowledge

In the figure, design decisions are related to architectures, requirements, and to the stakeholders that make the decision. Architecture descriptions are represented in terms of views, while design decisions comprise reusable chunks of knowledge like patterns and style. We have to remark that the “rationale” of a decision is defined as an attribute in the proposed model, but from a higher level perspective, rationale is considered an entity in the ANSI/IEEE 1471-2000, currently under review. In addition, to the evolution of the decisions is already supported by a specific entity which records all the modifications and changes performed over a particular decision. The attributes included in the entities of the model can be extended with new ones to add more functionality. Therefore, keeping the model small, it is easier to extend this with those features specific to product lines and after if needed, add all those additional entities and attributes required to support extra functionality like that one included in the tools analyzed. Next section outlines the product line features added to the core data model of Figure 2.

5.2 Extended data model to support product line features

The need stated at the beginning of this work to provide support for capturing design decisions belonging to software product lines is realized by extending the model of Figure 2 with the features described in section 3. We have refined the description given by Figure 1 to add the entities and attributes we perceived necessary to link architectural design decisions to the product line variability model. The result of this customization is shown in Figure 3.

In the figure, we have added three different entities as well as some attributes. One new entity captures the information representing the variations points, including a constraint rule that defines the logical relationship between the variants. In addition, a *category* attribute is used to perform a classification of different variation points that can be filtered for visualization purposes. This *variation point* decision is attached to the architectural design decision which explains and motivates the definition of a particular variation point.

Similarly, we did the same for the variants defined in the variability model. This fine grained decision has also its corresponding architectural design decision which justifies the selection of particular variant in the architecture. A new *binding time decision* class was also added to indicate the binding time of a particular variation point or variant, which is related to the architectural design decision that indicates why a binding time has been chosen but also to the variant or variation point affected by such binding time. Therefore, we can obtain an extremely fine granularity to define different binding times for different variants and variation points. Also, the *design decision* class has a new attribute, *ProductDecision*, which defines if a design decisions concerns to a product line decisions (“yes”) or to a single architecture (“no”).

Other two entities provide attributes for supporting additional product line features. The *architectural significant requirement* class uses an attribute (“ProductSpecific”) to discriminate between common requirements for an entire product line or single architecture, or if it belongs to a product specific requirement. Finally, the *architectural description* entity uses a new attribute to indicate if the architecture belongs to a concrete product architecture (this will imply that the variability model has been resolved and the decisions are made).

