



## Engineering Virtual Domain-Specific Service Platforms

Specific Targeted Research Project: FP7-ICT-2009-5 / 257483

# Decision Support Framework for Platforms as a Service (Final)

### **Abstract**

*INDENICA aims at providing a basis for decision making in the architecting and variability resolution of customizable service platforms. This decision making is a rather complex activity, which provides a sophisticated engineering challenge. Correctly identifying the many design alternatives and resolving the corresponding decisions is a challenge that requires adequate support.*

*This deliverable describes the support that the INDENICA architecture and variability tools provide for decision making. We emphasize on architectural decision making and its integration with variability decision making. (The details of the variability decision making are described in deliverables D2.2.2 and D2.4.2)*

Document ID:	INDENICA – D1.3.2
Deliverable Number:	D1.3.2
Work Package:	WP1
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2013-09-30
Author(s):	SUH, SAP, SIE, PDM, TEL, UV
Project Start Date:	October 1 <sup>st</sup> 2010, Duration: 36 months
Document ID:	INDENICA - D1.3.2

---

---

**Version**

0.1	17 September 2013	Document structure for integration defined
0.2	24 September 2013	Section 2 added
0.3	27 September 2013	Revised Section 2
0.4	30 September 2013	Section 3 added
1.0	1 October 2013	final version

***Document Properties***

The spell checking language for this document is set to UK English.

---

## Table of Contents

1	Introduction .....	5
2	Architecture Decision Making – ADvISE Tooling.....	6
2.1	Related Work and Tools.....	6
2.2	ADvISE Eclipse plug-ins .....	7
2.2.1	ADvISE .....	8
2.2.2	FuzzDS .....	8
2.2.3	AK Transformation Language.....	9
2.3	Plug-in Dependencies and Technologies .....	9
2.4	Architectural Decision Meta-model.....	10
2.5	Installation .....	11
2.6	User’s Guide .....	12
2.6.1	ADvISE .....	12
2.6.2	FuzzDS .....	21
2.6.3	AK Transformation Language.....	23
3	A Tool-Based Approach to Integrated Variability- and Architecture-Decision-Making .....	24
3.1	Model Variability.....	24
3.2	Model Architectural Decisions.....	25
3.3	Define Mapping .....	26
3.4	Resolve Variability.....	27
3.5	Resolve Remaining Architectural Decisions.....	28
3.6	Wrap-Up.....	29
4	Summary and Conclusion .....	30
5	References .....	31
6	Appendix: Conceptual Integration of Variability Decision Making and Architecture Decision Making.....	33

---

**Table of Figures**

Figure 2-1: Plug-ins Overview .....	8
Figure 2-2: Plug-ins Dependencies.....	10
Figure 2-3: Architectural Decision Model .....	11
Figure 2-4: Installation from Update Site .....	12
Figure 2-5: ADvISE Perspective .....	13
Figure 2-6: ADvISE View .....	14
Figure 2-7: Questionnaire View .....	14
Figure 2-8: Design Patterns Tab .....	15
Figure 2-9: Design Solutions Tab.....	16
Figure 2-10: Architectural Decisions Tab .....	17
Figure 2-11: Select the type of question.....	18
Figure 2-12: Question expecting alternative options .....	19
Figure 2-13: Question expecting free-text answer .....	20
Figure 2-14: Example of a questionnaire .....	21
Figure 2-15: Gaussian membership functions for 3 linguistic values of property performance .....	22
Figure 2-16: Fuzzy Logic Based Approach for Supporting Architectural Decision Making.....	23
Figure 3-1: Textual view of the variability model of the WMS example modeled in EASy-Producer.....	25
Figure 3-2: Reusable Architectural Decision Model .....	26
Figure 3-3: Synchronize Architectural Decisions Launcher.....	27
Figure 3-4: Excerpt of the mapping between variability and architectural decisions.	27
Figure 3-5: Providing values to the variability decisions of the WMS example .....	28
Figure 3-6: Architectural Decision Making.....	29

## 1 Introduction

Creating virtual domain-specific service platforms requires very complex decision making both on a domain-level and on an engineering level. Supporting this decision making in a systematic manner provides the basis for reliable, consistent, and well-engineered service platforms. In the interim version (D1.3.1), we discussed the breadth of the problem and discussed many implications of this decision making, along with corresponding approaches to support this. This deliverable provides a tool-based approach that supports the decision making both from a variability and from architecture point of view. It integrates the variability support with an architecture support tool and thus addresses the decision in an integrated manner.

This deliverable is structured as follows. In the next section, we first introduce the core contribution of D1.3.2: the ADvISE tool with its architectural decision support. In Section 3, we describe how ADvISE is combined with EASy-Producer to create an integrated decision support approach. Section 4 provides a brief summary. In Section 6 we add as additional background a paper, which further discusses the integration of both sides. We provide this only as an appendix, because it is a paper currently under review and was not originally written as a deliverable. However, it is exclusively a result of the INDENICA corporation and discusses only work relevant to this deliverable.

Further relationships to other INDENICA deliverables are:

- D 1.3.1: Realizes the approaches described in D.1.3.1
- D 2.1: Open Variability Modelling Approach for Service Ecosystems provides the basis for modelling of variability.
- D 2.4.2: Integrates with the Variability Engineering Tool in a specific way that addresses the needs specific to architecture decision making. Details on the Variability Engineering Tool and the EASy-Producer User Guides are contained in this deliverable.
- D 3.3.2: Integrates with the Tool Suite for Virtual Service Platform Engineering.

Comments on the relation to previous work:

- The contributions described in Sections 2 and 3 were solely developed as part of the INDENICA project and were motivated by the project. There exist relations (as described above) to previous deliverables, however.
- Further, especially the contributions in Section 3 are currently submitted for publication or already published.

## 2 Architecture Decision Making – ADvISE Tooling

The **Architectural Design decision Support framEwork** – ADvISE is a prototype to support architecture decision making.

To assist decision making for reusable architectural decisions, such as the ones needed in service-based platform integration, ADvISE is proposed for modeling reusable architectural decisions and architectural decisions under uncertainty (using Fuzzy Logic, (see FuzzDS for more details) at different levels of abstraction, i.e., high-level as well as technology and domain specific levels. This approach enables us to semi-automate the decision making process for recurring architectural decisions and document the design rationale at low cost. This way, we support software architects in recurring design making processes, so that they can have more time left to spend on the challenging architectural problems that require creative thinking.

Apart from that, in order to keep architectural decisions and designs consistent and traceable to each other, we introduce formal links between the reusable architectural decisions and architectural designs. We achieve that by integrating ADvISE with VbMF (WP3). Thus, we provide maintenance support for evolving architectural decisions and designs. In particular, using the *Architectural Knowledge(AK) Transformations Toolkit*, a bridge between architectural decisions modeled in ADvISE and the architectural designs modeled with VbMF, we are able to automatically generate architectural designs from actual decisions and check for inconsistencies between them.

In the following, we present step by step how to model reusable architectural decisions and decisions under uncertainty and how to use these models to support software architects in architectural decision making. It also explains how to integrate architectural decisions in ADvISE with architectural designs using the AK Transformation Language. ADvISE, as well as the rest of the tools integrated with it, can be adapted and modified according to the developers' needs, as they are all available as open-source Eclipse plug-ins.

### 2.1 Related Work and Tools

In recent years, software architecture is no longer solely regarded as the solution structure, but also as the set of architectural design decisions that led to that structure [1]. The actual solution structure, or architectural design, is merely a reflection of those design decisions. Architectural design views [5] document the design rationale of the architecture and contribute to the gathering of Architectural Knowledge and its sharing among different stakeholders.

Capturing architectural design decisions is important for analyzing and understanding the rationale and implications of these decisions and reducing the problem of architectural knowledge vaporization [2]. Several approaches have been proposed for capturing architectural decisions. Akerman and Tyree defined a rich decision-capturing template [3]. Kruchten et al. presented an ontology for

architectural decisions, defining types of architectural decisions, dependencies between them and a decision lifecycle [4], [5]. Zimmermann et al. suggested a metamodel for decision capturing and modeling [8]. These approaches concentrate on the reasoning on software architectures, capturing and reusing of AK as well as on the communication of the design decisions between the stakeholders.

Architectural decisions are the result of making trade-offs for the quality attribute requirements. For example, in the Architecture Tradeoff Analysis Method (ATAM) and Attribute-Driven-Design Method (ADD) [6] the analysis of architectural trade-offs is an important part of the architectural decision making process. Bachmann et al. suggest a reasoning framework with quality attribute knowledge to help architects make trade-offs that impact individual quality attributes in an architecture [7]. These and other approaches for supporting architectural decision making have not been integrated with tools for modeling and documentation of architectural decisions.

In addition, there has been much effort on the documentation of reusable architectural decisions, that can be used as reusable architectural knowledge assets [8–10]. Several tools have also been developed to ease capturing, managing and sharing of architectural decisions [11], [12]. Knowledge Architect [13], Archium Tool [14], ADDSS [15], PAKME [16], Architech [17] and ADkwik [18] are examples of tools that provide support for architectural decision documentation and ease the architectural knowledge management. In most of the cases, the focus is set on the collaboration, the manipulation of architectural decision artifacts and their relationships, and the capturing and reuse of architectural knowledge. Automated support for architectural decision making and for architectural decisions under uncertainty, as well as their connection to the corresponding designs, for supporting traceability and consistency between decisions and designs, are not addressed in any of the aforementioned tools. This is mainly the gap we intend to bridge by introducing the **ADvISE** tooling.

## **2.2 ADvISE Eclipse plug-ins**

The ADvISE tool as well as the tools that have been integrated with ADvISE are developed as *Eclipse plug-ins* organized in *features*. An Eclipse based product is structured as a collection of plug-ins and each plug-in contains the code that provides some of the product's functionality. Product plug-ins are grouped together into features, i.e., units of separately downloadable and installable functionality. The Eclipse platform itself is structured as subsystems which are implemented in one or more plug-ins.

Figure 2-1 illustrates the available features (*ADvISE*, *FuzzDS* and *AK Transformation Language*) and the plug-ins they include<sup>1</sup>.

---

<sup>1</sup> Please note that the plug-ins here described in terms of their functionality and not in terms of physical Eclipse projects.

---

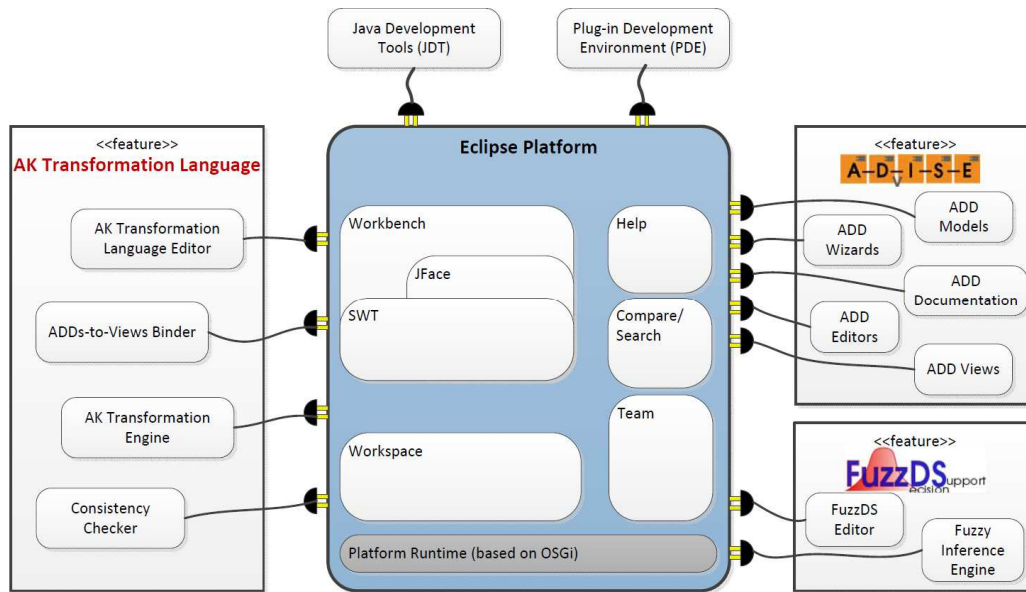


Figure 2-1: Plug-ins Overview

In the following, a short overview of the plug-ins' functionality is given.

### 2.2.1 ADvISE

**ADD Models:** It contains the definition of meta-models for modeling of reusable architectural decisions based on Questions, Options and Criteria (QOC) [19] and questionnaires for decision support. Also, the code for creating, modifying and persisting the architectural decision models.

**ADD Wizards:** It provides Eclipse wizards for creating ADvISE projects and models.

**ADD Documentation:** The documentation of architectural decisions and their rationale can be generated automatically from the answered questionnaires.

**ADD Editors:** Two kinds of editors are provided. One for editing the architectural decision models and one for editing the questionnaires generated from these models. Also, the questionnaires for assisting architectural decision making are generated automatically from the decision models.

**ADD Views:** It provides two customized Eclipse views, one for the management of resources and one for visualizing the decided and open decision points during architectural decision making. This plug-in contains also code for manipulating all related artifacts (create, delete, rename, etc.).

### 2.2.2 FuzzDS

**FuzzDS Editor:** It provides a textual Editor for editing reusable fuzzy models (based on Fuzzy Logic and fuzzy rules) to be used for architectural decision making under uncertainty (see 2.2.2 for more details).

**Fuzzy Inference Engine:** The Fuzzy Inference Engine infers the best-fitting solutions in a specific context (based on the fuzzy models) given specific requirements by leveraging the fuzzy models that have been edited with the FuzzDS Editor.

### 2.2.3 AK Transformation Language

**AK Transformation Language Editor:** The transformation from actual decisions to design views is achieved by executing transformation actions that apply to these views. These transformation actions (simple or compound) can be edited using the AK Transformation Language Editor. Not only transformation actions but also templates of transformation actions can be edited using this editor. The .action files can be, thus, also edited using a template language editor (Velocity).

**ADDs-to-Views Binder:** Transformation actions that are edited in template form need to be bound to actual values when actual decisions are made. The bound actions are executable and can afterwards transform the design views.

**AK Transformation Engine:** The transformation actions enact on the design views using the AK Transformation Engine. Also, consistency checking rules (constraints) for checking the conformance of the architectural decisions to the corresponding designs are generated as soon as the transformation actions modify the design views.

**Consistency Checker:** It validates the constraints for consistency checking between architectural decisions and designs and highlights detected inconsistencies.

## 2.3 Plug-in Dependencies and Technologies

Figure 2-2 shows the dependencies between the various plug-ins introduced before, as well as the technologies and frameworks they are based on. ADvISE (with its integrated tools) is an Eclipse RCP application<sup>2</sup>. FuzzDS can be used either stand-alone or through the ADvISE user interface. The AK Transformation Language requires both ADvISE and the VbMF Framework plug-ins as it works as a bridge between these tools.

The grammars for the AK Transformation Language (DSL) as well as the FuzzDS DSL were created with Xtext framework<sup>3</sup>. Xtext is also used to generate Eclipse-based textual editors that can support several useful features such as syntax highlighting, content assist and auto-completion, validation and quick fixes, automated external cross-references resolutions, and so on. The Eclipse Modeling Framework (EMF)<sup>4</sup> project was used to create all required models (e.g., architectural decision models) and the code for editing these models. Where model-to-model or model-to-text generation is required (e.g., AK Transformation Engine) Xtend2<sup>5</sup> was used. The

---

<sup>2</sup> <http://www.eclipse.org/home/categories/rcp.php>

<sup>3</sup> <http://www.eclipse.org/Xtext/>

<sup>4</sup> <http://www.eclipse.org/modeling/emf/>

<sup>5</sup> <http://www.eclipse.org/xtend/>

---

template binding of transformation actions and constraints in template form was done with the Velocity Template Engine<sup>6</sup>. Finally, the fuzzy inference system of the open-source jFuzzyLogic package<sup>7</sup> was adapted for the needs of FuzzDS.

