University of Magdeburg
School of Computer Science

Master's Thesis

Consistent Concepts for Variant-Management Tool Integrations during the Complete Product Lifecycle

Author:

Maria Papendieck

May 31, 2013

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dr.-Ing. Janet Siegmund
Department of Technical and Business Information Systems

Dr. Danilo Beuche
Dr. Michael Schulze
pure-systems GmbH

Papendieck, Maria:
Consistent Concepts for Variant-Management Tool Integrations during the Complete Product Lifecycle
Master's Thesis, University of Magdeburg, 2013.

Abstract

Software Product Line Engineering (SPLE) is an increasingly often employed approach for developing variant-rich software-intensive systems. Its basic idea is to exploit commonalities of several similar, yet different products throughout the entire product lifecycle. To exploit commonalities, developers of software product lines need to explicitly define and manage the differences (variabilities) between products. This is not trivial, since variabilities can occur in all types of documents used throughout the product lifecycle. Due to the heterogenous nature of the used tools, the notations, in which variabilities are represented, differ greatly. Thus, developers need to learn a new approach for managing variability in each tool. To address this problem, pure::variants, a leading variant-management tool, integrates a variant-management user interface into several tools that are used frequently during the development of software-intensive systems. When using these tool integrations, users only need to have a rough idea how variability is represented in the respective tool. However, each tool provides different extension mechanisms. On the one hand, this introduces inconsistencies between integrations, which is bad for the users, and, on the other, it slows down the development process of new integrations.

To address these problems, we aim to improve the user-interface consistency between integrations, and reduce the development time of new integrations. To this end, we propose a workflow for producing new tool integrations based on real-life requirements and use cases. Apart from improving the user-interface consistency and reducing development time, the workflow should promote good user-experience, and relate to the users' tasks.

To improve user-interface consistency, and reduce the time-to-market, we utilize the concepts of SPLE: We denote variability for the defined requirements, taking into account the different extension mechanisms that each tool provides. Thus, tool-integration developers can derive a requirements document for the respective integration, instead of creating a new one from scratch. Based on the example of three tool integrations and an interview with a pure::variants customer, we show that the workflow fulfills our goals, while there is still room for improvement.

Contents

List of Figures

List of Tables

List of Abbreviations

DME Documentation Management Environment

DOORS IBM Rational DOORS

EA Enterprise Architect

EAT Executable Acceptance Test

EC Expertise Constraint

Excel Microsoft Office Excel

IDE Integrated Development Environment

LED Light-Emitting Diode

NN Not Necessary

pvSCL pure::variants Simple Constraint Language

Rhapsody IBM Rational Rhapsody

SPES 2020 Software Plattform Embedded Systems 2020

SPES$_{XT}$ Software Plattform Embedded Systems - Extended

SPL Software Product Line

SPLE Software Product Line Engineering

TEC Technical Constraint

TIC Time Constraint

UML Unified Modeling Language

VDM Variant Description Model

VRM Variant Result Model

1. Introduction

The complexity of software-intensive systems, such as embedded systems, increases steadily (Pohl et al. [PHAB12], p.245). At the same time, stakeholders want to keep the time-to-market as short as possible, while still maintaining good quality (Clements and Northrop [CN01], p.17). Software Product Line Engineering (SPLE) has the potential to achieve these goals (van der Linden et al. [LSR07], p.287).

When employing SPLE, software producers achieve a higher efficiency of the software development process by strategically reusing common parts (*commonalities*) of multiple similar, yet different products, throughout the complete product lifecycle (Jepsen et al. [JDB07]; van der Linden et al. [LSR07], p.279). Instead of engineering the products individually, the goal is to develop them as one Software Product Line (SPL), from which *variants* can be derived. Ideally, only few or none adaptations are necessary to convert these variants to the final products. For deriving variants from an SPL, it is necessary to explicitly denote the differences between each product (*variabilities*). Systematically managing these variabilities is fundamental to the successful development of an SPL (van der Linden et al. [LSR07], p.7).

Since variability management influences the success of SPLE, it is important that good approaches for variability management exist. Developers need to define, represent, exploit, implement, and evolve variability throughout the SPLE lifecycle (van der Linden et al. [LSR07], p.8). However, several issues have not been solved so far:

1. lack of support of variability evolution (Chen and Babar [CB10]; van der Linden et al. [LSR07], p.308; Pohl et al. [PBL05], p.436f;)

2. lack of support for quality assurance in SPLs (Chen and Babar [CB10]; van der Linden et al. [LSR07], p.308; Pohl et al. [PBL05], p.436f; Lee et al. [LKL12])

3. lack of variability-management tool support (Chen and Babar [CB10]; Pohl et al. [PBL05], p.437; Czarnecki [Cza11])

4. lack of integration into software-development tools, that are used throughout the entire product lifecycle, such as tools for requirements engineering, architecture design, implementation, or quality assurance (van der Linden et al. [LSR07], p.309f; Chen and Babar [CB10])

With this thesis, we address some of these issues, and thus enable industrial practioners to further exploit the advantages of SPLE. Since well-developed tool support is important for the success of a software-engineering approach (Steimann [Ste06]), we concentrate on issues concerning tool support.

To address lack of tool support, some commercial and many research tools have been developed for variability management (van der Linden et al. [LSR07], p.310; Berger et al. [BRN+13]). We focus only on commercial tools, because research tools typically do not provide a stable production environment (van der Linden et al. [LSR07], p.310), which is necessary for industrial stakeholders. The two commercial tools *pure::variants*[1] and *Gears*[2] provide variability-management solutions throughout the entire product lifecycle. They already provide extensions for different tools used in different phases of the product lifecycle. By using these extensions, it is possible to denote variability in documents of the respective tool and generate variants from these documents. Both, pure::variants and Gears, already solve Issue 3: *Lack of tool support*, so we can focus on Issue 4. Since we write this thesis in a pure-systems context, we choose to improve pure::variants.

Regarding Issue 2: *lack of integration into software-development tools*, pure-systems already provides several integrations into tools used during the software-development process. For example, the integrations to IBM Rational Rhapsody (Rhapsody), Enterprise Architect (EA), or Microsoft Office Word support users in defining and reviewing variability in the respective tool. However, for several software-development tools, pure-systems only provides a means to generate variants, but no support for editing and reviewing variability information. Since variability is represented differently in development tools, or may even be defined in a separate model (Pohl [PBL05], p.75; Schmid and John [SJ04]), users have to learn a new approach for defining variability in each development tool.

This problem has also been identified in the context of the SPES_{XT} (Software Plattform Embedded Systems - Extended) research project. The project is a collaboration of 21 industrial and research partners to create a framework for the development of embedded systems. One challenge of the research project is to include variant-management solutions in the framework. Variant management is closely related to variability management. It is an "approach for managing the development of product variants [..., which

[1] http://www.pure-systems.com/ (last accessed on May 30, 2013)
[2] http://www.biglever.com/solution/product.html (last accessed on May 30, 2013)

enables] the development of a group of related product variants as a whole, rather than as individual, independent projects" (Dalgarno and Beuche [DB07]). pure::variants is a tool for both, variant and variability management. In a questionnaire to assess the current state-of-practice of variant management, the SPES_{XT} research partners identified that there is a demand for more consistent variant-management approaches. Based on the results of the questionnaire, they defined requirements for variant management. Regarding variant-management tool integrations, the most important requirements are:

1. Variant management shall always be done in the same way for the user independent of the used development tool.
2. Variant management shall seamlessly integrate itself into the development tools by providing a user interface.

To fulfill these requirements for pure::variants, two tasks are necessary: First, we need to improve the user-interface consistency between the existing pure::variants integrations. Second, we need to implement tool integrations for all tools that pure::variants supports. Furthermore, the number of tools that pure::variants supports constantly grows. Thus, constantly new tool integrations will be necessary. Although each tool integration should be consistent with all other integrations, and thus should provide the same functions using a similar user interface, the implementation of new integrations is not trivial. For each new integration, developers need to consider the different extension mechanisms the respective tool provides. This slows down the development process. Furthermore, with each new tool integration, the chance of inconsistencies grows, because developers choose different solutions to circumvent technical constraints of the extended tool.

To address these problems, we suggest to ease the development process of new pure::variants tool integrations by providing a consistent workflow for developing new integrations. The workflow should take into account the different technical constraints of the extended tools. To ensure consistency between tool integrations, we suggest that the workflow should include detailed requirements for all functions that the integration should support. When each tool integration is based on the same requirements, the user-interface consistency between the integrations should be high.

We refer to these tool integrations as variant-management tool integrations, because pure::variants is commonly referred to as variant-management tool, and because the requirements of the SPES_{XT} project speak of variant management instead of variability management.

Based on our suggestions, we define goals for this thesis and set tasks that are necessary for achieving these goals.

Goals and Tasks

With this thesis, we aim to provide a workflow for developing new variant-management tool integrations. The workflow should:

G1. improve the consistency between different variant-management tool integrations

G2. reduce the time-to-market for new variant-management tool integrations

By improving the consistency, we aim to reduce the time necessary for learning to operate new tool integrations and the number of errors made while using a tool integration. Furthermore, when users can access the same functions always in a consistent way, they do not need to learn how variability is represented in the respective tool. However, good consistency is not sufficient. In some cases, other factors are more important than consistency (Grudin [Gru89]). For instance, it is most important that users achieve their goals using the tool integration, and that they do so in an efficient, error-preventing way. Hence, we additionally specify quality goals for the requirements on which the workflow is based. The requirements should:

G3. promote good user experience

G4. relate to the everyday tasks of variant-management practioners

To produce a workflow that fulfills these goals, we define tasks. For addressing the quality goals (Goal 3 and 4), we first need to know which activities variant-management practitioners want to perform with the tool integrations. Hence, we define the following task:

T1. Collect and define variant-management use cases concerning the work with different product-lifecycle tools

Second, we need to define which functions the tool integrations should support. Therefore, we define Task 2:

T2. Derive a set of requirements from different sources, such as the specified use cases, the requirements of the SPES_{XT} project, and usability guidelines

Finally, we need to compile the gathered information into a workflow for developing new tool integrations. Thus, we define Task 3:

T3. Based on the requirements, propose a workflow for planning and implementing a new tool integration

Structure

We structure the remainder of our thesis as follows: In Chapter 2, we explain the background knowledge necessary to understand the rest of this thesis, such as variant management, pure::variants tool integrations, consistency, and usability. Then, in Chapter 3, we complete the tasks we defined, and thus propose a workflow for producing consistent variant-management tool integrations. To verify whether it is possible to produce tool integrations based on the workflow, we then apply the workflow to a representative selection of tools (Chapter 4). In Chapter 5, we evaluate whether the workflow meets the goals defined above. Finally, we present related work in Chapter 6, and summarize the results of our thesis in Chapter 7.

2. Background

In this chapter, we introduce background knowledge necessary to understand the rest of this thesis. For illustration of the explained concepts, we use a running example, which we introduce in Section 2.1. Since we publish this thesis in a SPES_{XT} context, we give an overview of the research project in Section 2.2. Because one of our goals is to improve the consistency of variant-management tool integrations, we describe in Section 2.3 what variant management is and how it can be realized. In Section 2.4, we explain in detail the existing pure::variants integrations for external tools. Finally, in Section 2.5, we introduce human-computer interaction terms that we need for evaluating the usability of tool integrations and defining requirements (Task 2).

2.1 The CarLight Example

For better illustration, we use a running example throughout this thesis: the *CarLight Example*. It consists of a set of requirements, test cases, and process definitions for different configurations of automotive lighting systems. We use this example, because it is sufficiently complex, but still understandable. Furthermore, it is a real-world example, since it resembles real automotive lighting requirements.

One automotive lighting system can differ in many aspects from other automotive lighting systems. This depends, for instance, on how expensive a car should be, in which country it will be sold, and how much assistance the target customer requests. The CarLight example specifies, among other things, the requirements for different parts of a car lighting system, such as the car's head lights, its cornering lights, and daytime running lights. In Figure 2.1, we show requirements for three different variants of car lighting systems: (a) represents the car lighting system of a low-budget car produced for the US-market. (b) refers to the same car produced for the European market. The only differences between (a) and (b) are the standards the system needs to fulfill: In the USA, head lights have to be conform to the Federal Motor Vehicle Saftety Standard

108 [FMV], whereas in Europe, halogen lamps need to comply with UN regulation 112 [UNR]. In Figure 2.1(c), we show the requirements for a high-end car, produced for the European market, and containing several driver assistance systems, such as automatic switch between high and low beam or daytime running light. The number of requirements compared to (a) and (b) has increased significantly. The only requirements that are the same for all three variants are *3.1 Turn Lights*.

Throughout the rest of this chapter, we extend this example with models for generating variants and more tool-specific example documents. Additionally, we use it in the main part of this thesis to visualize our concepts and implementations. Next, we present the SPES_{XT} project.

2.2 The SPES_{XT} Project

To better understand the context of this master thesis, we give an overview of the research project in this section. This thesis fulfills part of the requirements defined in the research project.

The Software Plattform Embedded Systems - Extended (SPES_{XT}) project is a follow-up of the Software Plattform Embedded Systems 2020 (SPES 2020) research project. Both projects are funded by the German Federal Ministry of Education and Research[1] and both aim to improve model-based development of embedded systems. The SPES 2020 project was an alliance of 21 industrial and research partners that focused " [...] on the professionalization of a cross-domain, model-based development method for embedded systems" (Pohl et al. [PHAB12], p.v). In 2012, SPES 2020 came to an end. Its major achievement was the creation of the SPES 2020 modeling framework and its successful evaluation. The framework enables, among other things, seamless model-based development. However, open engineering challenges were identified, such as better support of adding variation points and managing variants (Pohl et al. [PHAB12], p.254). Hence, the SPES_{XT} research project was initiated, in which the research partners address the open issues and transfer the results of SPES 2020 into an industrial-suited form (Böhm [Böh]).

The planned duration of the SPES_{XT} project is three years. It is structured in three ways: It contains six engineering challenges, four cross-cutting issues and three application domains. The research partners work on each engineering challenge with respect to each cross-cutting issues and each application domain. This thesis is part of *engineering challenge 5: variant management and reuse* and *cross-cutting issue 3: tools and tool platforms*. More precisely, it is closely related to *requirement 6.3.2* of cross-cutting issue 3, which states that "[v]ariant management shall always be done in the same way for the user independent of the used development tool." (Bodmann et al. [BGH+13]). Next, we explain what variant management is.

[1] http://www.bmbf.de/en/ (last accessed on May 30, 2013)

1. Head Lights
The beam pattern must fulfil the Federal Motor Vehicle Safety Standard 108.
1.1 High Beam
The high beam is activate if the user presses the high beam lever and the light mode switch is set to full light mode.
1.2 Low Beam
3. Indicator Lights
3.1 Turn Lights
All turn lights on a side must blink simultaenously with a frequency of 1.5 Hz when the blink lever for the respective side is activated
All turn lights must blink simultaenously as long as the hazard blinking switch is activated. The blinking frequency is 1.5 hz.

(a) Standard - USA

1. Head Lights
1.1 High Beam
The high beam is activate if the user presses the high beam lever and the light mode switch is set to full light mode.
The beam must conform to R112 — Headlamps emitting an asymmetrical passing beam and/or a driving beam and equipped with filament bulbs
1.2 Low Beam
The beam must confom to R112 — Headlamps emitting an asymmetrical passing beam and/or a driving beam and equipped with filament bulbs
3. Indicator Lights
3.1 Turn Lights
All turn lights on a side must blink simultaenously with a frequency of 1.5 Hz when the blink lever for the respective side is activated
All turn lights must blink simultaenously as long as the hazard blinking switch is activated. The blinking frequency is 1.5 hz.

(b) Standard - Europe

1. Head Lights
1.1 High Beam
The high beam is deactived temporarily if incoming traffic is detected by the camera.
The beam must conform to R112 — Headlamps emitting an asymmetrical passing beam and/or a driving beam and equipped with filament bulbs
The high beam is activated if the user presses the high beam lever and the light mode switch is set to full light mode or light mode switch is to automatic and light conditions require full light.
1.2 Low Beam
The beam must confom to R112 — Headlamps emitting an asymmetrical passing beam and/or a driving beam and equipped with filament bulbs
1.3 Fog Lights
Front fog lamps have to provide a wide, bar-shaped beam of light with a sharp cutoff at the top, and are generally aimed and mounted low.
They may be either white or selective yellow.
2. Assistance Systems
2.1 Cornering Light
2.1.1 Adaptive Forward Lighting
The adaptive forward lighting system is activated only when high or low beam is operating in full light mode.
3. Indicator Lights
3.1 Turn Lights
All turn lights on a side must blink simultaenously with a frequency of 1.5 Hz when the blink lever for the respective side is activated
All turn lights must blink simultaenously as long as the hazard blinking switch is activated. The blinking frequency is 1.5 hz.
3.2 Daytime Running Light
The DRL must be compliant with R87. It must emit white light of 450 candelas.
The LED pulse frequency above 50kHz.

(c) Luxury Car Lighting

Figure 2.1: Three different variants of car lighting requirements: (a) the requirements for a low budget car for sale in the USA, (b) the requirements for the same car for sale in Europe and (c) for a high-budget car with light assistance systems.

2.3 Variant Management

Producers of software-intensive systems strive to develop products as fast as possible, while still maintaining good quality (Clements and Northrop [CN01], p.17). If they develop multiple similar, but different products, a way to meet these demands is Software Product Line Engineering (SPLE) (Clements and Northrop [CN01], p.19). The basic idea of SPLE is to strategically reuse common parts of the product throughout the entire product lifecycle, such as the product's requirements, architecture, documentation, or test data (van der Linden et al. [LSR07], p.6). To this end, developers implement similar products as one SPL, from which they can derive different variants.

Managing an SPL throughout the entire product lifecycle can be very complex. One way is called variant management, which we explain next. For a more detailed introduction to SPLE, refer to [CN01].

2.3.1 What is Variant Management?

Variant management is an "[...] approach for managing the development of product variants [..., which enables] the development of a group of related product variants as a whole, rather than as individual, independent projects" (Dalgarno and Beuche [DB07]). Basically, this means that variant management provides a method to develop similar products as one project. From this project, developers can derive variants, which should ideally require only few adaptations to convert them to the final products.

In Figure 2.2, we show the structure of the variant-management workflow. In the domain-engineering phase, SPL developers need to model the problem space and the solution space. The *problem space* describes commonalities and variabilities of the SPL (Dalgarno and Beuche [DB07], Czarnecki and Eisenecker [CE00], p. 132). For example, problem-space modelers would specify that cars of the CarLight SPL could be for sale in the USA or in Europe. However, the same car could not be sold in both countries, because the according safety standards contradict each other.

The *solution space* describes the artifacts from which the final products will be built and their relation to the problem space (Dalgarno and Beuche [DB07], Czarnecki and Eisenecker [CE00], p. 132). One solution for this is to define variation points for all variable artifacts of the solution space. A *variation point* represents a design decision that is delayed until the generation of a variant (Bosch et al. [BFG+02]). For instance, for the CarLight example, a modeler of the solution space would add a variation point to requirements referring to different safety standards. In each variation point, he would map the requirement to a country or region defined in the problem space. Since the domain-engineering phase encompasses the usual software-development phases, he also needs to add variation points to artifacts of the architecture-design, implementation, and test phase (van der Linden et al. [LSR07], p.7).

Finally, in the application-engineering phase, SPL developers define which problem-space features each variant will contain, and then generate each variant from the solution space.

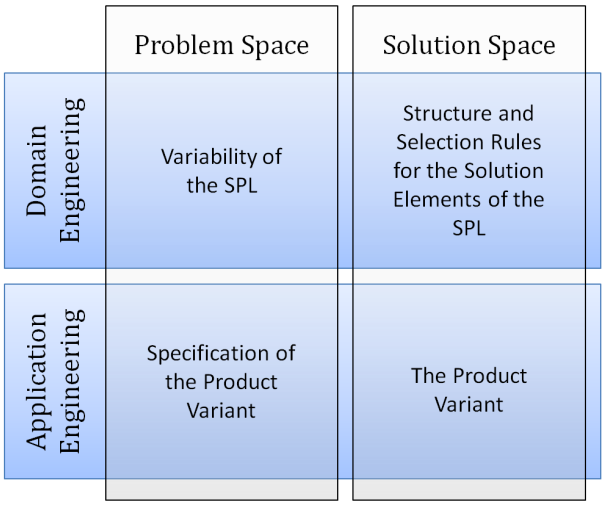


Figure 2.2: Overview of SPLE activities, based on [DB07].

There are only few commercial tools for variant management (Van der Linden et. al. [LSR07], p.309). In industry, the tools *pure::variants* by pure-systems[2] and *Gears* by BigLever Software[3] are used frequently (Berger [BRN+13]). Since we implement the proposed concepts in a pure::variants context, we explain how variant management is done with pure::variants, in the next section.

2.3.2 Variant Management with pure::variants

pure::variants is a tool for variant management that supports developers of SPLs throughout the entire product lifecycle (pure-systems [pur04]). Its user interface is an extension to the successful Eclipse Integrated Development Environment (IDE) (Garvin [Gar11]). pure::variants supports the definition of the problem and solution space with different types of models. Since we use pure::variants models in this thesis, we explain them next and describe how users work with these models to create new variants.

Defining the Problem Space To model the problem space, users create a *feature model* (Kang et al. [KCH+90], p.30), which specifies all possible features of an SPL and the relations between them. In Figure 2.3, we show such a feature model in pure::variants.

Each element with the icon Ⓕ is a feature. Its first icon refers to its *variation type*, which describes dependencies between features. Four different variation types exist in pure::variants feature models: *or*, *alternative*, *optional*, and *mandatory*. Table 2.1 matches each variation type to its icon and meaning. Apart from features, the feature

[2] http://www.pure-systems.com (last accessed on May 30, 2013)
[3] http://www.biglever.com (last accessed on May 30, 2013)

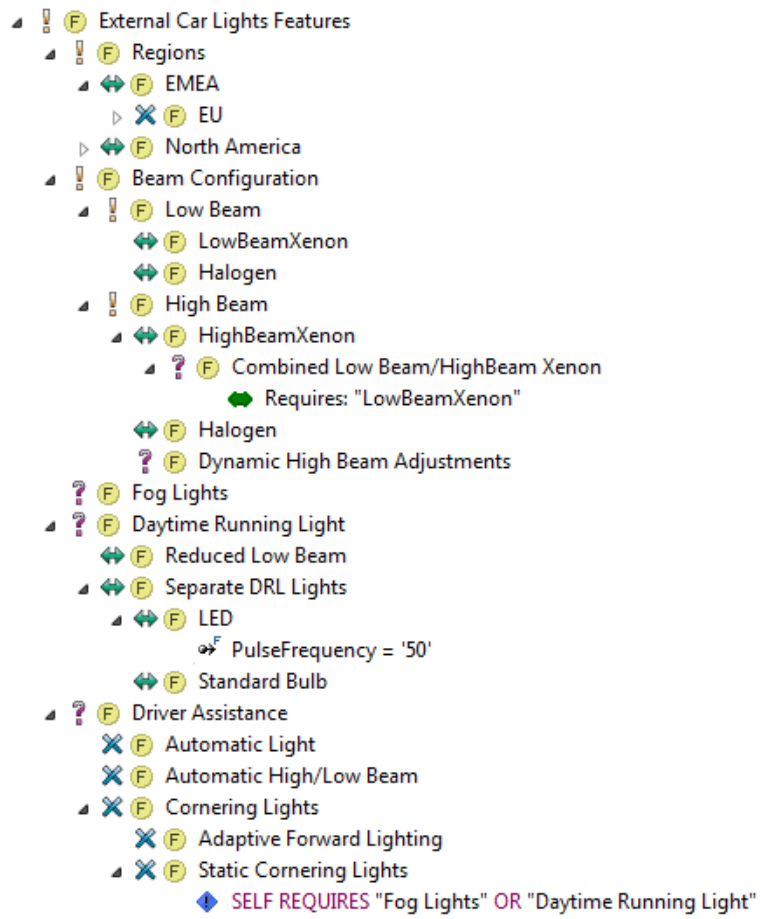


Figure 2.3: Feature model of the carlight example.

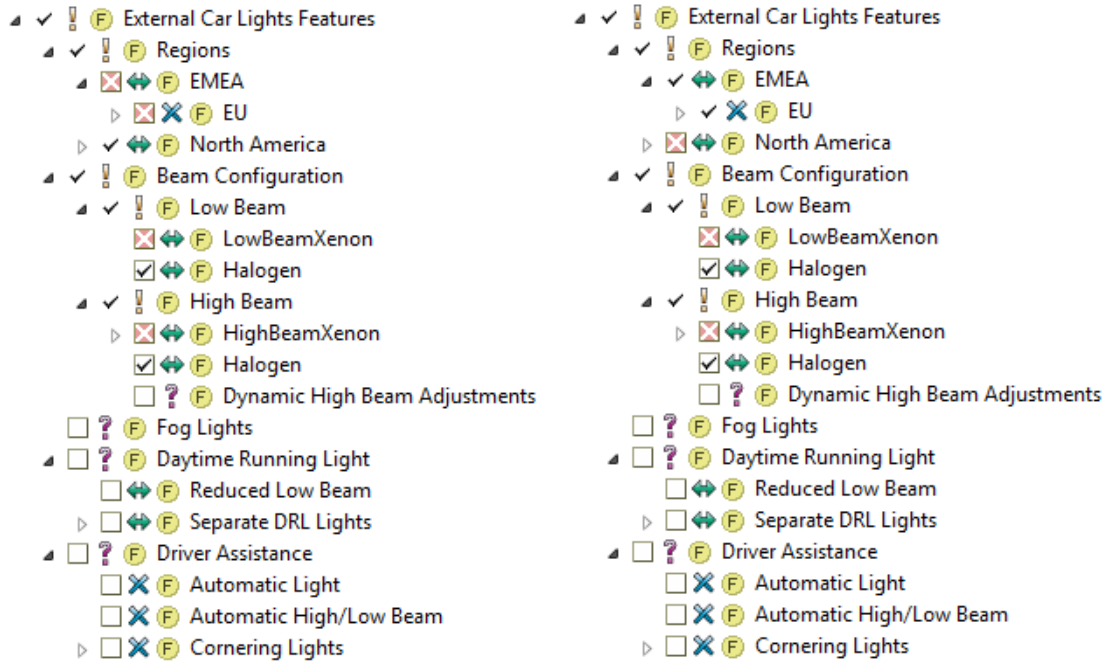
model can contain other elements, such as *constraints* ( ) and *relations* ( ). They define rules that are used to check whether a selection of features is valid. For instance, the relation *Requires: "LowBeamXenon"* states that feature *Combined LowBeam/HighBeam Xenon* can only be selected when feature *LowBeamXenon* is selected. Furthermore, *attributes* ( ) can contain values. For example, the attribute *PulseFrequency* defines that the pulse frequency of the Light-Emitting Diode (LED) is 50 kHz or higher. For a more detailed description of feature models, refer to the pure::variants documentation [pur].

To be able to derive variants, users need to select which features should be included in a variant. To this end, they define different *Variant Description Models (VDMs)*. In Figure 2.4 on the facing page, we show the two standard automotive lighting VDMs used for the CarLight-example requirements.

Defining the Solution Space When the features of an SPL and their relations are defined, variant modelers can start adding variation points to the solution space. To this end, they usually edit a document or project, from which they can later generate

Variation Type	Icon	Description
mandatory		A mandatory element is implicitly selected if the parent element is selected.
optional		Optional elements are selected independently if the parent element is selected.
alternative		Alternative elements are organized in groups. Exactly one element has to be selected from a group if the parent element is selected.
or		Or elements are organized in groups. At least one element has to be selected from a group if the parent element is selected.

Table 2.1: pure::variants variation types, according icons and descriptions taken from [pur], p.111.



(a) Standard - USA (b) Standard - Europe

Figure 2.4: CarLight VDMs that relate to the requirement-document examples shown in Figure 2.1(a) and (b).

variants. We refer to such documents or projects as *master artifact models*. Many different types of documents can be master artifact models, such as IBM Rational DOORS (DOORS) modules for requirements definition, Microsoft Word documents, or Unified Modeling Language (UML) models. To define variation points, users annotate elements of the document that contain variability. How this annotation process works in detail depends on the type of master artifact model and how variation points are represented in the model. For example, variation points can be represented by UML constraints, DOORS attributes, or by elements in a separate model that reference to variable artifacts (Heidenreich et al. [HSS+10]). In general, users attach rules to elements that define which features relate to these elements. Users can define different types of variation points, such as restrictions and calculations. *Restrictions* represent the decision whether the annotated element will be part of a variant. *Calculations* replace the annotated element with a value specified in a pure::variants model or a calculated value. The variation-point rules are given in the pure::variants Simple Constraint Language (pvSCL), a simple language for expressing constraints and restrictions, providing logical and relational operators to build boolean expressions (pure-systems [pur], p.128), or, in the case of calculations, to reference attributes of pure::variants models. For example, in Figure 2.5, a UML constraint in the *FogBulb* class represents a restriction. A variant of this class diagram will only contain the class *FogBulb* if the variant should contain feature *FogLights*.

