Supporting Variability Management in Architecture Design and Implementation

Iris Groher and Rainer Weinreich
Johannes Kepler University Linz, Austria
{iris.groher, rainer.weinreich}@jku.at

Abstract

Variability management and architecture design and implementation are mostly separate activities. Existing variability management approaches focus on documenting the variable properties of a product line and on deriving products as members of the product line. Their support for architecture modeling is limited to capabilities needed for product derivation. Existing architecture design and implementation tools, on the other hand, support different architectural and implementation views on a software system but lack support for variability tracing and modeling during development and design. To close this gap, we present an approach for integrated variability management during software architecture design and implementation. The approach is an extension of LISA, a model and toolkit for architecture management and analysis. Variability modeling is provided by an additional view on a single consistent architecture model leading to a tight integration of variability and architecture modeling and implementation. Architects and developers are thus constantly aware of the variants they are working on and their implications on architecture design and implementation.

1. Introduction

A vital part of software product line engineering (SPLE) [1,2] is the definition of a sustainable product line architecture [3], which comprises a variable structure that is the basis for the structures of all applications built as members of the product line [1]. The effectiveness of a software product line approach directly depends on how well variability within the portfolio is realized in the product line architecture and implementation. Though software product lines aim at simplifying software development and improving reuse by managing variations of a software across different operational contexts, they themselves tend to become quite complex. Variations often have widespread impacts on multiple artifacts in various life cycle stages, making their effective management a predominant engineering challenge in SPLE.

Existing variability management approaches [4] typically support the creation of variability models that document the variable properties of a product line. They describe what can vary (the so-called variation points) and the types of variation offered for a particular variation point (the so-called variants). Different types of assets can be linked to variation points and variants to explicitly document how the variability is realized in the product line. Apart from documenting the variability, a strong focus of existing approaches lies in product derivation [13,14], that is, in the creation of products as members of the product line. During application engineering, products are generated (semi-)automatically based on the links between the variability model and the specified assets.

The actual development and design of the product line architecture and its implementation during domain engineering are often neglected by existing variability management approaches [11,12,13,14]. The product line architecture is usually modeled with traditional UML tools and implemented using IDEs without explicit support for variability management. Variability management tools come into play after the architecture design and implementation phase to help configure the reusable assets during product derivation. Variability models, including the links to development assets, are created and maintained by only a few experienced developers and product managers. Variability-related activities are thus separated from software engineering activities in existing approaches. This lack of integration hinders variability awareness of all developers during the product line engineering life cycle.

We argue that a view on the variability of a system should be present at any time during architecture development, design, and implementation. For architects and developers this creates a continuous awareness of all the variation points and variants and their realization in architecture and implementation. Full traceability between variability models and architecture and implementation models requires navigation from variation points and variants to development assets and vice versa. With full
traceability the impacts of changes of the variability model on architecture design and implementation are immediately visible to developers working on the assets.

In this paper we present an approach and toolkit that addresses the afore-mentioned challenges. We integrate variability management into LISA (Language for Integrated Software Architecture), a model and toolkit for continuous architecture management and analysis of heterogeneous component-based software systems [7]. Our approach is based on a single formalized architecture model that is used for architecture representation throughout the software architecture life cycle. The model integrates and connects requirements, design decisions, architectural abstractions, implementation artifacts, and information needed for both manual and automatic architecture analysis. The architecture-related activities are supported by an integrated set of tools working on this model.

Variability management becomes an integral part of architecture definition and implementation when it is integrated into LISA. New variation points and variants can be added at any time during architectural modeling and can be linked to arbitrary architectural elements and implementation artifacts. Since the product line architecture is continuously synchronized with the underlying product line implementation, the approach enables the iterative development of the product line architecture and implementation together with seamless variability management. A view on the variability of the product line and its connection to product-line architecture and implementation is present for all developers at any time in the software life cycle.

The remainder of the paper is structured as follows: Section 2 introduces the LISA model and toolkit. Section 3 discusses our research methodology. In section 4 we present our approach for integrating variability and software architecture management. Related work is discussed in section 5. We provide an outlook on future work and conclude the paper in section 6.