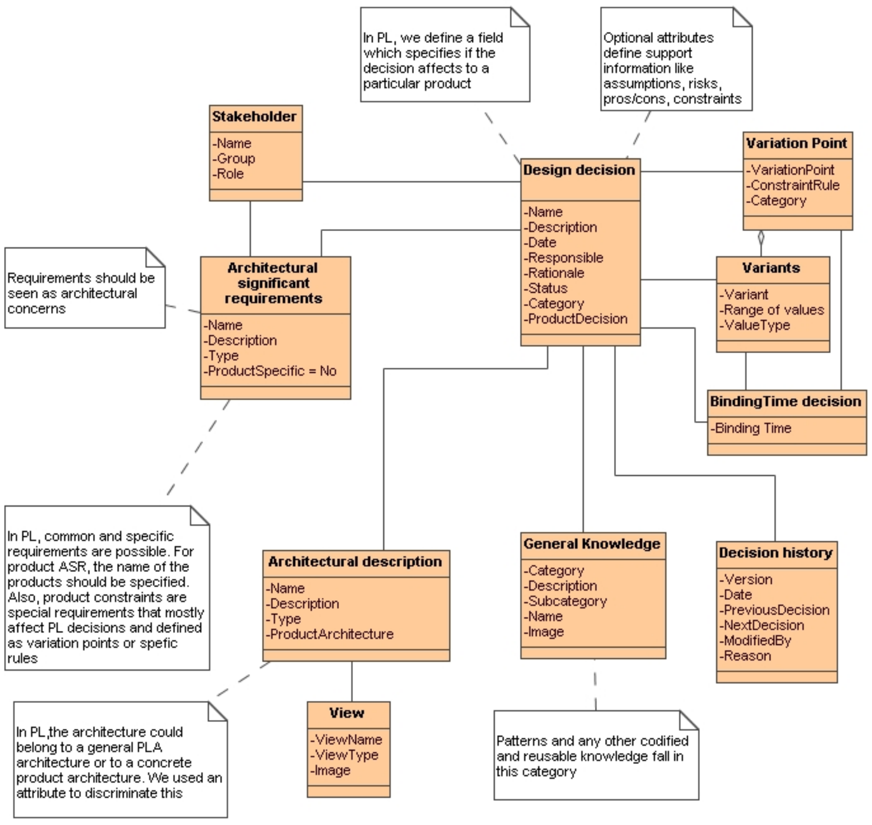
In PL, we define a field which specifies if the decision affects to a particular product
Optional attributes define support information like assumptions, risks, pros/cons, constraints
Stakeholder
-Name
-Group
-Role
Design decision
-Name
-Description
-Date
-Responsible
-Rationale
-Status
-Category
-ProductDecision
Variation Point
-VariationPoint
-ConstraintRule
-Category
Requirements should be seen as architectural concerns
Architectural significant requirements
-Name
-Description
-Type
-ProductSpecific = No
Variants
-Variant
-Range of values
-ValueType
BindingTime decision
-Binding Time
In PL, common and specific requirements are possible. For product ASR, the name of the products should be specified. Also, product constraints are special requirements that mostly affect PL decisions and defined as variation points or spefic rules
Architectural description
-Name
-Description
-Type
-ProductArchitecture
General Knowledge
-Category
-Description
-Subcategory
-Name
-Image
Decision history
-Version
-Date
-PreviousDecision
-NextDecision
-ModifiedBy
-Reason
In PL,the architecture could belong to a general PLA architecture or to a concrete product architecture. We used an attribute to discriminate this
View
-ViewName
-ViewType
-Image
Patterns and any other codified and reusable knowledge fall in this category

Figure 3: Extended data model to incorporate product line feature to an architectural design decisions model

The approach described in Figure 3 doesn’t consider multi-product lines (a product line composed by single product lines) and we didn’t represent the dependencies between design decisions as these are supported in PAKME and ADDSS as intermediate classes that relate one decision to others. The same will apply to define relationships between different variation points, but including this stuff will complicate unnecessarily the description of Figure 3.

5.3 Impact of product line features in the reasoning activity

In this section we analyze the impact product line features may have in the reasoning process. In addition to the information aimed to support product line decisions, the process by which a variation point or variant is selected or configured, has some influence on the steps of the reasoning activity. For instance, typical architectural design decisions are selected after an evaluation of the best or optimal choices among several, but the instantiation of a variation point only depends of the selection of their variants and values. Hence, no alternative design decisions are evaluated or stored for this case. Otherwise, the selection of a particular binding time may imply to consider and evaluate different binding time alternatives, which might be not the same for different architecture subsystems.

In addition, current architectural design decision tools like PAKME and ADDSS must capture new relationships that are now established between the variability model and the decisions that explain such variability model. To make more agile the capturing process, these relationships should be defined internally by the tools to alleviate the effort spent by the user in defining new links. Only in those cases where a relationship between variation points and variants has to be defined, the user must reason about such dependency and make it explicit when capturing the decision. We can reduce also such effort if the dependency that models a relationship in the variability model serves also to model the dependency between the decisions attached to variation points and variants. Hence, product lines slightly modified the way in which dependencies are modeled respect to single architectures, as the architect would define the relationship in the variability model and the tool implicitly uses this to internally define the relationships between its associated decisions.