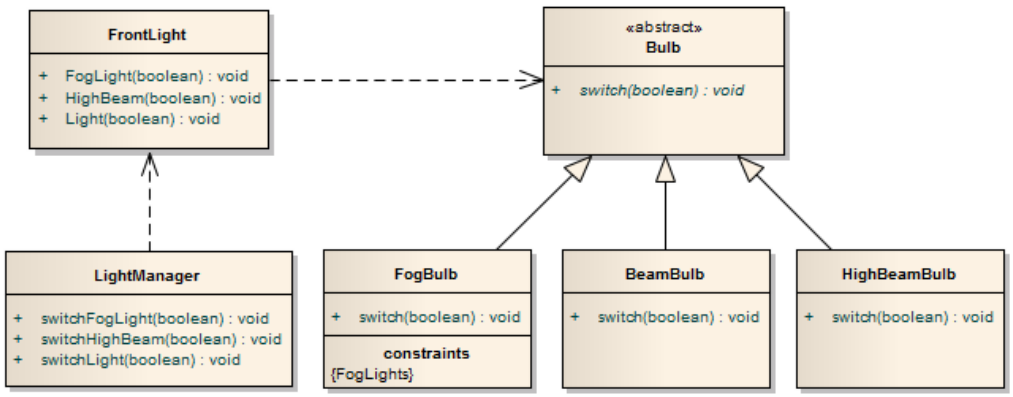


Figure 2.5: Variation point in an Enterprise Architect model.

Now, users still need to link the master artifact model to the pure::variants project. To this end, they create a *family model*, which stores information concerning the implementation (for more information regarding family models, refer to [pur], p.25).

Generating Variants Finally, for each VDM, users can trigger a transformation of the solution space, which results in a transformed copy of the master artifact model, containing only elements conforming to the selected features. In Figure 2.6, we show a variant of the class diagram presented in Figure 2.5 for one of the VDMs of Figure 2.4. Since the *FogLights* feature is not selected in these VDMs, the *FogBulb* class is deleted.

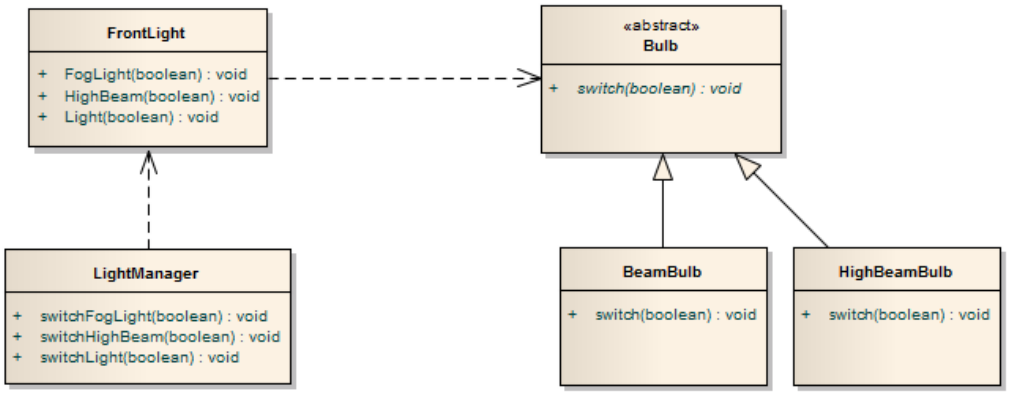


Figure 2.6: Variant of the EA model of Figure 2.5.

To support users in defining variation points, pure::variants integrations exist for different external tools. Next, we present these integrations.

2.4 pure::variants Integrations

pure::variants supports managing variability throughout the entire product lifecycle. However, developers use many different development tools during the lifecycle phases (Tell and Babar [TAB12]), such as tools for requirements engineering, designing the software architecture, implementation, or test management. To manage the variability in each of these tools, users need to be able to define variation points, which pure::variants can evaluate during variant generation. How these variation points are represented in each application, depends on which technical possibilities each tool provides. Therefore, pure-systems provides connectors for several product-lifecycle tools, such as the connector for IBM Rational DOORS (DOORS)[4], IBM Rational Rhapsody (Rhapsody)[5] or Sparx Systems Enterprise Architect (EA)[6]. Most connectors provide a transformation module that enables generating variants from a certain type of master artifact model. In Table 2.2, we present a list of pure::variants connectors. We group them by the general development-lifecycle phases, which occur in most software development lifecycle models: *Requirements engineering*, *software-architecture design*, *implementation*, and *test management* (Sommerville [Som10], p.29ff). Additionally listed are *defect tracking* and *software configuration management*, because they span multiple phases, and the category *other* which refers to all applications that we could not assign to a specific phase.

[4] http://www.pure-systems.com/pure_variants_for_Doors.166.0.html (last acc. on May 30, 2013)
[5] http://www.pure-systems.com/162+M54a708de802.0.html (last accessed on May 30, 2013)
[6] http://www.pure-systems.com/155+M54a708de802.0.html (last accessed on May 30, 2013)

Phase	pure::variants Connector for ...
Requirements Engineering	IBM Rational DOORS CaliberRM PTC Integrity HP Application Lifecycle Management Microsoft Office
Software-Architecture Design	AUTOSAR / ARTOP EMF Feature Mapping (e.g., Papyrus) Enterprise Architect IBM Rational Rhapsody Simulink
Implementation	Source Code Management
Lifecycle Management	Bugzilla ClearQuest
Test Management	HP Application Lifecycle Management
Other	Reporting with BIRT Freemind (experimental)

Table 2.2: Selection of pure::variants connectors.

The connectors we are interested in enable users to generate variants for a certain type of document and, in some cases, support users in editing variation points. For example, the connector for Rhapsody allows users to transform variants of Rhapsody UML projects and edit certain pure::variants constraints, which represent variation points. To support users when editing variation points, the connector for Rhapsody also ships with an integration into Rhapsody's user interface. This integration provides an editor for variation points with autocompletion and syntax highlighting. Furthermore, it enables users to preview variants and find errors in variation points. For the following external tools (meaning non-eclipse-based tools) pure::variants provides user-interface integrations: EA, Rhapsody, and Microsoft Office Word.

In Chapter 1, we stated that we will apply the proposed workflow to a representative selection of tools. Therefore, we need to select tools that are sufficiently different and successful. We choose to apply the workflow to IBM Rational DOORS (DOORS), Microsoft Office Excel (Excel) and Enterprise Architect (EA), since they fulfill these requirements: DOORS is a leading requirements-management tool (Hull et al. [HJD11], p.181; Alexander and Maiden [AM04], p.350). Excel is a spreadsheet application that can be used for many different tasks. It is part of Microsoft's Office Suite[7], which is used by approximately one billion people [Mic] in 2012. EA is a tool for designing the software architecture, which is currently used by approximately 300.000 people [Spa]. Next, we present each of the selected tools in detail to better understand how users work with the tools and what they want to achieve.

2.4.1 IBM Rational DOORS

IBM Rational DOORS (Dynamic Object Oriented Requirements System)[8] is a leading commercial requirements-management tool (Hull et al. [HJD11], p.181; Alexander and Maiden [AM04], p.350). It is "[...] a multi-platform, enterprise-wide requirements management tool designed to capture, link, trace, analyse and manage a wide range of information to ensure a project's compliance to specified requirements and standards" (Hull et al. [HJD11], p. 181).

A special feature of DOORS is that projects are stored on a server. Hence, people working on the same project access the same DOORS project. This is useful, because in large industrial projects, the requirements are managed usually by a team. A DOORS project consists of *folders* containing *modules*, which store the project's information. In Figure 2.7, we show such a module. Each row represents a requirement. For each row, users can define *attributes*, which contain additional information for each requirement. For example, the column *ID* in Figure 2.7 represents an attribute, which assigns an ID to each requirement.

The pure::variants connector for DOORS enables users to import information from DOORS modules into pure::variants models and synchronize it. Furthermore, requirements engineers can use it to create variants of DOORS modules. The variants are

[7] http://office.microsoft.com/en-us/ (last accessed on May 30, 2013)
[8] http://www-142.ibm.com/software/products/us/en/ratidoor/ (last accessed on May 30, 2013)

generated based on special DOORS attributes, which represent variation points. In Figure 2.7, the attribute *pvRestriction* represents the restrictions for each row. However, users need to define variation points without support, since no user-interface integration to DOORS exists.

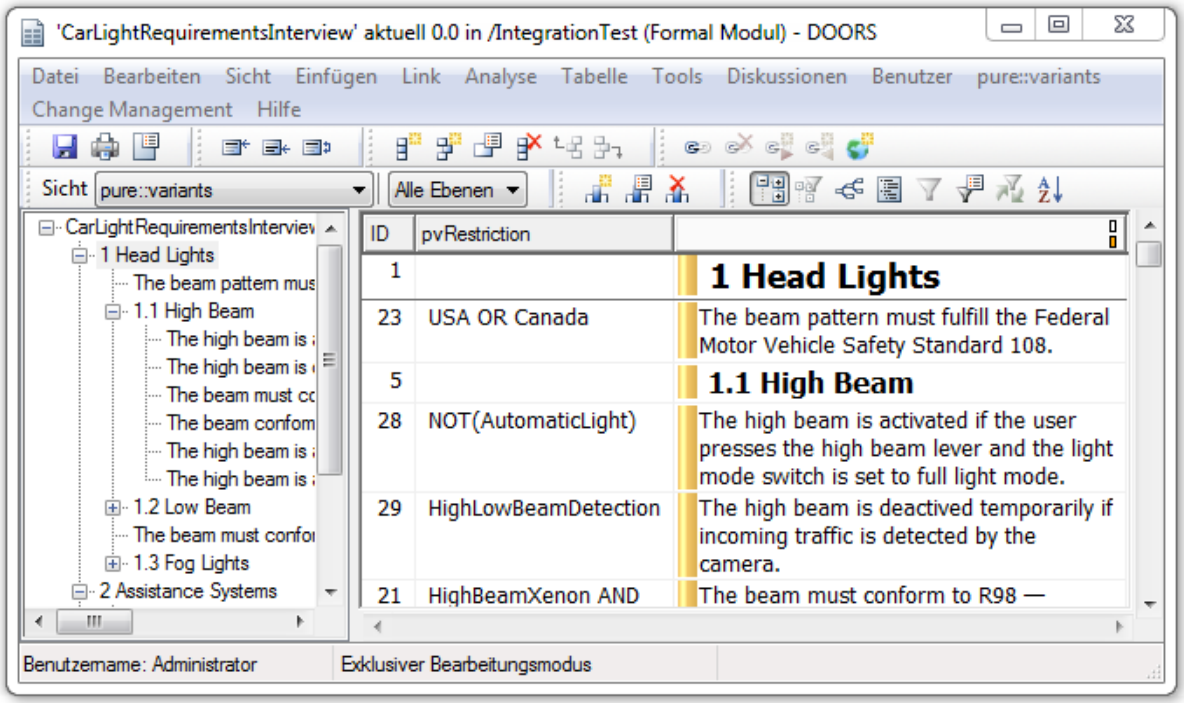


Figure 2.7: The CarLight example module in the DOORS user interface.

2.4.2 Microsoft Excel

Microsoft Office Excel (Excel) is a spreadsheet application, which supports managing large sets of data in a tabular form. It enables users to reference data, apply various computations, and visualize the data using diagrams. The main document type is a *workbook*, which can contain several *worksheets*. In Figure 2.8, we show such a workbook.

Excel can be used for many different tasks that involve managing data. Some employ it as low-budget alternative to professional requirements-management tools (Tell and Babar [TAB12], Damian and Zowghi [DZ03], Wahyudin et al. [WHE+08]) or test management tools (Lieberman [Lie], Parveen et al. [PTG07]).

Currently, the pure::variants connector for Microsoft Office only supports transformation of Word documents, for which it also provides a user-interface integration for editing variation points, previewing variants, and finding pvSCL errors.

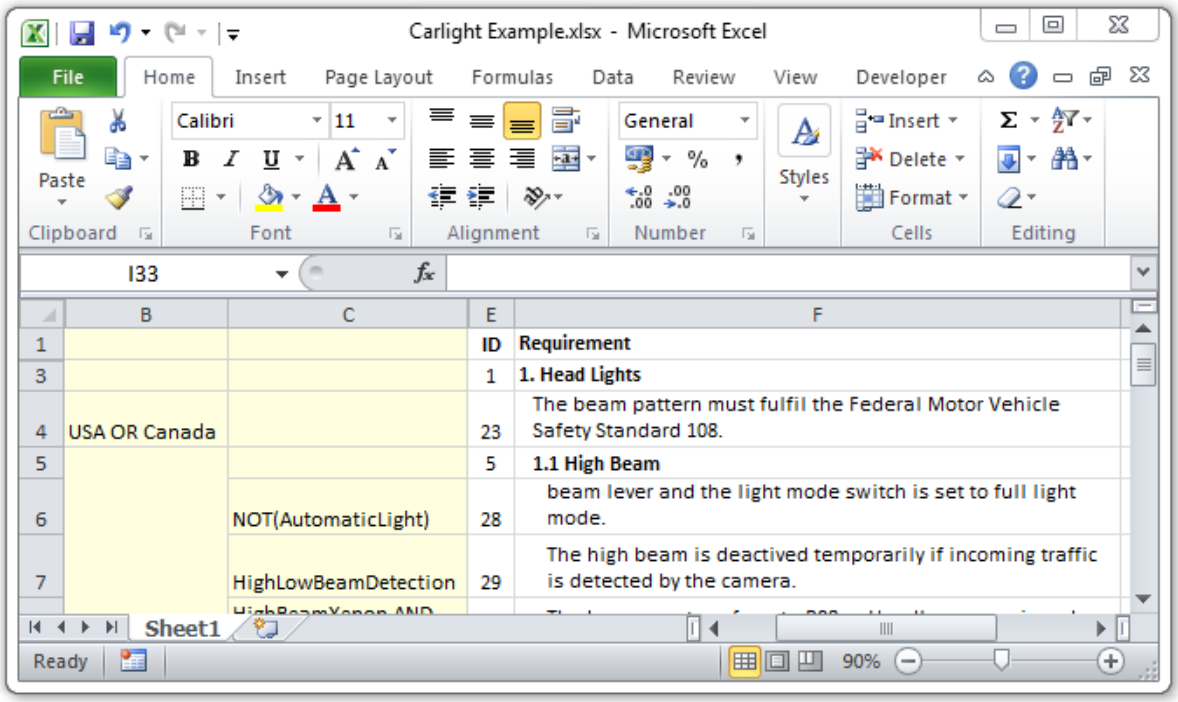


Figure 2.8: Excel user interface with CarLight example

2.4.3 Sparx Systems Enterprise Architect

Sparx Systems' Enterprise Architect (EA)[9] is a UML tool, which is mainly used for designing the system's architecture. There are many different types of elements that make up the system's architecture. Software architects organize these elements and the relationships between them in a hierarchical model. Moreover, they can visualize elements and their relations in diagrams. In Figure 2.9, we show an EA CarLight example project. The graph on the right represents a class diagram, which describes the different bulb types of an automotive lighting system. For a more detailed introduction to UML and its diagrams, refer to [RJB04].

The pure::variants connector for EA enables software architects to generate variants from an EA master artifact model based on variation points in the model. Other than the DOORS connector, it provides an integration (left in Figure 2.9). It supports users in creating variation points and provides visualizations for reviewing variability information. Next, we give an overview of the integration's user interface and its functions.

2.4.4 pure::variants Integration User Interfaces

There already exist integrations for EA, Rhapsody and Microsoft Office Word. To understand the functions and differences of the integration user interfaces, we show

[9] http://www.sparxsystems.com/enterprise-architect/index.html (last accessed on May 30, 2013)

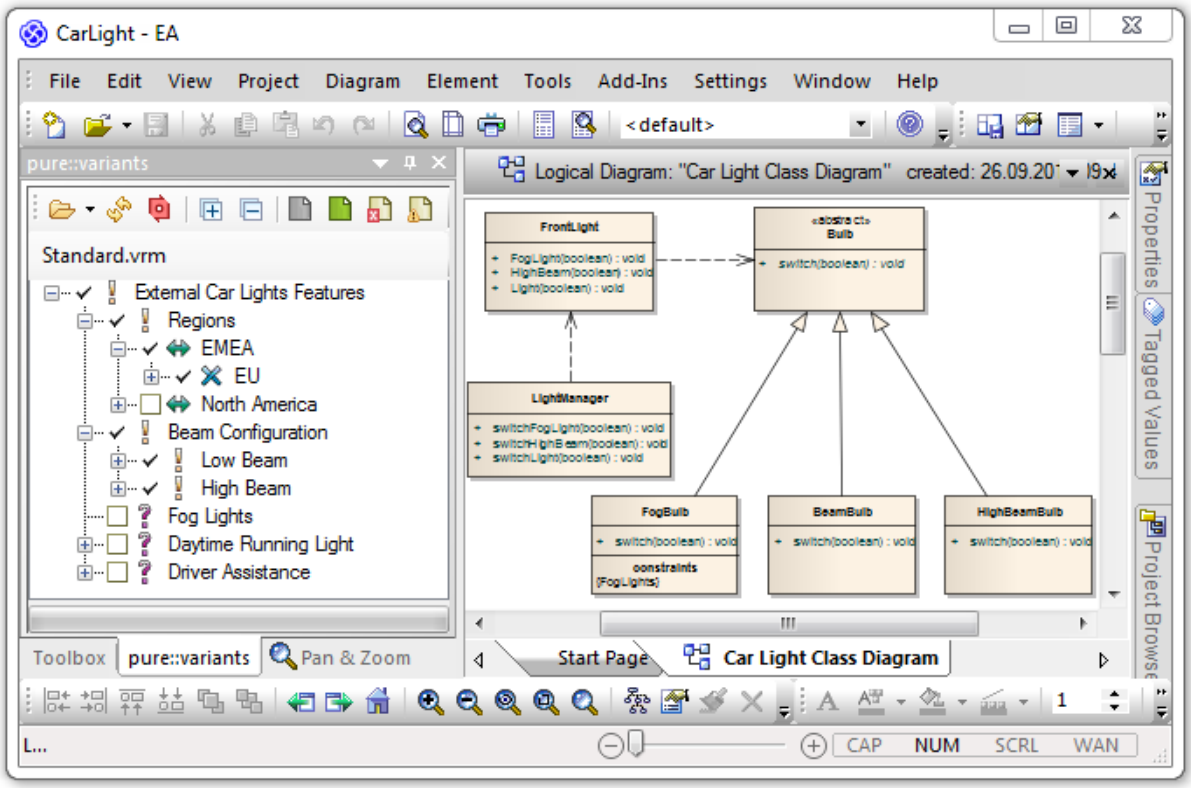


Figure 2.9: EA user interface showing CarLight example class diagram.

them in Figure 2.10. In Table 2.3, we explain the functions of each user-interface element. We use the numbers of Figure 2.10 to refer to the user-interface elements.

The figures of the different pure::variants-integration user interfaces show that the integrations are already very similar. For instance, the user-interface elements for loading and viewing pure::variants models are the same in all integrations. In all integrations users can load feature models and VRMs (A model that stores, similar to a VDM, which features are selected for the respective variant. However, it contains less information than a VDM). Nevertheless, there are several differences. For example, unlike for EA and Word, the Rhapsody integration is not embedded in Rhapsody. Moreover, the methods for adding restrictions differ: The UML integrations provide context menus for adding restrictions, whereas the Word Integration provides a button on the ribbon. Further differences concern the naming of user-interface elements and the icons used on the pure::variants ribbon tab in Microsoft Word.

This concludes the overview of the pure::variants integrations. Next, we explain human-computer interaction terms.

2.5 Human-Computer Interaction

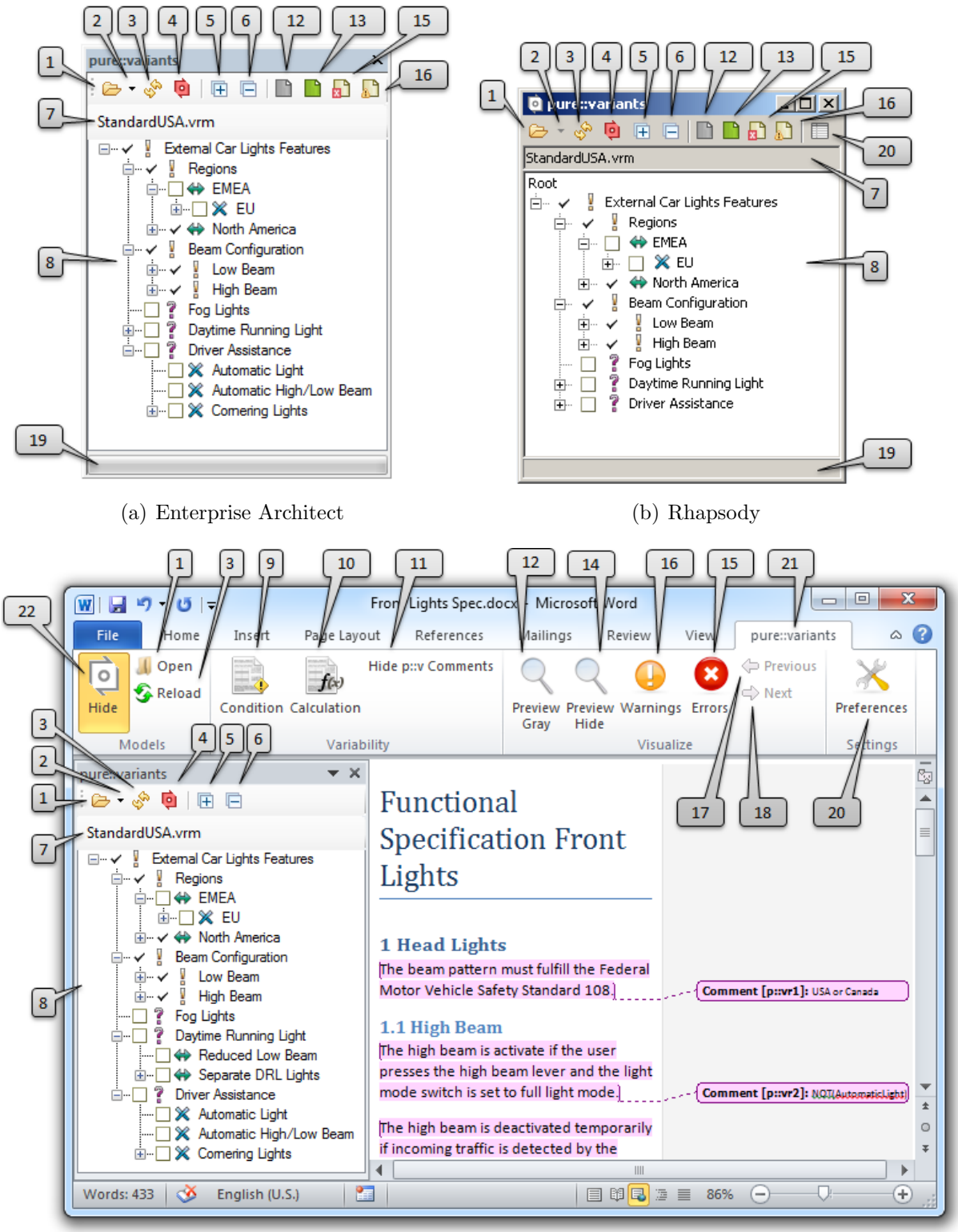
In this section, we introduce knowledge related to human-computer interaction that is relevant to this thesis. Since we aim to propose a workflow that improves consistency between tool integrations, we first explain *consistency*. Furthermore, the proposed workflow should promote good user experience. Hence, we also introduce *usability*.

2.5.1 Consistency

Consistency in the context of user interfaces means that "[...] similar elements are presented in similar ways across different displays" (Rosson and Carroll [RC02], p.127). This means that the design, icons, colors, and terminology used for user-interface elements should be the same as for other elements with the same meaning. Contrarily, elements should be clearly distinguishable from other elements with a different meaning (Preim and Dachselt [PD10], p.215).

Two types of consistency exist: internal and external consistency. *Internal consistency* relates to consistency within a single tool, whereas *external consistency* refers to the use of similar elements across different applications or, for example, an operating system (Rosson and Carroll [RC02], p.128). In this thesis, we concentrate on the external consistency between all integrations.

Consistent user interfaces can shorten the time to learn the functions of a new user interface (Polson et al. [PME86], Nielsen [Nie93], p.28f), reduce errors (Rhee et al. [RMC06]), and shorten the time needed to complete a task (Mendel [Men10]). Thus, if our integrations are consistent, users working with different tool integrations should be able to complete their tasks faster and with less errors. However, in some cases, other factors are more important for good user experience than consistency (Grudin [Gru89]). Therefore, we next explain usability, which closely relates to good user experience (Goal 3).



(a) Enterprise Architect

(b) Rhapsody

(c) Microsoft Word

Figure 2.10: Different pure::variants-integration user interfaces. We describe each user-interface element in Table 2.3 based on the annotated numbers.

Category	ID	Description
Models	1	Open a pure::variants model (a Variant Result Model (VRM) or a feature model)
	2	Quickly load recently opened pure::variants models
	3	Reload the current pure::variants models and update all visualizations
	4	Unload all pure::variants models and release the license
	5	Expand all branches of the pure::variants model
	6	Collapse the pure::variants model to the first level
	7	Shows the name of the currently loaded pure::variants model
	8	Shows a basic tree representation of the current pure::variants model
Variability	9	Add a condition (a restriction) to the selected text
	10	Add a calculation to the selected text
	11	Hide the pure::variants conditions and calculations, which are represented by Microsoft Word comments
Visualizations	12	Variant Preview: Gray out all elements of the master artifact model that are not part of the currently loaded VRM
	13	Variant Preview: Highlight all elements of the master artifact model that are part of the currently loaded VRM and are annotated with a variation point
	14	Variant Preview: Hide all elements of the master artifact model that are not part of the currently loaded VRM
	15	Error Visualization: Highlight all variation points that contain pvSCL syntax errors
	16	Error Visualization: Highlight all variation points that contain semantic errors, such as unknown feature names
	17	Jump to the previous error of the current error visualization
	18	Jump to the next error of the current error visualization
Other	19	Shows the progress of the current variant management action
	20	Show a preferences window
	21	The pure::variants tab in the Microsoft Office ribbon
	22	Show or hide the pure::variants task pane

Table 2.3: The descriptions for all pure::variants integration user interface elements. The numbers refer to the annotation of Figure 2.10.

2.5.2 Usability

Usability is "[...] a quality attribute relating to how easy something is to use" (Nielsen et al. [NL06], p.xvi). It is traditionally composed of different usability attributes that can be used to evaluate or measure the usability of a system. Next, we explain each usability attribute in detail.

2.5.2.1 Usability Attributes

We choose to define usability and its attributes based on Nielsen's definition from [Nie93]. Other definitions of usability and its attributes can be found, for example, in [Sha91] by Shackel, [ISO98] by the International Organization for Standardization or in [Lin94] by Lindgaard. The advantage of Nielsen's definition is that it subdivides usability into precise and measurable components, such that it can be systematically approached and evaluated (Nielsen [Nie93]). Nielsen defines five usability attributes: Learnability, efficiency, memorability, error rate, and satisfaction (Nielsen [Nie93], p.26). We later refer to these usability attributes, when defining requirements for consistent tool integrations and evaluating our results.

1. *Learnability* refers to the time needed to learn the functions of a new system. This time is reduced significantly when users already know, for example, a previous version of a system, or a similar system (Nielsen [Nie93], p.26,28).

2. *Efficiency* refers to how many cognitive resources are expended in relation to the accuracy and completeness with which users achieve goals (International Organization for Standardization [ISO98]). Using a system should be efficient, so that users can reach a high level of productivity after they have learned the system's functionality (Nielsen [Nie93], p.26).

3. *Memorability* describes how easy it is to remember the functionality of a system. After a period of not using the system, users should be able to return to it without having to learn its functionality again (Nielsen [Nie93], p.26).

4. The *error rate* of a system refers to the number of errors users make while using the system. For good user experience, this rate should be low. When an error occurs, users should be able to recover from it easily. Fatal errors that cause users to loose all unsaved work, or small errors that are not discovered by users and lead to faulty results should not occur (Nielsen [Nie93], p.32-33).

5. *Satisfaction* relates to how much users like using aspects of a system. It is a subjective attribute, which can be measured, for example, by an interview with users of the system (Shneiderman [SP04], p.16).

To address these attributes, Nielsen advises to follow usability principles, which we explain next.

2.5.2.2 Usability Heuristics

Optimizing the introduced usability attributes improves the overall usability of a system. To indicate usability problems and fix them, there are numerous sets of design guidelines (Johnson [Joh10], p.xi). When followed, they help developers in creating user interfaces providing a good user experience, and when used to analyze a system, they help identifying usability issues. For consistency, we use Nielsen's design guidelines introduced in [Nie94]. The guidelines are named *usability heuristics*. Similar heuristics can be found in [SP04] by Shneiderman and Plaisant or [GP96] by Gerhardt-Powals.