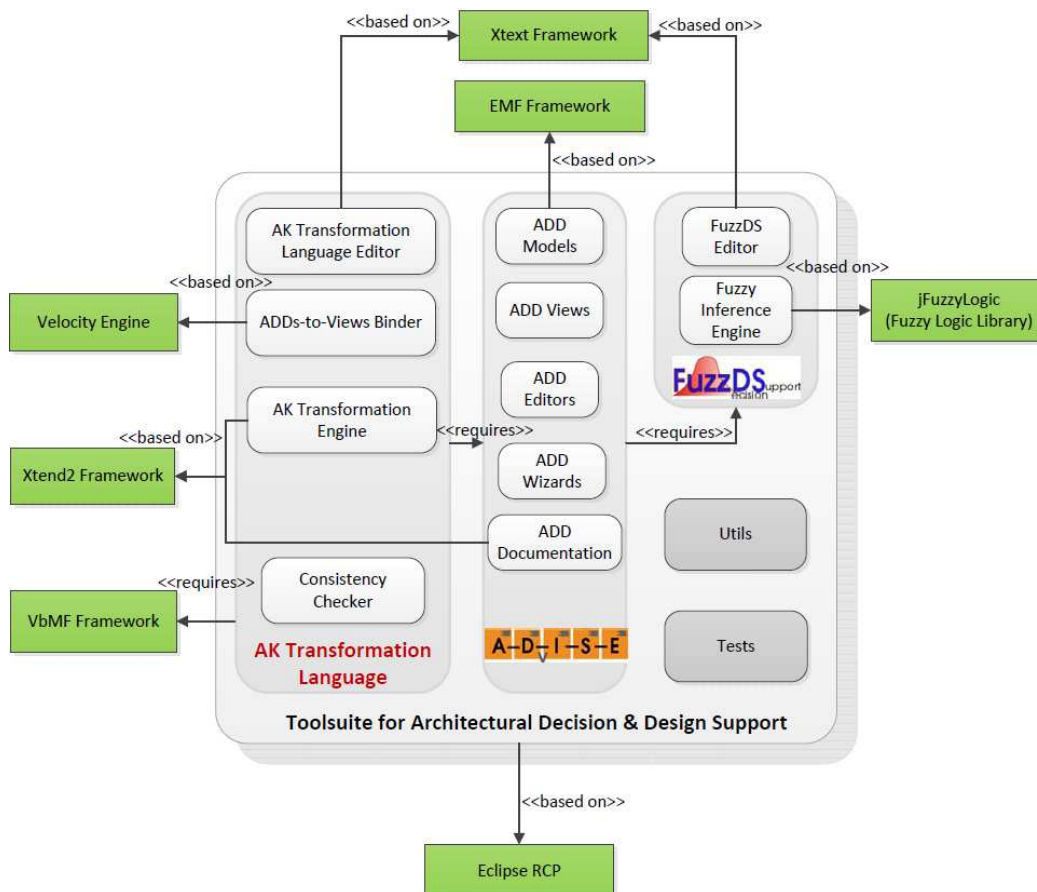


Figure 2-2: Plug-ins Dependencies

## 2.4 Architectural Decision Meta-model

Architectural decision models (see in Figure 2-3) contain architectural decisions - decision points and for each decision point a set of questions along with potential options or answers are introduced. The selection of an option can lead either to a solution (often pattern-based) or trigger follow-on decisions and questions. A question that requires a free-text answer can also be followed by a next decision or question. Also, an option can constrain other options (e.g., force or be incompatible with).

<sup>6</sup> <http://velocity.apache.org/>

<sup>7</sup> <http://jfuzzylogic.sourceforge.net/>

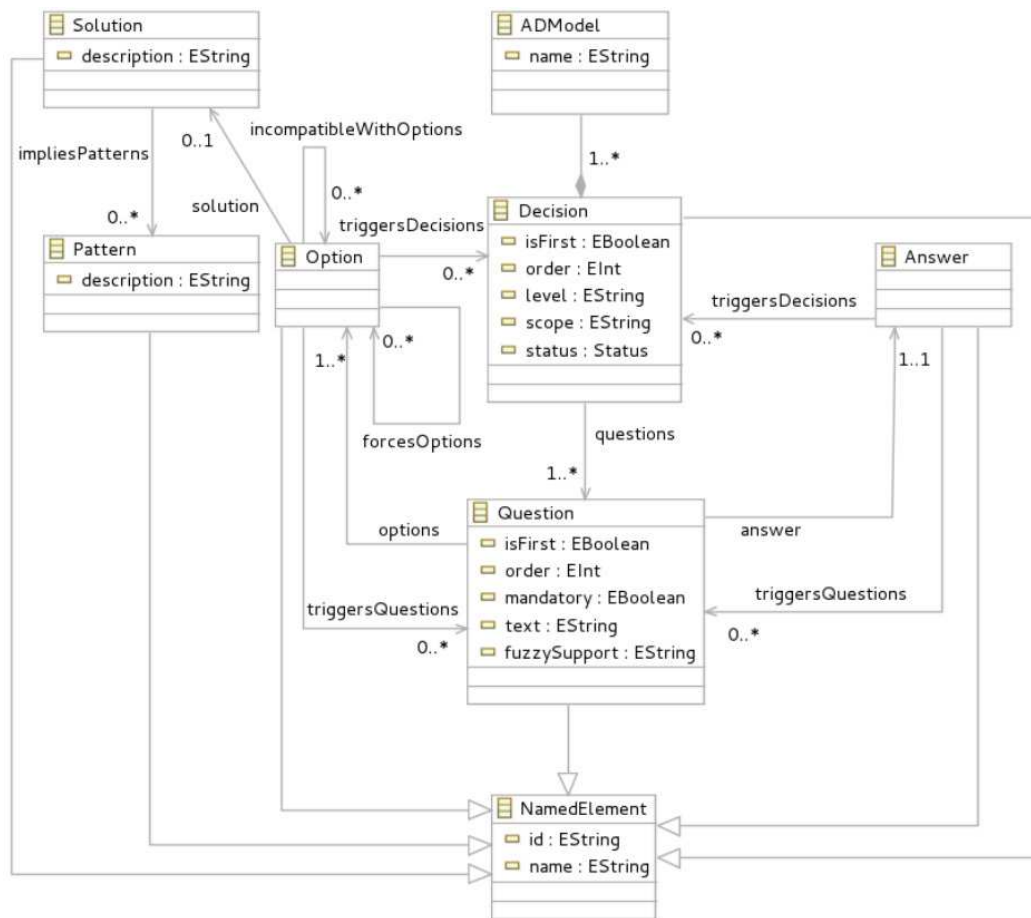


Figure 2-3: Architectural Decision Model

## 2.5 Installation

The Eclipse plug-ins have been developed and currently work stably in <http://www.eclipse.org/indigo/Eclipse 3.7.2> (Indigo). To install ADvISE go to Help → Install New Software... and add the ADvISE Update Site – <http://indenica.swa.univie.ac.at/public/advise> to download the latest version of the tool.

At this point you can select the features you want to install. Note that ADvISE requires FuzzDS and that AK Transformation Language requires ADvISE and VbMF. To save time you can download the Eclipse version with the Eclipse Modeling Tools from <http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/indigosr2> and afterwards download the latest versions of Xtext and Xtend2 from the corresponding Update Site. In order to install and use ADvISE you have to install the Eclipse Modeling Framework (EMF), Xtend2 and Xtext.

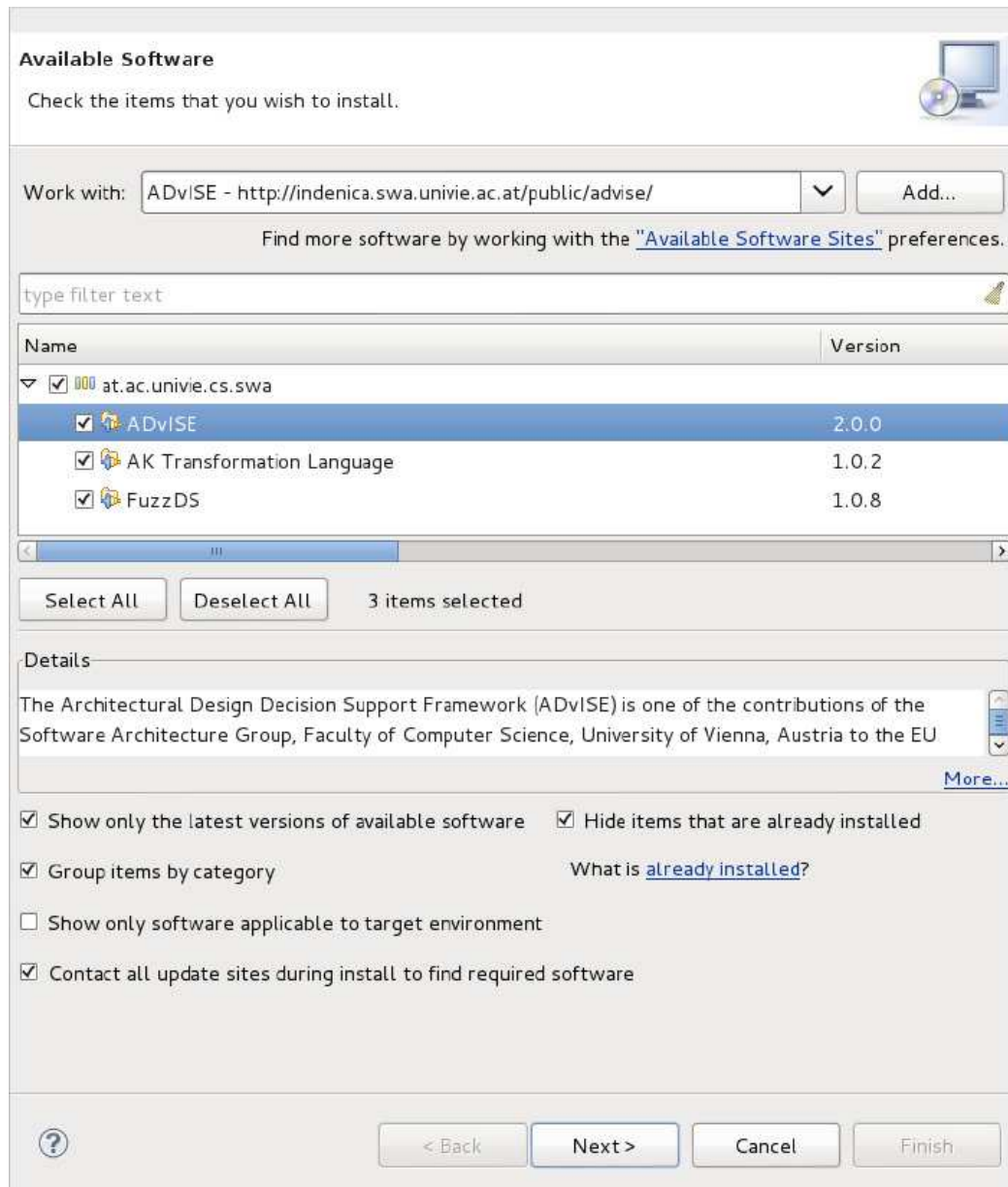


Figure 2-4: Installation from Update Site

## 2.6 User's Guide

In this guide, the functionality of the ADVISE tooling is presented step-by-step.

### 2.6.1 ADVISE

#### **ADvISE Perspective**

The ADvISE perspective (see Figure 9) contains all views and editors (parts) that are used when working with ADvISE and the tools integrated with it. To activate the perspective go to Window → Open Perspective → Other... → ADvISE.

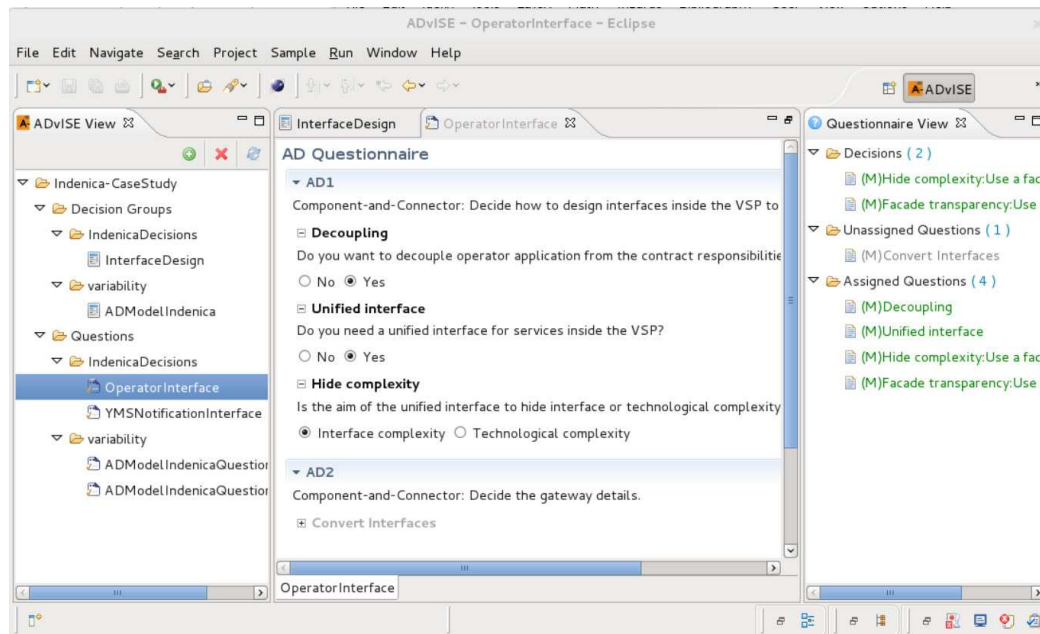


Figure 2-5: ADvISE Perspective

### ADvISE Views

ADvISE offers two types of views: the *ADvISE View* for the management of models and questionnaires and the *Questionnaire View* which gets synchronized when questionnaires for architectural decision making are filled in, and shows unassigned, assigned and made decisions in real-time. Questionnaires are generated from the reusable architectural decision models modeled using the ADvISE tool and contain questions with alternative options that can be selected or answer fields. For a detailed presentation of the questionnaires refer to Architectural Decision Questionnaires.

*Available actions provided in ADvISE View:*

1. Create new architectural decision model
2. Create new group for organizing architectural decision models
3. Rename group/architectural decision model/questionnaire
4. Delete group/architectural decision model/questionnaire
5. Refresh all projects
6. Validate architectural decision model
7. Generate Questionnaire (from architectural decision model)

*Questionnaire View:*

- Decisions group contains all made decisions
- Unassigned questions have not been answered yet

- Assigned questions have been already answered
- Questions indicated with (M) are mandatory
- Questions indicated with (O) are optional
- Questions in green color are activated
- Questions in gray color are deactivated

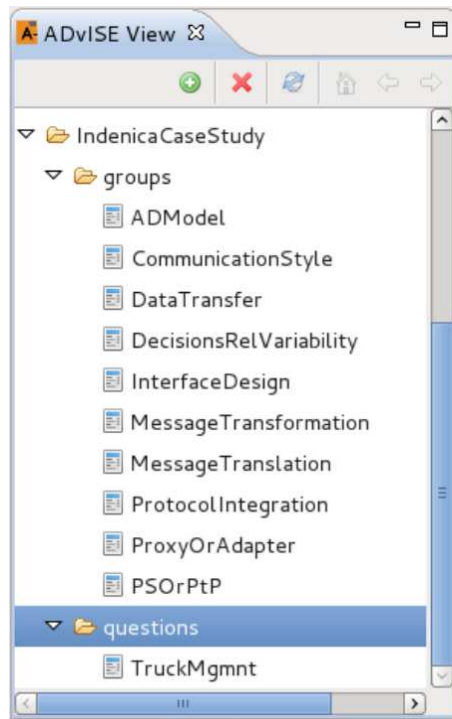


Figure 2-6: ADvISE View

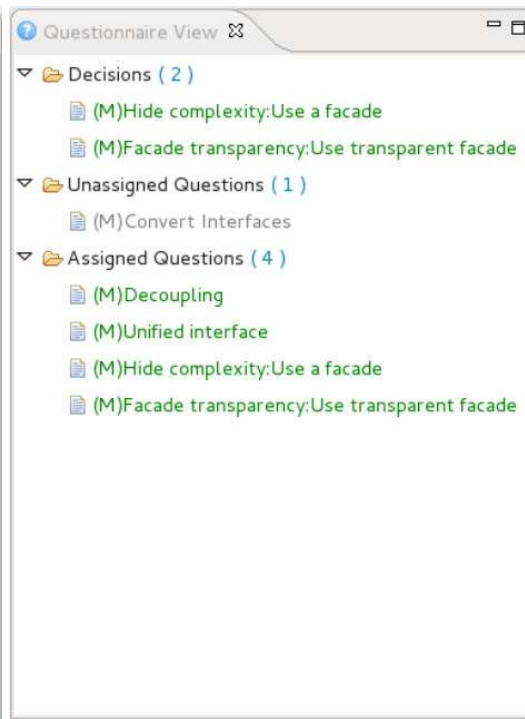


Figure 2-7: Questionnaire View




**Activate Views:** To activate the ADvISE and Questionnaire Eclipse views separately go to Window → Show View → Other... Under the group ADvISE activate the views ADvISE View and Questionnaire View.