6. Related Work

Recently, many researchers have emphasized the importance of treating architectural design decisions as first class entity, however, only a few have mentioned the use of design decisions in SPLE. In [1], the authors present a framework for representing design decisions in a SPL, which is structured as a design decision tree (DDT) where nodes represent design decisions and branches relate nodes to each other. Some of these nodes represent the variations in a product family. Lago and Vliet [15] show how to introduce assumptions in UML models representing architectures and variability concerns in SPLE. They present assumptions with feature models to show the influence of the assumptions on the features used to model variability. The work reported in [21] focuses on the reuse of design decisions in order to customize product line using composition techniques as a step-wise refinement for product derivation. Design decisions are captured in XML files that can be reused during the transformations needed to obtain a final product. Extensions to products and their design decisions can easily be traced by viewing the ways decisions extend architectures during through successive refinements. The COVAMOF model for managing variability described in [20] is used to support the notion of architectural design decisions in a SPL. The authors map architectural concepts (including decisions) to COVAMOF concepts to demonstrate the feasibility of capturing architectural knowledge and link this to variability models by mapping similar concepts.

7. Conclusions

This paper presents the continuation of our efforts in building two similar tools for capturing and documenting architectural design decisions. Because there is a lack of specific support for product line architecture decisions, we have merged the data models of both tools and we have distilled the common model to support the decisions made in a product line context in order to explain the decisions made in variability models. Thus thriving research area provides the necessary infrastructure for systematically and rigorously incorporating the notion of design decisions and their rationale in designing and maintaining PLAs. Our work identifies the characteristics of architecture design decisions in the context of SPLE.

From the common model distilled from both tools, we have observed that is no so difficult to incorporate the decisions made in a variability model to support the specificities of product lines. However, accommodating this new information may result a bigger number of medium-size or fine grained decisions. From our perspective, we believe that the same dependencies defined for the architectural design decisions can be used to define the relationships in the variability model in order to not duplicate dependency links.

Additionally, supporting common and specific requirements is also necessary to distinguish those decisions that are specific to a single product. For future work we are planning to start the construction of a new web-based tool or to extend ADDSS in order to incorporate these new features in a new data model and test the new capabilities in a product line environment.

References

1. Alonso A, León, G. and Dueñas J.C. Framework for Documenting Design Decisions in product Families Development, ICECSS IEEE CS, 206-211, (1997).
2. Babar, M. A. and Gorton, I. A Tool for Managing Software Architecture Knowledge. Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge, ICSE Workshops, (2007).
3. Batory D. S., Sarvela, J.N. and Rauschmayer, A. Scaling Step-Wise Refinement. IEEE Transanctions on Software Engineering 30(6), 355-371, (2004).
4. Benavides, D., Trinidad, P. and Ruiz Cortés, A. Automated Reasoning on Feature Models. 17th International Conference on Advanced Information Systems Engineering (CAiSE) Springer-Verlag LNCS 3520, 491-503, (2005).
5. Bosch, J. Design and Use of Software Architectures, Addison-Wesley (2000).
6. Bosch, J. Software Architecture: The Next Step, Proceedings of the 1st European Workshop on Software Architecture (EWSA 2004), Springer-Verlag, LNCS 3047, pp. 194-199 (2004).