Next, we enumerate and explain Nielsen's heuristics.

1. *Visibility of system status*: A system should always provide feedback about what it is doing. Each user input should be followed by a reaction of the system. Especially, when a system has long response times, feedback is important. When the delay is greater than 10 seconds, users have to be informed of how long they have to wait, for example, by a progress indicator (Nielsen [Nie93], p.135f; Card et al. [CRM91]).

2. *Match between system and real world*: A system should employ the users' terminology to communicate with them, rather than using its internal terminology (Nielsen [Nie93], p. 123). Furthermore, it should take the users' background knowledge into account(Nielsen [Nie94]).

3. *User control and freedom*: Users should always feel in control of the system they are using. For example, the system should provide an option to undo or redo an action, close an open dialog, and interrupt a process that takes too long (Nielsen [Nie93], p.138f).

4. *Consistency and standards*: A system should be consistent, which means that required actions should be consistent in similar situations (see Section 2.5.1). Terminology in prompts, menus and help screens should be identical, and consistent color, layout, fonts, etc., should be used (Shneiderman [SP04], p.74; [Joh10], p.133ff, 143).

5. *Error prevention*: User-interface elements should be chosen such that the error rate is kept at a minimum (Nielsen [Nie94]; Shneiderman [SP04], p.75).

6. *Recognition rather than recall*: Since users are much better at recognizing something than at remembering it without help, users should not enter data completely from scratch. Instead, the system should rely on user-interface elements that let users choose from a list of possible items (Shneiderman [Shn97], p.129). For example, typing errors can be reduced by proposing possible words, because users have to type less.

7. *Flexibility and efficiency of use*: A system should be flexible and efficient to use. This can be achieved, for example, by providing shortcuts to frequently used functionality (Nielsen [Nie94]).

8. *Aesthetic and minimalist design*: Users should like the design of a system. Therefore, it should be aesthetic. Furthermore, the number of information pieces processed in short-term memory is limited to seven plus or minus two units (Miller [Mil56]). Hence, displays and the system's design should be kept as simple as possible. (Nielsen [Nie94])

9. *Error recognition and recovery*: In case an error occurs, the system should detect the error immediately and offer simple, constructive, and specific instructions for recovery (Nielsen [Nie94]; Shneiderman [SP04], p.75).

This concludes the necessary background knowledge. In the following chapter, we propose concepts for consistent tool integrations.

3. Concepts for Consistent Tool Integrations

Having introduced all necessary background knowledge, we now address the main goals of this thesis. We propose a workflow for developing new variant-management tool integrations that:

G1. improves the consistency between different variant-management tool integrations

G2. reduces the time-to-market for new variant-management tool integrations

G3. promotes good user experience

G4. relates to the everyday tasks of variant-management practioners

To fulfill these goals, we complete the tasks we specified in Chapter 1:

T1. Collect and define variant-management use cases concerning the work with different product-lifecycle tools

T2. Derive a set of requirements from different sources, such as the specified use cases, the requirements of the SPES_{XT} project, and usability guidelines

T3. Based on the requirements, propose a workflow for planning and implementing a new tool integration

In Section 3.1, we present the collected use cases. Then, we derive requirements in Section 3.2. Based on these requirements, we propose a workflow for planning and implementing a new tool integration in Section 3.3.

3.1 Tool-Integration Use Cases

In this section, we address Task 1: *Collect and define variant-management use cases concerning the work with different product-lifecycle tools.*

In the context of the SPES$_{XT}$ project, such a use case already exists: Use Case *QT3-UC-1* describes a scenario for consistent variant management throughout the entire product lifecycle (see Figure 3.1). This use case already addresses our aim for consistent tool integrations. However, it is kept very general. Hence, we next define more detailed use cases that focus on the work with different product-lifecycle tools. We identify two main activities that should be represented by the use cases: First, users always need to add variability information to a master artifact model. Second, users may want to verify whether the variability information is correct. Especially in an industrial context, verification may be part of a workflow. We define the use cases independent of the used product-lifecycle tools, because the main activities are the same for every tool. The scenarios would only differ in details, such as the representation of variation points or how users can access variation points. Next, we describe each use case and show its main description. For the full use cases, refer to Appendix A. We present the use cases in the same order users would execute them. Hence, we start with use cases that address adding variability to master artifact models. Then, we define use cases that concern reviewing variability information.

3.1.1 Use Case 1: Adding Restrictions

In Use Case 1, we describe a scenario for adding restrictions in an efficient and error-preventing way. We define such a use case, because users frequently need to add restrictions when they edit the variability of a master artifact model. We describe the scenario as follows:

> *A variant modeler wants to edit the variability of a master artifact model. To this end, he identifies an element that should not be contained in all variants. Since the project is very complex, he needs to consult the pure::variants feature model or variant model to find out which features relate to which elements. Therefore, he loads a feature model or variant result model, which he can consult during variation-point creation. Then, he selects the identified element and opens the pvSCL editor for defining a restriction. A restriction is typically composed of feature names and keywords. To enter the correct feature names, the variant modeler looks at the feature model loaded in the pure::variants integration and uses the editor's autocompletion feature. When he presses the OK button, the rule is checked for syntax errors and unknown feature names. If an error is found, he is notified and can choose to correct or ignore the error.*

As stated in SPES$_{XT}$ Use Case *QT3-UC-1*, users should be able to edit variability in the same way for different types of master artifact models (see Figure 3.1). Providing

ID	**QT3-UC-1**	**Consistent variant management in each phase of a system's development**	
Detailed Description	A team of product managers and portfolio managers is responsible for the development of a central car locking system. The team starts brainstorming and they envision a system with a lot of different features, depending on how much a customer is prepared to pay. The output of that brainstorming is a high level requirements document and list of features. The features are then modeled within a feature diagram describing the relationships between them. That model is the location and base of each and any variability related information. After this, a detailed planning is started and the steps following the V-Model are processed. Every time an artifact is being handled concerning variability, the relation to the feature model has to be established and this always in the same kind. This ensures that learning the variant management is done only once, which the later maintenance additionally crucially eases. To enable this, the variant management helps the user during setting up the linking for example by providing a seamless UI integration for the respective tool.		
Rationale	The aim of consistent variability management in general is to: • establish the *identification and representation of variability* in all relevant artifacts (e.g. requirements models, behavior models, and so on • ensure the consistency between all of these models by *specifying and maintaining the dependencies* among them • enable v*ariant specification and configuration* to describe a valid selection of features from a set of features where all feature interdependencies are satisfied.		
Related QT3-Goal(s)	Interoperability, Variant Management		
Related Use Cases	None		
Actor(s)	• Product managers and portfolio managers • Variant Modeler • Requirements Engineer • Software Developer • Test Engineer • Tester		
Tools/Utilities	• Variant Management tool (e.g. pure::variants) • Requirements tool (e.g. Doors), etc.		
Pre-condition	No other artifacts exist		
Expected Results	Consistent variant management is established for the system in question.		
Post-condition	The variant management is used consistently throughout the whole development and further on in the next phases of the system's life cycle.		
Main Scenario	**Step**	**Actor**	**Interaction**
	1	Product managers and portfolio managers	Brainstorming about the system's features, Feature List and High Level Requirements are the output.
	2	Variant Modeler	Creates Feature Model from input
	3	Requirements Engineer	Writes detailed requirements
	4	Variant Modeler	Linking of requirements to related features
	5	Tester Engineer	Specifying Tests for the requirements and linking them accordingly
	6	Software Developer	Realizes the requirements and link created artifacts with the related features.
	7	Variant Modeler	Specifies different variants of the system
	8	Tester	Executes the tests for each variant and documents the results
Alternative Scenario 1	**Step**	**Actor**	**Interaction**
	4	Requirements Engineer	Linking of requirements to related features
Alternative Scenario 2	**Step**	**Actor**	**Interaction**
	4	Variant Modeler and Requirements Engineer	Linking of requirements to related features

Figure 3.1: SPES_{XT} use case for consistent variant-management tool integrations

a pvSCL editor that is the same in all supported tools should address this goal. The described editor is based on [Pap11], where we first introduced a tool-integration editor for pvSCL rules that supports autocompletion, syntax highlighting, and error checking.

We argue that the presented editor improves efficiency and reduces the error rate for several reasons: First, users should not enter information from scratch, because this reduces efficiency and may lead to typographic errors (Shneiderman and Plaisant [SP04], p.77; usability heuristic *recognition rather than recall*). Since variant modelers already know the pure::variants pvSCL editor, it makes sense to provide a pvSCL editor for consistency. Hence, it should support autocompletion and syntax highlighting. Second, to reduce the error rate, errors should be prevented before writing them to the master artifact model. Since pvSCL syntax errors and unknown feature names can be checked automatically, such a check should be part of the editor. A further source of inefficiency is the need to switch to pure::variants when users want to view the features they are referencing. Therefore, they should be able to load and display pure::variants models directly in the external tool.

3.1.2 Use Case 2: Adding One Restriction to Multiple Elements

In Use Case 2, we extend the scenario of Use Case 1, such that users can add multiple restrictions at once. We specify such a use case, because in some master artifact models the same restriction applies to different elements. In this case, users would need to repeat the scenario from Use Case 1 repeatedly. Hence, we describe an alternative scenario:

> *While editing the variability of a master artifact model, a variant modeler encounters several elements that should be annotated with the same restriction. To save time, he creates one restriction element and attaches it to all elements. The advantage of this approach is that he only needs to edit one restriction. However, it is not possible in all development tools or in all cases (for example, in UML models, adding one restriction to different elements would only be possible on the same diagram). Therefore, he can alternatively select all elements and trigger the restriction editor. When confirming the entered rule, it is added to all selected elements. When he later needs to edit the rule, he again selects all relevant elements. If their rules are the same, the restriction editor shows the existing restriction. When the variant modeler confirms the entered rule, it is written to all selected elements.*

Use Case 2 increases the efficiency of adding restrictions, if one restriction often applies to different artifacts. This may not be the case if the extended tool supports adding one restriction to multiple elements. In this case, one restriction can relate to multiple elements. However, some development tools do not support the relation of one restriction to different elements. Hence, the integration should enable this approach to ensure efficient editing of restrictions.

3.1.3 Use Case 3: Adding Calculations

In Use Case 3, we describe a scenario for adding another type of variation point: a *calculation*. Since customers requested support for calculations, we define this use case. In contrast to restrictions, a calculation does not represent the transformation decision whether an annotated element exists in a document. Instead, the annotated element is replaced with a value specified in the problem space (or calculated based on problem-space values). The scenario of this use case is as follows:

> *A variant modeler wants to edit the variability of a master artifact model. The model contains, for example, a set of parameters. Most variable parts of the master artifact model apply to multiple variants. The variant modeler annotates those parts with restrictions as described in Use Case 1. However, some parameters of the master artifact model differ for nearly every variant. Since, in this case, restrictions would clutter the document unnecessarily, he uses the concept of calculations instead. Hence, he opens the pure::variants project and defines each parameter in an attribute of the project's feature model. In each variant description model, he sets a different value for this attribute. Back in the external tool, he selects the text fragment that should contain one of the parameters after variant generation and triggers the pure::variants calculation editor. Here, he references the attribute. Consistent with the restriction editor, the calculation editor supports editing rules supported by autocompletion and error check. Furthermore, it provides the same options to ignore errors and to cancel editing. Later, during variant generation, the text fragments annotated with calculations will be replaced with the value of the referenced attribute.*

pure::variants customers stated that they frequently use parameters in master artifact models (e.g., requirement documents). It would be possible to create the described master artifact model using only the concept of restrictions. However, it would clutter the model with restricted elements that would each be contained in only one variant. Calculations solve this problem, because only one calculation is required instead of multiple restrictions, and users can better trace parameters to each VDM. For consistency between the editors, the calculation editor should provide the same functions as the restriction editor.

3.1.4 Use Case 4: Finding Errors in Variation Points

In Use Case 4, we describe a method for supporting users during their search for errors in pvSCL rules. We describe this use case, because users often need to verify whether the entered pvSCL rules are correct. Hence, they should be supported in this task. The description of Use Case 4 is as follows:

> *After adding variability, the variant modeler or a variant verifier wants to check if all entered pvSCL rules and the pure::variants models are correct*

and consistent. First, he wants to verify whether all pvSCL rules of variation points comply with the pvSCL syntax. To this end, he uses the pure::variants integration to load all feature models relevant for the master artifact model. Then, he triggers a syntax-error visualization, which highlights all elements that are not compliant with pvSCL syntax. To find errors that are not in the active viewport, he uses the integration's navigation buttons. When he has found an error, he selects the related element and triggers the according pvSCL editor, which shows the type of error in the title bar. After correcting all syntax errors, he wants to check whether all features referenced in pvSCL rules exist in the pure::variants feature model and are spelled correctly. To this end, he triggers a semantic-error visualization, which highlights all elements that contain unknown names, such as feature or attribute names. When he has found an error using the same navigation methods as for syntax errors, he selects the related element and triggers the calculation editor, which underlines all unknown names.

With this use case, we aim to improve the efficiency of the variability-verification process. We choose to provide a visualization for errors in pvSCL rules, because colored elements are perceived by users at first glance (preattentively) (Goldstein [Gol10], p.144). Without this visualization, users would have to check the transformation errors and manually search for the element of the error message. This approach would be inefficient, since users would need to switch to pure::variants and do a time-consuming transformation. Furthermore, unknown feature or attribute names would not be detected at all. Since we already proposed such a visualization, refer to [Pap11] for a detailed reasoning on pvSCL error visualizations.

3.1.5 Use Case 5: Finding Design Errors

In Use Case 5, we describe a scenario for checking whether the edited variation points produce the desired results. We specify such a use case, because variant modelers or variant verifiers often need to check whether the edited variation points lead to the correct variant. The common way to do this is to transform a variant and examine the result. However, this would be slow, since the user would need to switch to pure::variants and transform a variant. Hence, we propose a more efficient scenario:

A variant verifier (or variant modeler) wants to check whether the entered variation points, combined with the problem-space definition in pure::variants, produce the desired variants. To this end, he opens the master artifact model and loads a variant description model. To see which elements would be included in the variant and which would not, he triggers a preview. He can choose to either gray out or hide elements that would be deleted. Graying out elements has the advantage that deleted elements are still visible, which makes it easier to check whether they should really be deleted. Hiding elements may be better to get an overview when large parts of the master

artifact model would be deleted. If the variant verifier finds a design error in a variation point, he triggers the respective pvSCL editor and corrects it. When all errors are corrected, the variant verifier triggers a transformation for a final check of the variant.

We believe that previewing variants is more efficient than transforming a variant and looking at the results, because transforming a variant and opening the output document can be time consuming. When using preview visualizations, the variant verifier needs to open the master artifact model only once. Furthermore, all pvSCL rules are visible during preview, which makes it easier to check whether they are correct. We choose two different visualizations, because they have different advantages. On the one hand, the gray-out preview has the advantage that, although users can differentiate between included and excluded elements at first glance, all elements are still visible. Thus, identifying falsely deleted elements is easier.

On the other hand, the hide preview declutters large documents and gives an overview of how the variant will look like. In [Pap11], we already proposed a preview that grays out elements not included in a variant.

This concludes the use cases for consistently manipulating and verifying a master artifact model. Next, we derive requirements from the presented use cases.

3.2 Requirements Analysis

In this section, we present requirements for consistent variant-management tool integrations. Since existing tool integrations are quite complex, we define the requirements hierarchically. Thus, few top-level requirements provide an overview of all requirements, whereas subrequirements define sufficient details. In Section 3.2.1, we specify all top-level requirements. Then, we define subrequirements for each of the main requirements in Section 3.2.2.

3.2.1 Top-Level Requirements

We derive the top-level requirements from different sources:

1. Requirements specified in the SPES_{XT} project
2. Use cases of Section 3.1
3. Experience with previous tool integrations
4. Usability heuristics (see Section 2.5.2.2)

Next, we list for each of the sources the requirements we derived from it. To get a first overview of the requirements, we present only their title. We give a more detailed description in Section 3.2.2.

SPES$_{XT}$ Requirements

In the SPES$_{XT}$ project, five requirements are defined that relate to variant-management tool integrations (Bodmann et al. [BGH+13]):

1. Variant management shall always be done in the same way for the user independent of the used development tool.
2. Variant management shall seamlessly integrate itself into the development tools by providing a user interface.
3. Variant management shall support the visualizing of variability-affected model elements.
4. Variant management shall support the previewing of variants.
5. Development tools shall support the user during the creation of variation points.

Since these requirements relate to variant-management tool integrations, we reuse them as top-level requirements.

Requirements Derived from Use Cases

The SPES$_{XT}$ requirements already cover part of the use cases we specified in Section 3.1: Requirement 4 addresses Use Case 5: *Finding Design Errors*, while Requirement 5 relates to all use cases concerned with adding variation points (Use Cases 1 - 3). Only Use Case 4: *Finding Errors in Variation Points* is not addressed by any of the requirements. Hence, we add a top-level requirement

6. The tool integration shall support users in finding variation-point errors.

Requirements Derived from Experience

Since the tool integration should support users in adding variation points, deleting these variation points should also be possible. From the integration to EA, we already know that deleting variation points can be inefficient. Hence, the integration should support users in deleting variation points if the method provided by the extended tool is inefficient. To this end, we add a top-level requirement

7. The tool integration shall support users in deleting variation points.

Requirements Derived from Usability Heuristics

One of our goals is to provide concepts that foster good user experience. Since none of the defined requirements explicitly mentions user experience, we add a top-level requirement

8. The tool integration shall comply with usability heuristics.

To ensure good user experience, this requirement should be further detailed with sub-requirements that relate to Nielsen's usability heuristics (see Section 2.5.2.2).

This concludes all top-level requirements. Since the current order of the requirements does not relate to their contents, we reorder the requirements. To emphasize the importance of consistency and usability, we put all general user-interface requirements first. For consistency with the use cases, we order the rest of the requirements in the same way as the related use cases:

1. Variant management shall always be done in the same way for the user independent of the used development tool.
2. The tool integration shall comply with usability heuristics.
3. Variant management shall seamlessly integrate itself into the development tools by providing a user interface.
4. Development tools shall support the user during the creation of variation points.
5. The tool integration shall support users in deleting variation points.
6. Variant management shall support the visualizing of variability-affected model elements.
7. The tool integration shall support users in finding variation-point errors.
8. Variant management shall support the previewing of variants.

Next, we shortly describe each requirement and define more detailed subrequirements.

3.2.2 Detailed Requirements

In this section, we describe for each top-level requirement which detailed subrequirements we define and why. For brevity, we show only the full description of the top-level requirements. For the subrequirements, we omit the description. Instead, we show a tree of subrequirements to get an overview of the requirement's hierarchy. Refer to Appendix B for the full requirement descriptions.

Requirement 1: Variant management shall always be done in the same way for the user independent of the used development tool.

ID: 1 (6.3.2)	Variant management shall always be done in the same way for the user independent of the used development tool.
Description	If a developer or a maintainer works with different tools during the development, such as DOORS for requirements and Matlab/Simulink for model-based software development, variant management will be always done and used in the same manner in these tools.
Rationale	If the requirement can be fulfilled, the overall learning effort decreases because just a single method has to be known, leading usually to fewer errors during the development.
Use Case	QT3-UC-1
Source	SPESXT
Acceptance Criterion	All subrequirements are fulfilled

SPES$_{XT}$ Requirement 1 relates to Goal 1: The workflow should improve the consistency between different variant-management tool integrations. It is a very general description of consistent variant management in external tools. To produce a consistent user interface, its design, messages, labels, and icons need to be consistent (see Section 2.5.1). Hence, we detail Requirement 1 as follows:

1. Variant management shall always be done in the same way for the user independent of the used development tool.
   1.1 Tool integrations shall use the same or similar icons for the same functions.
   1.2 Tool integrations shall communicate similar messages in similar situations.
   1.3 The integrations' user-interface design shall be similar for all integrations.

To realize these requirements, we propose that only one implementation should be used for the base user interface. It should be possible to embed this user interface in most external tools. The base implementation should also contain icons, messages, and labels, so these can be referenced in tool-specific implementations.

Requirement 2: The tool integration shall comply with usability heuristics.

ID: 2	The tool integration shall comply with usability heuristics.
Description	The tool integration should fulfill rules for achieving good user experience.
Source	Usability
Acceptance Criterion	All subrequirements are fulfilled

Since one of our thesis goals is to foster good user experience, we define Requirement 2. It states that tool integrations should comply with usability heuristics. In Section 2.5.2.2, we introduced Nielsen's usability heuristics. They are basic rules that help create user interfaces that provide a good user experience. The obvious way to integrate the heuristics into the requirements document would be to add a requirement for each heuristic. However, the heuristics are formulated very general. Therefore, we instead identify for each heuristic specific requirements and add them to Requirement 2:

2. The tool integration shall comply with usability heuristics.
   2.1 The tool integration shall display a wait cursor for actions that take longer than two seconds.
   2.2 Tool integrations shall provide progress bars for actions that take longer than ten seconds.
   2.3 Tool integrations shall be consistent with pure::variants and the extended tool.
   2.4 Users shall be able to interrupt all actions that take longer than 10 seconds.
   2.5 For all actions that change the master artifact model, the tool integration shall provide a means to undo or redo the action.
   2.6 Users shall always be able to cancel a dialog.

2.7 Tool integrations shall warn users before performing an irreversible action.
2.8 The tool integration shall always support users in entering data.
2.9 The tool integration's user interface and visualizations shall be kept as simple as possible.
2.10 If an error occurs, the tool integration shall provide simple, constructive, and specific instructions for recovery.

Regarding Heuristic 1: *Visibility of system status*, Nielsen states that progress bars are important for all actions that take longer than ten seconds. For actions with a duration between two and ten seconds, a wait cursor is sufficient (Nielsen [Nie93], p.136f; Myers [Mye85]). Hence, we add Requirements 2.1 and 2.2.

Heuristic 2: *Match between system and real world* states that a system should employ the user's language instead of internal terminology. Also it should make use of the user's background knowledge. Since users are already familiar with pure::variants and its terminology, we add Requirement 2.3: *Tool integrations shall be consistent with pure::variants and the extended tool.*

For Heuristic 3: *User control and freedom*, we define three requirements: Users are more efficient, and learn faster in low-risk environments (Johnson [Joh10], p.149). Thus, they should always be able to cancel a dialog (Requirement 2.4; Nielsen [Nie93], p.138), undo and redo an action (Requirement 2.5; Johnson [Joh10], p.149), and interrupt processes that take longer than ten seconds (Requirement 2.6; Nielsen [Nie93], p.139).

For Heuristic 4: *Consistency and Standards*, we do not add a requirement, because Requirement 1 already covers consistency between tool integrations, and Requirement 2.3 refers to consistency with pure::variants and the extended tool.

Heuristic 5: *Error prevention* states that errors should be avoided. Especially errors that cannot be reversed should be prevented (Nielsen [Nie93], p.146). Therefore, we add Requirement 2.7: *Tool integrations shall warn users before performing an irreversible action.* We do not add more requirements for Heuristic 5, because we already try to prevent errors by providing autocompletion and error check.

Regarding Heuristic 6: *Recognition rather than recall*, Nielsen states that having to enter data from scratch involves the risk of spelling errors ([Nie93], p.144f). Hence, we add Requirement 2.8. It specifies that the tool integration should support users when they enter data, such as a pvSCL rule, or the path to a pure::variants model.

For Heuristic 7: *Flexibility and efficiency of use*, we do not add an explicit requirement, because we already address efficiency by providing autocompletion, preview and error visualizations, and by displaying pure::variants models directly in the development tool.

To address Heuristic 8: *Aesthetic and minimalist design*, we add Requirement 2.9. It states that all parts of the user interface and its visualizations should be as simple as possible, because the number of information pieces processed in short-term memory is limited to seven plus or minus two units (Miller [Mil56]). We do not add a requirement

for aesthetic design, because it is a subjective feature, and thus we cannot add a precise acceptance criterion.

For Heuristic 9: *Error recognition and recall*, we add Requirement 2.10, which specifies that error and warning messages should be understandable and helpful (Nielsen [Nie93], p.142ff).

Requirement 3: Variant management shall seamlessly integrate itself into the development tools by providing a user interface.

ID: 3 (6.3.4)	Variant management shall seamlessly integrate itself into the development tools by providing a user interface.
Description	Seamless integration is meant in the sense that the developer is not just able to get access to variability information directly within its used environment but also to interact with the variant management for example the developer is guided during the specification of variant restrictions and those restrictions are also validated against the variability model.
Rationale	Providing a seamless integration of variant management with development tools a) eases the usage, b) decreases the risk of a wrong usage, and c) usually fosters the usage. Seamless integration usually also increases the acceptance through the user, because she has not to switch between tools anymore to get her job right.
Use Case	QT3-UC-1
Source	SPESXT
Acceptance Criterion	All subrequirements are fulfilled

SPES_{XT} Requirement 3 specifies that tool integrations should provide a user interface that is embedded in the development tool and users should be able to trigger variant-management actions from the development tool. However, embedding a user interface has already proven difficult. For instance, in Rhapsody, it was not possible to integrate the user interface (Papendieck [Pap11]). Therefore, we define alternative subrequirements. When implementing a new tool integration, developers only need to fulfill one of the following subrequirements:

3. Variant management shall seamlessly integrate itself into the development tools by providing a user interface
   - 3.1 The tool integration's user interface shall integrate itself directly into the development tool's user interface.
   - 3.2 The tool integration's user interface shall be displayed in its own window, which can be triggered from the development tool's user interface.
     - 3.2.1 The tool integration's user interface shall be brought to front when the user triggers a variant-management action from within the development tool.

3.2.2 The tool integration's user interface shall stay in front of the development-tool window.

Requirement 3.1 covers the ideal situation, in which the user interface can be integrated. Requirement 3.2 specifies the alternative solution. Here, the user interface resides in an extra window.

However, Requirement 3.2 is not sufficient, since showing the user interface in a separate window introduces more problems: Whenever a user selects an element within the development tool, the integration window would be hidden behind the development tool's window. To still be able to refer to the pure::variants models when needed, we consider two possible solutions: Either the integration window could stay always on top, or it could be automatically called to front when users trigger variant-management actions from within the development tool. In Requirement 3.2.1, we choose the latter solution, because it does not hide part of the development tool's user interface when the integration is not required. However, if no actions can be triggered from within the development tool, this solution would not be possible, since users would need to trigger all actions from the tool integration's window. In this case, it would be more sensible to always show the user-interface window in front of the development tool (Requirement 3.2.2).

Requirement 4: Development tools shall support the user during the creating of variation points.

ID: 4 (6.3.8)	Development tools shall support the user during the creating of variation points.
Description	Development tools shall support the user during the creating/instantiating of variation points by either choosing the appropriate variant handling mechanism automatically if possible or propose different variant handling mechanisms to the user.
Rationale	Due to the fact that the variability handling mechanisms are different depending on the used language, tool, DSL, and so on, the user can start working directly also regarding variability without learning or knowing the different variability handling concepts upfront.
Use Case	QT3-UC-1
Possible Realization	Depends on the used language (IDE), tool, DSL, and so on.
Source	SPESXT
Acceptance Criterion	All subrequirements fulfilled

Requirement 4 states that users should be supported when creating variation points. Variability is represented differently in development tools. For example, it can be

represented by UML constraints, DOORS attributes, or by elements in a separate model that reference to variable artifacts of the master artifact model (Pohl [PBL05], p.75). Users should not need to learn how variability is represented in each tool. Instead they should be supported by the integration. In Use Cases 1 – 3, we already specified scenarios for creating variation points. Based on these use cases, we define the following subrequirements.

4. Development tools shall support the user during the creating of variation points.
   - 4.1 The tool integration shall provide editors for different kinds of pvSCL rules.
   - 4.2 The tool integration shall allow loading and displaying feature models and variant result models.
   - 4.3 The tool integration shall provide an editor for writing restrictions.
   - 4.4 The tool integration shall provide an editor for writing calculations.

In the scenarios of Use Cases 1 – 3, users add all types of variation points using a pvSCL editor. Hence, we add Requirement 4.1. Furthermore, we state in these use cases that users can view feature models and VRMs directly in the development tool. Therefore, we define Requirement 4.2. Since the use cases refer to two types of variation points, we also add Requirements 4.3 and 4.4.

These requirements are still very general, since they do not yet specify which features the pvSCL editors should support, or how users should trigger the editors. Thus, we define further subrequirements. Next, we show each of the requirements, and describe which detailed subrequirements we define and why.

Requirement 4.1: The tool integration shall provide editors for different kinds of pvSCL rules.

ID: 4.1	The tool integration shall provide editors for different kinds of pvSCL rules.
Description	Users should be able to write different kinds of variation points, such as restrictions or calculations, using an editor that enables editing of pvSCL rules in an efficient and error-preventing way.
Rationale	Necessary to fulfill Requirement 4
Use Case	QT3-UC-1, Use Case 1
Acceptance Criterion	All subrequirements are fulfilled

Requirement 4.1 states that the tool integration should provide editors for different kinds of pvSCL rules. We further describe these pvSCL editors in the following requirements:

- 4.1 The tool integration shall provide editors for different kinds of pvSCL rules.
  - 4.1.1 All pvSCL editors shall provide autocompletion based on the loaded models.
    - 4.1.1.1 The autocompletion of all pvSCL editors shall show all possible pvSCL keywords.