### ADvISE Wizards

ADvISE also provides two wizards to create new ADvISE projects and ADvISE decision models.

**New ADvISE Project:** To create a new ADvISE project go to File → New → Other... → ADvISE wizards and select ADvISE Project. A new project with the following structure is created:

- project\_name
  - Decision Groups (root folder for architectural decision models)
    - \* ADModel (empty architectural decision model)
  - Questions (root folder for questionnaires)

**New ADvISE Decision Model:** To create a new ADvISE model go to File → New → Other... → ADvISE wizards and select Architectural Decision Model. An empty decision model (with extension .admodel) is created in the selected folder. The decision model editor contains 3 tabs: Architectural Decisions, Design Solutions and Design Patterns for editing the architectural decisions and the solutions and patterns related to them. Every time you edit your model (and the workspace resources are refreshed) the “Save” and “Save all” buttons get activated. Press “Save All” to save all the changes you have done so far or the “Undo”  button to undo the changes (one by one) after the last save. You can switch between horizontal and vertical view of your editor by using the buttons  and  respectively.

### Edit Design Patterns

Go to the Tab Design Patterns and press “Add”. A pattern with the name *NewPattern* will be created. You can give the *Name* and *Description* of the pattern and change your selection to see the changes in the list of patterns on the left. You can delete one or more patterns by selecting the pattern(s) and pressing “Delete”.

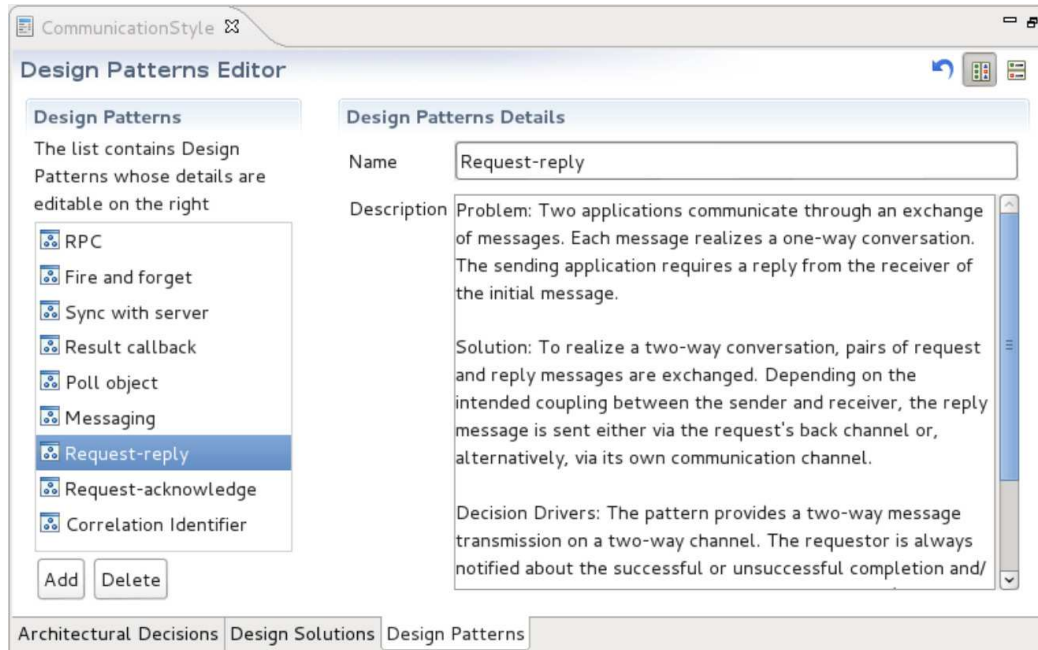




Figure 2-8: Design Patterns Tab

### Edit Design Solutions

Go to the Tab Design Solutions and press “Add”. A solution with the name *NewSolution* will be created. You can give the *Name* and *Description* of the solution and relate the solution to one or more Design Patterns. To add/remove related Patterns press the buttons  and  under the Patterns table respectively. After you change your selection you can see the changes in the list of solutions on the left. You can delete one or more solutions by selecting the solution(s) and pressing “Delete”.

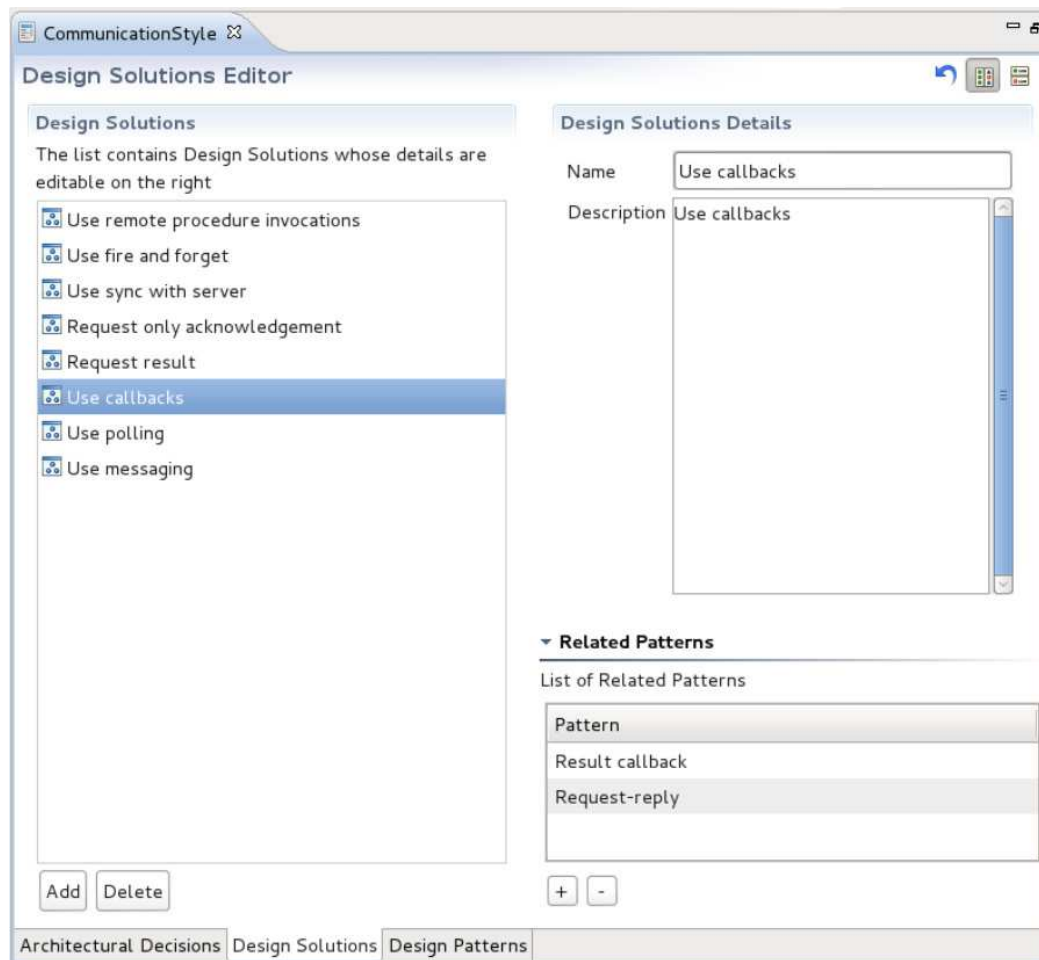




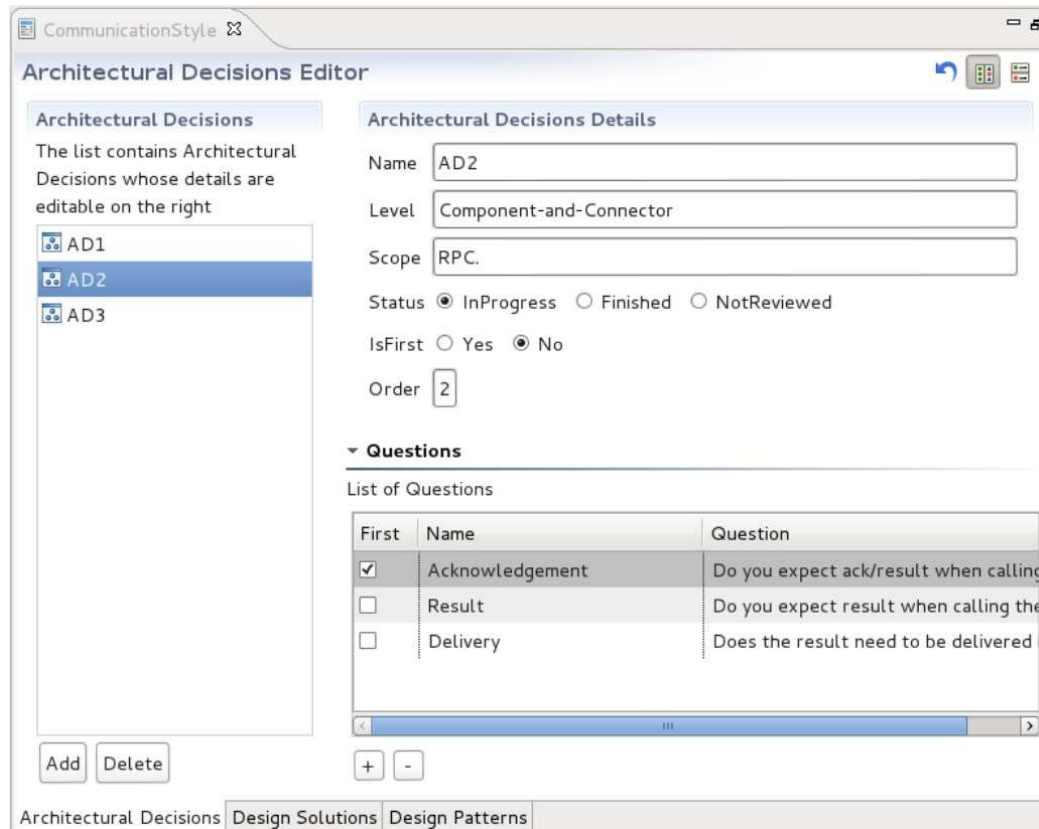
Figure 2-9: Design Solutions Tab

### ***Edit Architectural Decisions***

Go to the Tab Architectural Decisions and press “Add”. A decision with the name NewAD will be created. You can give the following information on the decision:

- Name: name of decision
- Level: design level for decision
- Scope: description of decision
- Status: status of decision (In Progress, Finished, Not Reviewed)
- IsFirst: indicates if it is the first decision to make before making follow-on decisions (i.e., it indicates the first decisions to be made in the generated questionnaire)
- Order: indicates the order of the decision as it will appear in the generated questionnaire

Each decision entails a list of questions that have to be answered in order to make a specific decision. To add/remove a question press the buttons  and  under the table of questions respectively. You can edit a question by double-clicking it. To set a question as first question (i.e., first question(s) to be answered for the corresponding decision in the generated questionnaire) select the checkbox “First”. You can delete one or more decisions by selecting the decision(s) and pressing “Delete”.



**Architectural Decisions Editor**

**Architectural Decisions**  
The list contains Architectural Decisions whose details are editable on the right

- AD1
- AD2**
- AD3

**Architectural Decisions Details**

Name:

Level:

Scope:

Status: ☒ InProgress ☐ Finished ☐ NotReviewed

IsFirst: ☐ Yes ☒ No

Order:

**Questions**

List of Questions



First	Name	Question
<input checked="" type="checkbox"/>	Acknowledgement	Do you expect ack/result when calling
<input type="checkbox"/>	Result	Do you expect result when calling the
<input type="checkbox"/>	Delivery	Does the result need to be delivered

Architectural Decisions | Design Solutions | Design Patterns

Figure 2-10: Architectural Decisions Tab

**Edit Questions:** You can add either a Question with Answer or a Question with Options.

A question expecting alternative options contains the following information:

- Name: symbolic name of the question
- Question: actual question
- mandatory/optional: check if question is mandatory or optional
- Options: press the buttons  and  to add/delete options respectively

When an option is selected the following get activated:

- Solution: select the solution that corresponds to the selected option (if exists)
- next Decisions: add/delete follow-on decisions
- next Questions: add/delete follow-on questions
- forces Options: add/delete options that are forced by the selection of this option
- incompatible Options: add/delete options that are incompatible with the selection of this option

The Fuzzy Decision Support field can be used to add fuzzy logic decision support (see 2.2.2 for more details) between alternative options. Only .fuzzypattern files can be selected.

A question expecting a free-text answer contains the following information:

- Name: symbolic name of the question
- Question: actual question
- mandatory/optional: check if question is mandatory or optional
- next Decisions: add/delete follow-on decisions
- next Questions: add/delete follow-on questions

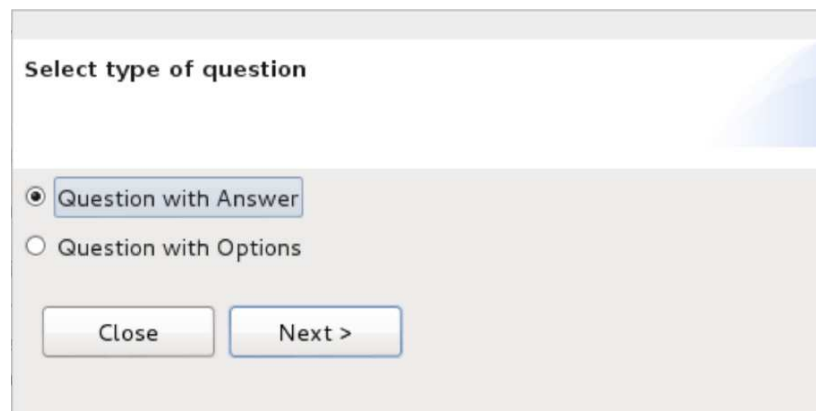
A screenshot of a software dialog box titled "Select type of question". The dialog has a light gray background with a blue gradient on the right side. It contains two radio button options: "Question with Answer" (which is selected, indicated by a filled circle) and "Question with Options" (which is unselected, indicated by an empty circle). Below the radio buttons are two buttons: "Close" and "Next >". The "Next >" button is highlighted with a blue border.

Figure 2-11: Select the type of question

Edit the question containing options. Select an option to edit details.

Name

Question

☒ mandatory ☐ optional

Option

No

Yes

+ -

Solution

next Decisions

next Questions

Result

Delivery

forces Options

incompatible Options

Fuzzy Decision Support

< Back Close OK

Figure 2-12: Question expecting alternative options

**Edit the question containing free-text answer**

Name

Question

☒ mandatory ☐ optional

next Decisions

next Questions

+ -

+ -

< Back Close OK

Figure 2-13: Question expecting free-text answer

### ***Architectural Decision Questionnaires***

ADvISE framework uses model-to-model transformation to generate Questionnaires from the architectural decision models. To generate a questionnaire from an architectural decision model right-click on the model in the ADvISE View and click “Generate Questionnaire”. Choose the name and location of the generated questionnaire. This questionnaire will be used to make a decision for a specific problem at hand and is based on the reusable architectural decision model.

Double-click on the generated questionnaire to edit the questionnaire with the corresponding editor. Answering the questionnaire contains clicking on the available options to the questions and filling-in the required information. Whenever an option is selected you are guided to the next questions you have to consider for making your decision, as well as to the follow-on decisions. Decisions and questions that are grayed-out can not be answered at this stage. The Questionnaire View shows which options have been decided and which are left open for decision and is synchronized with the questionnaire. Questions indicated as (M) or (O) are mandatory or optional respectively. Changing a previous option may lead to invalidated follow-on questions and options which are recalculated in real-time from the constraints introduced in the respective architectural decision model (incompatible with, etc.). Questions that get invalidated by decision changes have to be answered again.

The screenshot displays the INDENICA D1.3.2 questionnaire interface. It is divided into three main panes. The left pane, titled 'AD Questionnaire', contains three sections: AD1, AD2, and AD3. AD1 includes questions about decoupling, unified interface, and hide complexity. AD2 includes a question about convert interfaces. AD3 includes a question about facade transparency. The right pane, titled 'Questionnaire View', shows a tree structure of decisions and assigned questions. The bottom pane, titled 'OperatorInterface', shows the current design context.