7. Capilla, R., Nava, F., Pérez, S. and Dueñas, J.C. A Web-based Tool for Managing Architectural Design Decisions, Proceedings of the Workshop on Sharing and Reusing Architectural Knowledge, ACM Digital Library, Software Engineering Notes 31 (5).
8. Capilla, R., Nava, F..and Dueñas, J.C. Modeling and Documenting the Evolution of Architectural Design Decisions, Proceedings of the 2nd Workshop on Sharing and Reusing Architectural Knowledge, ICSE Workshops, (2007).
9. Capilla, R. Sánchez, A. Dueñas, J.C. An Analysis of Variability Modelling and Management Tools for Product Line Development. In Proceedings of the Software and Services Variability Management Workshop – Concept Models and Tools, Helsinki University of Technology Software Business and Engineering Institut, HUT-SoberIT-A3, ISBN: 978-951-22-8747-5, Helsinki, Finland, 32-47, (2007).
10. Clements, P., Bachman, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. Documenting Software Architectures. Views and Beyond, Addison-Wesley (2003).
11. Dueñas, J.C. and Capilla, R. The Decision View of Software Architecture, Proceedings of the 2nd European Workshop on Software Architecture (EWSA 2005), Springer-Verlag, LNCS 3047, pp. 222-230 (2005).
12. Dhungana, Rabiser, R., D., Grünbacher, P.and Prähofer, H., Federspiel, C. and Lehner K. Architectural Knowledge in Product Line Engineering: An Industrial Case Study. 32rd Euromicro Conference on Software Engineering and Advanced Applications, 186-197, (2006).
13. Dhungana, D., Grünbacher, P.and Rabiser, R. DecisionKing: A Flexible and Extensible Tool for Integrated Variability Model. In Proceedings of the 1st Workshop on Variability Modelling of Software-intensive Systems (VAMOS), LERO, UL, Ireland (2007).
14. Fritsch, C., Lehn, A. and Strohm, T. Evaluating Variability Implementation Mechanisms. Procs of International Workshop on Product Line Engineering (PLEES'02), Technical Report at Fraunhofer IESE (No. 056.02/E) 59-64 (2002).
15. Jansen, A. and Bosch, J. Software Architecture as a Set of Architectural Design Decisions, 5th IEEE/IFIP Working Conference on Software Architecture, pp. 109-118, (2005).
16. Jaring, M. and Bosch, J. Variability Dependencies in Product Family Engineering. 5th International Workshop on Product family Engineering (PFE), Springer-Verlag, LNCS 3014, pp. 81-97, (2004).
17. Kang K. C., Cohen S., Hess J. A., Novak W. E., Peterson A. S.. Featured-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report, CMU/SEI-90-TR-21 ESD-90-TR-22, Software Engineering Institute, Carnegie Mellon University, Pittsburgh (1990).
18. Kruchten P. Architectural Blueprints. The "4+1" View Model of Software Architecture, IEEE Software 12 (6), pp.42-50 (1995).
19. Kruchten, P., Lago, P., and van Vliet, H., T. Building up and Reasoning About Architectural Knowledge, QoSA2006, LNCS, pp. 43-58 (2006).
20. Lago, P and van Vliet, H. Explicit Assumptions Enrich Architectural Models, ICSE 2005, IEEE CS, 206-214, (2005).
21. Lee, K. and Kang, K.C. Feature Dependency Analysis for Product Line Component Design. 8th International Conference on Software Reuse (ICSR), Madrid, Springer-Verlag LNCS 3107, pp. 69-85, (2004).
22. Perry, D.E. and Wolf, A.L. "Foundations for the Study of Software Architecture", Software Engineering Notes, ACM SIGSOFT, October 1992, pp. 40-52.
23. Rozanski, N. and Woods. E. Software Systems Architecture: Working with Stakeholders Using viewpoints and Perspectives, Addison-Wesley (2005).
24. Sinemma, M., Deelstra, S., Nijhuis, J. and Bosch J. COVAMOF: A Framework for Modeling Variability in Software Product Families, 3rd SPLC Springer-Verlag LNCS 3154, 197-213 (2004).