4.1.1.2 The autocompletion of all pvSCL editors shall show all features of the loaded models.
4.1.1.3 The autocompletion of all pvSCL editors shall show all attributes of the entered feature.
4.1.1.4 The autocompletion of all pvSCL editors shall show all components of the loaded models.
4.1.2 All pvSCL editors shall provide pvSCL compliant syntax highlighting.
4.1.3 All pvSCL editors shall check the pvSCL rule for errors before writing pvSCL rules to variation points.

Since pure::variants users already know the pvSCL editor of pure::variants, the tool integration's pvSCL editors should be similar (usability heuristic *consistency and standards*). Hence, they should provide autocompletion (Requirement 4.1.1) and syntax highlighting (Requirement 4.1.2). In Requirements 4.1.1.1 – 4.1.1.4, we further specify which keywords and elements of pure::variants models the editor should propose. To reduce errors in variation points, the editors also should check whether the entered rules are correct. Thus, we add Requirement 4.1.3.

Requirement 4.2: The tool integration shall allow loading and displaying feature models and variant result models.

ID: 4.2	The tool integration shall allow loading and displaying feature models and variant result models.
Description	The integration should provide a means to load feature models and variant description models. The models should be displayed in a tree that resembles the pure::variants model tree. Especially displaying pure::variants features and attributes is important, because they are needed when editing pvSCL rules.
Rationale	Besides the visual representation, the integration needs feature models and variant description models as input data for autocompletion, preview, and error visualization.
Use Case	QT3-UC-1, Use Case 1, Use Case 3
Source	Usability

As described in Use Case 1, showing pure::variants models directly in the extended tool is of advantage. Furthermore, pure::variants models are needed for autocompletion and visualization. Therefore, we define Requirement 4.2 and its subrequirements:

4.2 The tool integration shall allow loading and displaying feature models and variant result models.
4.2.1 Loaded models shall be saved and reloaded when opening a master artifact model.
4.2.2 The last five opened models shall be quickly accessible.

4.2.3 Feature models and variant result models shall show all elements needed for entering pvSCL rules.

To increase user satisfaction and efficiency, we add Requirements 4.2.1 and 4.2.2. We define the requirements for two reasons: First, a master artifact model is often related to the same feature model(s). Hence, manually reopening the same feature model on every startup of the master artifact model is unnecessary (Requirement 4.2.1). Second, for previewing variants, users need to switch between different VRMs. To reduce the time needed for switching between models, we suggest a quickload option for the previously loaded models (Requirement 4.2.2).

With displaying feature models or VRMs inside the extended tool, we intend to help users in entering pvSCL rules (see Use Case 1). Users should not need to switch back to pure::variants to view the related model. To prevent users from switching back to pure::variants, the model representation should show all elements that are relevant for entering a pvSCL rule. Thus, we add Requirement 4.2.3.

Requirement 4.3: The tool integration shall provide an editor for writing restrictions.

ID: 4.3	The tool integration shall provide an editor for writing restrictions.
Description	Users should be able to write restrictions using a pvSCL editor that complies with Requirements 4.1.1 - 4.1.3. When they press OK, the tool integration should write the entered rule to the variation point. Depending on the extended tool, the rule can be stored either in the master artifact model, using a specific type of artifact, or outside, using a file that maps pvSCL rules to variable artifacts. Either way, the user should notice no different behavior of tool integrations.
Rationale	If the transformation for the development tool supports restrictions, it should be possible to edit them in an efficient and error-preventing way.
Use Case	QT3-UC-1, Use Case 1
Acceptance Criterion	It is possible to write restrictions using a pvSCL editor, and the subrequirements are fulfilled.

In Use Case 1, we described a scenario for adding restrictions to master artifact models. Hence, we add Requirement 4.3, which defines how users can add restrictions. Since we already defined features of pvSCL editors in Requirement 4.1, we do not need to specify how the restriction editor should work. Thus, we only add subrequirements that concern how users can trigger the editor, and how pvSCL rules are written to the master artifact model:

4.3 The tool integration shall provide an editor for writing restrictions.
- 4.3.1 It shall be possible to add multiple restrictions at once.
- 4.3.2 It shall be possible to edit multiple restrictions at once.
- 4.3.3 The restriction editor shall be easily accessible.
  - 4.3.3.1 The restriction editor shall be accessible through the context menu of each element.
  - 4.3.3.2 The restriction editor shall be accessible through a button in the integration's user interface.
  - 4.3.3.3 The restriction editor shall be accessible through a menu of the development tool.
  - 4.3.3.4 Triggering the restriction editor for an already defined variation point shall be possible in the same or a way as adding a new one.

As described in Use Case 2, adding multiple restrictions at once can be of advantage. Therefore, we define Requirement 4.3.1, which states that users should be able to select multiple elements of a master artifact model and trigger a restriction editor for all of these elements. The entered rule is then written to all elements. To later edit these rules efficiently, users should also be able to *edit* multiple existing restrictions. In Requirement 4.3.2, we define an approach for editing multiple restrictions. Since how the restriction editor is triggered influences the time needed to add or edit a restriction, we define Requirement 4.3.3: *The restriction editor shall be easily accessible.* How this requirement can be fulfilled, strongly depends on the extension mechanisms of the respective tool. We believe that it would be best to trigger the editor from the context menu of each master-artifact-model element (Requirement 4.3.3.1), because it emphasizes which elements the user is editing. However, it is not possible to add custom context menus in all tools. Hence, we also define the alternative Requirement 4.3.3.2: *The restriction editor shall be accessible through a button in the integration's user interface.* Furthermore, when the user interface cannot be embedded, pvSCL editors need to be triggered from the development tool, so that the tool integration can be called to front (Requirement 3.2.1). However, in some tools it may not be possible to fulfill Requirement 4.3.3.1 or 4.3.3.2, because the development tool may not support adding context menus to elements or adding buttons to the user interface. Therefore, we add Requirement 4.3.3.3: *The restriction editor shall be accessible through a menu of the development tool.* Moreover, editing an existing restriction should be consistent with adding a new one. Thus, we add Requirement 4.3.3.4: *Triggering the restriction editor for an already defined variation point shall be possible in the same or a way as adding a new one.* When the required actions are the same, users do not need to learn a new approach for editing existing variation points (usability heuristic *consistency and standards*).

Requirement 4.4: The tool integration shall provide an editor for writing calculations.

ID: 4.4	The tool integration shall provide an editor for writing calculations.
Description	Users should be able to write calculations using a pvSCL editor that complies with Requirements 4.1.1 – 4.1.3. When they press OK, the tool integration should write the entered rule to the variation point. Depending on the extended tool, the rule can be stored either in the master artifact model, using a specific type of artifact, or outside, using a file that maps pvSCL rules to variable artifacts. Either way, the user should notice no different behavior of tool integrations.
Rationale	If the transformation for the development tool supports calculations, it should be possible to edit them in an efficient and error-preventing way.
Use Case	QT3-UC-1, Use Case 3
Source	Customer request for calculations
Acceptance Criterion	All subrequirements are fulfilled

Based on Use Case 3, we propose Requirement 4.4, which describes how users can add calculations. Similar to Requirement 4.3, we define subrequirements:

4.4 The tool integration shall provide an editor for writing calculations.
- 4.4.1 The calculation editor shall be to the restriction editor.
- 4.4.2 The calculation editor shall be easily accessible.
  - 4.4.2.1 The calculation editor shall be accessible through the context menu of each element.
  - 4.4.2.2 The calculation editor shall be accessible through a button in the integration's user interface.
  - 4.4.2.3 The restriction editor shall be accessible through a menu of the development tool.
  - 4.4.2.4 Triggering the calculation editor for an already defined variation point shall be possible in the same or a way as adding a new one.

For consistency, users should be able to edit calculations in the same way as they edit other pvSCL rules. Hence, we define Requirement 4.4.1, which states that the calculation editor should be similar to the restriction editor. The only differences should be the autocompletion keywords and the error check, because restrictions are only valid if the result is a boolean value, while calculations can return different types. To realize this, we propose to base both editors on the same class. Triggering the calculation editor should also be consistent with the restriction editor. Therefore, we add Requirement 4.4.2, which is similar to Requirement 4.3.2.

Requirement 5: The tool integration shall support users in deleting variation points.

ID: 5	The tool integration shall support users in deleting variation points.
Description	If the development tool does not support a simple option to delete a variation point, the tool integration should provide such an option. It should be consistent with the action to add or edit a variation point.
Rationale	The tool integration eases the way of entering variation points. Therefore, it should also be easy to delete them.
Use Case	QT3-UC-1, Use Case 1
Possible Realization	If, for example, adding and editing variation points is possible through an element's context menu, deleting a variation point should also be possible through such a context menu (see Enterprise Architect Integration).

Since a tool integration already provides a mechanism to add and edit variation points, it should also be possible to delete them. If the development tool does not provide an efficient solution for this, the tool integration should support users in deleting variation points. Thus, we define Requirement 5 without subrequirements.

Requirement 6: Variant management shall support the visualizing of variability-affected model elements.

ID: 6 (6.3.5)	Variant management shall support the visualization of variability-affected model elements.
Description	The variant management is in charge for giving the information what elements belong to what variability information. That means if a user wants to know what elements within a given model are related to certain variability information and the tool is able to visualize them accordingly, the exploration will be much easier than without such capability.
Rationale	E.g. visualizing variability-affected model elements is helpful a) to get an overview and b) to analyze whether all relevant model elements are referring to the correct variability information.
Use Case	QT3-UC-1
Source	SPESXT
Acceptance Criterion	All subrequirements are fulfilled

SPES$_{XT}$ Requirement 6 states that variant management is responsible for visualizing which elements are affected by variability. Hence, the tool integration should support visualizing variability-affected elements. Since this requirement can refer to different types of visualizations, we add more detailed subrequirements:

6. Variant management shall support the visualizing of variability-affected model elements.
   - 6.1 Variation points shall be visibly different from other elements of the master artifact model.
     - 6.1.1 The tool integration shall provide a means to highlight restrictions.
     - 6.1.2 The tool integration shall provide a means to highlight calculations.
   - 6.2 The tool integration shall provide a means to quickly find all model elements related to a specific variation point.

We interpret SPES_{XT} Requirement 6 in two different ways: First, it should always be clear which elements are affected by variability. When looking at a master artifact model, the variable elements should be visible at first glance. Otherwise, variant modelers or variant verifiers would waste time searching for existing variation points. To this end, we specify Requirement 6.1. If this requirement is not already fulfilled per default by the development tool, the tool integration should provide a visualization that solves this problem (see Requirement 6.1.1 and 6.1.2). Second, visualizing variability-affected elements could also mean that users want to know which elements relate to a certain variation point. In Requirement 6.2, we propose a method that solves this problem. The integration could provide a list of all variation points contained in the master artifact model. When clicking on a list entry, the related model elements would be highlighted. This could be further refined by a filter function, so that only those rules were shown that contain a search string specified by the user. Such a feature would be useful, since it reduces the time needed to find all elements related to a variation point. However, we classify it as optional, since in most cases, users are able to use a tool-specific search to find all elements related to a specific variation point.

Requirement 7: The tool integration shall support users in finding variation-point errors.

ID: 7	The tool integration shall support users in finding variation-point errors.
Description	Variation-point errors that can be detected automatically include pvSCL syntax errors and unknown feature names. The integration should help users in finding these errors.
Rationale	Helping users to find errors in variation points increases the efficiency of the error finding process and leads to less errors in master artifact models.
Use Case	QT3-UC-1, Use Case 4
Source	[Pap11]
Acceptance Criterion	All subrequirements are fulfilled

Based on Use Case 4, we define Requirement 7, which states that the integration should support users in finding errors in variation points. To fulfill this requirement, we propose the following subrequirements:

7. The tool integration shall support users in finding variation-point errors
   - 7.1 The tool integration shall provide a means to highlight pvSCL syntax errors and unknown names.
   - 7.2 When searching variation-point errors, users shall be able to distinguish between syntax errors and unknown names
     - 7.2.1 Syntax errors and unknown names shall be distinguished by different colors.

In Requirement 7.1, we state that the tool integration should fulfill Requirement 7 by providing a visualization that highlights all variation-point errors. Furthermore, we specify that users should be able to distinguish between pvSCL syntax errors and unknown names, because both errors have different consequences. pvSCL syntax errors have to be always corrected in the variation-point definition. However, an unknown feature or attribute name can also result from an error in the feature-model definition. The existing integrations fulfill these requirements by providing two different visualizations: One visualization highlights all pvSCL syntax errors in red. Another visualization highlights variation points that contain unknown feature or attribute names in yellow (Requirement 7.2.1).

Requirement 8: Variant management shall support the previewing of variants.

ID: 8 (6.3.6)	Variant management shall support the previewing of variants.
Description	The variant management is in charge of giving the information what elements belong to a variant to allow tools visualizing them perhaps by highlighting them. Another possibility would be graying out those elements that are not part of a concrete variant. This functionality helps the user to get an impression of how that variant will look like when finalized.
Rationale	Previewing means visualizing those elements that belong to a variant and which do not.
Use Case	QT3-UC-1
Source	SPESXT
Acceptance Criterion	All subrequirements are fulfilled

Both Use Case 5 and SPES_{XT} Requirement 8 state that it should be possible to preview a variant without executing a time-consuming transformation. To further specify how such a preview should work, we define the following subrequirements:

8. Variant management shall support the previewing of variants.
   - 8.1 The tool integration shall provide a preview that is faster than a transformation of the same master artifact model.
   - 8.2 The tool integration shall provide a preview that grays out elements not included in the variant.

8.3 The tool integration shall provide a preview that hides elements not included in the variant.
8.4 The tool integration shall provide a preview that highlights all variability-affected elements that are included in the variant.
8.5 Preview visualizations shall replace calculation values.

A general requirement that a preview implementation should always fulfill is Requirement 8.1: It should be faster than a transformation of the same master artifact model. Otherwise, the preview would be unnecessary, since users could instead transform a variant. To verify whether this requirement is met, the transformation and preview duration should be measured for the same master artifact model.

In Requirement 8.2 and 8.3, we specify how preview visualizations should look like. We choose to either gray out or hide elements not included in a variant, because these visualizations have different advantages: On the one hand, nothing is hidden when graying out elements. This reduces the chance that variant verifiers overlook falsely deleted elements. Hiding elements, on the other hand, may be good for large master artifact models when a variant verifier wants to get an overview of the contents of the variant.

In EA and Rhapsody we have already implemented a third variant of the preview, which highlights all variability-affected elements that stay in a variant. For consistency with these integrations, we add Requirement 8.4: *The tool integration shall provide a preview that highlights all variability-affected elements that are included in the variant.* Such a preview is useful in combination with the gray-out preview, because users can check whether (a) all variable elements are attached to variation points and (b) the variation points lead to the correct transformation decision. However, when Requirement 6.1: *Variation points shall be visibly different from other elements of the master artifact model* is fulfilled, the same can be achieved using only the gray-out visualization. Therefore, we classify Requirement 8.4 as optional.

Finally, we specify that, if the respective transformation supports calculations, all preview visualizations should replace calculation values (see Requirement 8.5). This is necessary, because the preview should be as similar to the transformation as possible. Otherwise, users would have to resort to the transformation for correctness.

3.3 Workflow for Implementing Tool Integrations

Having defined use cases and requirements for variant-management tool integrations, we propose a workflow for creating new integrations that fulfills the goals specified in Chapter 1. Developers should follow this workflow to efficiently create consistent and usable integrations. We start by analyzing the current workflow.

3.3.1 Current Workflow

From our experience with developing new tool integrations, the first step is to gather all necessary information about the tool we want to extend. This includes:

- Conceptual details
  - How do users work with the tool?
  - Which tool-integration functions would make sense in the context of the extended tool?
- Technical details
  - What extension mechanisms does the tool provide?
  - What is not possible?

Based on the gathered information, we would then create a specification document and start implementing the integration.

Following these steps *can* lead to consistent and usable tool integrations. However, creating a specification document for each integration anew is time-consuming and may introduce inconsistencies. In Section 3.2, we argued that the use cases and requirements for variant management are similar for all integrations, although some differences exist. Hence, we choose to develop the specification document as an SPL. This ensures that all documents are as consistent as possible, since they are based on the same text, and the effort for creating a new specification document is kept low, because it is generated from a master artifact model. Furthermore, it ensures that updates to the specification are applied to all integrations. Thus, the workflow supports developers in keeping the integrations consistent after the initial implementation.

To model the specification document as an SPL, we follow the steps we described in Section 2.3.2. Therefore, we still need the respective pure::variants models and a master artifact model. Next, we present these models.

3.3.2 The pure::variants Integration Project

In this section, we present the pure::variants integration project, which enables developers of new tool integrations to generate a specification document for the desired integration. We choose to base the specification document on the requirements we defined in Section 3.2, since they describe what is expected of a tool integration and already cover alternative requirements. For creating a new SPL, we follow the steps we described in Section 2.3.2. Hence, we start with defining the problem space.

To decide which requirements are relevant for an integration, we first need to know which variation-point types the respective pure::variants connector supports. Therefore, we define the different variation-point types in feature model *Connector Features* in Figure 3.2.

Then, the tool-integration developer needs to find out which extension mechanism the tool provides and which technical constraints exist. In feature model *Technical Features*, we specify all technical aspects that are important for implementing new tool

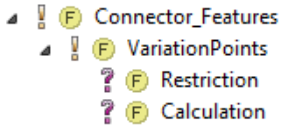


Figure 3.2: Connector Features: The first feature model of the pure::variants integration project. It contains features of the respective pure::variants connector that are relevant for the integration.

integrations (see Figure 3.3). For example, it defines whether the integration's user interface can be embedded directly in the user interface of the development tool (*CanIntegrateUI*), whether elements of the master artifact model can be colored during visualizations (*CanColorElements*), whether custom context menus can be added (*CanAttachContextMenuToElement*), or whether restrictions are visibly different from other artifacts (*UsersCanIdentifyRestrictions*).

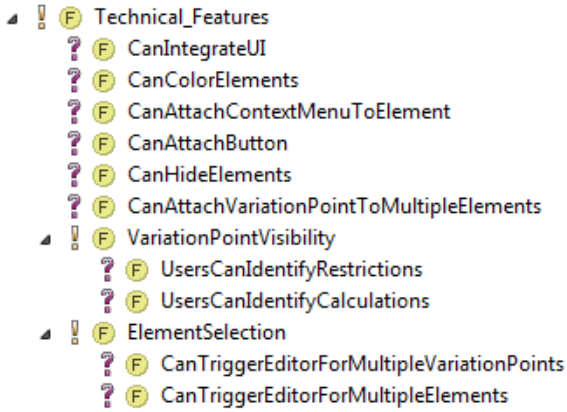


Figure 3.3: Technical Features: The second feature model of the pure::variants integration project. It describes technical details of the extended tool.

Finally, we need to decide which features should be supported. It is not always possible or sensible to provide all features for each tool integration, because customers may only need a small subset of functions, or the first version of the tool integration should only contain basic features. Hence, we provide the feature model *Design Features*, which defines all possible features of a tool integration, such as the supported types of visualizations, or the possible variation-point actions. We display this feature model in Figure 3.4. Most of the features depend on technical details of the extended tool. For example, developers can only select feature *PreviewGrayOut*, if feature *CanColorElements* of the technical feature model is selected. Therefore, we add pure::variants relations and constraints to some features. They describe the relations between technical features and design features. pure::variants checks, based on these relations and constraints whether the selection of a VDM is valid.

Having defined the feature models, we still need to edit the variability of the solution space. Since we want to generate requirements documents, we use a Microsoft Word

Figure 3.4: Design Features: The third feature model of the pure::variants integration project. It specifies which functions an integration shall provide. The restrictions and constraints ensure that all necessary features are selected, and that users can only select technically possible features.

document containing all requirements of Section 3.2 as master artifact model. We add variation points to this model using the existing pure::variants integration for Microsoft Word. In Table 3.1, we give an overview of the master artifact model. For each requirement, we display the attached restriction, its ID, and its title.

We added two kinds of restrictions to the requirements document. First, we reference technical features when an alternative implementation is needed due to technical constraints (e.g., Requirement 3.1 and 3.2 are alternatives that depend on feature *CanIntegrateUI*). Second, we reference design features that define which functions the tool integration should support. For instance, for Requirements 8.2 – 8.4, the variation points specify the respective type of preview. Even though we could define the master artifact model based completely on technical features, we choose to also refer to design features, because developers may not always want to provide the full range of functions.

Next, we define the workflow steps for creating a new tool integration.

3.3.3 Workflow Definition

Based on the presented pure::variants integration project, we propose the following workflow for creating a new tool integration:

1. *Define a VDM of the pure::variants Integration Project*:
   (a) Specify, which types of variation points the respective connector shall support (see Figure 3.2 on page 50)
   (b) Gather all information about the tool that is necessary for creating a VDM of the *Technical Features* feature model (see Figure 3.3 on page 50)
   (c) Select the features the integration shall support (see Figure 3.4 on the previous page)
   (d) If pure::variants reports errors caused by constraints or restrictions, an important requirement cannot be met due to technical constraints. In this case, provide an alternative requirement and modify the feature models accordingly.
2. *Generate a requirements document*: Generate a variant of the requirements master artifact model.
3. *Analyze and adapt the requirements*: Analyze based on test implementations and previous experience, whether all requirements can be met. If a requirement cannot be met, the feature models and/or the master artifact model is incomplete or incorrect. In this case, update the respective models and generate a new variant.
4. *Implement the requirements*: Implement the requirements step by step starting with the most important ones.

Restriction	ID	Requirements Title
	1	Variant management shall always be done in the same way for the user independent of the used development tool.
	1.1	Tool integrations shall use the same or similar icons for the same functions.
	1.2	Tool integrations shall communicate similar messages in similar situations.
	1.3	The integrations' user-interface design shall be similar for all integrations.
	2	The tool integration shall comply with usability heuristics.
	2.1	The tool integration shall display a wait cursor for actions that take longer than two seconds.
	2.2	Tool integrations shall provide progress bars for actions that take longer than ten seconds.
	2.3	Tool integrations shall be consistent with pure::variants and the extended tool.
	2.4	Users shall be able to interrupt all actions that take longer than 10 seconds.
	2.5	For all actions that change the master artifact model, the tool integration shall provide a means to undo or redo the action.
	2.6	Users shall always be able to cancel a dialog.
	2.7	Tool integrations shall warn users before performing an irreversible action.
	2.8	Tool integrations shall always support users in entering data.
	2.9	The tool integration's user interface and visualizations shall be kept as simple as possible.
	2.10	If an error occurs, the tool integration shall provide simple, constructive, and specific instructions for recovery.
CanIntegrateUI	3	Variant management shall seamlessly integrate itself into the development tools by providing a user interface.
NOT(CanIntegrateUI)	3.1	The tool integration's user interface shall be embedded in the development tool's user interface.
CanAttachContextMenuToEleme	3.2	The tool integration's user interface shall be displayed in its own window, which can be triggered from the development tool's user interface.
NOT(CanAttachContextMenuTo	3.2.1	The tool integration's user interface shall be brought to front when the user triggers a variant-management action from within the development tool.
CanIntegrateUI	3.2.2	The tool integration's user interface shall stay in front of the development-tool window.
CalculationsSupported	4	Development tools shall support the user during the creating of variation points.
	4.1	The tool integration shall provide editors for different kinds of pvSCL rules.
	4.1.1	All pvSCL editors shall provide autocompletion based on the loaded models.
SupportsKeywords	4.1.1.1	The autocompletion of all pvSCL editors shall show all possible pvSCL keywords.
SupportsFeatures	4.1.1.2	The autocompletion of all pvSCL editors shall show all features of the loaded models.
SupportsAttributes	4.1.1.3	The autocompletion of all pvSCL editors shall show all attributes of the entered feature.
SupportsComponents	4.1.1.4	The autocompletion of all pvSCL editors shall show all components of the loaded models.
	4.1.2	All pvSCL editors shall provide pvSCL compliant syntax highlighting.
	4.1.3	All pvSCL editors shall check the entered rule for errors before writing pvSCL rules to variation points.
LoadPVModels	4.2	The tool integration shall allow loading and displaying feature models and variant result models.
SaveModels	4.2.1	Loaded models shall be saved and reloaded when opening a master artifact model.
Quickload	4.2.2	The last five opened models should be quickly accessible.
	4.2.3	Feature models and variant result models shall show all elements needed for entering pvSCL rules.
EditRestriction	4.3	The tool integration shall provide an editor for writing restrictions.
AddMultipleRestrictions	4.3.1	It shall be possible to add multiple restrictions at once.
EditMultipleRestrictions	4.3.2	It shall be possible to edit multiple restrictions at once.
	4.3.3	The restriction editor shall be easily accessible.
CanAttachContextMenuToEleme	4.3.3.1	The restriction editor shall be accessible through the context menu of each artifact.
NOT(CanAttachContextMenuTo AND CanAttachButton	4.3.3.2	The restriction editor shall be accessible through a button in the integration's user interface.
NOT(CanAttachContextMenuTo AND NOT(CanAttachButton)	4.3.3.3	The restriction editor shall be accessible through a menu of the development tool.
	4.3.3.4	Triggering the restriction editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
EditCalculation	4.4	The tool integration shall provide an editor for writing calculations.
	4.4.1	The calculation editor shall be similar to the restriction editor.
	4.4.3	The calculation editor shall be easily accessible.
CanAttachContextMenuToEleme	4.4.3.1	The calculation editor shall be accessible through the context menu of each element.
NOT(CanAttachContextMenuTo AND CanAttachButton	4.4.3.2	The calculation editor shall be accessible through a button in the integration's user interface.
NOT(CanAttachContextMenuTo AND NOT(CanAttachButton)	4.4.3.3	The restriction editor shall be accessible through a menu of the development tool.
	4.4.3.4	Triggering the calculation editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
DeleteCalculation OR DeleteR-estriction OR DeleteConstraint	5	The tool integration shall support users in deleting variation points.
	6	Variant management shall support the visualizing of variability-affected model elements.
	6.1	Variation points shall be visibly different from other elements of the master artifact model.
HighlightRestrictions	6.1.1	The tool integration shall provide a means to highlight restrictions.
HighlightCalculations	6.1.2	The tool integration shall provide a means to highlight calculations.
VariationPointSearch	6.2	The tool integration shall provide a means to quickly find all model elements related to a specific variation point.
ErrorVisualization	7	The tool integration shall support users in finding variation-point errors.
	7.1	The tool integration shall provide a means to highlight pvSCL syntax errors and unknown names.
	7.2	When searching variation-point errors, users shall be able to distinguish between syntax errors and unknown names.
SeparateErrorVisualizations	7.2.1	Syntax errors and unknown names shall be distinguished by different colors.
	7.3	The tool integration shall provide a means to navigate between errors.
Preview	8	Variant management shall support the previewing of variants.
	8.1	The tool integration shall provide a preview that is faster than a transformation of the same master artifact model.
PreviewGrayOut	8.2	The tool integration shall provide a preview that grays out elements not included in the variant.
PreviewHide	8.3	The tool integration shall provide a preview that hides elements not included in the variant.
PreviewHighlight	8.4	The tool integration shall provide a preview that highlights all variability-affected elements that are included in the variant.
CalculationsSupported	8.5	Preview visualizations shall replace calculation values.

Table 3.1: An excerpt of the final requirements master artifact model. Only the requirements ID, title and related restrictions are displayed. For the full requirements descriptions refer to Appendix B.

When employing this workflow, the created integrations should be both consistent and usable. Furthermore, the time to create a specification document for each new integration is reduced, because developers can generate it from the pure::variants integration project. To evaluate whether the proposed workflow can be used to create tool integrations, we test it on three different external tools in the next chapter.

4. Applying the Proposed Workflow

To verify whether it is technically possible to create tool integrations based on the proposed workflow, we apply it to three different tools. We select the tools IBM Rational DOORS (DOORS), Microsoft Office Excel (Excel), and Enterprise Architect (EA), since they are typically used in different phases of the software-development lifecycle, and are all successfull tools (see Section 2.4). Part of the implementation is already done: For EA, pure-systems already provides an integration; for DOORS, no integration exists, but users can already transform variants; and for Excel, neither integration nor transformation is supported (for more details, refer to Section 2.4).
We structure this chapter based on the workflow steps. Therefore, we first define a VDM of the pure::variants integration project in Section 4.1. Then, we generate the requirements document in Section 4.2, and analyze whether the requirements can be met in Section 4.3. If we cannot fulfill a requirement, we propose an alternative. Finally, we present the resulting implementation for the new integrations.

4.1 Defining a VDM of the pure::variants Integration Project

The first step of the proposed workflow is to define a VDM that describes the existing pure::variants connector, the technical features of the extended tool, and the integration's desired features. To define a VDM, we need to select which features should be part of the final requirements document.

4.1.1 Selecting Connector Features

In Figure 4.1, we show the selection of the connector features for the connectors to DOORS, Excel, and EA.