2. Background – The LISA Approach

The work described in this paper is an extension of the LISA approach. LISA consists of two main elements: an architecture description meta-model (LISA model) and a set of integrated tools (LISA toolkit) that operate on this model and support architecture-related activities like architecture design, analysis and implementation.

2.1 The LISA model

The LISA model is a meta-model for describing heterogeneous component-based and service-oriented software architectures [6,7]. It provides elements for describing lower-level structures like classes and interfaces as well as higher-level structures like components, composites, and system configurations. The LISA model also provides the basis for architecture knowledge management by including requirements and design decisions as first-class elements in the architecture model [8]. The model is independent of any concrete component or implementation technology and can be synchronized with an implementation by using technology-specific implementation bindings [7].

The LISA model itself is composed of a number of integrated submodels describing different aspects of a software architecture. These submodels can again be divided into LISA core models and LISA model extensions. LISA core models contain structural elements at different levels of abstraction like classes, components, ports, connections, configurations, and systems. LISA model extensions contain additional models for aspect-oriented programming [16], reference architecture specification, application environment modeling, and for binding architecture models to different implementation technologies.

Elements of the LISA core models that are important in the context of this paper are shown in Figure 1. LISA-based architecture models can be partitioned into modules containing different architectural elements. The base class for architectural elements is AbstractArchitectureElement. As shown in Figure 1, nearly all other elements including requirements and design decisions are architectural elements. The figure also shows that the core models can be roughly divided into four parts:

1) The requirements or decision model contains elements for describing requirements, design decisions, and relationships between them. Design decisions are used for capturing and documenting decisions in the design process that lead to actual designs [18]. Requirements and design decisions are special kinds of decisions in our model (see [19] for the relationship between requirements and design decisions). Traces can be defined from requirements to design decisions, and from both requirements and design decisions to other architectural elements. Through support for synchronization of architecture and implementation full traceability is provided from requirements and design decisions to both architecture and implementation elements.
2) The implementation and language model contains elements that can be directly mapped to an implementation language, like classes, interfaces, methods, and enumerations.

3) The structure and types model contains higher-level structural elements for defining packages, modules, components, and component types.

4) The configuration or runtime model supports describing runtime configurations, which are essentially component instances and connections among these instances.

Additionally, elements from the various models can be bound to a specific implementation technology or component model through technology bindings. Many pre-defined technology bindings exist. LISA currently provides technology bindings for J2EE, BPEL, Eclipse, Java, Spring, SCA, OSGi, and C#.

2.2 The LISA toolkit

The LISA toolkit is a set of plugins for the Eclipse IDE. It supports architecture modeling, visualization, and analysis of architecture models based on the LISA meta-model. The toolkit provides multiple architectural views for exploring and editing architecture information [7]. Examples are different kinds of structural views like module, component, and configuration diagrams, views for presenting architecturally significant requirements and design decisions, views for exploring traceability relationships, and also (albeit limited) support for behavioral views like sequence diagrams. All views are generated from a single integrated architecture model, which avoids inconsistencies among views.

In addition, the toolkit supports continuous synchronization of an architecture with an implementation and automated consistency and completeness analysis [7], which ensures that the presented architectural information is consistent and up-to-date.

To summarize, the LISA approach provides a highly integrated set of tools for supporting architecture related-activities during the software life cycle. An emphasis of the approach is the integration of architecture modeling, architecture knowledge management, architecture analysis, and system implementation, which is particularly suited for incremental and agile development processes.

3. Research Methodology

The LISA approach has been developed using action research [17], which is characterized by a close interaction of industry and academia with continuous feedback loops and observation. Currently we are working with a company in the banking domain where our approach is used for architecture extraction and evaluation as part of a company-wide enterprise architecture management initiative. The LISA approach has also been applied in projects with other
companies for tracing requirements and design decisions to architecture and implementation artifacts.

In some of our recent work with industrial partners we experienced a need for continuous variability management during architecture design and implementation. Development teams have created their own solutions for embedding configuration information and different variants into implementation artifacts to manage variability during architecture design and implementation. External variability management and design tools were avoided because of the effort involved in keeping the required models consistent and up-to-date.