Figure 2-14: Example of a questionnaire

**Save/Reset a Questionnaire:** To save a questionnaire press on button “Save”, so you can re-open and re-edit your questionnaire. Press “Reset” to discard all answers and re-start with the questionnaire.

**Fuzzy Logic Support:** If a question has been configured to use fuzzy decision support a button Fuzzy Decision Support will appear next to the question. Press the button to use the wizard that will guide you through the decision making (see subsec:fuzzds for more details).

**Generate Documentation:** Press the “Export” button to generate documentation (in HTML format) of the made decisions. The exported documentation is based on the answers that have been given to the questionnaires and the solutions that are implied by the selected options and can be also edited manually.

## 2.6.2 FuzzDS

FuzzDS aims to provide semi-automated support for specific recurring ADDs and resolve their inherent uncertainty. Rather than creating a new design from scratch, it automates the decision making for design problems that emerge repetitively in a specific context. Our purpose is to cover the whole design space for a design situation at hand consisting of generic, as well as technology-specific decisions. To address uncertainty we use Fuzzy Logic [20], which allows the numerical encoding of the vague linguistic values software engineers use to describe requirements, as well

as forces and consequences of reusable ADDs. Key concepts of Fuzzy Logic are fuzzy sets and their membership functions, which express *degrees of membership* spanned in the interval  $[0,1]$  for the elements of the fuzzy sets. The linguistic values can be interpreted using fuzzy sets which get mapped to overlapping membership functions (e.g., gaussian, trapetzoidal, etc.). For example, the property *performance* could be described as *high*, *medium* or *low* and these linguistic values can be mapped to overlapping membership functions, as shown in Figure 2-15.

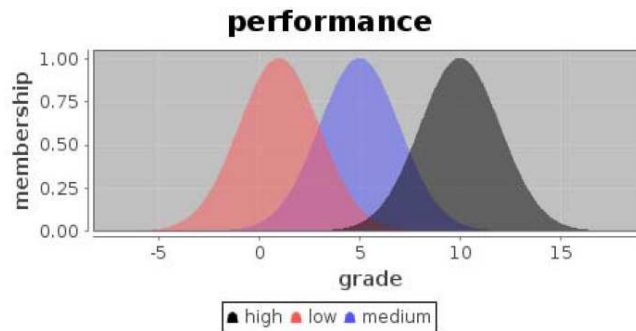


Figure 2-15: Gaussian membership functions for 3 linguistic values of property performance

Figure 2-16 presents an overview of our approach, namely the participating tools and roles. We distinguish between two stakeholder roles: *software architect (expert)* and *software architect (user)*. That is, different levels of experience are expected for architects who create the fuzzy logic models and users of our approach. The software architects (experts) use the *Fuzzy Decisions Models Editor* (a textual DSL editor) to capture architectural knowledge. A decision model contains alternative design solutions along with their properties and quality attributes and a set of expert IF–THEN fuzzy rules that guide the design decisions. From these decision models we derive specialized fuzzy decision models in which domain and technology specific knowledge can be included. Both kinds of decision models get stored in a *Fuzzy Decision Model Repository* for reuse by a *Fuzzy Inference System*. The software architects (users) use the *Requirements Editor* to give the desired requirements in crisp values using a grading system (e.g., 1–10) for fuzzy input variables like *performance* and *reliability* and binary values (i.e., 0, 1) for variables that accommodate only two values (e.g., Yes, No) like *supports acknowledgment*.

The *Fuzzy Inference System* returns the appropriate design alternatives and their ranking for the given requirements by combining and evaluating the fuzzy rules already defined in the fuzzy models. The list of the best-fitting design solutions is supposed to be used as a decision aid for the software architect who makes the final decision. After that, the input requirements and the inferred design solutions can be synthesized to produce *ADD Documentations*.

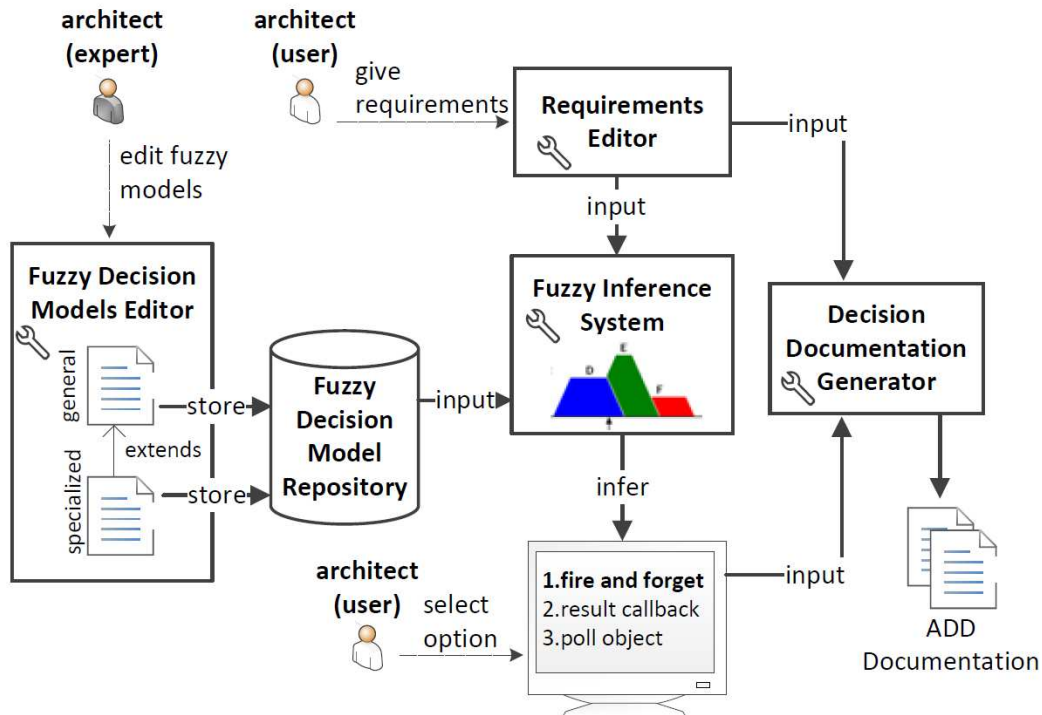


Figure 2-16: Fuzzy Logic Based Approach for Supporting Architectural Decision Making

For more information about editing and using fuzzy models, as well as the integration of FuzzDS with ADvISE please refer to the following manual <http://indenica.swa.univie.ac.at/public/advise/FuzzDS.pdf>.

### 2.6.3 AK Transformation Language

The AK Transformation Language works as a bridge between ADvISE for modeling architectural decisions and supporting architectural decision making and VbMF for modeling architectural views. It consists of simple and compound transformation actions that enact on the component views modeled in VbMF in order to add, delete, modify its components, connectors, properties, etc.

For a complete reference of the AK Transformation Language, usage examples and its integration with ADvISE please refer to the following manual <http://indenica.swa.univie.ac.at/public/advise/AK Transformation Language.pdf>.

### 3 A Tool-Based Approach to Integrated Variability- and Architecture-Decision-Making

This section describes the realization of the integration of variability and architecture decision making, which is now possible through the combination of the ADvISE and Easy-Producer tool sets. This follows the discussion, which is provided in more detail in Section **Error! Reference source not found.**. However, here we describe the workflow with a focus on tool usage.

The description follows these steps:

1. Model Variability
2. Model Architectural Decisions
3. Define Mapping
4. Resolve Variability
5. Resolve Remaining Architectural Decisions

We also describe the usage of the tool accordingly. However, prior to starting this, we need to prepare the tool environment. Thus, we assume that the Easy-Producer tool and the ADvISE tool are installed correctly in the Eclipse environment.

#### 3.1 Model Variability

The first step is to model the variability relevant to our joint example. The simplest way to do so is to create a corresponding IVML-description. Thus, we create a new Easy-Producer project (*File -> New -> Project... -> EASy-Producer -> New EASy-Producer Project*). We will choose “PL\_WMS” as the name of the new project in the wizard and click the *Finish*-button. The creation of a new product line project automatically opens the *Product Line Editor*. However, this editor does not support the definition of variability. We will open the *IVML Editor* instead by double-clicking the IVML-file, located in the *EASy*-folder of the *PL\_WMS*-project. We model as variability the variability of a warehouse management system with four variabilities:<sup>8</sup>

- PickingRate
- PartialPalletStrategy
- StaplerCraneStrategy
- UIDeviceType

The result is shown in Figure 3-1. In lines 5-8 each variability is defined as an individual enumeration along with their resolutions in curly brackets. These types are used to define the variability decisions “VP1” to “VP4” in lines 10-13. The variability

---

<sup>8</sup> Of course, four variabilities is a very small number, this is due to the fact that the example is explicitly artificial to demonstrate interoperability of ADvISE and Easy-Producer in a nutshell.

decisions “VP1”, “VP3”, and “VP4” are supposed to be bound at design-time (lines 15-16), while “VP3” is left open until runtime (lines 17-19). Finally, we save this definition (Ctrl + S).

```

1 project PL_WMS {
2
3     version v0;
4
5     enum PickingRateType {high, medium, low};
6     enum PartialPalletStrategyType {highSpeed, optimalReduction};
7     enum StaplerCraneStrategyType {single, multiple};
8     enum UIDeviceType {computer, mobile};
9
10    PickingRateType      VP1;
11    PartialPalletStrategyType VP2;
12    StaplerCraneStrategyType VP3;
13    UIDeviceType         VP4;
14
15    enum BindingTime {designTime, compileTime, initTime, runTime};
16    attribute BindingTime bindingTime = BindingTime.designTime to PL_WMS;
17    assign (bindingTime = BindingTime.runTime) to {
18        PartialPalletStrategyType VP2;
19    }
20 }

```

Figure 3-1: Textual view of the variability model of the WMS example modeled in EASy-Producer

The decisions defined in the IVML-file are also shown in the *IVML Configuration Editor*-tab of the *Product Line Editor* to determine a specific configuration in an interactive manner. We will describe the definition of a configuration in EASy-Producer in Section 3.4 in detail.

### 3.2 Model Architectural Decisions

As a next step we model the architectural decisions, relevant to the WMS platform.

First of all, we need to define the design space that provides the basis for making architectural decisions at the product line and product level. An example of an architectural decision at product line is the type of Interprocess Communication (IPC) that will be used (fix or variant). An example of an architectural decision at product level is the type of IPC software that will be used (open source, medium price, or very expensive). Given the information about the architectural design alternatives and their drivers (forces and consequences) we model the reusable architectural decisions using questions, options and criteria. Please refer to Section 2 for more details about modelling reusable architectural decisions and to Section 6 for more examples of architectural decisions at product line level as well as at product level.

In Figure 3-2 the user interface for modelling the architectural decisions using the ADvISE tooling is shown.

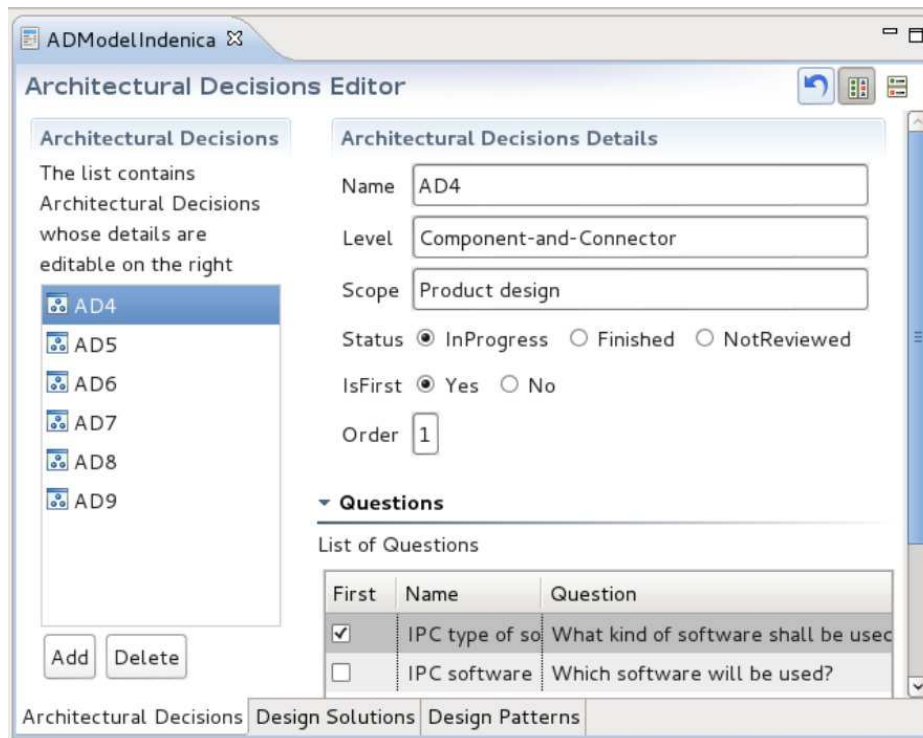


Figure 3-2: Reusable Architectural Decision Model

### 3.3 Define Mapping

The ADVISE tooling provides a launcher for modifying actual architectural decisions according to constraints introduced by variability decisions.

In order to run the Synchronize Architectural Decisions Launcher the following arguments are needed:

- *EasyProducer PL Project*: the project that contains variability decision models that can affect current architectural decisions
- *Mapping File*: an XML file that describes which variants are mapped to which architectural decisions and the kind of the dependency (e.g., enforces) – see listing of Figure 3-4
- *Questionnaire File*: ADVISE questionnaire used for architectural decision making
- *Output*: output questionnaire file where architectural options may be invalidated according to the selected variants

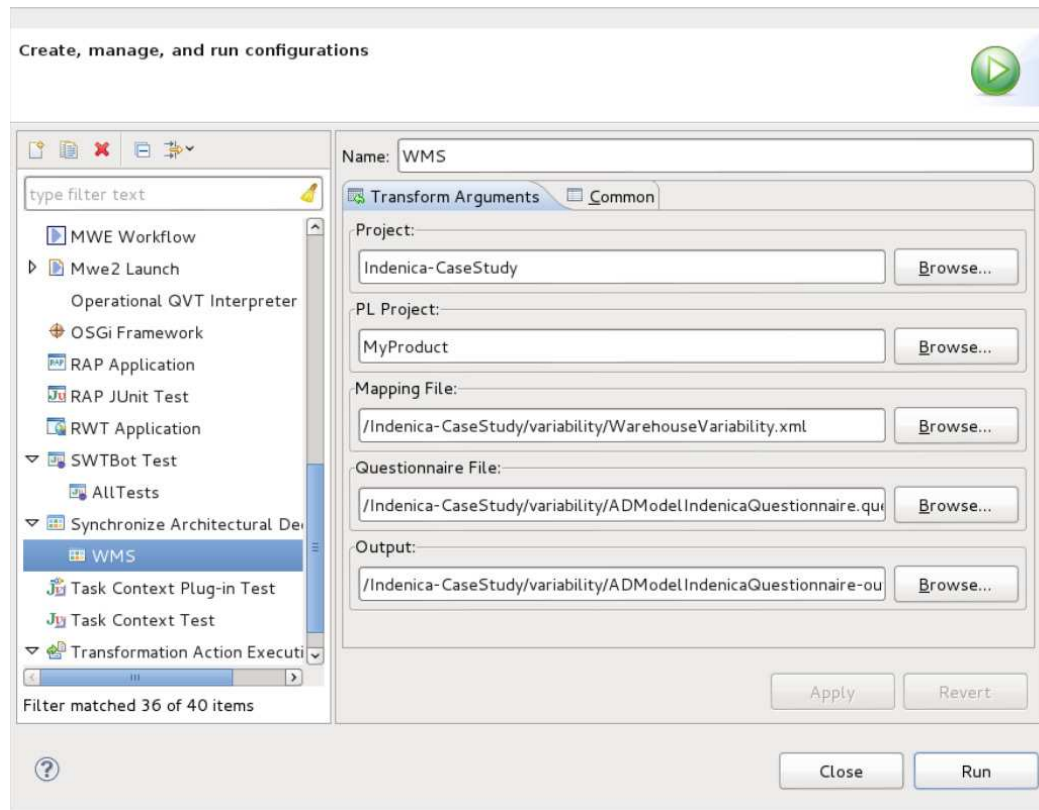


Figure 3-3: Synchronize Architectural Decisions Launcher

```

1 <mappings>
2   <aModel>ADModelIndenica</aModel>
3   <vModel>PL_WMS</vModel>
4   <vp name="VP1">
5     <relation type="excludes">
6       <vd id="PickingRateType.medium"/>
7       <add id="AD4.IPC type of software.IPC open source"/>
8     </relation>
9   </vp>
10   ...
11 </mappings>

```

Figure 3-4: Excerpt of the mapping between variability and architectural decisions

### 3.4 Resolve Variability

The next step after the definition of the mapping between variants and architectural decisions is to resolve the variability. This is done by assigning values to the variability decisions defined in Section 3.1. For this purpose, EASy-Producer provides the interactive IVML Configuration Editor as part of the Product Line Editor (Right-click on the project -> Edit Product Line -> IVML Configuration Editor). This interactive view guides the user through the configuration of a specific instance. For example, in Figure 3-5 the possible values of decision “VP2” are shown in a drop-down menu limiting the user to select only one of the two (valid) alternatives. Such selections also change the status of a decision from unassigned to assigned. As the

values are taken as defaults, they need to be set as final (freeze) so they have an effect in an instantiation. (The concept of explicitly freezing decisions is described in detail in Deliverable D2.1.) Finally, we save the configuration (Ctrl + S).