All three connectors already support, or shall support, restrictions. Furthermore, calculation support shall be added to the DOORS and Excel connector, due to customer

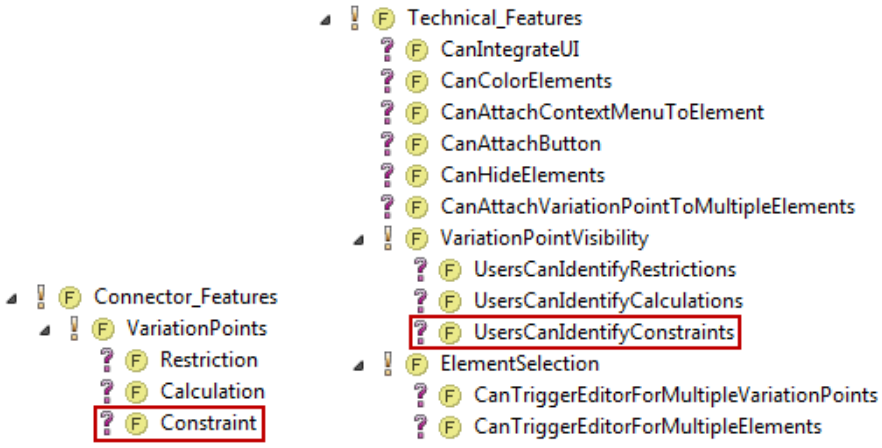
Model Elements	Level	Doors	Excel	EA
Connector_Features				
Connector_Features		✓	✓	✓
SupportedVariationPoints	1	✓	✓	✓
Restriction	1.1	☑	☑	✓
Calculation	1.2	☑	☑	☐

Figure 4.1: Variant selections for the pure::variants connectors to DOORS, Excel, and EA.

requests. This completes the selection of the connector features. However, the connector for DOORS also supports pure::variants constraints. Hence, we add Requirement 4.5 and 6.1.3 to the requirements master artifact model. Since we add requirements, we also need to modify all feature models accordingly. In Figure 4.2, we show the updated feature models. For better readability, we highlight all new features.

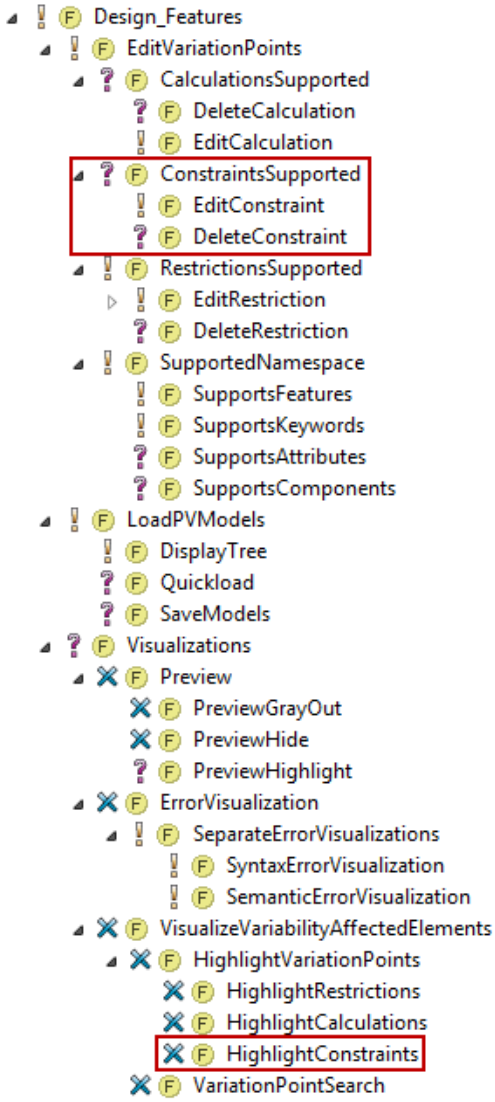
ID: 4.5	The tool integration shall provide an editor for writing constraints.
Description	Users should be able to write constraints using an editor that enables editing of pvSCL rules in an efficient and error-preventing way. Using the editor and accessing it should happen in the same way as for the restriction editor. However, users should always be able to distinguish whether they are editing a constraint or a restriction. Thus, labels and menus should be different.
Rationale	If the transformation for the extended tool supports constraints, it should be possible to edit them using a pvSCL editor.
Use Case	QT3-UC-1
Possible Realization	Use the same code base for restriction and constraint editor.
Acceptance Criterion	The same implementation is used for restriction and constraint editor. Only the dialog title, labels, and menu entries refer to the constraint editor.

ID: 6.1.3	The tool integration shall provide a means to highlight constraints.
Description	Since constraints and the related artifacts are not visibly different from other elements, the integration needs to provide a means to highlight constraints and the related elements.
Rationale	Necessary for fulfilling Requirement 6.1
Use Case	QT3-UC-1
Possible Realization	For example, provide a toggle button that highlights all constraints and related elements when selected.
Acceptance Criterion	Constraints can be or are always highlighted.



(a) Connector Features

(b) Technical Features



(c) Design Features

Figure 4.2: Updated pure::variants integration feature models. Now pure::variants constraints are also supported. New features are highlighted with a red box.

4.1.2 Selecting Technical Features of the Extended Tool

Next, we need to define which extension mechanisms each of the tools provides. To this end, we select the features of the technical feature model. If the extended tool does not provide the extension mechanism represented by a feature, we exclude the feature. We show the results in Figure 4.3.

Model Elements	Level	Doors	Excel	EA
Technical_Features				
Technical_Features		✓	✓	✓
CanIntegrateUI	1	⊖	☑	☑
CanColorElements	2	☑	✓	✓
CanAttachContextMenuToElement	3	⊖	⊖	☑
CanAttachButton	4	⊖	☑	⊖
CanHideElements	5	☑	✓	⊖
CanAttachVariationPointToMultipleElements	6	⊖	⊖	☑
VariationPointVisibility	7	✓	✓	✓
UsersCanIdentifyRestrictions	7.1	☑	⊖	⊖
UsersCanIdentifyCalculations	7.2	☑	⊖	⊖
UsersCanIdentifyConstraints	7.3	☑	⊖	⊖
ElementSelection	8	✓	✓	✓
CanTriggerEditorForMultipleVariationPoints	8.1	⊖	☑	✓
CanTriggerEditorForMultipleElements	8.2	⊖	☑	✓

Figure 4.3: Variant selections for the technical features of DOORS, Excel, and EA. Red signs mean that a feature is excluded, since the external tool does not provide the respective extension mechanism.

Notable is that DOORS does not allow for embedding of custom user interfaces (*CanIntegrateUI*), and it is not possible to trigger an action for multiple elements, because users can only select one element at a time (*CanTriggerEditorForMultipleVariationPoints* and *CanTriggerEditorForMultipleElements*). Furthermore, in Excel, we cannot add custom context menus[1], and restrictions or calculations do not visually differ from other cells of the workbook (*UsersCanIdentifyRestrictions* and *UsersCanIdentifyCalculations*). Similarly, in EA, restrictions are visible for some element types, but not for all. Therefore, we deselect *UsersCanIdentifyRestrictions*.

4.1.3 Selecting Design Features of the Integration

Finally, we can select the desired features for each integration. In Figure 4.4, we show the selection of the design features.

[1]Excel does support custom context menus. However, macro support is needed for this feature. Since users can disable all macros, we deselect feature *CanAttachContextMenu*.

Model Elements	Level	Doors	Excel	EA
Design_Features				
Design_Features		✓	✓	✓
EditVariationPoints	1	✓	☑	✓
CalculationsSupported	1.1	☑	☑	☐
DeleteCalculation	1.1.1	☐	☐	☐
EditCalculation	1.1.2	✓	✓	☐
ConstraintsSupported	1.2	☑	☐	☐
EditConstraint	1.2.1	✓	☐	☐
DeleteConstraint	1.2.2	☐	☐	☐
RestrictionsSupported	1.3	✓	✓	✓
EditRestriction	1.3.1	✓	✓	✓
EditMultipleRestrictions	1.3.1.1	☒	☑	☑
AddMultipleRestrictions	1.3.1.2	☒	☑	☑
DeleteRestriction	1.3.2	☐	☐	☑
SupportedNamespace	1.4	✓	✓	✓
SupportsFeatures	1.4.1	✓	✓	✓
SupportsKeywords	1.4.2	✓	✓	✓
SupportsAttributes	1.4.3	☑	☑	☑
SupportsComponents	1.4.4	☐	☐	☐
LoadPVModels	2	✓	✓	✓
DisplayTree	2.1	✓	✓	✓
Quickload	2.2	☑	☑	☑
SaveModels	2.3	☑	☑	☑
Visualizations	3	☑	☑	✓
Preview	3.1	☑	☑	☑
PreviewGrayOut	3.1.1	☑	☑	☑
PreviewHide	3.1.2	☑	☑	☒
PreviewHighlight	3.1.3	☐	☐	☑
ErrorVisualization	3.2	☑	☑	☑
SeparateErrorVisualizations	3.2.1	✓	☑	☑
SyntaxErrorVisualization	3.2.1.1	✓	✓	✓
SemanticErrorVisualization	3.2.1.2	✓	✓	✓
VisualizeVariabilityAffectedElements	3.3	☐	☑	✓
HighlightVariationPoints	3.3.1	☐	☑	✓
HighlightRestrictions	3.3.1.1	☐	☑	✓
HighlightCalculations	3.3.1.2	☐	☑	☐
HighlightConstraints	3.3.1.3	☐	☐	☐
VariationPointSearch	3.3.2	☐	☐	☐

Figure 4.4: Variant selections describing the supported features of the integrations to DOORS, Excel, and EA.

Three things are notable: First, we could not select features *AddMultipleRestrictions* and *EditMultipleRestrictions* for DOORS, because it is not possible to trigger a pvSCL editor for multiple restrictions. Second, EA does not support hiding elements programatically. Thus, we cannot provide a preview that hides all excluded elements. Instead, the integration provides another preview visualization that highlights all variability-affected elements included in a variant. Third, in Excel and EA, we need to additionally highlight variation points, because they are not visibly different from the rest of the document. Having defined valid VDMs, we can now generate the requirement documents for each integration.

4.2 Generating Requirements Documents for each Integration

In Table 4.1, we give an overview of the generated variants. An 'x' means that the requirements document of either DOORS, Excel, or EA contains the respective requirement.

Most requirements apply to all tool integrations. However, some differ between tool integrations. The differences mainly relate to the supported variation-point types, the supported preview types, to how users can access pvSCL editors, or whether the integration window is embedded in the development tool's user interface. Next, we analyze whether the generated requirements can be met.

4.3 Analyzing and Adapting Requirements

The next step is to analyze the generated requirements and modify them, if necessary. Therefore, we argue for each requirement whether and why we believe that we can fulfill it. We structure this section based on the main requirements (see Section 3.2.1).

Requirement 1: Variant management shall always be done in the same way for the user independent of the used development tool.

Requirement 1 and its subrequirements apply to all three integrations. They mainly relate to a consistent look and feel of the integration's user interface. The text of labels and dialog boxes, or the used icons are independent of the development tool. Since it is easy to change such user-interface elements, we argue that Requirement 1 and its subrequirements can be fulfilled.

Requirement 2: The tool integration shall comply with usability heuristics.

It should be possible to fulfill nearly all subrequirements of Requirement 2, because they do not depend on the extended tool (e.g., showing wait cursors, or progress bars). However, it can be difficult to fulfill Requirement 2.4:*For all actions that change the master artifact model, the tool integration shall provide a means to undo or redo the*

ID	DOORS	Excel	EA	Requirement Title
1	x	x	x	Variant management shall always be done in the same way for the user independent of the used development tool.
1.1	x	x	x	Tool integrations shall use the same or similar icons for the same functions.
1.2	x	x	x	Tool integrations shall communicate similar messages in similar situations.
1.3	x	x	x	The integrations' user-interface design shall be similar for all integrations.
2	x	x	x	The tool integration shall comply with usability heuristics.
2.1	x	x	x	The tool integration shall display a wait cursor for actions that take longer than two seconds.
2.2	x	x	x	Tool integrations shall provide progress bars for actions that take longer than ten seconds.
2.3	x	x	x	Tool integrations shall be consistent with pure::variants and the extended tool.
2.4	x	x	x	Users shall be able to interrupt all actions that take longer than 10 seconds.
2.5	x	x	x	For all actions that change the master artifact model, the tool integration shall provide a means to undo or redo the action.
2.6	x	x	x	Users shall always be able to cancel a dialog.
2.7	x	x	x	Tool integrations shall warn users before performing an irreversible action.
2.8	x	x	x	Tool integrations shall always support users in entering data.
2.9	x	x	x	The tool integration's user interface and visualizations shall be kept as simple as possible.
2.10	x	x	x	If an error occurs, the tool integration shall provide simple, constructive, and specific instructions for recovery.
3	x	x	x	Variant management shall seamlessly integrate itself into the development tools by providing a user interface.
3.1		x	x	The tool integration's user interface shall be embedded in the development tool's user interface.
3.2	x			The tool integration's user interface shall be displayed in its own window, which can be triggered from the development tool's user interface.
3.2.1	x			The tool integration's user interface shall be brought to front when the user triggers a variant-management action from within the development tool.
3.2.2				The tool integration's user interface shall stay in front of the development-tool window.
4	x	x	x	Development tools shall support the user during the creating of variation points.
4.1	x	x	x	The tool integration shall provide editors for different kinds of pvSCL rules.
4.1.1	x	x	x	All pvSCL editors shall provide autocompletion based on the loaded models.
4.1.1.1	x	x	x	The autocompletion of all pvSCL editors shall show all possible pvSCL keywords.
4.1.1.2	x	x	x	The autocompletion of all pvSCL editors shall show all features of the loaded models.
4.1.1.3	x	x	x	The autocompletion of all pvSCL editors shall show all attributes of the entered feature.
4.1.1.4				The autocompletion of all pvSCL editors shall show all components of the loaded models.
4.1.2	x	x	x	All pvSCL editors shall provide pvSCL compliant syntax highlighting.
4.1.3	x	x	x	All pvSCL editors shall check the entered rule for errors before writing pvSCL rules to variation points.
4.2	x	x	x	The tool integration shall allow loading and displaying feature models and variant result models.
4.2.1	x	x	x	Loaded models shall be saved and reloaded when opening a master artifact model.
4.2.2	x	x	x	The last five opened models should be quickly accessible.
4.2.3	x	x	x	Feature models and variant result models shall show all elements needed for entering pvSCL rules.
4.3	x	x	x	The tool integration shall provide an editor for writing restrictions.
4.3.1		x	x	It shall be possible to add multiple restrictions at once.
4.3.2		x	x	It shall be possible to edit multiple restrictions at once.
4.3.3	x	x	x	The restriction editor shall be easily accessible.
4.3.3.1			x	The restriction editor shall be accessible through the context menu of each artifact.
4.3.3.2		x		The restriction editor shall be accessible through a button in the integration's user interface.
4.3.3.3	x			The restriction editor shall be accessible through a menu of the development tool.
4.3.3.4	x	x	x	Triggering the restriction editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
4.4	x	x		The tool integration shall provide an editor for writing calculations.
4.4.1	x	x		The calculation editor shall be similar to the restriction editor.
4.4.3	x	x		The calculation editor shall be easily accessible.
4.4.3.1			x	The calculation editor shall be accessible through the context menu of each element.
4.4.3.2		x		The calculation editor shall be accessible through a button in the integration's user interface.
4.4.3.3	x			The restriction editor shall be accessible through a menu of the development tool.
4.4.3.4	x	x		Triggering the calculation editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
4.5	x	x		The tool integration shall provide an editor for writing constraints.
5			x	The tool integration shall support users in deleting variation points.
6	x	x	x	Variant management shall support the visualizing of variability-affected model elements.
6.1	x	x	x	Variation points shall be visibly different from other elements of the master artifact model.
6.1.1		x	x	The tool integration shall provide a means to highlight restrictions.
6.1.2		x		The tool integration shall provide a means to highlight calculations.
6.1.3				The tool integration shall provide a means to highlight constraints.
6.2				The tool integration shall provide a means to quickly find all model elements related to a specific variation point.
7	x	x	x	The tool integration shall support users in finding variation-point errors.
7.1	x	x	x	The tool integration shall provide a means to highlight pvSCL syntax errors and unknown names.
7.2	x	x	x	When searching variation-point errors, users shall be able to distinguish between syntax errors and unknown names.
7.2.1	x	x	x	Syntax errors and unknown names shall be distinguished by different colors.
7.3	x	x	x	The tool integration shall provide a means to navigate between errors.
8	x	x	x	Variant management shall support the previewing of variants.
8.1	x	x	x	The tool integration shall provide a preview that is faster than a transformation of the same master artifact model.
8.2	x	x	x	The tool integration shall provide a preview that grays out elements not included in the variant.
8.3	x	x		The tool integration shall provide a preview that hides elements not included in the variant.
8.4			x	The tool integration shall provide a preview that highlights all variability-affected elements that are included in the variant.
8.5	x	x		Preview visualizations shall replace calculation values.

Table 4.1: An excerpt of the generated requirements documents. An 'x' means that the requirements document of the respective integration contains the requirement. We only display the ID and title of each requirement. For the full descriptions refer to Appendix B.

action. From our experience with implementing tool integrations, we already know that it is difficult to let users undo or redo actions of an integration. For instance, when an integration executes code in EA or Rhapsody, the complete undo and redo information is deleted. Nevertheless, we do not modify the requirements document, because, to the best of our knowledge, no meaningful alternative exists.

Requirement 3: Variant management shall seamlessly integrate itself into the development tools by providing a user interface.

For Requirement 3 and its subrequirements, two alternatives exist: The integrations to Excel and EA shall embed their user interface into the respective tool (see Requirement 3.1), and the integration to DOORS shall provide a stand-alone user interface that is always called to front when triggering a pvSCL editor (see Requirement 3.2). For defining the VDMs, we already tested whether the development tool supports embedding a custom user interface. Therefore, we can fulfill Requirement 3.1 for Excel and EA. For DOORS, we argue that Requirement 3.2 can be met, because it is possible to integrate a context menu into DOORS. Thus, we can call the user interface to front when triggering a pvSCL editor.

Requirement 4: Development tools shall support the user during the creating of variation points.

To support users when creating variation points, the extended tool needs to provide a mechanism to programmatically add or edit variation points and trigger the pvSCL editor. All three tools provide such mechanisms. Furthermore, the integration needs to fulfill the requirements for the pvSCL editor's user interface. We already know that these requirements can be fulfilled, because the editor is already implemented and will be reused in new integrations. We identify only one requirement that we cannot meet: Requirement 4.4.1: *The calculation editor shall be similar to the restriction editor* cannot be fulfilled for DOORS and Excel, because it is not possible to annotate text in these tools, which is necessary to add a calculation to the master artifact model. Therefore, the pure::variants connectors for DOORS and Excel provide an alternative annotation mechanism for calculations, which is based on markers. Hence, we adapt the requirements document to reflect this modification (see Requirement 4.4.2).

Per default, a calculation should begin with '[', and end with ']'. To ignore a term in brackets, it should start with '$['. However, in some tools, these characters already have a different meaning. Thus, we add calculations to the requirements master artifact model that specify which marker characters are used in the respective integrations[2]. Since we changed the requirements document, we also need to update the pure::variants integration project. In Figure 4.5, we show the updated feature model *Connector Features*. The attributes define the used marker characters.

[2]In the original requirements document, we use calculations to set the marker characters, so that the used characters can differ between tool integrations. However, due to formatting reasons, we do not show the calculations in this thesis.

ID: 4.4.2	The calculation editor shall support users in editing all calculations of the selected element.
Description	If calculations cannot be added to selected text, markers in the text are used to identify calculations. The following markers should be used per default: • Start of a calculation: '[' • End of a calculation: ']' • Escape character: '$' For example, "...[featurename->attributename] ..." is transformed to "...attributevalue ...". To still be able to use a pvSCL editor, it should be possible to trigger a calculation editor for an artifact that contains a calculation. The editor should show a selection of all calculations found in the element. It should be possible to select a calculation and edit it using a pvSCL editor that complies with Requirements 4.1.1 – 4.1.3. When the user presses OK, the element's text should be replaced so that the calculations are updated.
Rationale	In some tools it is not possible to attach a variation point directly to a text fragment. In this case, users cannot trigger a calculation editor for a single calculation. Therefore, the selection of the calculation to edit should happen in the editor.
Use Case	QT3-UC-1, Use Case 3
Possible Realization	Provide a list of all calculation rules contained in the selected element's text. Below, the currently selected rule can be edited using a standard pvSCL editor field. While typing, the list of calculation rules is updated. When the user closes the editor successfully, the element's text should be replaced with the edited text.

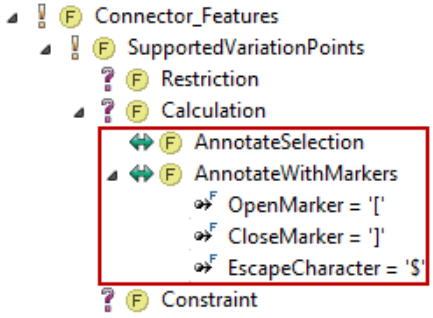


Figure 4.5: Updated *Connector Features*. Changes are highlighted.

Requirement 5: The tool integration shall support users in deleting variation points.

Requirement 5 depends on whether it is difficult to delete a variation point using the standard approach of the extended tool. This is only the case in EA, in which users need to navigate to the properties menu to delete a restriction. Since the pure::variants integration for EA already supports deletion of restrictions through a context menu, the requirement is already fulfilled.

Requirement 6: Variant management shall support the visualizing of variability-affected model elements.

To fulfill Requirement 6, the integrations need to comply at least with Requirement 6.1, which states that variation points shall be visibly different from other artifacts. We believe that this requirement can be met for all three integrations, because, first, in DOORS all variation points reside in an extra column. Therefore, users can easily identify variation points by scanning for rows that contain a pvSCL rule. Second, it should be possible to highlight variability-affected elements in EA and Excel, since coloring elements is possible.

Requirement 7: The tool integration shall support users in finding variation-point errors.

To meet Requirement 7, the extended tool needs to provide a mechanism to color elements. Since all three tools support coloring elements programmatically, we argue that Requirement 7 can be fulfilled. Additionally, we noticed while testing the extension mechanisms of DOORS, that icons can easily be added to each row of a module. This enables us to provide an error visualization that is similar to the errors and warnings in pure::variants. Hence, we propose Requirement 7.2.2. Since we introduce an alternative to Requirement 7.2.1, we need to update the feature models and VDMs accordingly. Therefore, we add the technical feature *CanAttachIcon*, and the design feature *OneErrorVisualization*. For brevity, we omit the updated feature models (refer to Figure B.1 on page 99 and Figure B.2 on page 100 for the updated models).

Requirement 8: Variant management shall support the previewing of variants.

To fulfill Requirement 8, the extended tool needs to at least provide a mechanism to color or hide elements. All three tools support coloring elements programmatically, and both Excel and DOORS provide a means to hide elements. Thus, we argue that Requirement 8 can be fulfilled.

After adapting the master artifact model, we need to generate new requirements documents. For brevity, we omit a table containing all requirements, since only few requirements have changed. This concludes Step 3 of the proposed workflow. Next, we describe the implementation results of the three integrations.

ID: 7.2.2	Syntax errors and unknown names shall be distinguished by icons.
Description	Syntax errors and unknown feature or attribute names should be distinguished by icons. Users will need to trigger only one error visualization to show both types of errors. During visualization, an Eclipse error icon should be visibly attached to elements containing syntax errors, and an Eclipse warning icon should be shown for variation points in which unknown names are referenced.
Rationale	pure::variants users already know Eclipse's error and warning icons.
Use Case	QT3-UC-1, Use Case 4
Possible Realization	In column- and row-based development tools an extra column could be added that displays error or warning icons.
Source	Consistency with pure::variants
Acceptance Criterion	See Description

4.4 Implementing Requirements

Now that we generated the final requirements documents, we can start implementing the tool integrations, or in the case of EA, adapt the existing integration to match the requirements. To verify whether our assumptions from Section 4.3 are correct, we present for each tool which requirements are fulfilled in the current version. If a requirement is not met, the reason why it is not fulfilled is important, since it has different consequences for our workflow. Therefore, we differentiate between four kinds of reasons:

1. *Technical Constraint (TEC)*: The requirement cannot be fulfilled, because the extended tool does not provide the necessary extension mechanisms.
2. *Expertise Constraint (EC)*: The requirement cannot be fulfilled, because we lack the necessary expertise.
3. *Time Constraint (TIC)*: The requirement can be met, but is not yet fulfilled. It will be addressed in future.
4. *Not Necessary (NN)*: It is not necessary to explicitly address this requirement, because the situation to which the requirement refers does not occur.

For this thesis, requirements with technical constraints for which no alternative exists are most relevant, because they may introduce problems that cannot be fixed. Requirements with other constraints should be addressed in future. Consistent with previous sections, we structure this section based on the top-level requirements. To get an overview of the current status of each tool integration, we present a table for each top-level requirement and its subrequirements.

Requirement 1: Variant management shall always be done in the same way for the user independent of the used development tool.

ID	D	EX	EA	Requirement Title
1				Variant management shall always be done in the same way for the user independent of the used development tool.
1.1	TIC	TIC	TIC	Tool integrations shall use the same or similar icons for the same functions.
1.2	TIC	TIC	TIC	Tool integrations shall communicate similar messages in similar situations.
1.3	✓	✓	✓	The integrations' user interface design shall be similar for all integrations.

Table 4.2: Status of Requirement 1 and its subrequirements. '✓' means that the requirement is fulfilled by the current integration version. If it is not fulfilled, the column contains an abbreviation referring to the reason (Technical Constraint (TEC), Expertise Constraint (EC), Time Constraint (TIC), or Not Necessary (NN)). We leave the cell blank when a subrequirement is not fulfilled. Since not all requirements apply to all tool integrations, we gray out all cells of requirements that are not part of the respective requirement document.

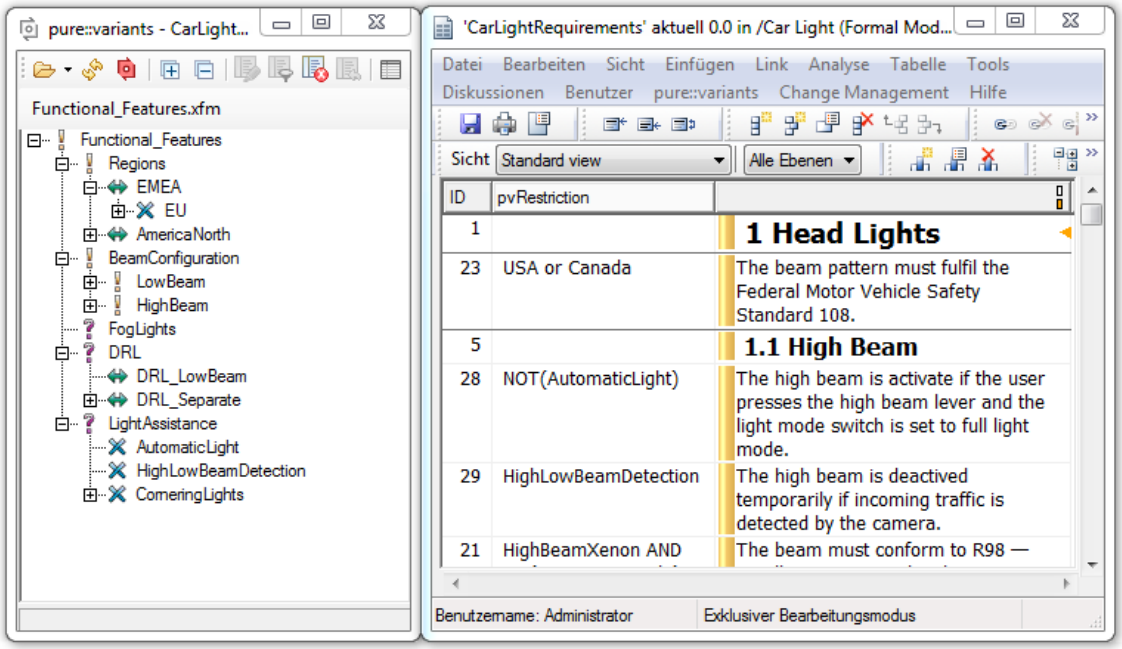
Requirement 1 and its subrequirements are partially fulfilled. The user interface is already designed in the same way for all tools (see Figure 4.6). However, currently the icons, error messages, and labels differ between tools. Furthermore, restrictions are named differently: In DOORS, a restriction is correctly called *restriction*, in Excel, it is named *condition*, and in EA, *constraint*. This may confuse users of multiple tool integrations, and therefore should be fixed in future releases.

Requirement 2: The tool integration shall comply with usability heuristics.