Since we already support architecture extraction in the LISA approach, we focused on providing means for connecting variability models and architecture. The main aim is to support evolution of a product line architecture by identifying required flexibility early but at the same time preventing over engineering, that is, the inclusion of unnecessary flexibility in the architecture design.

When reviewing existing approaches for variability management we found that while product derivation is usually supported well, variability management and architecture design are typically separated in existing approaches leading to an additional management overhead developers wanted to avoid beforehand. Also, an integration of existing variability management approaches proved to be insufficient for providing immediate feedback on variable requirements and violations during architecture design and implementation.

Therefore, we developed a solution to directly integrate variability management into the LISA approach. This allows us to facilitate full traceability, not only between elements of the variability model and structural elements of the architecture model but also to other elements of LISA architecture models including requirements and design decisions. In addition, we are able to facilitate helpful features of the toolkit like automatic creation of traces, continuous analysis, and continuous synchronization of architecture and implementation.

For validating the applicability of the developed approach we used a previous project from the home automation domain (conducted by one of the authors) as an initial case study. Currently we are working with our partners to apply and refine the approach through action research.

4. Integrated Variability Management with LISA

The following subsections show how variability management has been integrated in the LISA approach. This includes a description of the variability model and its facilitation in the LISA toolkit.

4.1 Representing Variability

We use the Orthogonal Variability Modeling (OVM) approach [1] as the basis for our variability model. The OVM meta-model presented in [1] is integrated into the LISA model as a model extension as described in section 2.1. OVM advocates the use of a separate model to capture the variability of the product line. This model is then related to traditional software development models such as feature models, use case models, design models, component models, and test models. To document what varies, the variable properties of the different development artifacts are documented by variation points. In addition to what varies, it is important to also capture the rationales of variability. We distinguish between external and internal variability [1]. External variability is visible to customers, whereas internal variability is hidden from customers. How the variable properties vary is documented by variants and their realization in development artifacts.

Capturing the variability of the system in a separate model instead of integrating the information directly into the artifacts has many advantages [1]. Most importantly, it provides a single view of the variability across all development artifacts. The variability information is not spread across different artifacts and does not need to be kept consistent. Also, how variability influences the different kinds of artifacts is better visible when using a separate model for variability documentation. As our aim was to support variability management in the software architecture life cycle, following an orthogonal variability modeling approach was obvious.

We provide concepts like variation points, variants, and variability dependencies to model the central concepts of OVM. A variability dependency is an association between a variation point and a variant. It states that a variation point offers a certain variant. Variability dependencies can be optional, mandatory or alternative. An optional variability dependency states that a variant related to a variation point can be part of a product but does not need to be part of it. A mandatory dependency states that a variant must be part of a product if the associated variation point is part of it. Alternative dependencies define the number of optional variants that need to be selected for the associated variation point.

Constraint dependencies describe restrictions between variants, between variation points, or between variants and variation points. Therefore, we distinguish between variant constraint dependencies, variation...
point constraint dependencies and variant to variation point constraint dependencies [1]. Constraint dependencies describe requires or excludes relationships between variants or variation points. The selection of a variant or variation point can require or exclude the selection of another variant or variation point.

To document how the variability is realized in the system, variation points and variants can be linked to development artifacts. In our case, variation points and variants can be linked to all kinds of architectural elements in LISA. Linking variability models and architectural elements is explained in more detail in Section 4.3.

4.2 Variability and Architecture Modeling

Figure 2 shows how the variability model of the home automation product line we used as the initial case study is visualized in the LISA toolkit. A home automation system connects the devices of a house and enables inhabitants to monitor and control these devices. The status of the devices can either be changed manually by the inhabitants or automatically by the system using predefined policies. For example, in case of smoke detection the system automatically closes all windows and calls the fire brigade. The main causes of variability in home automation systems are varying types of houses, different customer demands, the need for short time-to-market, and cost savings.

For visualizing the orthogonal variability model we use a similar graphical notation as introduced in [1]. Variation points are represented by triangles and variants are represented by rectangles. Variability dependencies are labeled with either “mandatory”, “optional”, or the “range” in case of alternative choices.