The configuration given in Figure 3-5 defines the product from a variability perspective. Based on the connections to the architectural decisions (cf. Section 3.3) a number of architectural decisions can be automatically derived and further ones can be constrained.

**Product Configuration Editor: PL\_WMS**

► **Filtering Options**

Decision Name	Current value	+	-	Freeze
✓ VP1	medium			
▲ VP2	highSpeed			freeze
✓ VP3	highSpeed			
✓ VP4	optimalReduction			

Figure 3-5: Providing values to the variability decisions of the WMS example

### 3.5 Resolve Remaining Architectural Decisions

In our running example, the mapping we defined in Figure 3-4 (interdependence between variability and architectural decisions) and the configuration we made as shown in Figure 3-5 will result in the deactivation of an architectural option. In particular, the selection of the variant “PickingRate-medium” will cause the option “IPC open source” to be deactivated in the related questionnaire. This is shown in Figure 3-6. At this point, further architectural decisions –related to variability or not—can be made in order to design the product architecture.

**AD Questionnaire**

## ▼ AD4

Component-and-Connector: Product design

☐ **IPC type of software**

What kind of software shall be used for managing IPC?

☒ IPC open source   ☐ IPC medium price   ☐ IPC very expensive☐ **IPC software**

Which software will be used?

☐ Software 1   ☐ Software 2

(a) Questionnaire excerpt for product level architectural decisions

**AD Questionnaire**

## ▼ AD4

Component-and-Connector: Product design

☐ **IPC type of software**

What kind of software shall be used for managing IPC?

☒ IPC open source   ☐ IPC medium price   ☐ IPC very expensive☐ **IPC software**

Which software will be used?

☐ Software 1   ☐ Software 2

(b) Architectural option deactivated due to variability decision

Figure 3-6: Architectural Decision Making

**3.6 Wrap-Up**

As a result of the previous activities, we derived a set of customized architectural decisions, which correspond to the specific variability as described in the variability model, introduced in Section 3.1 and the variability resolutions introduced in Section 3.4. Of course, the variability does not completely determine the architecture. Rather it constrains the range of possible architectural decisions. In the last step, the remaining open architectural decisions are made (Section 3.5).

## 4 Summary and Conclusion

Customizing service platforms is a complex activity, which either requires significant understanding by the development personnel of the existing implementation or it requires support to aid the personnel in making these decisions in an informed way. In this deliverable, we focussed on the second way: how to support developers in dealing with the complexity of sophisticated decision making in the context of developing service platforms. In relation to the previous Deliverable (D1.3.1) we focused on a somewhat narrower range of decision making, but with a considerably more ambitious goal: creating a tool environment that supports complex architectural decision making and is able to support the integration of this architectural decision making with variability. Both has been developed as part of this work package, in addition, the integration allows to connect (and exploit) the results of WP2. The main part of the deliverable is the running prototype; hence we focused in our description on a description of how to effectively use it.

## 5 References

- [1] A. Jansen and J. Bosch, "Software Architecture as a Set of Architectural Design Decisions," in *5th Working IEEE/IFIP Conf. on Software Architecture (WICSA)*, Pittsburgh, PA, USA, 2005, pp. 109–120.
- [2] N. B. Harrison, P. Avgeriou, and U. Zdun, "Using Patterns to Capture Architectural Decisions," *IEEE Software*, vol. 24, pp. 38–45, 2007.
- [3] J. Tyree and A. Akerman, "Architecture Decisions: Demystifying Architecture," *IEEE Softw.*, vol. 22, pp. 19–27, 2005.
- [4] P. Kruchten, "An Ontology of Architectural Design Decisions," in *Proceedings of 2nd Workshop on Software Variability Management*, 2004.
- [5] P. Kruchten, R. Capilla, and J. C. Dueñas, "The Decision View's Role in Software Architecture Practice," *IEEE Softw.*, vol. 26, pp. 36–42, Mar. 2009.
- [6] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998.
- [7] F. Bachmann, L. Bass, M. Klein, and C. Shelton, "Designing software architectures to achieve quality attribute requirements," *Software, IEEE Proc.*, vol. 152, pp. 153–165, Aug. 2005.
- [8] O. Zimmermann, T. Gschwind, J. Küster, F. Leymann, and N. Schuster, "Reusable architectural decision models for enterprise application development," in *Proceedings of the Quality of software architectures 3rd international conference on Software architectures, components, and applications*, 2007, pp. 15–32.
- [9] O. Zimmermann, J. Grundler, S. Tai, and F. Leymann, "Architectural Decisions and Patterns for Transactional Workflows in SOA," in *Proceedings of the 5th international conference on Service-Oriented Computing (ICSOC)*, 2007, pp. 81–93.
- [10] O. Zimmermann, U. Zdun, T. Gschwind, and F. Leymann, "Combining Pattern Languages and Reusable Architectural Decision Models into a Comprehensive and Comprehensible Design Method," in *7th IEEE/IFIP Conf. on Software Architecture*, 2008, pp. 157–166.
- [11] M. Shahin, P. Liang, and M. R. Khayyambashi, "Architectural design decision: Existing models and tools," in *IEEE/IFIP Conf. on Software Architecture/European Conference on Software Architecture (WICSA/ECSA)*, 2009, pp. 293–296.
- [12] W. Bu, A. Tang, and J. Han, "An analysis of decision-centric architectural design approaches," in *Proceedings of the ICSE Workshop on Sharing and Reusing Architectural Knowledge (SHARK)*, 2009, pp. 33–40.
- [13] A. Peng Jansen Liang and P. Avgeriou, "Knowledge Architect: A Tool Suite for Managing Software Architecture Knowledge," 2009.
- [14] A. Jansen, J. V. D. Ven, P. Avgeriou, and D. K. Hammer, "Tool Support for Architectural Decisions," in *Proceedings of the 6th working IEEE/IFIP Conference on Software Architecture*, 2007.

- [15] R. Capilla, F. Nava, J. Montes, and C. Carrillo, "ADDSS: Architecture Design Decision Support System Tool," in *ASE*, 2008, pp. 487–488.
- [16] M. A. Babar and I. Gorton, "A Tool for Managing Software Architecture Knowledge," in *Proceedings of the Second Workshop on SHaring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent*, 2007, p. 11–.
- [17] D. Ameller, O. Collell, and X. Franch, "ArchiTech: Tool support for NFR-guided architectural decision-making," in *RE*, 2012, pp. 315–316.
- [18] N. Schuster, O. Zimmermann, and C. Pautasso, "ADkwik: Web 2.0 Collaboration System for Architectural Decision Engineering," in *SEKE*, 2007, pp. 255–260.
- [19] A. MacLean, R. Young, V. Bellotti, and T. Moran, "Questions, Options, and Criteria: Elements of Design Space Analysis," *Human-Computer Interaction*, vol. 6, pp. 201–250, 1991.
- [20] L. A. Zadeh, "Fuzzy sets," *Information and Control*, vol. 8, pp. 338–353, 1965.

## **6 Appendix: Conceptual Integration of Variability Decision Making and Architecture Decision Making**

*This section describes in more detail the background of the integration of ADvISE and EASy-Producer. Both tools are combined to integrate variability decision making with architectural decision making.*

*This paper has been submitted to the Journal of Universal Computer Science (J.UCS), Special Issue: Software Components, Architectures and Reuse.*

*So far, we did not receive feedback regarding the acceptance of the paper.*

# On the Interdependence and Integration of Variability and Architectural Decisions

**Ioanna Lytra, Huy Tran, Uwe Zdun**

(Faculty of Computer Science  
University of Vienna, Austria  
firstname.lastname@univie.ac.at)

**Holger Eichelberger, Klaus Schmid**

(Software Systems Engineering, Institute of Computer Science  
University of Hildesheim, Germany  
lastname@sse.uni-hildesheim.de)

**Georg Leyh**

(Siemens AG  
Erlangen, Germany  
georg.leyh@siemens.com)

**Abstract:** In software product line engineering, the design of assets for reuse and the derivation of software products involves low-level and high-level decision making. In this process, two major types of decisions must be addressed: variability decisions, i.e., decisions made as part of variability management, and architectural decisions, i.e., fundamental decisions to be made during the design of the architecture of the product line or the products. In practice, variability decisions often overlap with or influence architectural decisions. For instance, resolving a variability may enable or prevent some architectural options. This inherent interdependence has not been explicitly and systematically targeted in the literature, and therefore, is mainly resolved in an ad hoc and informal manner today. In this paper, we discuss possible ways how variability and architectural decisions interact as well as their management and integration in a systematic manner. For this, we leverage two existing tools for variability management and architectural decision support to demonstrate the (semi-)automated integration between the two types of decisions. We apply and evaluate our approach in a case study from the industry automation area and discuss the lessons learned.

**Key Words:** variability decisions, architectural decisions, software product lines, product derivation

**Category:** D.2.2, D.2.10, D.2.11

## 1 Introduction

Variability management and architecture-centric development are fundamental aspects of software product line engineering [1, 22]. Variability management aims at the explicit modeling of differences (variabilities) among the products that can be derived from a product line and, in particular, the interdependencies among individual variabilities. From a variability management perspective, the software architecture of a product line describes the design of all products in a product line in terms of reusable assets. This requires, first of all, that commonalities and variabilities among the different products of a given product line are identified. The aim of Software Product Line Engineering

(SPLE) is to create a single architecture for a range of related products that can be tailored and customized to meet the requirements of the derivable products, for instance, imposed by different customers. This single architecture is often called the *reference architecture* of the product line and may contain variabilities to represent the difference among the products [22]. Finally, each product has its own architecture derived from the reference architecture.

From an architectural perspective, variabilities may reflect different architectural options considered during the design of the product line that are independent of the products' features. Bachmann and Bass pointed out two causes of variability in the software architecture of a product line: (1) product line architectures encompass a collection of different alternatives that must be resolved during product configuration, and (2) at design time multiple alternatives may exist and need to be captured [2]. Other approaches consider product line architectures as a set of architectural decisions organized by the features in a feature model and the product architecture as a subset of the decisions associated with the chosen product features [29].

Currently in research, variability management and architecture design are mostly treated as separate activities. For their respective needs, product line and architecture communities use a variety of methods and tools for modeling, documenting, and making specific types of decisions. The product line community has mainly adopted feature models (e.g., [21]) and decision models (e.g., [36]) to specify variability. The software architecture community has exploited techniques from variability modeling (e.g., COVAMOF [39]) and used architectural decision modeling (e.g., [20]) to describe variabilities and connect them to quality attributes [44]. Existing variability management approaches focus on describing variabilities in a product line and managing their impact on the derived products. On the other hand, existing architectural design and decision support tools [37] cover various architectural aspects, views, and reasoning of decisions but lack adequate support for interaction with variability decisions. To our best knowledge, the interdependency and integration of variability decisions and architectural decisions have neither been studied nor addressed in a systematic way, yet. This work intends to fill this gap and addresses tool support for the semi-automated integration of the two types of decisions.

Based on a motivating case, we will discuss the interdependence of variability decisions and architectural decisions in the development of the product line and the derived products. For modeling product line variability, we opt for using variability decision models [35, 36]. For assisting and capturing architectural decisions, we use reusable architectural decision models designed for resolving recurring design issues (such as [23, 45]). The dependencies in the variability decision model are expressed by constraints. By defining a mapping of the variability model onto the architectural decision model, we explicitly enable the resolution of variability decisions onto the architectural decisions. In our approach, variability and architectural decision options of the reference architectures of product lines and the products' architectures are taken into

account. Further, we propose integrated tool support for the management and harmonization of both types of decisions. The integrated tool incorporates and extends the functionality provided by two existing tools: EASy-Producer<sup>1</sup> for variability management and ADvISE<sup>2</sup> for assisting architectural decision making. We apply our approach in an industrial case study from the warehouse automation domain and discuss the results and lessons learned in this context.

The remainder of the paper is structured as follows. In Section 2, we briefly present the background and terminology for variability decisions and architectural decisions in product line engineering, as well as the implicit relations of both kinds of decisions documented in the literature. In Section 3, we introduce an industrial case study and discuss variability and architectural decision interdependencies in this context. The details of our proposal and the related prototypical tool support are presented in Section 4 and Section 5, respectively. We discuss the results of our evaluation in Section 6. In Section 7, we compare our approach to related work and, finally, in Section 8 we conclude and outline future work.

## 2 Background

In this section, we provide some background definitions that are relevant to this paper. Based on this we will also provide a basic discussion of the dependencies between variability decisions and architectural decisions.

### 2.1 Product Line Engineering

To understand product line engineering it is important to notice a fundamental characteristic: the separation between development at the level of the product line as a whole and the level of an individual product. This distinction is typically referred to as the two-lifecycle model [22]. The two lifecycles are usually referred to as *domain engineering* and *application engineering*, respectively. However, we will mostly use the simplifying terms *product line level* and *product level* in this paper. At the product line level, all engineering activities that are relevant to a range of products—the product line—are performed. The logically first step is to determine what variability needs to be supported by the product line as a whole. As a basis for this, the scoping activity identifies which variability should actually be supported in a reusable manner [30, 31]. Together with domain analysis, a precise model of the product line is derived from this. It should be noted that this excludes product-specific parts from further consideration. Variability decisions are captured in a variability (decision) model, which represents all variabilities (differences among the individual products) at an abstract level. Typically,

<sup>1</sup> <https://www.uni-hildesheim.de/fb4/institute/ifi/software-systems-engineering-sse/forschung/projekte/easy-producer/>

<sup>2</sup> [http://swa.univie.ac.at/Architectural\\_Design\\_Decision\\_Support\\_Framework\\_\(ADvISE\)](http://swa.univie.ac.at/Architectural_Design_Decision_Support_Framework_(ADvISE))

a variability model includes constraints among the individual variabilities in order to restrict the variability space, i.e., to make explicit which instances of the model describe a valid product. In later stages, the variability model is used to derive a valid product configuration for instantiation.

The reference architecture defines the realization of the product line, i.e., any decisions made in the reference architecture will be available in all products. The reference architecture may also contain variability in the sense that some architectural decisions are not finally taken for the product line as a whole, but several resolutions remain possible for different variants. Thus, the fundamental potential architectural decisions are determined at the product line level. Besides, some architectural decisions may also be introduced as part of product-specific parts of the system.

The development of individual products is commonly referred to as *application engineering*. We will use the terminology *product level* to indicate artifacts that are specific to products. Specific variants are determined that are in accordance with the variability model on the product line level. Similar to the product architecture, the corresponding architectural decisions that have not yet been defined at the product line level are made. It may also happen at the product level that product requirements are not fully covered by the product line. Thus, one of two situations may occur: (a) the additional requirements can be covered as product-specific functionality as they interfere only little with the parts covered by the product line, and (b) there is a strong impact (e.g., the change relates to some variability, but would require a variability option which is not available). In the latter case, it is necessary to re-evaluate the existing variability. This may lead to introducing new variabilities to the variability model; likewise, the architecture needs to be extended.

## 2.2 Variability Decisions and Architectural Decisions

When creating variability models or reference architectures, a large number of decisions must be made, such as how to model a certain variability or which design pattern provides adequate support for particular variabilities. Other decisions are left open to be determined at product derivation time. We refer to the decisions taken as part of variability management activities as *variability decisions*. Further, we will call the decisions related to the software design of the architectures of the product line and the products *architectural decisions*.