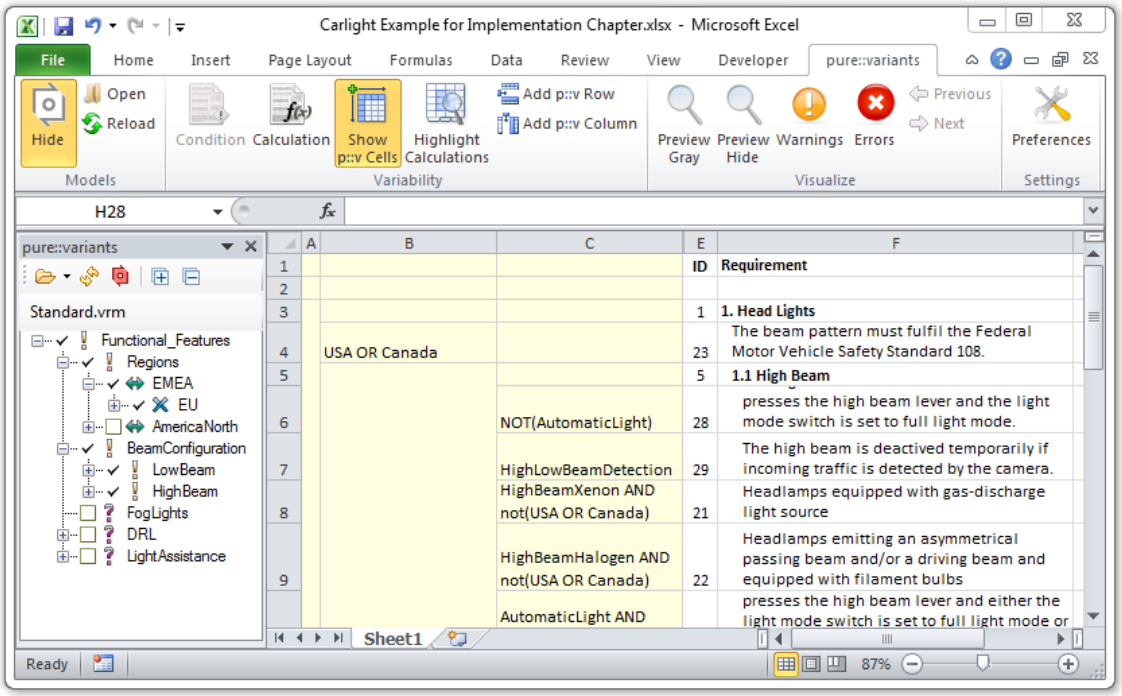
ID	D	EX	EA	Requirement Title
2				The tool integration shall comply with usability heuristics.
2.1	✓	✓	✓	The tool integration shall display a wait cursor for actions that take longer than one second.
2.2	✓	TIC	✓	Tool integrations shall provide progress bars for actions that take longer than ten seconds.
2.3	✓	✓	✓	Tool integrations shall be consistent with pure::variants and the extended tool.
2.4	TIC	TIC	TIC	Users shall be able to interrupt all actions that take longer than 10 seconds.
2.5	✓	TEC	TEC	For all actions that change the master artifact model, the tool integration shall provide a means to undo or redo the action.
2.6	✓	✓	✓	Users shall always be able to cancel a dialog.
2.7	NN	NN	NN	Tool integrations shall warn users before performing an irreversible action.
2.8	✓	✓	✓	Tool integrations shall always support users in entering data.
2.9	✓	✓	✓	The tool integration's user interface and visualizations shall be kept as simple as possible.
2.10	TIC	TIC	TIC	If an error occurs, the tool integration shall provide simple, constructive, and specific instructions for recovery.

Table 4.3: Status of Requirement 2 and its subrequirements. We use the same notation as in Table 4.2.

We could not fulfill Requirement 2 completely, since Excel and EA do not support undoing or redoing actions triggered by a custom extension. This is a usability problem. However, we cannot fix it using the undo functions of Excel and EA. Instead, we could try to implement a workaround. Furthermore, the integration to Excel does not



(a) DOORS



(b) Excel

Figure 4.6: DOORS and Excel with the respective pure::variants integration. For the integration to EA, refer to Figure 2.9.

show a progress bar for long processes, and long processes cannot be canceled in all integrations. Additionally, we have not yet optimized the error messages to comply with Requirement 2.10. Since it is possible to fulfill these requirements, they should be addressed in future work.

Requirement 3: Variant management shall seamlessly integrate itself into the development tools by providing a user interface.

ID	D	EX	EA	Requirement Title
3		✓	✓	Variant management shall seamlessly integrate itself into the development tools by providing a user interface.
3.1	TEC	✓	✓	The tool integration's user interface shall integrate itself directly into the development tool's user interface.
3.2				The tool integration's user interface shall be displayed in its own window, which can be triggered from the development tool's user interface.
3.2.1	(EC)			The tool integration's user interface shall be brought to front when the user triggers a variant-management action from within the development tool.
3.2.2				The tool integration's user interface shall stay in front of the development-tool window.

Table 4.4: Status of Requirement 3 and its subrequirements. We use the same notation as in Table 4.2.

We could fulfill almost all subrequirements of Requirement 3. For the only technical constraint (DOORS does not allow embedding custom user interfaces), we provided an alternative requirement, which we could satisfy almost completely. Only Requirement 3.2.1 is not satisfied entirely: Currently, the loaded pure::variants model is hidden by the DOORS window while editing pvSCL rules. We should check whether users need to view the model tree when editing pvSCL rules. If this is the case, we need to fix the implementation.

Requirement 4: Development tools shall support the user during the creating of variation points.

The current tool integrations satisfy almost all of the requirements in the respective requirements document. Only Requirement 4.2.3 is not fulfilled, since currently attributes are not displayed in the representation of pure::variants models. For nearly all requirements with technical constraints, we provide an alternative requirement. Only for Requirements 4.3.1 and 4.3.2, no alternative exists. Hence, it is not possible to add or edit multiple restrictions in DOORS. We believe that this is not critical, since elements are arranged hierarchically in DOORS. Thus, an element's variation point also relates to all child elements. In our opinion, editing multiple restrictions is only necessary, when a variation point can only relate to one element. We need to verify whether users really do not need to edit multiple restrictions using the DOORS tool integration.

ID	D	EX	EA	Requirement Title
4				Development tools shall support the user during the creating of variation points.
4.1	✓	✓	✓	The tool integration shall provide editors for different kinds of pvSCL rules.
4.1.1	✓	✓	✓	All pvSCL editors shall provide autocompletion based on the loaded models.
4.1.1.1	✓	✓	✓	The autocompletion of all pvSCL editors shall show all possible pvSCL keywords.
4.1.1.2	✓	✓	✓	The autocompletion of all pvSCL editors shall show all features of the loaded models.
4.1.1.3	✓	✓	✓	The autocompletion of all pvSCL editors shall show all attributes of the entered feature.
4.1.1.4	TIC	TIC	TIC	The autocompletion of all pvSCL editors shall show all components of the loaded models.
4.1.2	✓	✓	✓	All pvSCL editors shall provide pvSCL compliant syntax highlighting.
4.1.3	✓	✓	✓	All pvSCL editors shall check the entered rule for errors before writing pvSCL rules to variation points.
4.2	✓	✓	✓	The tool integration shall allow loading and displaying feature models and variant result models.
4.2.1	✓	✓	✓	Loaded models shall be saved and reloaded when opening a master artifact model.
4.2.2	✓	✓	✓	The last five opened models should be quickly accessible.
4.2.3	TIC	TIC	TIC	Feature models and variant result models shall show all elements needed for entering pvSCL rules.
4.3	✓	✓	✓	The tool integration shall provide an editor for writing restrictions.
4.3.1	TEC	✓	✓	It shall be possible to add multiple restrictions at once.
4.3.2	TEC	✓	✓	It shall be possible to edit multiple restrictions at once.
4.3.3	✓	✓	✓	The restriction editor shall be easily accessible.
4.3.3.1	TEC	TEC	✓	The restriction editor shall be accessible through the context menu of each artifact.
4.3.3.2	TEC	✓		The restriction editor shall be accessible through a button in the integration's user interface.
4.3.3.3	✓			The restriction editor shall be accessible through a menu of the development tool.
4.3.3.4	✓	✓	✓	Triggering the restriction editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
4.4	✓	✓		The tool integration shall provide an editor for writing calculations.
4.4.1	TEC	TEC		The calculation editor shall be similar to the restriction editor.
4.4.2	✓	✓		The calculation editor shall support users in editing all calculations of the selected element.
4.4.3	✓	✓		The calculation editor shall be easily accessible.
4.4.3.1	TEC	TEC		The calculation editor shall be accessible through the context menu of each element.
4.4.3.2	TEC	✓		The calculation editor shall be accessible through a button in the integration's user interface.
4.4.3.3	✓			The restriction editor shall be accessible through a menu of the development tool.
4.4.3.4	✓	✓		Triggering the calculation editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
4.5	✓	✓		The tool integration shall provide an editor for writing constraints.

Table 4.5: Status of Requirement 4 and its subrequirements. We use the same notation as in Table 4.2.

Requirement 5: The tool integration shall support users in deleting variation points.

No subrequirements exist for Requirement 5. Thus, we omit a table showing the current status of the requirement. The requirement is fulfilled, because, in EA, the tool integration supports users in deleting variation points. In DOORS and Excel, users can easily delete existing variation points using the functions of the extended tool.

Requirement 6: Variant management shall support the visualizing of variability-affected model elements.

ID	D	EX	EA	Requirement Title
6	✓	✓		Variant management shall support the visualizing of variability-affected model elements.
6.1	✓	✓	✓	Variation points shall be visibly different from other elements of the master artifact model.
6.1.1		✓	TIC	The tool integration shall provide a means to highlight restrictions.
6.1.2		✓		The tool integration shall provide a means to highlight calculations.
6.1.3				The tool integration shall provide a means to highlight constraints.
6.2	TIC	TIC	TIC	The tool integration shall provide a means to quickly find all model elements related to a specific variation point.

Table 4.6: Status of Requirement 6 and its subrequirements. We use the same notation as in Table 4.2.

The tool integrations partly fulfill Requirement 6. Only for EA, not all variability-affected elements are highlighted. For instance, restrictions attached to UML attributes are not visible in diagrams. Since it is possible to color elements in EA, we should be able to highlight diagram elements that contain variability. We address this issue in future work. We also do not fulfill Requirement 6.2, because of time constraints.

Requirement 7: The tool integration shall support users in finding variation-point errors.

ID	D	EX	EA	Requirement Title
7	✓	✓	✓	The tool integration shall support users in finding variation-point errors.
7.1	✓	✓	✓	The tool integration shall provide a means to highlight pvSCL syntax errors and unknown names.
7.2	✓	✓	✓	When searching variation-point errors, users shall be able to distinguish between syntax errors and unknown names.
7.2.1		✓	✓	Syntax errors and unknown names shall be distinguished by different colors.
7.2.2	✓			Syntax errors and unknown names shall be distinguished by icons.

Table 4.7: Status of Requirement 7 and its subrequirements. We use the same notation as in Table 4.2.

We satisfy Requirement 7 completely. No technical constraints exist. Notable is only that we introduce an alternative error visualization in DOORS (Requirement 7.2.2) to provide better consistency with the pure::variants error visualization. All tool integrations that can attach icons to elements can use this visualization.

Requirement 8: Variant management shall support the previewing of variants.

ID	D	EX	EA	Requirement Title
8	✓	✓	✓	Variant management shall support the previewing of variants.
8.1	✓	✓	✓	The tool integration shall provide a preview that is faster than a transformation of the same master artifact model.
8.2	✓	✓	✓	The tool integration shall provide a preview that grays out elements not included in the variant.
8.3	✓	✓	TEC	The tool integration shall provide a preview that hides elements not included in the variant.
8.4			✓	The tool integration shall provide a preview that highlights all variability-affected elements that are included in the variant.
8.5	✓	✓		Preview visualizations shall replace calculation values.

Table 4.8: Status of Requirement 8 and its subrequirements. We use the same notation as in Table 4.2.

The tool integrations fulfill Requirement 8. The only difference between the integrations is that, due to a technical constraint, EA does not provide a preview that hides excluded elements of a variant (*hide preview*). Instead, it supports highlighting all variability-affected elements that are included in a variant (*highlight preview*). This is inconsistent. For consistency, we could either support the highlight preview in all tool integrations, or omit this type of preview in EA. To find out which option is preferable, we need to ask users which preview they prefer, and whether they use the highlight preview at all. Furthermore, we need to find out whether the missing hide preview is important to users. If this is the case, we could try to introduce a complex workaround.

4.5 Summary

Apart from several limitations, we were able to apply the workflow for the tools DOORS, Excel, and EA. Limitations are two-fold: First, we have not yet implemented several requirements for all tools. We will address them in future work:

1.1 Tool integrations shall use the same or similar icons for the same functions (all)

1.2 Tool integrations shall communicate similar messages in similar situations (all)

2.2 Tool integrations shall provide progress bars for actions that take longer than ten seconds (Excel)

2.4 Users shall be able to interrupt all actions that take longer than 10 seconds (all)

2.10 If an error occurs, the tool integration shall provide simple, constructive, and specific instructions for recovery (all)

4.2.3 Feature models and variant result models shall show all elements needed for entering pvSCL rules (all)

6.1.1 The tool integration shall provide a means to highlight restrictions (EA)

7.3 The tool integration shall provide a means to navigate between errors (DOORS, EA)

Second, we could not fulfill some requirements due to technical constraints. We provided alternatives for most of the requirements. However, for the following requirements, no alternatives exist so far:

2.5 For all actions that change the master artifact model, the tool integration shall provide a means to undo or redo the action (Excel, EA)

4.3.1 It shall be possible to add multiple restrictions at once (DOORS)

4.3.2 It shall be possible to edit multiple restrictions at once (DOORS)

8.3 The tool integration shall provide a preview that hides elements not included in the variant (EA)

Therefore, we need to verify whether they are critical, and thus find out whether a complex workaround is necessary. Additionally, in DOORS, the tool integration window showing the loaded model is hidden behind the DOORS window. Thus, users cannot refer to the model when editing pvSCL rules. We need to verify whether users need to view the feature model while editing pvSCL rules. If this is the case, we need to fix this issue or provide an alternative solution.

Next, we evaluate whether we could fulfill our goals.

5. Evaluation

In this chapter, we evaluate whether the proposed workflow fulfills the goals we set in Chapter 1.

Our main thesis goals were to propose a workflow for developing new variant-management tool integrations that:

G1. improve the consistency between different variant-management tool integrations

G2. reduce the time-to-market for new variant-management tool integrations

Additionally, the quality of the produced tool integrations should still be at the same or a higher quality level than existing tool integrations. Thus, the workflow also should:

G3. promote good user experience

G4. relate to the everyday tasks of variant-management practioners

For evaluation of these goals, we first argue whether the workflow fulfills the main goals. Then, we present an interview with a pure::variants customer, and reason based on the results whether Goal 3 and 4 are met.

5.1 Main Thesis Goals

In this section, we describe whether we could fulfill the main goals of this thesis. We begin with Goal 1.

Goal 1. Consistency

To evaluate whether Goal 1 is met, we need to answer the question:

> Does the proposed workflow support consistency between tool integrations in a better way than the old workflow?

Consistency is hard to define (Grudin [Gru89]) and thus difficult to measure. Instead, we could measure the *effects* of good consistency, which are a reduced time needed to learn the functions of a tool integration and a reduced error rate (see Section 2.5.1). To this end, we could, for example, conduct a user study. The study would consist of multiple sessions. In each session, participants would use a different tool integration. We would state that the consistency is good, when the time needed to complete a task and the error rate reduces with each session.

However, we cannot conduct such a user study, because the current tool integrations do not fulfill all requirements, yet. Instead, we discuss based on plausibility whether the consistency has improved. We argue this is the case, for two reasons: First, consistency between tool integrations is explicitly supported by Requirement 1 and its subrequirements. In previous implementations of tool integrations, requirements for consistent icons, labels, and messages were not fulfilled. We hope that by including basic consistency requirements in the requirements document for each new tool integration, developers will focus more on consistency. In these requirements, we also propose that icons, labels, and messages should be based on the same implementation. Thus, developers can ensure that changes are propagated to all integrations, and hence ensure consistency.

Second, consistency is indirectly supported by reusing requirements whenever it is possible. If the requirements for a new tool integration are the same, the implemented functions should be consistent between different tool integrations. In Section 4.2, we applied the workflow to three tools. The generated requirements documents differed only for two reasons: Either technical constraints of the extended tool forced us to provide alternative requirements, or the corresponding pure::variants connectors supported different types of variation points (see Table 4.1). These alternative requirements introduce inconsistencies between tool integrations. Another solution would be to define simple requirements that we could fulfill in all tool integrations. For example, we could omit Requirement 3.1: *The tool integration's user interface shall integrate itself directly into the development tool's user interface*, because DOORS does not support it. Instead, all tool integrations could provide a separate window. Even though this solution would improve the consistency, it would also significantly reduce the efficiency of the integrations to EA and Excel, since users would need to manage multiple windows (consistency tradeoff: Rosson and Carroll [RC02], p.127; Grudin [Gru89]). With our workflow, we ensure that for the same technical constraints, always the same alternative requirement will be used. Thus, we still foster consistency with alternative requirements.

Goal 2. Reduced Time-to-Market

Our second goal was to reduce the time-to-market for new tool integrations. We argue that the proposed workflow eases the planning phase for new integrations, for two reasons: First, the feature model *technical features* contains features for all relevant extension mechanisms that the respective tool provides for embedding a tool integration (e.g., can elements be colored or hidden). Hence, tool-integration developers know exactly which extension mechanisms they need to test. This supports systematic testing of extension possibilities, and thus saves time. Second, the requirements document on which the implementation is based can be generated. Hence, developers do not need to write new specification documents. They only need to adapt the document in case of impossible requirements or improvements.

5.2 Quality Goals

Apart from being consistent and fast to develop, the produced variant-management tool integrations should still provide a good user experience (Goal 3) and relate to the users' tasks (Goal 4).

To evaluate whether the proposed workflow promotes good user experience, the best method is a user study based on tool integrations that fulfill the requirements proposed in Section 3.2 (Nielsen [Nie93], p.165). The setup would be similar to the user study we proposed for measuring consistency. However, we cannot conduct such a study for the same reason why we cannot measure the consistency between tool integrations: The current integrations do not fulfill all requirements, yet. Therefore, we argue whether the workflow promotes good user experience: To support usability, we defined Requirement 2: *The tool integration shall comply with usability heuristics*. Its subrequirements relate to usability problems identified in literature. In our opinion, fulfilling these requirements supports good user experience, because major usability problems are prevented. Nearly all usability heuristics are covered by one or more requirements (see Section 3.2.2, p.36f). We only omit an explicit requirement for Heuristic: *Flexibility and efficiency of use*, because we already address it by providing features, such as autocompletion, error check, previewing variants, etc.

Although fulfilling the usability requirements will prevent basic usability problems, this does not replace asking real users for their opinion or conducting a user study. Therefore, we prepared an interview with pure::variants customers, who use at least one tool integration regularly and can thus evaluate it. Based on their feedback, we can identify usability problems, and find out whether the proposed workflow relates to the users' everyday tasks (Goal 4). We describe the interview in the next section, and present the results in Section 5.2.2.

5.2.1 Interview Setup

During the interview, we observed customers who are experienced with the integration to DOORS execute simple tasks and asked them for feedback on the integration. We

chose to only conduct the interview for the integration to DOORS, because significantly more customers use this integration compared to Excel and EA. Nevertheless, we still had difficulties finding participants, presumably because of time constraints of customers. We conducted the interview with two participants. However, one participant had not used the integration before. Hence, we only evaluate the interview with one customer. The participant had ten years experience with DOORS and three years experience with pure::variants. He is not the end user of the integration, but has tested its functions, so that his colleagues can use it in their everyday tasks.

We conducted the interview in form of a WebEx[1] Meeting. The duration was planned for 30 to 45 minutes, depending on how much feedback the participant communicated. The procedure was as follows: After briefly explaining the purpose of the interview, we asked the participant to execute three simple tasks using the tool integration. The participant used his own project, which is more realistic than providing an example project, since we could better observe problems related to the nature of the participant's project, or the setup of his work environment. As a drawback, we cannot give detailed information on the project due to company policies. We defined the following typical tasks:

1. Add a restriction
2. Add a calculation
3. Check whether the changed pvSCL rules would produce the expected results

We chose to first watch users while they execute tasks, so that we can indentify usability problems based on errors the user makes, and the user is reminded of possible sources for comments. We were especially interested in whether the participant triggered and used the new calculation editor correctly, and which type of preview he chose for Task 3.

We then asked questions regarding the tool integrations. First, we wanted to find out which functions the participant used regularly, and get general feedback:

1. Which of the shown functions do you use regularly? Which do you use seldom or never?
2. Are you unhappy with the realization of some functions? If yes, with which functions are you unhappy? Which solution would you prefer?
3. Do you have more suggestions for improvements?

Then, we asked specific questions relating to the requirements we could not fulfill in DOORS, and for which no alternative requirement exists (see Section 4.5)

4. How useful would you find editing multiple restrictions or constraints?
5. How useful would you find it to be able to view the current pure::variants model when editing pvSCL rules?
6. Do you prefer one type of preview? If yes, which preview and why?

Based on the answers, we evaluated how important the implementation of the missing requirements is.

[1] http://www.webex.com/ (last accessed on May 30, 2013)

5.2.2 Interview Results

In this section, we describe the results of the interview with our participant. From the results, we derive usability problems and suggestions for improvements.

Execution of Tasks

The participant executed all tasks using one of his own DOORS projects in his own environment. He loaded a pure::variants VRM and added a restriction successfully (Task 1). However, he did not know how to use calculations (Task 2), because calculations were not necessary in his usual work projects. Hence, we gain no insight whether the approach for adding calculations (Requirement 4.4.2) introduces usability problems. For previewing a variant (Task 3), he chose to gray out excluded elements instead of hiding them. However, the elements were not colored in gray, but in orange, because the color scheme of the project was different from the default color scheme. With graying out elements, we intended to draw the user's attention away from the excluded elements of the variant. Unfortunately, orange color achieves the opposite. Thus, this is a usability problem. However, it is not caused by faulty requirements, since Requirement 8.2 states that elements should be colored gray. Hence, we classify it as a bug that we need to fix.

Suggestions for Improvements

During executing the tasks and answering Questions 2 and 3, the participant provided the following suggestions for improvements:

1. *Load Variant Description Models (VDMs) instead of Variant Result Models (VRMs)*: In pure::variants, the participant works with server-based VDMs. However, in the tool integration, he can only load VRMs, which are stored locally. Hence, he suggested to be able to load VDMs instead. (A VDM is a pure::variants model that contains the feature selections which are needed for generating a variant. A VRM is similar. However, it contains only information on features that are included in the variant.)
2. *Improve the editing of pvSCL rules for large feature models*: The feature models used by the participant are very large, and many feature names begin with the same text. Thus, the autocompletion is not very useful, since after entering the beginning of a feature, still many feature names are suggested. To fix this, the participant suggested to move features from the loaded pure::variants model to the pvSCL editor either by drag and drop or through the feature's context menu. Furthermore, he suggested that the autocompletion box should contain not only words that start with the entered text, but also words that contain the entered text.
3. *The pvSCL editor should visualize whether the entered rule evaluates to true*: To reduce the necessary calls of previews, the participant also proposes that the pvSCL editor shows to the user what the current pvSCL rule would evaluate to. This visualization should be updated during typing.

Thus, there is room for improvement, which we consider in future work.

Specific Questions

Additionally, we asked more detailed questions that relate to specific requirements (see Section 4.5). To Question 4: *How useful would you find editing multiple restrictions or constraints?*, the participant answered that he would not find this useful in DOORS. He reasoned that, since DOORS elements are hierarchical, editing multiple restrictions would only be useful for elements that are not neighbors. However, multiselection is only possible in DOORS for adjacent elements. Hence, we argue that we do not need alternative requirements for Requirements 4.3.1 and 4.3.2.

Regarding Question 5: *How useful would you find it to be able to view the current pure::variants model when editing pvSCL rules?*, the participant had already answered with his request to use pure::variants models during editing pvSCL rules (see Suggestion 2). Therefore, we need to improve the current requirements for editing pvSCL rules to support his use case.

To Question 6: *Do you prefer one type of preview? If yes, which preview and why?*, he answered that he would prefer graying out elements. He reasoned that, using this preview, he would still be able to see elements, and thus errors would be more present. Nevertheless, he would use the hide preview whenever many elements are excluded, or for checking the final version of the master artifact document. This confirms our argumentation of Use Case 5 and Requirement 8.2 – 8.3. Since graying out excluded elements is perceived as more valuable than hiding them, we argue that it is not critical that the hide preview is not implemented for EA.

5.2.3 Limitations

Several factors limit the validity of the interview: First, we conducted the interview with only one participant. Thus, generalizability is limited (Flyvbjerg [Fly06]). Nevertheless, the results are useful, since they provide suggestions for improvement, and show usability problems. Second, the participant was not the end user of the tool, but has tested its functions, so that his colleagues can use it in their everyday tasks. Hence, he may have focused on different problems than end users. However, since he tested the integration for his colleagues, he knew their work scenario in detail, talked to them about problems, and the setup of his work environment was similar. Therefore, we argue that he was in the position to identify the same usability problems as the real end users would have. Third, to keep the duration of the interview short, we asked the participant to execute only three tasks. Therefore, we did not gain insight how he worked with error visualizations. Similarly, we could not evaluate whether triggering and editing calculations involves usability problems, since he had not used calculations before. Hence, we can use the interview results only to evaluate whether our requirements for editing restrictions and previewing variants fulfill Goals 3 and 4.

5.3 Discussion

Based on our argumentation, we conclude that the proposed workflow fulfills our main goals: It increases consistency (Goal 1) and reduces the time-to-market for developing new variant-management tool integrations (Goal 2). We achieve these goals mainly by systematically reusing common requirements between tool integrations.

To further address these goals, we could convert the entire development process of new tool integrations to an SPL. This typically improves user-interface consistency and reduces the development time (van der Linden et al. [LSR07], p.27f). However, the tool integrations are currently not very complex, and common parts are already reused by inheritance of classes. Thus, we first need to evaluate whether the initial effort would pay off.

To evaluate whether the proposed requirements promote consistency (Goal 3), and relate to the tasks of variant-management practioners (Goal 4), we conducted an interview with a pure::variants customer. The results of the interview indicate that both goals are fulfilled with limitations: The participant found the integration useful. However, he criticized the efficiency for editing pvSCL rules for large feature models with similar names. Since large feature models are common among industrial practioners (Berger et al. [BRN+13]), we need to address this problem in future work. He also made other suggestions for improvement, which we will consider in the future.

We conclude that the proposed requirements satisfy Goal 3 and 4, because, first, the results of the interview indicate so. However, we still need to increase external validity by conducting more interviews. Second, the requirements reflect the functions of current tool integrations. Therefore, the quality of integrations should not decrease compared to existing tool integrations. Third, the proposed requirements include usability requirements that should prevent major usability problems. This addresses Goal 3. Fourth, five of the eight proposed main requirements are based on requirements of the SPES_{XT} project, which reflect the demands of industrial partners of the project. Hence, we argue that the majority of requirements is related to the tasks of variant-management practitioners (Goal 4).

In summary, we can state that we fulfilled the goals in general, but that there is still room for improvement.

6. Related Work

Having presented our work in the previous chapters, we discuss work related to our thesis. In this thesis, we proposed concepts for consistent tool integrations that support two basic tasks: First, the tool integrations support users in making variability explicit throughout the different phases of the software-development lifecycle. Second, the integrations support users in reviewing variability information using visualizations. Therefore, we next present publications and tools that also address these tasks. For a better overview, we first present related work that is concerned with making variability explicit. Then, we describe how the presented and other work supports visualizing variability in artifact models. Finally, we present work that focuses explicitly on supporting users throughout the complete product lifecycle.

Making Variability explicit

Numerous approaches how to denote variability were presented in literature. Heidenreich et al. give an extensive overview of how variability can be represented [HSS+10]. However, only few of the approaches provide tool support for making variability explicit. For example, Czarnecki and Antkiewicz propose a template-based method to represent variability in different types of artifact models [CA05]. The basic idea is to attach so-called *presence conditions* to artifacts. Similar to our concept of restrictions, presence conditions describe relations to features. If the referenced feature is not part of the configuration (in our case represented by a VDM or VRM), the annotated elements are removed during variant generation. To illustrate the approach, they provide a prototype implementation for UML models. The prototype *fmp2rsm* integrates their *feature modeling plugin* (Antkiewicz and Czarnecki [AC04]) with IBM Rational Software Modeler. Using the tool, users can define presence conditions by adding special UML profiles to the respective artifacts.

Trung and Jarzabek present another prototype for defining variability [TJ]: Their tool *Documentation Management Environment (DME)* is an add-in to Microsoft Office Word, which supports generating documentation files from a template. To make

variability explicit, users can define seven different types of fragments, such as unique fragments, repeated fragments, parameters, or links to other documents. To generate a variant, users define custom values, which DME uses to replace the defined fragments. In our work, calculations represent a similar solution: Users can attach a rule to a text fragment, which is replaced during variant generation with a value specified in the respective VDM.