In the home automation system shown in Figure 2 the Light Management of the house is variable and thus a variation point of the system. Dimmable Lights (the possibility of dimming the lights in a room by setting their light level) is thus an optional variant of the Light Management variation point. Bell, Light, and Siren are alternative variants of the Security variation point (representing different types of alarms the user can choose from) where at least 1 and at most 3 variants must be selected. Authentication and Light Simulation are optional variants of the Security variation point. The Light Simulation variant periodically turns on the lights in the house. Its purpose is to simulate inhabitants being at home in case they are on vacation. Outdoor Temperature (the capability of measuring the outdoor temperature) and Automatic Windows are optional variants of the Temperature Management variation point. The Automatic Windows variant automatically opens the windows if the temperature in a room raises above a certain threshold and closes them if the temperature falls below a certain threshold.
connected to the dimmer. The dimmer device dims the light of the connected lights. Automation components perform actions either periodically or in case of certain events. The LightSimulation component, for example, periodically turns on certain lights in the house.

The properties of variation points and variants are shown in the properties view. In Figure 4(3) the properties of the currently selected Automatic Windows variant are displayed. Variants have a name and can be assigned to developers that are responsible for it. A variant can be architecturally significant, meaning that its presence has an effect on the architecture of the system. A priority (low, medium, high) can be defined and its architecture and implementation status (open, later, ignore, addressed, verified, rework) can be documented. Additionally, for every variation point and variant a rationale for introducing it can be captured. We automatically document who created the variation point or variant at what time. The different properties of variation points and variants can be used for analysis purposes. For example, it can easily be checked whether variants with a high priority have already been addressed in architecture and implementation or not.

4.3 Linking Variability to Requirements, Architecture, and Implementation

To document how the variability is realized in the system, traceability links are defined between variation points or variants and development artifacts. A variant must be related to at least one development artifact and may be related to more than one development artifact (realized by relationship with multiplicity ‘1..n’). A variation point can but does not have to be related to one or more development artifacts (represented by relationship with multiplicity ‘0..n’). Examples of development artifacts representing variation points include abstract classes that realize common behavior for several variants [1].

Any AbstractArchitectureElement (e.g. requirements, decisions, components, ports, classes, methods, cf. Figure 1) can be linked to variation points and variants. This supports full integration of the variability model into the LISA model including traceability of variability from requirements to architecture and code and vice versa. Elements from the architecture navigator (Figure 4(1)) can be assigned to variation points and variants (Figure 4(2)) using drag & drop. Alternatively, architectural elements can be assigned to variability elements in the properties view in the Assigned Architecture Elements tab of the respective variability element (Figure 4(3)).

In the home automation system, the variation points and variants of the variability model in Figure 2 are linked to parts of the architecture in Figure 3 to document how the variability is realized in the product line.
In our example, the Dimmable Lights variant (cf. Figure 2) is linked to the DimmerContract (cf. Figure 3) (setting the desired light level of the connected lights). The variant is also linked to the DimmerComponent that provides the DimmerContract. The operations for setting the light level of the light itself are part of the LightContract and LightComponent respectively. They are also linked to the Dimmable Lights variant because they only need to be parts of the contract and component if the variant is part of the system.

In addition to manually linking architectural elements to the variability model as described above, we support the automatic creation of trace links from design activities as explored in [10]. To activate automatic tracing, variation points and variants can be set active by developers working on the architecture model. If a variability element is set active (indicating that the developer is currently working on its realization), all subsequent modeling and implementation events are recorded. The recording is performed until the active variability element is set inactive. After stopping the recording, the system suggests trace links to elements that have been created or modified during the recording. Designers and developers can review the suggested traces and exclude the ones that are not relevant for the active variation points or variants.

Additionally, the LISA approach supports the continuous synchronization of architecture and implementation through technology bindings and specific kinds of analysis [7]. In the home automation example, we use OSGi bindings to connect architecture and implementation. Differences between architectures and implementations are visualized as validation problems and shown in both representations. Problems can then either be resolved by changing the system implementation, or by changing the architecture description.

### 4.4 Variability-Related Analysis

Since the LISA toolkit permanently monitors architectures and implementations for changes, we are able to detect changes of artifacts that are linked to elements of the variability model like variants and variation points. If changes are detected, a manual review is proposed, since the correct realization of the defined requirements can no longer be ensured.