Conceptually, variability decisions and architectural decisions may pose distinctive parts and overlapping parts. Variability decisions are any decisions that describe differences among different products in a product line and are relevant to reuse (i.e., excluding product-specific aspects). Typically, variabilities are described in terms of optional (yes-no), alternative (one-out-of-many), or multiple (many-out-of-many) selections [22]. An architectural decision is the result of the evaluation of alternative design options in terms of architectural elements such as patterns, components, or connectors and the selection of the best-fitting solution. This may happen both at the product line and product level.

At product line level architectural decisions can also be kept open and be postponed to the product level. These open decisions become variabilities. Architectural decisions at different levels of granularity are usually taken first in the early stages of design. Later, throughout the software development process, as well as the maintenance and evolution of a product line, new architectural decisions may need to be considered and existing decisions might have to be revised.

### **2.3 Relation between Variability Decisions and Architectural Decisions**

As mentioned above, a variability decision model is often established at the product line level. This model describes the variability that is relevant to the product line as a whole and may contain decisions on various levels of abstraction starting from very abstract domain-related decisions to rather fine-grained technical decisions.

The product line architecture must be designed such that it can cover the desired variabilities. During this process, various architectural decisions for the design of the reference architecture must be made. Other architectural decisions can be left open and resolved later during the derivation of products, i.e., when variabilities are resolved respectively.

This is not a one-time process, as both engineering activities at the product line level may provide feedback to requirements engineering or variability management [8]. In particular, the variability that should be supported by the product line needs to be re-examined whenever new products are developed and may introduce the need for new variabilities. A variability in a system can be implicit (present at higher levels of abstraction), designed (explicit) and bound (to a particular variant) [42].

In general, there are two potential options for dealing with such feedback from later stages to variability modeling: (a) by using a single variability model that captures variability on all levels in a homogeneous way, and (b) by using a so-called staged configuration approach in which multiple models are used, and information in one model is used to configure the subsequent level [10]. In practice, especially in commercial tools, usually an approach with a single central variability model is used to simplify management and development.

## **3 Case Study**

As a case study for the proposed approach, we leverage a software product line of warehouse management systems. Based on the product line, custom-made software products for specific warehouses can be derived. The product line targets automated warehouses only; that is, goods in a warehouse are moved on pallets by conveyors or stapler cranes.

Usually, a warehouse management system is accompanied by an Enterprise Resource Planning (ERP) system that handles all financial aspects of warehouse transactions, and a base automation system that directly controls the conveyors and stapler cranes of a warehouse. This overall system architecture is typically layered, consisting

of a Resource Planning Layer, a Warehouse Management Layer and a Basic Automation Layer (see Figure 1).

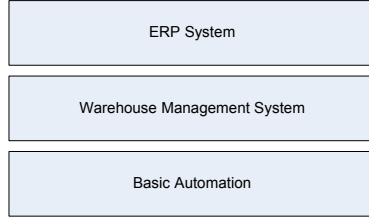


Figure 1: Layered architecture of a warehouse management system

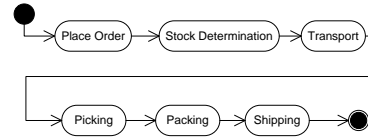


Figure 2: Goods out process of a warehouse

The most important business process of a warehouse is “Order Processing” as illustrated in Figure 2. The process is triggered when a client orders some goods stored in the warehouse. Next, the ERP system notifies the warehouse management system about what shall be delivered (Place Order). The warehouse management system then maps the orders to boxes of goods that are stored in the warehouse management system (Stock Determination). Transport orders are sent to the base automation. The base automation will transport the boxes to a picking station (Transport). There, a human worker picks up the goods that are specified by the order (Picking). After that, the goods are packed (Packing) and sent to the customer (Shipping).

### 3.1 Variability

To create a product line of warehouse management systems, the variability of the domain must be managed. A selected set of variabilities that we will use throughout this paper as running example is summarized in Table 1. One of the variabilities we consider is the scale of the warehouse that can be ranging from *high* (e.g., handling thousands of orders per day), *medium* (e.g., handling hundreds orders per day), or *low* (e.g., handling few dozen orders per day), typical numbers we found in real world warehouses. Another variability is the strategy for handling partial pallet quantities that considers either speed or optimal reduction. The *high speed* strategy tries to only pick from a single box whilst it may possibly leave a lot of partial pallet quantities while the *optimal reduction strategy* will remove as much partial pallet quantities as possible (thereby creating space in the warehouse) which leads to a higher amount of picks. The third variability represents different strategies for stapler cranes with one or several forks. The forth variability investigated in this example captures the user interface (UI) options including support for desktop/laptop computers or mobile devices.

The variabilities are summarized in Table 1. The table gives for each variability decision a name for the variation point, the possible values and the binding time. The binding time gives the latest point in the lifecycle when a decision on the variability must be made. All binding times are related to the product level.

**Table 1:** Selected Variation Decisions and Their Values in the Warehouse Product Line

ID	Variability Decision	Possible Values	Binding Time
VP1	Picking rate	High/Medium/Low	Design time
VP2	Partial pallet strategy	High speed/Optimal reduction	Runtime
VP3	Stapler crane strategy	Single fork/Multiple forks	Design time
VP4	UI device	Computer/Mobile device	Design time

### 3.2 Architectural Decisions

Table 2 depicts an illustrative subset of the design space under consideration that provides the basis for making architectural decisions at the product line level. Some fundamental architectural decisions are taken to form the reference architecture. We elaborate these decisions along with their corresponding rationales in the context of the aforementioned case study. We prefer an asynchronous call-oriented interaction style to a message-oriented interaction style because it leads to less complex and more readable code (cf. **AD1**); we prefer fix interprocess communication (IPC) to variant IPC because fix IPC provides higher performance (cf. **AD2**); and we prefer a service-oriented API style to resource-oriented API style as the underlying infrastructure functionality is already provided in terms of services (cf. **AD3**).

**Table 2:** Architectural Decisions at Product Line Level

ID	Decision Point	Options
AD1	Interaction Style	Asynchronous calls interaction Message-oriented interaction Synchronous calls interaction
AD2	Interprocess Communication (IPC)	Fix Variant
AD3	API Style	Service-oriented Object-oriented Resource-oriented

The architectural decisions at the product level are shown in Table 3 along with their variabilities. We note that some of the architectural decisions are influenced by the variabilities identified previously. For instance, we cannot select a single interprocess communication solution for **AD4** because of **VP1**. For *low* picking rate, the option *IPC open source* is sufficient, while for *medium* and *high* picking rates, the option *IPC very expensive* is necessary. This decision brings us to other subsequent architectural decisions respectively: we create an IPC abstraction interface to localize dependencies

**Table 3:** Architectural Decisions at Product Level

ID	Decision Point	Options	Variability Decision
AD4	IPC	IPC open source	VP1-low
		IPC medium price	-
		IPC very expensive	VP1-medium/VP1-high
AD5	IPC invocations	No abstraction interface	-
		Abstraction interface – facade	-
		Abstraction interface – gateway	-
AD6	IPC open source adaptation	Adapt IPC open source	-
		Create a wrapper component	-
AD7	IPC very expensive adaptation	Create a wrapper component	-
		Other	-
AD8	Deployment devices	Single server	VP1-low
		Multiple servers with round-robin	VP1-medium
		Multiple servers with load monitoring	VP1-high
AD9	Server identification	Business delegate proxy	-
		Business delegate adapter	-

to the changing IPC component in a single component (cf. **AD5**); we will create a wrapper component for *IPC open source* to support **VP1** (cf. **AD6**); or we will create a wrapper component for *IPC very expensive* (cf. **AD7**). Similar to the decision for an IPC solution, the deployment cannot be decided until **VP1** is chosen. A single server is necessary for *low* picking rates but multiple servers with a round-robin strategy should be used with respect to *medium* picking rates. For *high* picking rates, multiple servers with load monitoring are needed. To limit the effect on architecture, we decide to use the same component structure for all deployment options: always using a client side business delegate to identify the server, knowing that it is not necessary, and therefore, costly in single server deployments, to limit the variability of the architecture (cf. **AD9**).

### 3.3 Dependencies between Decisions

For application engineering (i.e., at the product level), the decisions **AD4** and **AD8** are still open and need to be decided. They are related to and influenced by the variation point **VP1** as following:

- Low picking rate implies *IPC open source* (**AD4**) and *Single server* (**AD8**)
- Medium picking rate implies *IPC very expensive* (**AD4**) and *Multiple server with round-robin* (**AD8**)
- High picking rate implies *IPC very expensive* (**AD4**) and *Multiple server with load monitoring* (**AD8**)

We will illustrate the application of our approach in capturing and resolving such dependencies systematically.

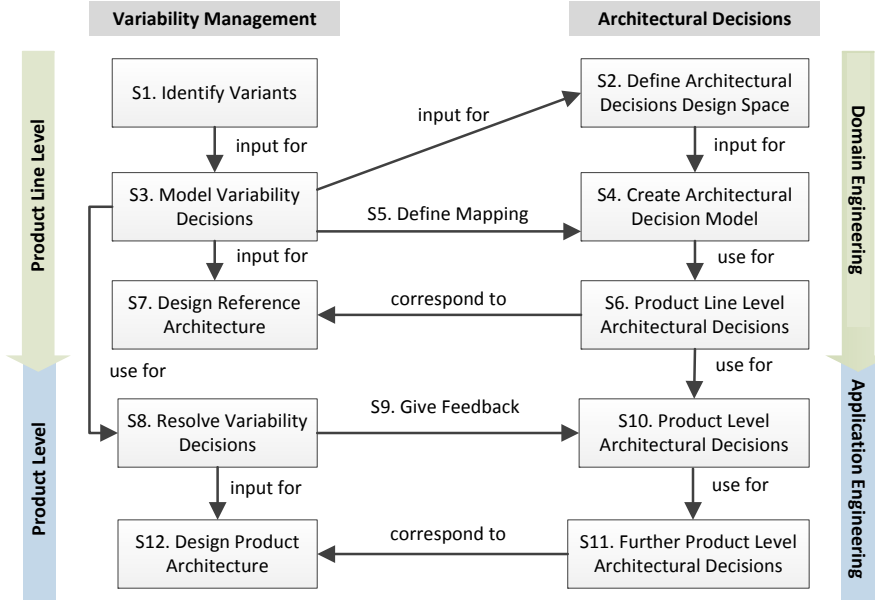
## 4 Proposed Approach

In this section, we introduce our approach for integrating variability and architectural decisions in a systematic manner. Our approach is presented in the context of both variability and architectural decision making processes at product line and product level for designing reference architectures and product architectures respectively. In particular, we present various steps of variability management and architectural decision making and their relationships along with our solutions for eliciting and harmonizing the interdependencies among different kinds of decisions.

### 4.1 Approach Overview

An overview of our approach is provided in Figure 3. We present the basic steps at the product line level and the product level for both variability modeling and architectural decision modeling. The variability model at the product line level is derived from scoping and subsequent domain analysis. It leads through a manual process of derivation to a corresponding architectural decision model. We discuss this derivation process in Section 4.2. As part of variability modeling, dependencies among variabilities are expressed using constraints such as selecting a certain kind of warehouse restricts the range of applicable partial pallet handling strategies. To support the creation of the reference architecture, architectural decisions for the product line are identified and documented in the architectural decision model. Our approach formally elicits the dependencies between the variabilities and architectural decisions in terms of mappings between the corresponding elements. At product line level, the resulting reference architecture will be designed to cover the whole range of variability specified by the variability decision model by selecting the appropriate architectural solutions from the architectural decision model.

At the product level, the variabilities of the product are resolved in order to obtain a valid configuration, i.e., all variability constraints are satisfied by the decisions made. Using the aforementioned formal mappings as input, we can automatically constraint the available architectural options that correspond to variability decisions made on product level. For example, if a specific variability option is chosen in the configuration, the resolution of the constraints and the aforementioned predefined mappings ensure that only architectural options that are associated with that specific variability option can still be chosen in the architectural decision model. The interdependencies that can be automatically enforced are discussed in Section 4.3. At this point, further architectural decisions at product level can be made in order to accomplish the product architecture. This way, variability and architectural decisions are kept consistent to each other and the product architecture conforms both to the variability and architectural constraints.



**Figure 3:** Approach overview

## 4.2 Product Line Level

The key idea of our approach is to first determine necessary variability decisions based on the requirements, as well as possible architectural solutions for implementing the required variability—and also architectural decisions not related to variability. Then, the possible variability resolutions (variants) are mapped to the corresponding architectural options. This is realized in the following steps:

- S1. Identify Variabilities:** Based on scoping and an analysis of the requirements, we identify potential variabilities. This is often done by determining main features that are relevant to specific system instances [31].
- S2. Define Architectural Decisions Design Space:** We consider existing documented architectural knowledge, such as the reusable architectural models in [45], in order to define the architectural decisions design space, i.e., architectural options and alternatives for the various decisions points related to the design of the reference architecture and product architectures. This information will be used as input for creating the architectural decision model.
- S3. Model Variability Decisions:** The individual variants that vary along an identifiable theme can be described as variability decisions. The advantage of having them as variability decisions—rather than using other variability modeling techniques

such as feature modeling—is mostly to make the inherent dimension of variability explicit<sup>3</sup>.

**S4. Create Architectural Decision Model:** The analysis made in **S2** helps us define the architectural decision model that will be used as guidance for making architectural decisions at product line and product level<sup>4</sup>.

**S5. Define Mapping:** The product line architect(s) identify which variability decisions correspond to architecturally relevant requirements. They determine potential architectural decisions that correspond to the individual variants, and make this interdependencies explicit by introducing a mapping between the two models. For instance, a variability decision may exclude or enforce a related architectural decision.

**S6. Product Line Level Architectural Decisions:** The architectural decisions that will cover the desired variability are derived manually. The aim is to create a strategy that covers the whole range of variants described by a variability decision, considering the architectural alternatives and options provided by the architectural decision model.

**S7. Design Reference Architecture:** The architectural decisions that are made to cover the whole range of variants implied in the variability decisions model will be realized in the reference architecture.

### 4.3 Product Level

The major goal at the product level is to derive configurations based on the reference architecture to create particular products. At this level, the architecture of a concrete product may incorporate additional features apart from the base configuration that are different from the others. The following steps can be leveraged to accomplish the architecture of a certain product:

**S8. Resolve Variability Decisions:** Let us assume that, at the product line level, the kind of variability decisions and architectural decisions described in the previous section have been determined. We can distinguish three situations for handling variability and architectural decisions:

- a) The variability identified at the product line level fits to the product level and we resolve the product line variability while developing the product.

---

<sup>3</sup> Note that we will use a decision-modeling approach [35] as our basis as the tool that we will discuss later is based on this approach. However, as discussed in [9] decision modeling and feature modeling are rather similar today and can even be proven to be equivalent for some cases [17]. Hence, our approach could just as well be applied with feature modeling.

<sup>4</sup> Note, that we will apply decision modeling based on Questions, Options and Criteria [25] as the tool for decision making support that will be used in our approach (ADvISE) is based on this approach. However, other decision models (such as [45]) can be used in a similar way.

- b) The variability determined at the product line level has not yet supported all product-relevant functionality. However, the additional functionality is only relevant to a single product.
- c) The variability identified in the previous step is insufficient and the needed variability is important for a range of products. This requires product line evolution [34].

Each situation will trigger the next steps for handling and resolving the variability and architectural decisions:

**S9. Give Feedback and S10. Product Level Architectural Decisions:** The first case **S8(a)** is rather straightforward. In this case, fitting variability decisions have already been developed at the product line level. The decisions are taken and related to corresponding architectural decisions. Thus, selecting the variants immediately constrains the architectural decisions through the mappings achieved in **S5**. If the variability decisions are sufficiently fine-grained, then the architectural decisions can be automatically resolved. Otherwise, the architectural decisions are constrained and the architect performs a tradeoff decision among the remaining cases.

**S11. Further Product Level Architectural Decisions:** The second case **S8(b)** is related to additional product-specific functionality that needs to be designed. Therefore, the variability decisions do not provide further orientation as this is outside the scope of functionality supported by reusable assets and we cannot expect to make it reusable (hence *product-specific*). This case is not fundamentally different from architecting a single-system. The only distinguishing point is that the existing architectural decisions have to be considered. This can be resolved automatically as constraints among decision points and architectural options are available in the architectural decision model. However, due to the similarity to the initial case of architecting and because there is no direct relation to variability, we will not discuss further.