Another example for a tool that supports denoting variability is *FeatureMapper* (Heidenreich et al. [HKW08][HcW08]). It extends Eclipse, such that users can make variability explicit for all models based on the Eclipse Modeling Framework (e.g., UML models). Other than in the tool integrations we presented, users can select features directly from the feature model to map them to the current selection. This variability is then stored in a separate *mapping model*, instead of the model itself.

Furthermore, Ghanam and Maurer propose a method for linking features to code artifacts by using *Executable Acceptance Tests (EATs)* [GM10]. An EAT documents the specifications of a feature and developers can run it to verify whether the implemented behavior is correct. The authors propose to extend all leaf features with one or more EATs. Since EATs natively link to code artifacts, variability is made explicit through EATs. After product derivation, all EATs selected in the configuration are executed. Thus, traceability is supported, since developers instantly get feedback when changes have introduced errors. To illustrate the concept, Ghanam and Maurer implemented a prototype based on Feature Model DSL[1], which extends Microsoft Visual Studio with a toolbox for feature modeling. This EAT tool mainly provides a method for linking EATs and features.

Visualizing Variability in Artifact Models

Several of the presented tools additionally provide visualizations that support users in better understanding the explicit variability. Thus, the tools support users in reviewing variability. For instance, the tool *fmp2rsm* performs automatic coloring, which assigns a different color for each presence condition. *FeatureMapper* supports a similar visualization: Users can assign colors to features, which are then used to color the related artifacts. It also provides a visualization that is similar to our gray-out preview: All excluded elements of a variant are grayed out. Furthermore, FeatureMapper supports visualizations that highlight feature-dependent changes, and highlight all artifacts related to the selected feature while graying out all others. The EAT tool by Ghanam and Maurer also provides a visualization that grays out deselected elements. Additionally, it enables users to execute tests, and visualize the results by coloring EAT elements either green (successfull) and red (failed).

Heuer et al. propose another prototype that visualizes variability in artifact models [HLMS10]. The authors identify challenges to visual variability modeling, and address these challenges by proposing several visualizations, which they implement in their prototype *Remmidemmi*. For artifact models, they propose a selection-based visualization.

[1] http://featuremodeldsl.codeplex.com/

It highlights all elements in the artifact model that are related to the currently selected variation point, and it grays out elements that belong to other variation points. Furthermore, they provide a view that resembles the method we proposed in Requirement 6.2: *The tool integration shall provide a means to quickly find all model elements related to a specific variation point*: Users can drop artifacts or variants to the view. For each dropped element, the tool displays all related variants or artifacts in a tree structure.

From the presented tools, we learn that other researchers also consider the gray-out preview useful. However, they focus more on visualizing the connection between features and artifacts; For example, by assigning feature colors to artifacts (Czarnecki et al. [CAK+05], Heidenreich et al. [HcW08]), or by providing a special view (Heuer et al. [HLMS10]). We could use these proposed visualization as inspiration for future work.

All of the presented approaches focus only on one type of artifact model, and thus only on one phase of the product lifecycle. However, we aimed to address the entire lifecycle. Therefore, we next present research and tools that consider all lifecycle phases.

Variability Management Throughout the Complete Product Lifecycle

With their tool *Gears*, BigLever Software also provides a commercial tool for SPLE throughout the entire product lifecycle. It supports similar tasks as pure::variants: Comparable to pure::variants' connectors, there are bridges to different product lifecycle tools. The bridges also integrate themselves into the respective tool, such that users can execute variant-management specific tasks. Like our tool integrations, they support a visualization that grays out excluded elements of a variant (Krueger and Bakal [KB09]), and, similar to our calculations, the DOORS bridge supports substituting texts (Krueger and Jackson [KJ09]).

Similar to our work, some authors provide a concept for creating tool support for variability management throughout the entire product lifecycle. For example, Schmid and John argue that to ease the transition from single-system development to SPLE as many of the existing approaches as possible should be maintained [SJ04]. However, the notation of variability-affected artifacts differs between the used tools. Therefore, they suggest a customizable approach to full-lifecycle variability management. Like our workflow, the approach consists of several steps. Basically, it is necessary to provide a means to model variability and a method for mapping variability to the different notations of artifact models. As in pure::variants and its connectors, only the mapping needs to be different for each solution-space tool. The variability modeling approach should be the same.

Hammouda et al. aim to address the same problem [HHPK05]. They describe an approach to create tool support for representing variability in heterogenous artifacts. To make variability explicit, they introduce the concept of feature variation patterns. These patterns should be managed by a central tool, which guides developers in denoting variability.

Although the presented papers also provide an approach for creating tool support throughout the different phases of the product lifecycle, they do not address the same problems as our work. They propose concepts that answer the questions how variability is represented, and how variants can be generated; whereas we propose concepts that support users in editing and reviewing variability in a consistent, efficient, and error-preventing way.

This concludes our presentation of related work. Next, we summarize our results, and present directions for future work.

7. Conclusion

Software Product Line Engineering (SPLE) is a powerful software-engineering approach that increases efficiency of the development process by strategically reusing common parts of multiple similar, yet different products. Systematically managing variabilities is fundamental to successful development of a Software Product Line (SPL).

However, in literature, a lack of integration into software-development tools has been reported. Since variability is represented differently in each development tool (Pohl [PBL05], p.75), SPL developers need to learn a new approach for denoting variability in each tool. With this thesis, we aimed to address this problem. Since the variant-management tool pure::variants already provides tool integrations, we suggested to improve pure::variants' tool integrations.

Based on requirements of the SPES_{XT} (Software Plattform Embedded Systems - Extended) project, and our experience with implementing pure::variants tool integrations, we identified two requirements for variant-management tool integrations: First, tool integrations shall be consistent between each other, so that the time to learn the functions of an integration is reduced for users of multiple tool integrations. Second, the development of new tool integrations shall be as simple as possible.

Based on these requirements, we set our thesis goals: We aimed to provide a workflow for developing new tool integrations. The workflow should:

G1. improve the consistency between different variant-management tool integrations

G2. reduce the time-to-market for new variant-management tool integrations

G3. promote good user experience

G4. relate to the everyday tasks of variant-management practioners

By addressing these goals, we intended to help tool-integration developers to faster create new integrations that provide good user-interface consistency. These integrations should then support variability modelers and verifiers in their work with different development tools.

To achieve the goals, we defined three tasks:

T1. Collect and define variant-management use cases concerning the work with different product-lifecycle tools

T2. Derive a set of requirements from different sources, such as the specified use cases, the requirements of the SPES_{XT} project, and usability guidelines

T3. Based on the requirements, propose a workflow for planning and implementing a new tool integration

To address the first task, we argued that variant-management tasks are basically the same in all software-development tools: Users need to add, edit, and review variability information. Hence, we defined use cases for these tasks that are independent of the development tool.

According to Task 2, we collected and derived requirements for variant-management tool integrations from different sources. We defined alternative requirements for all requirements that cannot be fulfilled in existing tool integrations, because of technical constraints of the development tool.

Regarding Task 3, we argued that we can increase consistency and reduce development time by systematically reusing common requirements between tool integrations. Therefore, we created the pure::variants integration project. It contains feature models that reflect all factors needed for choosing between alternative requirements (e.g., extension mechanisms supported by the development tool). Based on these feature models, we added variability information to the requirements document of Task 2. The steps of the proposed workflow are as follows:

1. Define a Variant Description Model (VDM) of the pure::variants Integration Project
2. Generate a requirements document
3. Analyze and adapt the requirements
4. Implement the requirements

To verify whether it is technically possible to implement variant-management tool integrations based on the workflow, we applied the suggested workflow to three tools. We found out that it is indeed possible with several limitations. The following features could not be implemented in all tools:

1. undo and redo functionality
2. adding and editing multiple restrictions at once
3. a variant preview that hides excluded elements of a variant

For evaluating whether the workflow and the included requirements fulfill the goals of this thesis, we argued for each goal whether it is fulfilled. We came to the conclusion that the workflow improves consistency, because explicit requirements for consistency exist, and functional requirements are reused across tool integrations. Thus, the implementation based on common requirements should be the same. Furthermore, the workflow reduces the time-to-market for new tool integrations, because requirements documents can be generated, which saves time. Additionally, the feature models of the pure::variants integration project provide a good base for the initial evaluation of the extension mechanisms supported by each tool.

Regarding Goal 3 and 4, we prepared an interview with pure::variants customers. We argued that we could satisfy both goals, for several reasons: First, the results of the interview were mainly positive, although there is room for improvement. However, we still need to conduct more interviews to increase external validity. Second, the quality of integrations should not decrease compared to existing tool integrations, since the proposed requirements reflect the functions of current tool integrations. Third, we argued that Goal 3 is fulfilled, because the proposed requirements include usability requirements that should prevent major usability problems. Fourth, five of the eight proposed main requirements are based on requirements of the SPES_{XT} project, which reflect the demands of industrial partners of the project. Therefore, we argued that the majority of requirements is related to the tasks of variant-management practitioners, and thus Goal 4 is fulfilled.

Hence, we fulfilled our goals. We proposed a workflow for developing consistent variant-management tool integrations. Thus, we support tool-integration developers to faster develop new integrations that provide good user-interface consistency. The created integrations support variability modelers and verifiers in their work with different development tools independent of the used variability representation. Next, we discuss how we plan to continue our work on variant-management tool integrations.

Future Work

First, we need to address the requirements that are currently not fulfilled due to time constraints:

- **1.1** Tool integrations shall use the same or similar icons for the same functions.
- **1.2** Tool integrations shall communicate similar messages in similar situations.
- **2.2** Tool integrations shall provide progress bars for actions that take longer than ten seconds.
- **2.4** Users shall be able to interrupt all actions that take longer than 10 seconds.
- **2.10** If an error occurs, the tool integration shall provide simple, constructive, and specific instructions for recovery.
- **6.1.1** The tool integration shall provide a means to highlight restrictions.
- **6.2** The tool integration shall provide a means to quickly find all model elements related to a specific variation point.
- **7.3** The tool integration shall provide a means to navigate between errors.

Furthermore, we need to evaluate the use of the highlight preview that is already implemented in Enterprise Architect (EA) and IBM Rational Rhapsody (Rhapsody). If it is useful, it should also be part of all other integrations; if not, we can remove it from the integrations to EA and Rhapsody to improve consistency between tool integrations.

Based on the interview with a pure::variants customer (see Section 5.2.2), we identified several suggestions for improvement:

1. *Load Variant Description Models (VDMs) instead of Variant Result Models (VRMs)*: To better address tasks of users with a server-based scenario, we could support loading another type of pure::variants models.
2. *Improve the editing of pvSCL rules for large feature models*: To better support users with large feature models that contain similar names, we could enable users to drag features from the current pure::variants model to the pvSCL rules they are editing, or autocompletion could evaluate more than the beginnings of feature names.
3. *The pvSCL editor should visualize whether the entered rule evaluates to true*: To further increase efficiency of reviewing variability information, the pvSCL editor could display the evaluation result of the entered rule.

We plan to address these suggestions. Especially, the second suggestion is important, because regarding the scenario of large feature models with similar feature names, our measures to improve efficiency of editing variation points are not sufficient. To address these suggestions, we need to adapt the requirements, and update all existing tool integrations accordingly.

Finally, we will apply this workflow to more tools used during the product lifecycle, so that variant-management practitioners are better supported in making variability explicit and reviewing it.

Appendix A: Use Cases

In this appendix, we present the full use case descriptions. The structure of the use cases is consistent with SPES$_{XT}$ use cases.

ID: 1	**Adding Restrictions**
Detailed Description	A variant modeler wants to edit the variability of a master artifact model. To this end, he identifies an element that should not be contained in all variants. Since the project is very complex, he needs to consult the pure::variants feature model or variant model to find out which features relate to which elements. Therefore, he loads a feature model or variant result model, which he can consult during variation-point creation. Then, he selects the identified element and opens the pvSCL editor for defining a restriction. A restriction is typically composed of feature names and keywords. To enter the correct feature names, the variant modeler looks at the feature model loaded in the pure::variants integration and uses the editor's autocompletion feature. When he presses the OK button, the rule is checked for syntax errors and unknown feature names. If an error is found, he is notified and can choose to correct or ignore the error.
Rationale	For most tools it is possible to edit restrictions without support by a pure::variants integration. However, editing pvSCL rules without help is error-prone, because users need to remember the exact spelling of feature names. Furthermore, it is inefficient, since users would need to switch back to pure::variants to check which features apply to the current rule and if these features are spelled correctly.
Related Goal(s)	Efficiency, Reduce Error-Rate
Related Use Cases	None
Actor(s)	Variant modeler, respective tool user

Tools/Utilities	pure::variants (for editing the pure::variants models), External tool (e.g., DOORS, EA, Excel, Word) and respective pure::variants integration		
Pre-restriction	The feature model and/or variant result model have already been defined using pure::variants.		
Expected Results	The selected element is annotated with a restriction, unless the editor was closed using the cancel button.		
Post-restriction	Except for the pvSCL rule, no changes have been made to the document		
Main Scenario	**Step**	**Actor**	**Interaction**
	1	variant modeler	Loads a pure::variants feature or variant result model
	2	p::v integration	Displays loaded model in a tree
	3	variant modeler	Selects element that contains variability
	4	variant modeler	Triggers the pvSCL editor
	5	p::v integration	Shows the editor with the currently defined restriction of the element (if a restriction already exists)
	6	variant modeler	Enter the rule: Press CTRL + Space to trigger autocompletion
	7	p::v integration	Shows all possible feature names based on the loaded pure::variants model and the entered text. The list updates during typing. Selecting a list entry inserts the selected text at the caret position.
	8	variant modeler	Presses the OK button
	9	p::v integration	Checks the entered rule for pvSCL syntax errors. No errors are found.
	10	p::v integration	Annotates the selected element with the entered pvSCL rule.
Alternative Scenario 1	**Step**	**Actor**	**Interaction**
	...	...	...
	9	p::v integration	Checks the entered rule for pvSCL syntax errors. Finds errors and notifies user with the option to correct or ignore the errors
	10	variant modeler	Chooses to ignore error
	11	p::v integration	Annotates the selected element with the entered pvSCL rule.
Alternative Scenario 2	**Step**	**Actor**	**Interaction**
	...	...	...

	9	p::v integration	Checks the entered rule for pvSCL syntax errors. Finds errors and notifies user with the option to correct or ignore the errors
	10	variant modeler	Chooses to correct errors.
	11	p::v integration	Shows entered rule in pvSCL editor. Underlines unknown feature names and displays an error message in title bar if a syntax error is found.
	12	variant modeler	Corrects error and presses OK.
Alternative Scenario 3	**Step**	**Actor**	**Interaction**
	...	...	...
	8	variant modeler	Presses the cancel button.
	9	p::v integration	Closes dialog and does not write entered rule to master artifact model.

ID: 2	**Adding one restriction to multiple elements**
Detailed Description	While editing the variability of a master artifact model, a variant modeler encounters several elements that should be annotated with the same restriction. To save time, he creates one restriction element and attaches it to all elements. The advantage of this approach is that he only needs to edit one restriction. However, it is not possible in all development tools or in all cases (for example, in UML models, adding one restriction to different elements would only be possible on the same diagram). Therefore, he can alternatively select all elements and trigger the restriction editor. When confirming the entered rule, it is added to all selected elements. When he later needs to edit the rule, he again selects all relevant elements. If their rules are the same, the restriction editor shows the existing restriction. When the variant modeler confirms the entered rule, it is written to all selected elements.
Rationale	Variation points of a master artifact model may often be the same for multiple elements. If in the extended tool a variation point can relate to multiple elements it is probably not critical to edit each element separately. However, not all tools support relation of one variation point to multiple elements (e.g., Excel). Having to enter the same rule over and over again, would be time-consuming and unnecessary. Also the repeated task might frustrate users.
Related Goal(s)	Efficiency, User Satisfaction
Related Use Cases	None

Actor(s)	Variant modeler, respective tool user		
Tools/Utilities	External tool and respective pure::variants integration		
Pre-restriction	The feature model and/or variant result model have already been defined using pure::variants.		
Expected Results	The selected elements are annotated with the same restriction, unless the editor was closed using the cancel button.		
Post-restriction	Except for the pvSCL rule(s), no changes have been made to the document		
Main Scenario	**Step**	**Actor**	**Interaction**
	1	variant modeler	Adds a variation-point element to the master artifact model using a tool-specific approach.
	2	variant modeler	Connects the variation-point element with all elements that shall contain the same restriction.
	3	variant modeler	Triggers the pvSCL editor for the new variation-point element.
	4	p::v integration	Shows the editor (for details see Use Case 1)
	5	variant modeler	Enters the desired rule and presses OK.
	6	p::v integration	Writes the entered pvSCL rule to the variation-point element
Alternative Scenario 1	**Step**	**Actor**	**Interaction**
	1	variant modeler	Selects all elements that should be annotated with the same restriction.
	2	variant modeler	Triggers the restriction editor.
	3	p::v integration	Shows the restriction editor. The initial text is either nothing, or the restriction defined for all elements (if all selected elements are already annotated with a restriction and the restriction is the same for all elements).
	4	variant modeler	Enters the desired rule and presses OK.
	5	p::v integration	Writes the entered restriction for all elements.

ID: 3	Adding Calculations
Detailed Description	A variant modeler wants to edit the variability of a master artifact model. The model contains, for example, a set of parameters. Most variable parts of the master artifact model apply to multiple variants. The variant modeler annotates those parts with restrictions as described in Use Case 1. However, some parameters of the master artifact model differ for nearly every variant. Since, in this case, restrictions would clutter the document unnecessarily, he uses the concept of calculations instead. Hence, he opens the pure::variants project and defines each parameter in an attribute of the project's feature model. In each variant description model, he sets a different value for this attribute. Back in the external tool, he selects the text fragment that should contain one of the parameters after variant generation and triggers the pure::variants calculation editor. Here, he references the attribute. Consistent with the restriction editor, the calculation editor supports editing rules supported by autocompletion and error check. Furthermore, it provides the same options to ignore errors and to cancel editing. Later, during variant generation, the text fragments annotated with calculations will be replaced with the value of the referenced attribute.
Rationale	Using only the concept of deleting artifacts of master artifact models can clutter the master artifact model significantly. For example, when an artifact differs for every variant, a master artifact model based on restrictions would contain all possible cases of the artifact. However, each variant would contain only one of the artifacts. In this case, it makes more sense to define the different values in the pure::variants models and replace one annotated element with the computed value. Furthermore, if parameters are involved, defining them as part of the pure::variants models is more sensible, since each parameter value can be easily traced to a variant by looking at the variant description model. Using this approach, master artifact models become better readable, since they are less complex, and defining the different values can be easily done in pure::variants variant description models.
Related Goal(s)	Efficiency, Reduce Error-Rate
Related Use Cases	Use Case 1
Actor(s)	Variant modeler, respective tool user
Tools/Utilities	pure::variants, external tool and respective pure::variants integration
Pre-restriction	The feature model and/or variant result model have already been defined using pure::variants.
Expected Results	The selected element is annotated with a pure::variants calculation, unless the editor was closed using the cancel button.
Post-restriction	Except the pvSCL rule, no changes have been made to the document

Main Scenario	**Step**	**Actor**	**Interaction**
	1	variant modeler	Loads a pure::variants feature or variant result model.
	2	p::v integration	Displays loaded model in a tree.
	3	variant modeler	Adds an attribute to the pure::variants model, if it is not defined already and saves the model(s).
	4	variant modeler	Triggers a refresh of the integration's loaded models.
	5	variant modeler	Selects element that contains variability.
	6	variant modeler	Triggers the pvSCL editor for calculations
	7	p::v integration	Shows the editor with the currently defined calculation of the element (if a calculation exists)
	8	variant modeler	Enters the rule: Press CTRL + Space to trigger autocompletion
	9	p::v integration	Shows all possible feature names based on the loaded pure::variants model and the entered text. The list updates during typing. Selecting a list entry inserts the selected text at the caret position.
	10	variant modeler	Presses the OK button.
	11	p::v integration	Checks the entered rule for pvSCL syntax errors. No errors are found.
	12	p::v integration	Annotates the selected element with the entered pvSCL rule.
Alternative Scenario 1	**Step**	**Actor**	**Interaction**
	...	...	Similar alternative actions as for restrictions (see Use Case 1)

ID: 4	Finding errors in variation points		
Detailed Description	After adding variability, the variant modeler or a variant verifier wants to check if all entered pvSCL rules and the pure::variants models are correct and consistent. First, he wants to verify whether all pvSCL rules of variation points comply with the pvSCL syntax. To this end, he uses the pure::variants integration to load all feature models relevant for the master artifact model. Then, he triggers a syntax-error visualization, which highlights all elements that are not compliant with pvSCL syntax. To find errors that are not in the active viewport, he uses the integration's navigation buttons. When he has found an error, he selects the related element and triggers the according pvSCL editor, which shows the type of error in the title bar. After correcting all syntax errors, he wants to check whether all features referenced in pvSCL rules exist in the pure::variants feature model and are spelled correctly. To this end, he triggers a semantic-error visualization, which highlights all elements that contain unknown names, such as feature or attribute names. When he has found an error using the same navigation methods as for syntax errors, he selects the related element and triggers the calculation editor, which underlines all unknown names.		
Rationale	During transformation of a variant, syntax errors cause the transformation to fail. Hence, it is possible to check for syntax errors without using syntax-error visualization. However, finding the faulty variation point based on the generated error message is difficult, and only one syntax error is communicated. Using the visualization, variant verifiers can find all errors at once, without having to search them. Semantic-error visualization is even more important, because unknown names do not cause a transformation to fail. Hence, variant verifiers could easily overlook these errors.		
Related Goal(s)	Efficiency, Reduce Error-Rate		
Related Use Cases	Use Case 1, Use Case 3		
Actor(s)	Variant modeler, variant verifier, respective tool user		
Tools/Utilities	(pure::variants), external tool and respective pure::variants integration		
Pre-restriction	The master artifact model contains variation points, and a feature model is already defined.		
Expected Results	Semantic and syntactic errors are found and corrected.		
Post-restriction	The variation points' pvSCL rules do not contain unknown feature names or pvSCL syntax errors.		
Main Scenario	**Step**	**Actor**	**Interaction**
	1	variant verifier	Loads all feature models that are used during transformation

	2	p::v integration	Displays models in tree
	3	variant verifier	Triggers syntax error visualization
	4	p::v integration	Highlights all variation points containing syntax errors and enables buttons for jumping to next or previous error.
	5	variant verifier	Corrects errors, or instructs variant modeler to do so.
	6	variant verifier	Triggers semantic error visualization
	7	p::v integration	Highlights all variation points containing unknown feature names and enables buttons for jumping to next or previous error.
	8	variant verifier	Corrects errors, or instructs variant modeler to do so.

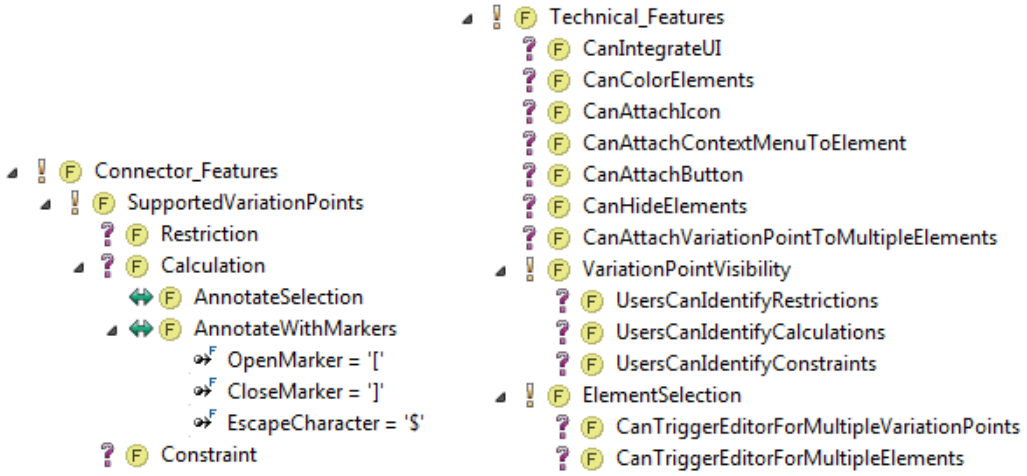
ID: 5	Finding design errors
Detailed Description	A variant verifier (or variant modeler) wants to check whether the entered variation points, combined with the problem-space definition in pure::variants, produce the desired variants. To this end, he opens the master artifact model and loads a variant description model. To see which elements would be included in the variant and which would not, he triggers a preview. He can choose to either gray out or hide elements that would be deleted. Graying out elements has the advantage that deleted elements are still visible, which makes it easier to check whether they should really be deleted. Hiding elements may be better to get an overview when large parts of the master artifact model would be deleted. If the variant verifier finds a design error in a variation point, he triggers the respective pvSCL editor and corrects it. When all errors are corrected, the variant verifier triggers a transformation for a final check of the variant.
Rationale	For finding design errors, previewing variants is more efficient and less error-prone than transforming each variant separately and examining the result, because: • Transforming a variant and opening the output document can be time consuming. When using preview visualizations, the variant verifier needs to open the master artifact model only once. • Identifying falsely deleted elements is easier when they are visible. This is the case when only graying out elements. • All pvSCL rules are visible during preview, which makes it easier to check whether they are correct
Related Goal(s)	Efficiency, Reduce Error-Rate
Related Use Cases	Use Case 1, Use Case 3
Actor(s)	Variant modeler, variant verifier, respective tool user
Tools/Utilities	pure::variants, external tool and respective pure::variants integration
Pre-restriction	A feature model and variant result models are already defined and the master artifact model contains variation points.
Expected Results	The contents of variation points and pure::variants models are correct and consistent. During transformation, the desired variants are produced.
Post-restriction	
Main Scenario	

Step	Actor	Interaction
1	variant verifier	Loads a variant description model.
2	p::v integration	Displays variant description model in tree

	3	variant verifier	Triggers the preview-hide visualization to get a first impression of the variant.
	4	p::v integration	Hides all elements with variation points that would be deleted during transformation and the related elements. Replaces all calculations.
	5	variant verifier	If he finds an error, he corrects it, or instructs variant modeler to do so
	6	variant verifier	For a detailed review of the variant, the verifier triggers the preview-gray-out visualization.
	7	p::v integration	Grays out all elements with variation points that would be deleted during transformation. Replaces all calculations.
	8	variant verifier	If he finds an error, he corrects it, or instructs the variant modeler to do so

Appendix B: Requirements and Feature Models

In this appendix, we present the full requirements definitions and the final version of the related feature models. The structure of the requirments is consistent with SPES_{XT} requirements. Not all fields are necessary for all requirements. For example, we omit the acceptance criterion if the description already contains all necessary information. Furthermore, not all requirements have an explicit source, since some requirements only detail parent requirements. For brevity, we omit all empty fields. To get a first overview of the requirements and the related feature models, we show the final versions in Figure B.1, Figure B.2, and Table B.1.



(a) Connector Features (b) Technical Features

Figure B.1: Final connector and technical feature models.

- Design_Features
  - EditVariationPoints
    - CalculationsSupported
      - Requires: "Calculation"
      - DeleteCalculation
      - EditCalculation
    - ConstraintsSupported
      - Requires: "Constraint"
      - EditConstraint
      - DeleteConstraint
    - RestrictionsSupported
      - Requires: "Restriction"
      - EditRestriction
        - EditMultipleRestrictions
          - NOT(CanAttachVariationPointToMultipleElements) RECOMMENDS SELF
          - Requires: "CanTriggerEditorForMultipleElements"
        - AddMultipleRestrictions
      - DeleteRestriction
    - SupportedNamespace
      - SupportsFeatures
      - SupportsKeywords
      - SupportsAttributes
        - Required For: "Calculation"
      - SupportsComponents
  - LoadPVModels
    - DisplayTree
    - Quickload
    - SaveModels
  - Visualizations
    - Preview
      - PreviewGrayOut
        - Requires: "CanColorElements"
      - PreviewHide
      - PreviewHighlight
    - ErrorVisualization
      - OneErrorVisualization
        - Requires: "CanAttachIcon"
      - SeparateErrorVisualizations
        - SyntaxErrorVisualization
          - Requires: "CanColorElements"
        - SemanticErrorVisualization
    - Requires: "CanColorElements"
      - HighlightVariationPoints
        - HighlightRestrictions
          - (NOT(UsersCanIdentifyRestrictions) AND Restriction) REQUIRES SELF
        - HighlightCalculations
        - HighlightConstraints
      - VariationPointSearch

Figure B.2: Final design feature model.