In addition we facilitate the analysis framework provided by the LISA approach [9] to support other kinds of variability analysis. This includes detecting variation points and variants that have not yet been addressed in architectures or implementations, detecting variation points or variants that need to be validated/verified, and detecting variation points or variants that have not been linked to architecture and implementation artifacts. Analysis is based on user-definable constraint sets [9]. Each user-defined analysis can be configured to include the desired analysis constraints. Figure 5 shows how the analysis is configured. An analysis is a named set of selected constraints. After the analysis is activated, its constraints are continuously evaluated in response to modifications of the architecture description and the system implementation. Problems can thus be detected and visualized (in editors and architectural diagrams) immediately.
4.5 Variability Awareness

Variability awareness is provided by a variability view of the system that is always present during architecture design and implementation (cf. Figure 4(2)). We view variability awareness as an important concept to guide developers in providing flexibility where it is really needed. This also addresses the problem of over-engineering, that is, the introduction of unnecessary flexibility in the design. A permanent view on the variability model reminds the developers to focus on implementing variability in the right places.

We use different kinds of highlighting to show constraints between different elements of the variability model and also to show implications for elements of the architecture model.

The use of highlighting of relations between variation points and variants in the home automation example is illustrated in Figures 2 and 4. The Outdoor Temperature variant is highlighted in green in the variability view in Figure 2 and Figure 4(2), because it is required by the selected variant Automatic Windows. Excluded variants are highlighted in red.

Figure 4 also shows how highlighting is used to illustrate how variation points and variants are linked to elements of architectures and implementations. For example, the WindowAutomation component shown in the architecture view in Figure 4(1) is linked to the Automatic Windows variant in the variability view (highlighted in yellow). The operations getOutDoorTemperature() in the IThermometer interface and ThermometerImpl class are highlighted in green because they are linked to the variant Outdoor Temperature that is required by the selected variant Automatic Windows. Architectural elements linked to excluded variants are highlighted in red. This makes implicit dependencies between architectural elements visible, based on the dependencies between related variability elements. Making implicit relationships between architectural elements visible helps developers to make the right decisions in architecture design. Without the knowledge of those kinds of relationships it might easily happen that, for example, developers unintentionally introduce dependencies between elements that are linked to alternate and excluding variants.

Support for variability awareness is not only provided in architecture design but also in system implementation by annotating variable implementation artifacts with variability information in source code editors. This is shown in Figure 4(4), where the operation getOutdoorTemperature() of the IThermometer interface is linked to the Outdoor Temperature variant. The operation is only relevant if the respective variant is part of a product. A marker (variant symbol) next to the operation makes this information available to developers. The visualization of variability in architecture and implementation artifacts supports navigating from the artifacts to the variability model. This means that navigation is provided in both directions: from variation points and variants to their realizations and from variable artifacts to the variability model.

5. Related Work

Many different variability modeling approaches and tools exist [4]. Here, we focus on the ones that we believe are most relevant in our context. In general, most existing approaches and tools are focused towards product derivation and configuration of assets during application engineering. The actual development of a product line architecture and its implementation are often neglected.

Feature Modeling Plugin [11] is an Eclipse plugin for developing feature models. It integrates with the Rational Software Modeler or Rational Software Architect to support product line modeling. Features can be mapped to UML models during domain engineering to document how the variability is realized in design. During application engineering, the models are configured based on a valid feature selection. Similar to our approach, Feature Modeling Plugin supports linking variability models and (architecture) design models. However, the approach does not provide a single environment for architecture design and variability modeling. Developers need to switch between tools, which hinders a permanent view of the variability. Traceability of variability to implementation is not supported in the approach. In
general, the Feature Modeling Plugin approach is focused towards model configuration during application engineering.

**Gears** [12] is a development environment that supports variability modeling using so-called feature profiles where optional and varying features can be defined. Different kinds of assets (e.g. code files, requirements, documentation) can be linked to features and configured during application engineering. Several third party tools are integrated in Gears. The Rhapsody/Gears bridge supports the configuration of UML and SysML models based on feature choices in Gears feature profiles. The Gears approach is focused on configuration during application engineering. In contrast to our approach, Gears does not support variability modeling, architectural modeling, and implementation in a single, integrated environment. A permanent view on the variability is not provided because of the need to switch between tools.