The third case **S8(c)** denotes that, at the product level, the capabilities provided by the reusable assets (and hence the variabilities) are insufficient. There are two possible approaches to handling this circumstance. In the ideal case, we can go back to the product line level and evolve the product line infrastructure to cover the special case. This would entail augmenting the variability model providing (if needed) additional architectural decisions and establishing the relations between them. As a result, the steps from **S3** to **S7** are repeated and the variability decision model, architectural decision model, and their interdependencies are reconsidered. Afterwards, the rest can be achieved similarly to the first situation. Nevertheless, sometimes, especially if there is an urgent need for shipping the product, a different decision can be made: changes are made at the product level that might raise inconsistencies at the product line level. In this case, the product line level should

be evolved or adapted at a later point in time. While such an approach can speed up the development process, it may also expose extra costs through introducing technical debt, which needs to be addressed at a later point in time [32].

**S12. Design Product Architecture:** The resulting product architecture will be created based on both variability decisions and product-related architectural decisions.

## 5 Tool Support and Case Study Revisited

In this section, we present the main tools that are the basis for our work and demonstrate their integration in the context of the case study discussed in Section 3. We have developed an integration of the two tools, which are EASy-Producer (cf. [18])—for variability management—and ADvISE (cf. [24])—for architectural decision support. In the subsequent section, we describe the main features of these tools and demonstrate their integration through the warehouse case study.

### 5.1 EASy-Producer

The EASy-Producer tool aims at providing modeling and realization support for software product lines and software ecosystems [33]. It provides some capabilities that are standard to all product line engineering tools, like the capability to model variability, to support the configuration process by determining consistency and consequences of a partial configuration (e.g., some value may be derived based on constraints and other given values). In addition, the tool has some capabilities that make it special and particularly well suited for our case. One is that it provides a very generic approach to instantiating artifacts. This allows that artifacts can be as diverse as requirements, different forms of code, or, as in our case, architectural information. Another characteristic is that it has an extremely powerful language for describing variability per se as well as constraints [16]. The constraint language (actually a variant of the Object Constraint Language [26]) can also be used to describe implementation related decisions. This allows to easily describe and manage dependencies between variability decisions and architectural decisions. The tool also has further characteristics that are related to its high flexibility and ecosystem support like the ability to support multi-staged derivation and composition. However, these capabilities are not particularly relevant to our discussion here, thus, we will not go into more detail on this. A final capability, which is important in our context, as it helps to support the evolution scenarios, is that EASy-Producer keeps both the product line infrastructure (the product line level) as well as the individual products in separate projects. While the tool can help to support consistency among the two levels, it allows for temporary violations. Thus, we can perform cases, where we first extend the product level and only later add it to the product line level.

## 5.2 Architectural Design Decision Support Framework

The Architectural Design Decision Support Framework (ADvISE) is an Eclipse-based tool that supports the modeling of reusable architectural decisions using Questions, Options and Criteria (QOC) [25] for systematizing the design space and providing decision support. In particular, it assists the architectural decision making process by introducing for a group of design issues a set of questions along with potential options, answers and related (often design pattern based) solutions, as well as dependencies and constraints between them. ADvISE has been developed with focus on reusable architectural knowledge that can be also transformed into reusable architecture designs (cf. [24]) rather than on product lines. However, it is generic enough to support both product line as well as product related architectural decision making. The advantage of the reusable architectural decision models is that the models need to be created only once for a recurring design situation. In similar application contexts, corresponding questionnaires can be automatically instantiated and used for making concrete decisions, from which architectural decision documentations are generated. In our work, we integrate architectural decisions of the architectural decision models with variability decisions of the variability decision models by introducing mappings between the corresponding architectural options and variabilities.

## 5.3 Decision Integration Tool Support

As discussed in Section 4, the first step in the tool support (cf. **S3**) is to represent the variability outlined in Table 1 (cf. **S1**) in the form of an EASy-Producer decision model. The EASy-Producer tool supports two representations for variability models: an interactive view, where a configuration of the variability can be determined in an interactive manner (see Figure 4(a)) and a textual view where the variability can be described in a programmatic manner (see Figure 4(b)). In our example, four types of variability decisions are defined including “PickingRateType”, “PartialPalletStrategyType”, “StaplerCraneStrategyType”, and “UIDeviceType” along with their resolutions in the lines 5–8. These types are used to define the variability decisions “VP1” to “VP4” according to Table 1. The variability decisions “VP1”, “VP3”, and “VP4” are supposed to be bound at build time (lines 15–16) while “VP2” is left open until runtime (lines 17–19).

Afterwards, we model the architectural decisions summarized in the Table 2 and 3 (cf. **S2**) using ADvISE (cf. **S4**). The architectural decisions editor allows us to edit for each decision point a list of questions, and for each question a number of options which may be mapped to specific solutions and can be related to follow-on decisions and questions or constrained by other architectural options. In Figure 5, we give an example of the product-level architectural decision **AD4** of Table 3 that defines the types of IPC that can be used in the warehouse product. While Figure 5(a) gives general information about the underlying architectural decision, we can see the alternative options related to

Product Configuration Editor: PL_WMS			
<div>  Validate Product            Instantiate Product            Propagate Values            Freeze All         </div>			
	Decision Name	Current value	Freeze
	VP1	medium	freeze
	VP2		
	VP3	single	freeze
	VP4	computer	freeze

(a) Interactive view of the WMS example

```

1 project PL_WMS {
2
3     version v0;
4
5     enum PickingRateType {high, medium, low};
6     enum PartialPalletStrategyType {highSpeed, optimalReduction};
7     enum StaplerCraneStrategyType {single, multiple};
8     enum UIDeviceType {computer, mobile};
9
10    PickingRateType      VP1;
11    PartialPalletStrategyType VP2;
12    StaplerCraneStrategyType VP3;
13    UIDeviceType         VP4;
14
15    enum BindingTime {designTime, compileTime, initTime, runTime};
16    attribute BindingTime bindingTime = BindingTime.designTime to PL_WMS;
17    assign (bindingTime = BindingTime.runTime) to {
18        PartialPalletStrategyType VP2;
19    }
20 }

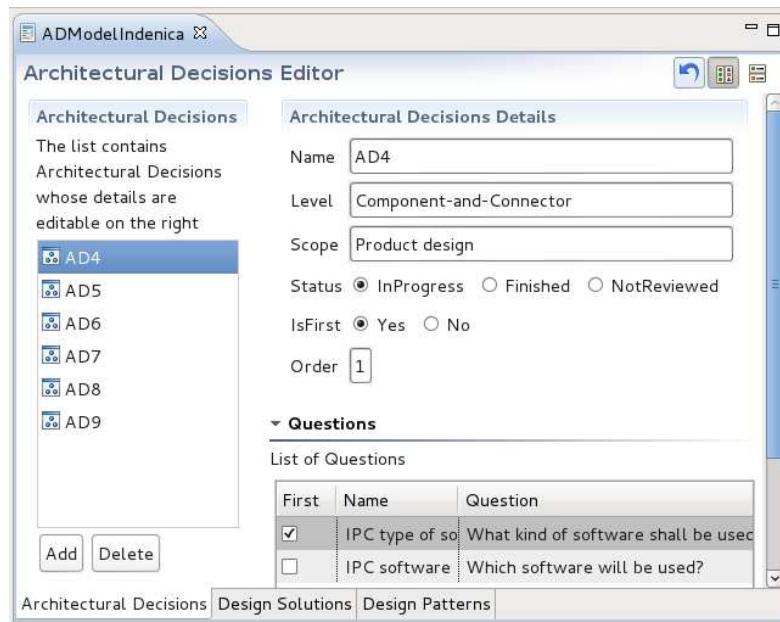
```

(b) Textual view of the WMS example

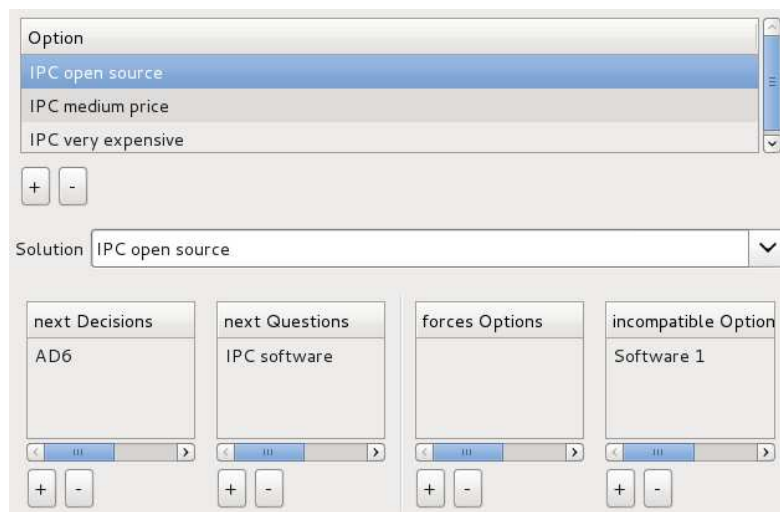
**Figure 4:** Variability model of the WMS example modeled in EASy-Producer

a specific question in the detailed view of Figure 5(b). In this example, the alternative options “IPC open source”, “IPC medium price”, and “IPC very expensive” are provided for the question “IPC type of software”.

The next step of our approach (cf. S5) requires that we define the mapping between the variability and architectural decisions. For this purpose, we establish a set of mappings from the variability decision model onto the corresponding architectural decision model for the different variabilities in XML format. In particular, for each variability decision, relations of specific type (e.g., *excludes*, *enforces*, etc.) can be specified onto an architectural option. In Listing 1, the variability decision “PickingRateType.medium” is mapped to the architectural option “IPC open source” of the architectural decision AD4 with type of relation “excludes”. It means that selecting the *medium* picking rate will result in the rejection of *IPC open source*.



(a) Architectural decisions



(b) Options related to a question

**Figure 5:** ADvISE architectural decisions model editor

```

<mappings>
  <aModel>ADModelIndenica</aModel>
  <vModel>PL_WMS</vModel>
  <vp name="VP1">
    <relation type="excludes">
      <vd id="PickingRateType.medium"/>
      <add id="AD4.IPC type of software.IPC open source"/>
    </relation>
  </vp>
  ...
</mappings>

```

**Listing 1:** Excerpt of the mappings between variability and architectural decisions

After designing the reference architecture to cover the whole range of variability (cf. **S6-S7**), it is now time to apply it to the problem of creating the product architecture. In step **S8** we resolve the variability decisions by assigning values to the variability decisions. This is shown in Figure 6, where the lines 6–10 contain the decisions made for each variation point at design time. For instance, in our running example, the value “PickingRate.medium” was selected for “VP1”. In the EASy-Producer tool the configuration is typically provided as a second file, referencing the definition file.

The case above actually corresponds only to step **S8(a)**, thus the configuration is straightforward. If we have the case of **S8(b)** or **S8(c)**, then the situation becomes more complex. An example of **S8(b)** would be that our customer requires an integration to a specialized ERP system, while the product line typically only supports SAP-integration. Then, a typical approach would be to introduce an extension point and a connector to this specialized ERP-system. This would not necessarily be visible in the variability model or it would be simply modeled as an option to activate the extension point. An example for **S8** would be that a new customer would like to have a Partial Pallet Strategy, which is not yet supported, like max two, i.e., at most two pallets may be opened for one type of article. If we would like to support this in the future we would extend the “PartialPalletStrategyType” to include the “maxTwo” option.

The configuration given in Figure 6 defines the product from a variability perspective. Based on the connections to the architectural decisions a number of architectural decisions can be automatically derived and further ones can be constrained. In our example, the mapping we defined in Listing 1 will enable us to reflect variability decisions on the architectural decision model at the product level design (cf. **S9-S10**). To support architectural decision making, ADvISE tooling provides automatically generated questionnaires from the architectural decision models for guiding software architects. In Figure 7, the selection of the variant “PickingRate-medium” will cause the option “IPC open source” in the related ADvISE questionnaire to be deactivated. At this point, further architectural decisions, related to variability or not, can be made in order to design the product architecture (cf. **S11-S12**).

```

1 project WMS_product {
2
3     version v0;
4     import PL_WMS;
5
6     VP1 = PickingRateType.medium;
7     /* partialPalletStrategy = PartialPalletStrategyType.high;
8        // This is not assigned, as this is only done at runtime */
9     VP3 = StaplerCraneStrategyType.single;
10    VP4 = UIDeviceType.computer;
11 }

```

**Figure 6:** Providing values to the variability decisions in the WMS example

**AD Questionnaire**

▼ AD4

Component-and-Connector: Product design

☒ **IPC type of software**  
What kind of software shall be used for managing IPC?

☒ IPC open source ☐ IPC medium price ☐ IPC very expensive

☒ **IPC software**  
Which software will be used?

☐ Software 1 ☐ Software 2

(a) Questionnaire excerpt for product level architectural decisions

**AD Questionnaire**

▼ AD4

Component-and-Connector: Product design

☒ **IPC type of software**  
What kind of software shall be used for managing IPC?

☒ IPC open source ☐ IPC medium price ☐ IPC very expensive

☒ **IPC software**  
Which software will be used?

☐ Software 1 ☐ Software 2

(b) Architectural option deactivated due to variability decision

**Figure 7:** Architectural decision making

## 6 Discussion and Limitations

Through our study, we have observed that in many cases, interdependencies between variability and architectural decisions exist but are kept implicit and get resolved, in

practice, in an informal way. Very often, these decisions are made by different stakeholders and with different tool support and their overlaps and inconsistencies are often resolved in an ad hoc and manual manner. We showed that formally eliciting the interdependencies requires additional efforts at the beginning but it leads to better automated support in integrating and harmonizing variability management and architectural decision making in the long run.

This way, we capture and formalize the links between variabilities and architectural options at the product line level, such that the architectural decision making process is (semi-)automated and connected with the variability decision making process. As a result, architectural decisions can be changed or adapted whenever the variability is resolved and specific variants are selected.

The variability can be resolved at different binding times, and therefore, the architectural decisions can also be constrained at different stages respectively. The huge advantage of our approach is that variability and architectural decisions shall remain consistent at product derivation. Moreover, the introduction of mappings between the two kinds of decisions can significantly enrich the documentation of the design rationale. For instance, the rejection or selection of an architectural option can be justified by following the dependencies with the corresponding variability decisions. This would help enduring the life time of the architectural design decisions, and therefore, making the decisions more sustainable.

Although we have used two existing tools for demonstrating our approach and implementation we claim that our proposal is to a large extent generalizable. For the variability management, feature modeling can be used alternatively to the decision modeling approach applied in EASy-Producer; for architectural decision support, other architectural decision models or ontologies would also be applicable in our proposal. In this case, some effort for synchronizing the architectural decisions with the variability decisions automatically from the predefined mappings would be required.

In our approach, we assume that the variability decisions guide the architectural decisions for the product line and product design. In practice, an architectural decision may also influence a variability decision. For instance, a decision to use a low-cost software solution because of cost constraints may cause some variabilities to be invalid. However, that would mean that the variability needs to be reconsidered and possibly redesigned (i.e., repeat steps **S3** to **S7** in Figure 3).

We discussed the interaction of variability and architectural decisions with the focus on product line design and product derivation and have not investigated the evolution and maintenance of product lines and products. As architectural decisions contain also interdependencies, reconsidering a variability decision may cause inconsistencies to existing architectural decisions. It is challenging to be able to handle this situation and also predict the impact variability decisions will have on the product architecture, but we plan to address this in our future work.

Another limitation of our approach is that the evidence of variability and architec-

tural decision interdependence is discussed in the context of one case study. As discussed in Section 2 this interdependence is implicit but no such examples have been documented in the literature. The elaboration of more industrial case studies and interviews with practitioners would allow us to discover more types of interdependencies, extend our approach and evaluate it in different contexts.

## **7 Related Work**

### **7.1 Architectural Decisions in Variability Management Approaches**

Variability and architectural decisions have been studied often in the literature in the same context. Variability decisions mainly refer to decisions related to the differences among the products that derive from a product line. The variabilities described as optional, alternative or multiple selections [22] are often related to architectural elements. For instance, the Feature-Oriented Domain Analysis (FODA) [3, 21] usually mixes architecture-related decisions with domain properties and system configurations. Feature models mainly describe the solution space (i.e., focuses on modeling of commonalities and variabilities) and do not provide any guidance for selecting between alternative variants and reasoning about them. However, many approaches in the literature propose to enrich variability management with design rationale and decision support by introducing decision models [14, 35, 36] for describing variabilities. In some cases these variability decision models are designed to be reusable [30, 31]. In an approach that combines both methods, Perovich et al. [27] consider product line architectures as a set of architectural decisions and use feature models to represent the decisions associated to the product features and transformation models to transform decisions into product architectures. The aforementioned approaches mainly focus on variability decisions and handle architectural decisions also as variability decisions without setting any boundary between the two. The different nature of the two types of decisions is not studied in any of these works.