25. Sinemma, M., van der Ven, J.S., and Deelstra S. Using Variability Modeling Principles to Capture Architectural Knowledge, 1st SHARK Workshop (2006).
26. Svahnberg, M., van Gurp, J. and Bosch, J. A Taxonomy of Variability Realization Techniques. Software Practice & Experience, vol 35(8), 705-754, (2005).
27. Tang, A., Babar, M.A., Gorton, I. and Han, J.A. A Survey of the Use and Documentation of Architecture Design Rationale, 5th IEEE/IFIP Working Conference on Software Architecture, (2005).
28. Tang, A., Jin, Y. and Han, J. A rationale-based architecture model for design traceability and reasoning. Journal of Systems and Software, 80 (6). 918-934
29. Trujillo, S., Azanza, M., Diaz, O. and Capilla, R. Exploring Extensibility of Architectural Design Decisions, Proceedings of the Workshop on Sharing and Reusing Architectural Knowledge and Design Intent (SHARK/ADI'07), ICSE Workshops, Minneapolis, USA, May 2007, IEEE CS.
30. Tyree, J. and Akerman, A. Architecture Decisions: Demystifying Architecture. IEEE Software, vol. 22, no 2, pp. 19-27, (2005).
31. Wang, A., Sherdil, K. and Madhavji, N.H. ACCA: An Architecture-centric Concern Analysis Method, 5th IEEE/IFIP Working Conference on Software Architecture, (2005).

Appendix 1: PAKME-ADDSS Data-model Comparison

Table 1. Comparison of the entities supported in both PAKME and ADDSS data models.

PAKME	PAKME description	ADDSS	ADDSS description	Matched
Stakeholder	People interested in the architecture process or product	Users	Stakeholders with different roles interested in architectures	Yes
Stakeholder Group	Define the project access rights	Permissions	Rights for each user type	Yes
Architectural significant requirements (ASR)	Are NFR (QA) Quality goals can be derived. An ASR must be satisfied by one or several design decisions	Requirements	Functional and non-functional requirements. Only the type, the number, and a text description is provided	Yes
Scenario	Is a refinement of ASR and are QA	NFR	A non-functional requirement	Partially through requirements
Quality factors	Are the factors a QA should match			Only quality attributes are supported
Findings	A description for the QA that meet a particular scenario			Not supported in ADDSS
Analysis Model	Is a reasoning framework that reasons about the effect of different tactics on QA			Not supported in ADDSS

	scenarios			
Design Tactic	Is a design mechanism for achieving the desired level			Not supported in ADDSS
Pattern	Characterizes a design solution in a given context	Pattern	Describes a design solution. Patterns are classified by its type, description and an usage example	Yes
Effect of pattern	Defines the effect of the pattern on a particular QA			Not supported in ADDSS
Support Information	Captures the background information required to justify the choice of a decision for a particular scenario	Optional attributes	Optional attributes capture extra information	Partially supported
Architecture Decision	Is a high level decision that satisfies FR and NFR. There are dependencies between decisions.	Design Decision	Captures the design decision and its rationale through a set of attributes. Basic dependencies can be defined but not the of the dependency	Yes
Architecture Decision Rationale	Is the reason behind the architecture	Rationale	Is the reason behind the architecture	Yes
Design history	A history of decisions is supported	Version and responsible	Some attributes in ADDSS are used for the same goal	Partially supported
Alternative	Design decisions may be related to other design alternatives	Status and category attributes	ADDSS offers a category attribute to indicate if a decision is alternative design choice but also an status to know if the decisions has been approved or rejected	Yes
Architecture Description	Prescribes the architecture to be realized	Architecture	Provides the information about a particular architecture and a link to the views supported	Yes
Architectural View	Provides a description for architectural views with the images	View attribute	Describes the view of the architecture and provides a link to it	Yes
		Translation	Provides multilingual support	Not supported in PAKME

Appendix 2: Core Common Entities between PAKME and ADDSS

Table 2. Main common entities distilled from PAKME and ADDSS data models.

Common entity	**Entity description**
Stakeholder	Are those persons interested in the architecture process or product
Architectural significant requirements	Functional and non functional architectural significant requirements drive the selected design decisions
General Knowledge	Characterizes a design solution in a given context, like patterns or styles
Design Decision	Is a high level design decision that explains the decisions and its underpinning rationale
Decision history	A history of decisions is supported
Architectural Description	Prescribes the architecture to be realized
View	Provides a description for architectural views with the images