**Dopler** [14] is an approach that supports decision and asset modeling to capture the variability of the system and its realization. Dopler supports the definition of new asset types to reflect the specifics of a particular product line. To link assets and decisions, inclusion conditions can be defined which have to be fulfilled for a particular asset to be included in a product. During product derivation, products can be automatically generated based on the inclusion conditions. The Dopler approach focuses on decision and asset modeling in a way that they can be efficiently used for product derivation. Support for a permanent view of the variability during the actual development of the product line architecture and implementation is not provided. Also, Dopler does not support synchronization of architectures and implementations.

**Pure::variants** [13] is a variability and variant management environment integrated into the Eclipse IDE. Feature models support modeling the commonality and variability of the product line. Family models support modeling the assets that describe the system in terms of architectural elements. The family model is divided into components (representing functional features of the solution) that consist of logical parts of the software (e.g. classes, functions, variables). Physical elements (files) can be assigned to logical elements. This is similar to our approach where implementation bindings are used to link architectures and implementations. Pure::variants integrates with the Enterprise Architect modeling environment. Features defined in pure::variants can be linked to SysML and UML models. Similar to Gears, developers have to switch between tools to document how the variability is realized in the models. Also, pure::variants does not support synchronization of architecture and implementation.

**FeatureMapper** [15] is an approach and toolkit integrated into the Eclipse IDE that supports feature modeling and linking feature models to arbitrary EMF models. The mapping can be performed manually or, similar to our approach, using modeling event recording. Different forms of highlighting are used to visualize the dependencies between feature models and EMF models. In contrast to our approach FeatureMapper does not support traceability of features to implementations and synchronization of architectures and implementations.

### 6. Conclusions and Future Research

In this paper we presented an approach for integrating variability management in architecture design and implementation. Orthogonal variability modeling is integrated as an additional architectural viewpoint in the LISA toolkit, providing a permanent view on the variability of the system for all developers. Elements of the variability model can be linked to any architectural elements in LISA, which provides full traceability from variation points and variants to requirements and design decisions, architecture models, and implementations. Navigation is provided in both directions, from the variability model to requirements, architecture models, and implementations, and vice versa. Different highlighting facilities provide variability awareness during all kinds of modeling and implementation activities. Relationships between the models can be established manually or automatically by capturing traces during design and implementation activities. Synchronization between architectures and implementations not only avoids architectural drift but also supports the detection of the impacts of changes to variability models, architecture models, and implementations. Potential problems due to the deletion or modification of artifacts are immediately detected and reported to the developers.

While we have used the LISA approach in several projects with industrial partners we have currently only limited experience with the support for variability presented in this paper. Further validation and improvement using action research in close cooperation with our industrial partners is an important topic of current and future work. Additional topics we plan to address are support for application engineering, variability mining, evolution management, and variability extraction.

In the work presented in this paper the focus has been on variability awareness in developing product line architectures during domain engineering. Support for product derivation during application engineering is currently not provided in our approach. In the future
we also plan to integrate product-specific views into LISA, which allow developers to see which parts of the architecture and the implementation are part of a specific product and which are not.

We further envision a systematic approach that guides engineers in defining a product line architecture based on existing product architectures. To minimize the required up-front investment in existing product line engineering approaches we want to provide support for merging existing product architectures. Companies typically do not start from scratch when building a product line; they typically have many more or less similar products already delivered to customers. In our work we plan to provide a similarity analysis for existing product architectures including support for refactoring existing product architectures into a product line architecture.

We also plan to extend our approach to support variability extraction. In our industrial projects we are often faced with a situation where a system implementation is available while an architecture description is not. We currently support architecture extraction from various sources such as configuration files, code annotations, and program code. In the future we plan to support variability information extraction from code annotations provided by developers.

Additionally, we want to support product line evolution management. As our approach provides full traceability from requirements and design decisions to architecture and implementation, change impact analysis can be performed based on the trace information. All kinds of changes to any of the development artifacts can optionally be logged which further supports evolution management.

8. References