### **7.2 Variability Decisions in Architectural Decision Making Approaches**

Many approaches in the literature deal with modeling of reusable architectural decisions [45] or provide tool support for architectural decision making [38]. Unlike decision models for product lines that describe a set of variabilities relevant to product derivation, architectural decision models focus mainly on architecture-related options and alternatives for designing a software architecture. Many approaches in the literature suggest to integrate variability management with architectural knowledge and design rationale. For instance, Dhungana et al. suggest to capture variabilities in variability management as decisions and establish relationships between assets, such as components and decisions explicitly [13]. Also, Capilla and Babar suggest to integrate architectural decision models with variability models to support ADDs for product line

architectures [7]. For this, they map design decisions to variabilities and binding times to document reasoning about decisions related to product lines. In a different approach, Burge et al. capture rationale for decisions required for a product configuration in a product line by associating design criteria with decision alternatives [6]. They generate an hierarchy of decisions needed to be made for designing a product architecture from feature models, thus assisting decision making and documentation. Sinnema et al. integrate a variability management framework (COVAMOF) with concepts of architectural decision models, thus making the effects of different alternatives on the quality of a system explicit [39]. A number of approaches in software product line engineering focus on documenting architectural design decisions and their rationale [7]. Unlike variability management approaches based on feature models [21] or decision models [36], these approaches propose to view architectural design decisions in modeling and managing of product line variability models as first class citizens. For this, they distinguish between variations considered at product configuration and architectural decisions made at early stages of the design phase. Also, the work in [41] suggests the reuse of design decisions to customize a product line using composition techniques as a step-wise refinement for product derivation. As before, these approaches consider the design of product line and product architectures as an architectural decision making process and do not distinguish it from variability management. None of these approaches studies, elicits, and resolves the interdependence between variability and architectural decisions.

### **7.3 Variability in the Architecture Design**

Some researchers have proposed the characterization of variability decisions according to the stage they are resolved. For example, Rosenmüller et al. apply multi-dimensional separation of concerns in variability modeling, that is, they create different variability models for different stakeholder concerns, e.g., for non-functional properties, such as security and quality of service, for runtime contexts, etc., and use generalization and specialization mechanisms to model extension, composition and configuration of the variability dimensions [28]. Bidian et al. introduce variability decision boundaries at the different stages of requirements definition, architecture and detailed design and runtime which define at which stage the decisions have to be resolved [4]. In both approaches, variability and architectural decisions are considered to have overlaps and interrelationships. Some approaches set the focus of the variability modeling on the architectural level. For instance, the architectural description language xADL [11] allows the modeling of versions, options and variants in product line architectures. Thiel and Hein integrate variability with product line architecture design. For this, they introduce variabilities in architectural views and connect them to feature variability [40]. The aim of these approaches is to reflect variability on the architecture design. In a different approach [15], the architectural knowledge behind variability, i.e., the variability design rationale is captured in models. Diaz et al. propose to combine two metamodels, one for defining external and internal variability and one for documenting architectural knowl-

edge and variability design rationale. They also do not distinguish between variability and architectural decisions as architectural decisions are implicit in the derived product line and product architectures.

#### **7.4 Integration of Other Design Artifacts**

Apart from the linking between variability and architectural decisions and architecture design, the systematic integration and mapping between other artifacts of the software design have been studied extensively in the literature. For instance, the relation and interaction between requirements and architecture design [19, 43] and the explicit mapping and integration between architectural decisions and software architectures [12, 24]. Buhne et al. introduce links between variability information and requirements artifacts to support consistency checking of variability definitions among different product line sets [5]. Also, Durdik et al. study the effects of design decisions on requirements, for example, the prioritization of requirements or the consideration of additional requirements at design space exploration [46]. One of the main aims of the aforementioned approaches is to document the rationale and check the consistency of evolving artifacts that contain interdependencies, which is also the main concept in our approach. However, our approach is the first proposal for investigating and supporting the interaction between variability and architectural decisions systematically.

### **8 Conclusions**

In this paper, we studied the interdependence of variability and architectural decisions that need to be resolved during product line and product design. Although variability decisions constraint and influence architectural decisions in practice, this inherent interdependence has not been studied or addressed systematically in the literature yet. We propose to make the interdependence of variability management and architectural decision making explicit. For this, we propose a novel approach for managing variability and architectural decisions in an integrated manner. To ensure that variability and architectural decisions remain consistent to each other we introduce at the product line level mappings between them, which are afterwards leveraged at the product derivation, i.e., resolution of variability decisions, to give feedback—mainly introduce constraints—to the architectural decisions. We demonstrated our approach by integrating two existing tools for modeling and realization support for software product lines (EASy-Producer) and for architectural decision modeling and making support (ADvISE).

In the context of a case study from the warehouse automation area we documented variability and architectural decisions and their interdependencies and demonstrated our approach using the aforementioned tools and their integration. Our contributions can be summarized in the following: 1) we propose a novel approach for bridging the gap between variability management and architectural decision making in product line engineering, 2) we make the interdependencies between variability and architectural

decisions explicit, and 3) we ensure that variability and architectural decisions are consistent to each other at product derivation. These aspects are important not only when creating reference architectures and derive products but also when a product line or product architecture needs to be maintained. We consider the second to be an open challenge and plan in the future to study different forms of integrating both kinds of decisions and the impact of changing these decisions during product line and product evolution.

## Acknowledgement

This work was partially supported by the European Union FP7 project INDENICA (<http://www.indenica.eu>), grant no. 257483.

## References

- [1] BABAR, M. A., CHEN, L. C. L., AND SHULL, F. Managing Variability in Software Product Lines. *IEEE Software* 27, 3 (2010), 89–91, 94.
- [2] BACHMANN, F., AND BASS, L. Managing Variability in Software Architectures. In *2001 Symposium on Software Reusability: Putting Software Reuse in Context (SSR)* (2001), ACM, pp. 126–132.
- [3] BENAVIDES, D., SEGURA, S., AND CORTÉS, A. R. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Inf. Syst.* 35, 6 (2010), 615–636.
- [4] BIDIAN, C., AND YU, E. S. K. Towards Variability Design as Decision Boundary Placement. In *10th Workshop on Requirements Engineering (WER)* (2007), pp. 139–148.
- [5] BÜHNE, S., LAUENROTH, K., AND POHL, K. Modelling Requirements Variability across Product Lines. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering* (2005), IEEE, pp. 41–52.
- [6] BURGE, J. E., GANNOD, G. C., AND CONNOR, H. L. Using Rationale to Drive Product Line Architecture Configuration. In *6th Int'l Workshop on SHaring and Reusing Architectural Knowledge (SHARK)* (2011), ACM, pp. 29–36.
- [7] CAPILLA, R., AND ALI BABAR, M. On the Role of Architectural Design Decisions in Software Product Line Engineering. In *2nd European Conf. on Software Architecture (ECSA)* (2008), Springer, pp. 241–255.
- [8] CLEMENTS, P., AND NORTHROP, L. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, 2002.
- [9] CZARNECKI, K., GRÜNBACHER, P., RABISER, R., SCHMID, K., AND WASOWSKI, A. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *6th Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)* (2012), ACM, pp. 173–182.
- [10] CZARNECKI, K., HELSEN, S., AND EISENECKER, U. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process Improvement and Practice* 10, 2 (2005), 143–169.
- [11] DASHOFY, E. M., VAN DER HOEK, A., AND TAYLOR, R. N. An Infrastructure for the Rapid Development of XML-based Architecture Description Languages. In *24th Int'l Conf. on Software Engineering (ICSE)* (2002), ACM, pp. 266–276.
- [12] DERMEVAL, D., PIMENTEL, J., SILVA, C. T. L. L., CASTRO, J., SANTOS, E., GUEDES, G., LUCENA, M., AND FINKELSTEIN, A. STREAM-ADD - Supporting the Documentation of Architectural Design Decisions in an Architecture Derivation Process. In *36th Annual IEEE Computer Software and Applications Conf. (COMPSAC)*, Izmir, Turkey (2012), IEEE, pp. 602–611.

- [13] DHUNGANA, D., GRÜNBACHER, P., AND RABISER, R. DecisionKing: A Flexible and Extensible Tool for Integrated Variability Modeling. In *1st Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)* (2007), pp. 119–128.
- [14] DHUNGANA, D., GRÜNBACHER, P., AND RABISER, R. The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. *Automated Software Engineering* 18, 1 (Mar. 2011), 77–114.
- [15] DÍAZ, J., PÉREZ, J., GARBAJOSA, J., AND WOLF, A. L. A Process for Documenting Variability Design Rationale of Flexible and Adaptive PLAs. In *Confederated Int'l Conf. on On The Move to Meaningful Internet Systems (OTM)* (2011), Springer, pp. 612–621.
- [16] EICHELBERGER, H., AND SCHMID, K. A Systematic Analysis of Textual Variability Modeling Languages. In *Int'l Software Product Line Conference (SPLC'13)* (2013). (accepted).
- [17] EL-SHARKAWY, S., DEDERICH, S., AND SCHMID, K. From Feature Models to Decision Models and Back Again: An Analysis Based on Formal Transformations. In *16th Int'l Software Product Line Conference (SPLC'12)* (2012), ACM, pp. 126–135.
- [18] EL-SHARKAWY, S., KRÖHER, C., AND SCHMID, K. Supporting Heterogeneous Compositional Multi Software Product Lines. In *Joint Workshop of the 3rd Int'l Workshop on Model-driven Approaches in Software Product Line Engineering and the 3rd Workshop on Scalable Modeling Techniques for Software Product Lines (MAPLE/SCALE 2011) at the 15th Int'l Software Product Line Conference (SPLC '11)* (2011), ACM.
- [19] GRÜNBACHER, P., EGYED, A., AND MEDVIDOVIC, N. Reconciling Software Requirements and Architectures with Intermediate Models. *Softw. Syst. Model.* 3, 3 (2003), 235–253.
- [20] JANSEN, A., AND BOSCH, J. Software Architecture as a Set of Architectural Design Decisions. In *The 5th Working IEEE/IFIP Conf. on Software Architecture* (2005), IEEE, pp. 109–120.
- [21] KANG, K. C., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. rep., Carnegie-Mellon University Software Engineering Institute, November 1990.
- [22] LINDEN, F., SCHMID, K., AND ROMMES, E. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*. Springer, 2007.
- [23] LYTRA, I., SOBERNIG, S., AND ZDUN, U. Architectural Decision Making for Service-Based Platform Integration: A Qualitative Multi-Method Study. In *Joint 10th Working IEEE/IFIP Conf. on Software Architecture & 6th European Conf. on Software Architecture (WICSA/ECSA), Helsinki, Finland* (2012), IEEE.
- [24] LYTRA, I., TRAN, H., AND ZDUN, U. Supporting consistency between architectural design decisions and component models through reusable architectural knowledge transformations. In *European Conference on Software Architecture (ECSA)* (2013), Springer, pp. 224–239.
- [25] MACLEAN, A., YOUNG, R., BELLOTTI, V., AND MORAN, T. Questions, Options, and Criteria: Elements of Design Space Analysis. *Human-Computer Interaction* 6 (1991), 201–250.
- [26] OBJECT MANAGEMENT GROUP, INC. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, 2006. <http://www.omg.org/spec/OCL/2.0/> [validated: June 2013].
- [27] PEROVICH, D., ROSSEL, P. O., AND BASTARRICA, M. C. Feature Model to Product Architectures: Applying MDE to Software Product Lines. In *2009 Joint Working IEEE/IFIP Conference on Software Architecture European Conference on Software Architecture* (2009), vol. 11, IEEE, pp. 201–210.
- [28] ROSENMÜLLER, M., SIEGMUND, N., THÜM, T., AND SAAKE, G. Multi-dimensional Variability Modeling. In *5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS)* (2011), ACM, pp. 11–20.
- [29] ROSSEL, P. O., PEROVICH, D., AND BASTARRICA, M. C. Reuse of Architectural Knowledge in SPL Development. In *11th Int'l Conference on Software Reuse: Formal Foundations of Reuse and Domain Engineering (ICSR)* (2009), S. Edwards and G. Kulczycki, Eds., Springer, pp. 191–200.

- [30] SCHMID, K. Scoping Software Product Lines — An Analysis of an Emerging Technology. In *Software Product Lines: Experience and Research Directions; Proceedings of the First Software Product Line Conference (SPLC1)* (2000), P. Donohoe, Ed., Kluwer Academic Publishers, pp. 513–532.
- [31] SCHMID, K. A Comprehensive Product Line Scoping Approach and its Validation. In *Int'l Conf. on Software Engineering (ICSE'24)* (2002), ACM, pp. 593–603.
- [32] SCHMID, K. A Formal Approach to Technical Debt Decision Making. In *9th Int'l ACM SIGSOFT Conf. on Quality of Software Architectures* (2013), ACM, pp. 153–162.
- [33] SCHMID, K., AND DE ALMEIDA, E. S. Product Line Engineering. *IEEE Software* (2013). To appear.
- [34] SCHMID, K., AND EICHELBERGER, H. A Requirements-Based Taxonomy of Software Product Line Evolution. *Electronic Communications of the EASST* 8 (2008).
- [35] SCHMID, K., AND JOHN, I. A customizable approach to full lifecycle variability management. *Sci. Comput. Program.* 53, 3 (Dec. 2004), 259–284.
- [36] SCHMID, K., RABISER, R., AND GRÜNBACHER, P. A Comparison of Decision Modeling Approaches in Product Lines. In *5th Int'l Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)* (2011), ACM, pp. 119–126.
- [37] SHAHIN, M., LIANG, P., AND KHAYYAMBASHI, M. R. Architectural Design Decision: Existing Models and Tools. In *Joint Working IEEE/IFIP Conf. on Software Architecture and European Conf. on Software Architecture (WICSA/ECSA)*, Cambridge, UK (2009), IEEE, pp. 293–296.
- [38] SHAHIN, M., LIANG, P., AND KHAYYAMBASHI, M. R. Architectural design decision: Existing models and tools. In *Joint Working IEEE/IFIP Conf. on Software Architecture and European Conf. on Software Architecture (WICSA/ECSA)* (2009), IEEE, pp. 293–296.
- [39] SINNEMA, M., VAN DER VEN, J., AND DEELSTRA, S. Using Variability Modeling Principles to Capture Architectural Knowledge. *Quality* 31, 5 (2006), 5.
- [40] THIEL, S., AND HEIN, A. Systematic Integration of Variability into Product Line Architecture Design. In *2nd Int'l Conf. on Software Product Lines (SPLC)* (2002), Springer, pp. 130–153.
- [41] TRUJILLO, S., AZANZA, M., DIAZ, O., AND CAPILLA, R. Exploring Extensibility of Architectural Design Decisions. In *2nd Workshop on SHaring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI)* (2007), IEEE, p. 10.
- [42] VAN GURP, J., BOSCH, J., AND SVAHNBERG, M. On the Notion of Variability in Software Product Lines. In *Working IEEE/IFIP Conf. on Software Architecture (WICSA)* (2001), IEEE, pp. 45–54.
- [43] VAN LAMSWEERDE, A. From System Goals to Software Architecture. In *School on Formal Methods* (2003), M. Bernardo and P. Inverardi, Eds., vol. LNCS 2804, Springer, pp. 25–43.
- [44] ZDUN, U. Systematic Pattern Selection Using Pattern Language Grammars and Design Space Analysis. *Software Practice & Experience* 37 (July 2007), 983–1016.
- [45] ZIMMERMANN, O., GSCHWIND, T., KÜSTER, J., LEYMAN, F., AND SCHUSTER, N. Reusable Architectural Decision Models for Enterprise Application Development. In *3rd Int'l Conf. on Quality of Software Architectures (QoSA)*, Medford, MA, USA (2007), Springer, pp. 15–32.
- [46] ZOYA DURDIK, A. K., AND REUSSNER, R. How the Understanding of the Effects of Design Decisions Informs Requirements Engineering. In *Second Int'l Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)* (2013).