Restriction	ID	Requirements Title
	1	Variant management shall always be done in the same way for the user independent of the used development tool.
	1.1	Tool integrations shall use the same or similar icons for the same functions.
	1.2	Tool integrations shall communicate similar messages in similar situations.
	1.3	The integrations' user-interface design shall be similar for all integrations.
	2	The tool integration shall comply with usability heuristics.
	2.1	The tool integration shall display a wait cursor for actions that take longer than two seconds.
	2.2	Tool integrations shall provide progress bars for actions that take longer than ten seconds.
	2.3	Tool integrations shall be consistent with pure::variants and the extended tool.
	2.4	Users shall be able to interrupt all actions that take longer than 10 seconds.
	2.5	For all actions that change the master artifact model, the tool integration shall provide a means to undo or redo the action.
	2.6	Users shall always be able to cancel a dialog.
	2.7	Tool integrations shall warn users before performing an irreversible action.
	2.8	Tool integrations shall always support users in entering data.
	2.9	The tool integration's user interface and visualizations shall be kept as simple as possible.
	2.10	If an error occurs, the tool integration shall provide simple, constructive, and specific instructions for recovery.
CanIntegrateUI	3	Variant management shall seamlessly integrate itself into the development tools by providing a user interface.
NOT(CanIntegrateUI)	3.1	The tool integration's user interface shall be embedded in the development tool's user interface.
CanAttachContextMenu	3.2	The tool integration's user interface shall be displayed in its own window, which can be triggered from the development tool's user interface.
NOT(CanAttachContextMenu)	3.2.1	The tool integration's user interface shall be brought to front when the user triggers a variant-management action from within the development tool.
CanIntegrateUI	3.2.2	The tool integration's user interface shall stay in front of the development-tool window.
CalculationsSupported	4	Development tools shall support the user during the creating of variation points.
	4.1	The tool integration shall provide editors for different kinds of pvSCL rules.
	4.1.1	All pvSCL editors shall provide autocompletion based on the loaded models.
SupportsKeywords	4.1.1.1	The autocompletion of all pvSCL editors shall show all possible pvSCL keywords.
SupportsFeatures	4.1.1.2	The autocompletion of all pvSCL editors shall show all features of the loaded models.
SupportsAttributes	4.1.1.3	The autocompletion of all pvSCL editors shall show all attributes of the entered feature.
SupportsComponents	4.1.1.4	The autocompletion of all pvSCL editors shall show all components of the loaded models.
	4.1.2	All pvSCL editors shall provide pvSCL compliant syntax highlighting.
	4.1.3	All pvSCL editors shall check the entered rule for errors before writing pvSCL rules to variation points.
LoadPVModels	4.2	The tool integration shall allow loading and displaying feature models and variant result models.
SaveModels	4.2.1	Loaded models shall be saved and reloaded when opening a master artifact model.
Quickload	4.2.2	The last five opened models should be quickly accessible.
	4.2.3	Feature models and variant result models shall show all elements needed for entering pvSCL rules.
EditRestriction	4.3	The tool integration shall provide an editor for writing restrictions.
AddMultipleRestrictions	4.3.1	It shall be possible to add multiple restrictions at once.
EditMultipleRestrictions	4.3.2	It shall be possible to edit multiple restrictions at once.
	4.3.3	The restriction editor shall be easily accessible.
CanAttachContextMenu	4.3.3.1	The restriction editor shall be accessible through the context menu of each artifact.
NOT(CanAttachContextMenu) AND CanAttachButton	4.3.3.2	The restriction editor shall be accessible through a button in the integration's user interface.
NOT(CanAttachContextMenu) AND NOT(CanAttachButton)	4.3.3.3	The restriction editor shall be accessible through a menu of the development tool.
	4.3.3.4	Triggering the restriction editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
EditCalculation	4.4	The tool integration shall provide an editor for writing calculations.
AnnotateSelection	4.4.1	The calculation editor shall be similar to the restriction editor.
AnnotateWithMarkers	4.4.2	The calculation editor shall support users in editing all calculations of the selected element.
	4.4.3	The calculation editor shall be easily accessible.
CanAttachContextMenu	4.4.3.1	The calculation editor shall be accessible through the context menu of each element.
NOT(CanAttachContextMenu) AND CanAttachButton	4.4.3.2	The calculation editor shall be accessible through a button in the integration's user interface.
NOT(CanAttachContextMenu) AND NOT(CanAttachButton)	4.4.3.3	The restriction editor shall be accessible through a menu of the development tool.
	4.4.3.4	Triggering the calculation editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
EditConstraint	4.5	The tool integration shall provide an editor for writing constraints.
DeleteCalculation OR DeleteRestriction OR DeleteConstraint	5	The tool integration shall support users in deleting variation points.
	6	Variant management shall support the visualizing of variability-affected model elements.
	6.1	Variation points shall be visibly different from other elements of the master artifact model.
HighlightRestrictions	6.1.1	The tool integration shall provide a means to highlight restrictions.
HighlightCalculations	6.1.2	The tool integration shall provide a means to highlight calculations.
HighlightConstraints	6.1.3	The tool integration shall provide a means to highlight constraints.
VariationPointSearch	6.2	The tool integration shall provide a means to quickly find all model elements related to a specific variation point.
ErrorVisualization	7	The tool integration shall support users in finding variation-point errors.
	7.1	The tool integration shall provide a means to highlight pvSCL syntax errors and unknown names.
	7.2	When searching variation-point errors, users shall be able to distinguish between syntax errors and unknown names.
SeparateErrorVisualizations	7.2.1	Syntax errors and unknown names shall be distinguished by different colors.
OneErrorVisualization	7.2.2	Syntax errors and unknown names shall be distinguished by icons.
	7.3	The tool integration shall provide a means to navigate between errors.
Preview	8	Variant management shall support the previewing of variants.
	8.1	The tool integration shall provide a preview that is faster than a transformation of the same master artifact model.
PreviewGrayOut	8.2	The tool integration shall provide a preview that grays out elements not included in the variant.
PreviewHide	8.3	The tool integration shall provide a preview that hides elements not included in the variant.
PreviewHighlight	8.4	The tool integration shall provide a preview that highlights all variability-affected elements that are included in the variant.
CalculationsSupported	8.5	Preview visualizations shall replace calculation values.

Table B.1: An excerpt of the requirements master artifact model. Only the requirements ID, title and related restrictions are displayed.

ID: 1 (6.3.2)	Variant management shall always be done in the same way for the user independent of the used development tool.
Description	If a developer or a maintainer works with different tools during the development, such as DOORS for requirements and Matlab/Simulink for model-based software development, variant management will be always done and used in the same manner in these tools.
Rationale	If the requirement can be fulfilled, the overall learning effort decreases because just a single method has to be known, leading usually to fewer errors during the development.
Use Case	QT3-UC-1
Source	SPESXT
Acceptance Criterion	All subrequirements are fulfilled

ID: 1.1	Tool integrations shall use the same or similar icons for the same functions.
Description	Icons used for the same actions in all user interfaces should be the same, or at least similar if technical constraints prevent this. Technical constraints can be, for example, differences in icon size, which lead to different levels of detail.
Rationale	Necessary to fulfill Requirement 1
Use Case	QT3-UC-1
Possible Realization	Provide an icon set in a base project to which all integrations refer.
Source	Usability
Acceptance Criterion	All icons of the same size with the same meaning are equal. If the size differs, they have the same content with a different degree of detail.

ID: 1.2	Tool integrations shall communicate similar messages in similar situations.
Description	Except for tool-specific messages, information, error, and warning dialogs should show the same messages in the same situations.
Rationale	Necessary to fulfill Requirement 1
Use Case	QT3-UC-1
Possible Realization	Provide a class with common messages in a base project to which all integrations refer.
Source	Usability
Acceptance Criterion	Messages with the same meaning are equal.

ID: 1.3	The integrations' user-interface design shall be similar for all integrations.
Description	The user interface's layout should be similar in all integrations wherever it is possible.
Rationale	Necessary to fulfill Requirement 1
Use Case	QT3-UC-1
Possible Realization	Implement a base user interface that can be integrated in most external tools and that integrations can extend with tool-specific content.
Source	Usability
Acceptance Criterion	The base user interface is the same (has the same look) in all integrations. Tool-specific buttons are added always in the same location. User interface elements outside of the base user interface are added in a similar location as in other integrations.

ID: 2	The tool integration shall comply with usability heuristics.
Description	The tool integration should fulfill rules for achieving good user experience.
Source	Usability
Acceptance Criterion	All subrequirements are fulfilled

ID: 2.1	The tool integration shall display a wait cursor for actions that take longer than two seconds.
Description	For actions that usually take longer than two seconds, the integration should display a wait cursor.
Rationale	It is necessary to give users feedback during time-consuming actions, so they know the delay is expected behavior (usability heuristic *visibility of system status*).
Source	Usability

ID: 2.2	Tool integrations shall provide progress bars for actions that take longer than ten seconds.
Description	For actions that usually take longer than 10 seconds, the integration should show a progress bar.
Rationale	It is necessary to give users feedback during time-consuming actions, so they know the delay is expected behavior (usability heuristic *visibility of system status*).
Source	Usability

ID: 2.3	Tool integrations shall be consistent with pure::variants and the extended tool.
Description	The terms used in messages and labels, and the icons should be consistent with pure::variants and the extended tool: For all variant-management actions, the tool integrations should use pure::variants terms and (if possible) icons, because users are already familiar with them. When referring to elements of the extended tool, the tool integration should use the relevant terms of the extended tool's user interface.
Rationale	Users should be able to understand messages, labels, and icons quickly. This is easier, when they already know the relevant terms (usability heuristic *match between system and real world* and *consistency and standards*).
Source	Usability
Acceptance Criterion	Terms for all concepts that also exist in pure::variants are the same as in pure::variants. When referring to elements of the master artifact model, the respective term of the extended tool is used.

ID: 2.4	Users shall be able to interrupt all actions that take longer than 10 seconds.
Description	If an action usually takes longer than 10 seconds, users should be able to interrupt the action.
Rationale	When a time-consuming action was triggered by mistake, it may be annoying to wait for the action to finish. To improve efficiency and user satisfaction, it should be possible to interrupt the action (usability heuristic *user control and freedom*).
Possible Realization	Provide a cancel button (labeled 'x') next to the progress bar, which appears only when the progress bar is active.
Source	Usability

ID: 2.5	For all actions that change the master artifact model, the tool integration shall provide a means to undo or redo the action.
Description	The integration should take care of providing undo or redo for actions that change the master artifact model. To trigger an undo or redo action the development tool's undo/redo trigger should be used.
Rationale	In several development tools, automatically changing elements causes undo/redo to stop working, or undo/redo only works for actions done by the user, but not for changes applied by the integration. If it is possible, the integration should repair this behavior, such that users can undo/redo actions they triggered themselves and actions triggered by the integration. This requirement is necessary for satisfying usability heuristic *user control and freedom*.
Source	Usability

ID: 2.6	Users shall always be able to cancel a dialog.
Description	Users should be able to close a dialog by pressing the 'x' in the upper right corner or the 'cancel' button in the lower right corner. No changes should be applied when canceling a dialog.
Rationale	Users should always feel in control of the system they are using (usability heuristic *user control and freedom*)
Source	Usability
Acceptance Criterion	All dialogs can be closed, and all changes are discarded when closing a dialog no using the 'OK'-button.

ID: 2.7	Tool integrations shall warn users before performing an irreversible action.
Description	When a user triggers an irreversible action, such as overwriting a document, the tool integration should display a warning.
Rationale	By warning users, errors can be prevented (usability heuristic *error prevention*)
Source	Usability

ID: 2.8	Tool integrations shall always support users in entering data.
Description	When entering data, such as a pvSCL rule or a file path, the tool integration should always support users.
Rationale	Users are much better at recognizing something than at remembering it without help. Thus, the tool integration should provide UI elements that let users choose from a list of possible items, or let users modify data rather than entering it completely from scratch (usability heuristic *recognition rather than recall*).
Possible Realization	For opening documents or specifying directories, provide the respective dialog of the operating system. For entering pvSCL rules, provide auto-completion.
Source	Usability
Acceptance Criterion	For entering file or directory locations, the tool integration shows standard open file/directory dialogs. For all other data, the tool integration provides a list of possible values, or a default value is already entered.

ID: 2.9	The tool integration's user interface and visualizations shall be kept as simple as possible.
Description	All parts of the user interface and its visualizations should be as simple as possible, since the number of information pieces processed in short-term memory is limited to seven plus or minus two units. For example, visualizations should not use more than five colors.
Rationale	Necessary to satisfy usability heuristic *aesthetic and minimalistic design.*
Source	Usability
Acceptance Criterion	Visualizations do not use more than five colors.

ID: 2.10	If an error occurs, the tool integration shall provide simple, constructive, and specific instructions for recovery.
Description	Error and warning messages should be understandable and helpful. For example, when a certain document or element is concerned, its name should be mentioned in the warning. Furthermore, the message should give instructions for solving the problem.
Rationale	Good messages support user satisfaction. Necessary to satisfy usability heuristic *error recognition and recall.*
Source	Usability
Acceptance Criterion	All error messages mention the name of the concerned element, and give instructions for solving the problem.

ID: 3 (6.3.4)	Variant management shall seamlessly integrate itself into the development tools by providing a user interface.
Description	Seamless integration is meant in the sense that the developer is not just able to get access to variability information directly within its used environment but also to interact with the variant management for example the developer is guided during the specification of variant restrictions and those restrictions are also validated against the variability model.
Rationale	Providing a seamless integration of variant management with development tools a) eases the usage, b) decreases the risk of a wrong usage, and c) usually fosters the usage. Seamless integration usually also increases the acceptance through the user, because she has not to switch between tools anymore to get her job right.
Use Case	QT3-UC-1
Source	SPESXT
Acceptance Criterion	All subrequirements are fulfilled

ID: 3.1	The tool integration's user interface shall be embedded in the development tool's user interface.
Description	If possible, the integration's user interface should be part of the development tool's user interface.
Rationale	When the user interface is directly integrated into the development tool's user interface the user does not need to manage multiple windows.
Use Case	QT3-UC-1
Possible Realization	Use a tool-specific integration mechanism.
Acceptance Criterion	The tool integration is embedded in the user interface

ID: 3.2	The tool integration's user interface shall be displayed in its own window, which can be triggered from the development tool's user interface.
Description	If the extended tool does not support user-interface integration, the tool integration should provide its own window, which users can trigger from the development tool's user interface.
Rationale	Not all tools support integration into their user interface. Providing a stand-alone user interface is an alternative.
Use Case	QT3-UC-1
Source	Experience with IBM Rational Rhapsody Integration
Acceptance Criterion	The tool integration can be triggered from within the development tool. A new window opens, that contains the user interface of the tool integration.

ID: 3.2.1	The tool integration's user interface shall be brought to front when the user triggers a variant-management action from within the development tool.
Description	Actions, such as triggering a pvSCL Editor should be done from within the development tool. However, when users work with the development tool, it is brought to front and hides the tool integration. Therefore, the tool integration's user interface needs to be brought to front programmatically whenever it is helpful (E.g., when editing restrictions or calculations).
Rationale	Managing the tool-integration window and the development tool's window should be as simple as possible.
Use Case	QT3-UC-1
Possible Realization	Use a tool-specific mechanism to embed custom context menus.
Acceptance Criterion	The tool integration is called to front when users open the pvSCL editor or execute any other action that users trigger from within the development tool.

ID: 3.2.2	The tool integration's user interface shall stay in front of the development-tool window.
Description	All actions of the tool integration shall be done from the tool-integration window. However, when users work with the development tool, it is brought to front and hides the tool integration. Therefore, the tool-integration window shall always be displayed in front of the development tool. Or a toggle button on the tool-integration window shall control whether the window stays in front or acts regularly.
Rationale	Managing the tool-integration window and the development tool's window should be as simple as possible.
Use Case	QT3-UC-1
Acceptance Criterion	The tool integration always stays in front of the development tool, or it can be set to stay in front using a toggle button.

ID: 4 (6.3.8)	Development tools shall support the user during the creating of variation points.
Description	Development tools shall support the user during the creating/instantiating of variation points by either choosing the appropriate variant handling mechanism automatically if possible or propose different variant handling mechanisms to the user.
Rationale	Due to the fact that the variability handling mechanisms are different depending on the used language, tool, DSL, and so on, the user can start working directly also regarding variability without learning or knowing the different variability handling concepts upfront.
Use Case	QT3-UC-1
Possible Realization	Depends on the used language (IDE), tool, DSL, and so on.
Source	SPESXT
Acceptance Criterion	All subrequirements fulfilled

ID: 4.1	The tool integration shall provide editors for different kinds of pvSCL rules.
Description	Users should be able to write different kinds of variation points, such as restrictions or calculations, using an editor that enables editing of pvSCL rules in an efficient and error-preventing way.
Rationale	Necessary to fulfill Requirement 4
Use Case	QT3-UC-1, Use Case 1
Acceptance Criterion	All subrequirements are fulfilled

ID: 4.1.1	All pvSCL editors shall provide autocompletion based on the loaded models.
Description	Users shall be able to use a similar autocompletion mechanism as provided by the pure::variants pvSCL editors. The basic functions should be the same: Users should be able to press CTRL + space to show a list of possible words to enter. The list should update itself based on the entered text, containing only those words that start with the entered text. When double-clicking a list entry or pressing 'Enter' the selected word should be inserted at the caret position. Each entry should look like autocompletion entries in the pure::variants pvSCL editors (icon unique name – visible name).
Rationale	Without autocompletion, users would need to switch back to pure::variants to check which feature names to enter. Furthermore it would be inconsistent with the pure::variants constraint and restriction editor.
Use Case	QT3-UC-1, Use Case 1
Source	Usability, Consistency with pure::variants
Acceptance Criterion	See description

ID: 4.1.1.1	The autocompletion of all pvSCL editors shall show all possible pvSCL keywords.
Description	The autocompletion should show a list of all possible pvSCL keywords.
Rationale	pvSCL rules can contain keywords. Therefore, autocompletion must support them.
Use Case	QT3-UC-1, Use Case 1
Source	Consistency with pure::variants

ID: 4.1.1.2	The autocompletion of all pvSCL editors shall show all features of the loaded models.
Description	The autocompletion should show a list of all features of the loaded models that start with the entered text.
Rationale	Almost every pvSCL rule contains features. Therefore, autocompletion must support them.
Use Case	QT3-UC-1, Use Case 1
Source	Consistency with pure::variants

ID: 4.1.1.3	The autocompletion of all pvSCL editors shall show all attributes of the entered feature.
Description	The autocompletion should show a list of all attributes of a feature, if the user entered something like 'featurname->'.
Rationale	All pvSCL rules can contain attributes. Especially in calculations attributes are referenced. Therefore, it is necessary to support them.
Use Case	QT3-UC-1, Use Case 1
Source	Consistency with pure::variants

ID: 4.1.1.4	The autocompletion of all pvSCL editors shall show all components of the loaded models.
Description	The autocompletion should show a list of all components of the loaded models that start with the entered text.
Rationale	pvSCL rules can contain components. To support the complete functionality of pvSCL rules, it should be possible to select components from the autocompletion list.
Use Case	QT3-UC-1, Use Case 1
Source	Consistency with pure::variants

ID: 4.1.2	All pvSCL editors shall provide pvSCL compliant syntax highlighting.
Description	To understand pvSCL rules intuitively, the integrations' restriction editor shall provide the same syntax highlighting as the pure::variants constraint and restriction editor.
Rationale	Syntax highlighting is necessary for a good understanding of pvSCL rules and consistency with pure::variants.
Use Case	QT3-UC-1, Use Case 1
Source	Usability, Consistency with pure::variants
Acceptance Criterion	Keywords, comments, and numbers are highlighted in the same way as in the pure::variants pvSCL editor.

ID: 4.1.3	All pvSCL editors shall check the entered rule for errors before writing pvSCL rules to variation points.
Description	To prevent faulty pvSCL rules, all pvSCL editors should check the entered rule for pvSCL syntax errors and unknown feature or attribute names. This should happen during typing and when pressing the OK button. During typing, syntax errors should be communicated in the title bar and unknown feature names should be underlined. When an error is found after pressing the OK button, it should be communicated in a warning dialog that specifies the type of error and lets the user choose to ignore or correct the error.
Rationale	Error check during rule entry prevents errors before they are entered.
Use Case	QT3-UC-1, Use Case 1
Source	Usability (Error Prevention)
Acceptance Criterion	See description

ID: 4.2	The tool integration shall allow loading and displaying feature models and variant result models.
Description	The integration should provide a means to load feature models and variant description models. The models should be displayed in a tree that resembles the pure::variants model tree. Especially displaying pure::variants features and attributes is important, because they are needed when editing pvSCL rules.
Rationale	Besides the visual representation, the integration needs feature models and variant description models as input data for autocompletion, preview, and error visualization.
Use Case	QT3-UC-1, Use Case 1, Use Case 3
Source	Usability

ID: 4.2.1	Loaded models shall be saved and reloaded when opening a master artifact model.
Description	The tool integration should save a loaded pure::variants model for the master artifact model. If possible, it should be saved inside the master-artifact-model file. When opening the master artifact model, the tool integration should reopen the saved pure::variants model.
Rationale	Having to search for the same pure::variants model every time when opening a master artifact model is frustrating. Hence, the tool integration should save and reload models.
Use Case	QT3-UC-1, Use Case 1, Use Case 3
Source	Usability

ID: 4.2.2	The last five opened models shall be quickly accessible.
Description	The tool integration should save a last 5 loaded pure::variants model for the current master artifact model. If possible, they should be saved inside the master-artifact-model file. Users should be able to select each model from a list to reload it.
Rationale	For previewing variants, users may want to switch between variant result models. Choosing the model each time from a file open dialog may be frustrating.
Use Case	QT3-UC-1, Use Case 1, Use Case 3
Source	Usability

ID: 4.2.3	Feature models and variant result models shall show all elements needed for entering pvSCL rules.
Description	Variant result models or feature models should show all elements that can be part of a pvSCL rule. Currently, this includes features and attributes.
Rationale	The use of showing a feature or variant result model in the development tool is that users do not need to switch back to pure::variants to view the relevant model. When writing a pvSCL rule, users may want to refer to the current pure::variants model. This is only useful if users can view all elements that can be part of a pvSCL rule. Otherwise, they would need to switch back to pure::variants, and view the full model representation.
Use Case	QT3-UC-1, Use Case 1, Use Case 3
Source	Usability
Acceptance Criterion	The visual representation of feature models and variant result models includes all elements of the respective model that can be part of a pvSCL rule (e.g., features and attributes)

ID: 4.3	The tool integration shall provide an editor for writing restrictions.
Description	Users should be able to write restrictions using a pvSCL editor that complies with Requirements 4.1.1 - 4.1.3. When they press OK, the tool integration should write the entered rule to the variation point. Depending on the extended tool, the rule can be stored either in the master artifact model, using a specific type of artifact, or outside, using a file that maps pvSCL rules to variable artifacts. Either way, the user should notice no different behavior of tool integrations.
Rationale	If the transformation for the development tool supports restrictions, it should be possible to edit them in an efficient and error-preventing way.
Use Case	QT3-UC-1, Use Case 1
Acceptance Criterion	It is possible to write restrictions using a pvSCL editor, and the subrequirements are fulfilled.

ID: 4.3.1	It shall be possible to add multiple restrictions at once.
Description	Users should be able to select multiple elements and add a new variation point containing the same restriction to each of the elements.
Rationale	Variation points of a master artifact model may often be the same for multiple elements. Having to enter the same rule over and over again would be time-consuming and unnecessary. Also the repeated task might frustrate users. Adding multiple variation points at once may be especially important when the development tool does not support adding one variation point to multiple elements.
Use Case	QT3-UC-1, Use Case 2
Acceptance Criterion	After selecting multiple elements, editing the pvSCL rule and pressing OK, each selected element is annotated with a variation point that contains the entered rule.

ID: 4.3.2	It shall be possible to edit multiple restrictions at once.
Description	When the user selects multiple variation points and triggers the restriction editor, the behavior should be: 1. If no pvSCL rules are defined: Open the restriction editor. Initially, the shown rule is empty. 2. If pvSCL rules are defined, but they are different: Open the restriction editor. Initially, the shown rule is empty. The tool integration warns the user that selected restrictions will be overwritten when pressing OK (the warning dialog should provide a 'Do not show this dialog again' option). 3. If pvSCL rules are defined and they are equal: Open the restriction editor and initially show the pvSCL rule. When the user presses OK, the entered rule is written to all selected elements.
Rationale	Variation points of a master artifact model may often be the same for multiple elements. Having to enter the same rule over and over again would be time-consuming and unnecessary. Also the repeated task might frustrate users. However, users should be aware that they edit multiple restrictions.
Use Case	QT3-UC-1, Use Case 2
Acceptance Criterion	See description

ID: 4.3.3	The restriction editor shall be easily accessible.
Description	Because users need to frequently trigger the restriction editor, accessing the editor should require a simple action.
Use Case	QT3-UC-1, Use Case 1
Source	Usability

ID: 4.3.3.1	The restriction editor shall be accessible through the context menu of each artifact.
Description	If it is possible to add a custom context menu to artifacts, such a menu should be used to trigger the restriction editor.
Rationale	Integration into the context menu of artifacts has the advantage that it is clear to which element the variation point will be attached. Furthermore, users have to execute only one action to trigger the editor.
Use Case	QT3-UC-1, Use Case 1

ID: 4.3.3.2	The restriction editor shall be accessible through a button in the integration's user interface.
Description	Users should trigger the restriction editor for a specific element by selecting the element and pressing a button on the tool integration's user interface.
Rationale	If it is not possible to add a context menu to artifacts, selecting the element and using the button is a good alternative, which should be realizable in most tools.
Use Case	QT3-UC-1, Use Case 1

ID: 4.3.3.3	The restriction editor shall be accessible through a menu of the development tool.
Description	Users should trigger the restriction editor from the main menu of the development tool. The menu entry should be located under an entry 'pure::variants'. When the user selects the entry, the tool integration shows an editor for the currently selected artifacts.
Rationale	If the tool integration's user interface cannot be embedded, we need to trigger pvSCL editors from the development tool, so we can call the tool integration window to front. If it is impossible to add a context menu to artifacts, this requirement is a good alternative.
Use Case	QT3-UC-1, Use Case 1

ID: 4.3.3.4	Triggering the restriction editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
Description	It should be possible to reaccess the restriction editor for a specific element using the same or a similar method as for adding a restriction (specified either in Requirement 4.3.3.1 or 4.3.3.2). When reopening the editor, it should show the defined rule.
Rationale	Users only have to learn one action if the same action is necessary for creating and reaccessing variation points.
Use Case	QT3-UC-1, Use Case 1
Source	Usability

ID: 4.4	The tool integration shall provide an editor for writing calculations.
Description	Users should be able to write calculations using a pvSCL editor that complies with Requirements 4.1.1 - 4.1.3. When they press OK, the tool integration should write the entered rule to the variation point. Depending on the extended tool, the rule can be stored either in the master artifact model, using a specific type of artifact, or outside, using a file that maps pvSCL rules to variable artifacts. Either way, the user should notice no different behavior of tool integrations.
Rationale	If the transformation for the development tool supports calculations, it should be possible to edit them in an efficient and error-preventing way.
Use Case	QT3-UC-1, Use Case 3
Source	Customer request for calculations
Acceptance Criterion	All subrequirements are fulfilled

ID: 4.4.1	The calculation editor shall be similar to the restriction editor.
Description	If calculations are added directly to a text selection, the calculation editor should be similar to the restriction editor. The only differences should be the autocompletion keywords and the error check (restrictions are only valid if the result is a boolean value, calculations can return different types).
Rationale	For consistency users should be able to edit calculations in the same way as they edit other pvSCL rules.
Use Case	QT3-UC-1, Use Case 3
Acceptance Criterion	The calculation editor is based on the same class as the restriction editor. Only the error check is different.

ID: 4.4.2	The calculation editor shall support users in editing all calculations of the selected element[1].
Description	If calculations cannot be added to selected text, markers in the text are used to identify calculations. The following markers should be used per default: • Start of a calculation: '[' • End of a calculation: ']' • Escape character: '$' For example, "...[featurename->attributename] ..." is transformed to "...attributevalue ...". To still be able to use a pvSCL editor, it should be possible to trigger a calculation editor for an artifact that contains a calculation. The editor should show a selection of all calculations found in the element. It should be possible to select a calculation and edit it using a pvSCL editor that complies with Requirements 4.1.1 – 4.1.3. When the user presses OK, the element's text should be replaced so that the calculations are updated.
Rationale	In some tools it is not possible to attach a variation point directly to a text fragment. In this case, users cannot trigger a calculation editor for a single calculation. Therefore, the selection of the calculation to edit should happen in the editor.
Use Case	QT3-UC-1, Use Case 3
Possible Realization	Provide a list of all calculation rules contained in the selected element's text. Below, the currently selected rule can be edited using a standard pvSCL editor field. While typing, the list of calculation rules is updated. When the user closes the editor successfully, the element's text should be replaced with the edited text.

ID: 4.4.3	The calculation editor shall be easily accessible.
Description	Because users need to frequently trigger the calculation editor, accessing the editor should require a simple action.
Use Case	QT3-UC-1, Use Case 3
Source	Usability

[1]In the original requirements document, we use calculations to set the marker characters, so that the used characters can differ between tool integrations. However, due to formatting reasons, we do not show the calculations in this thesis.

ID: 4.4.3.1	The calculation editor shall be accessible through the context menu of each element.
Description	If it is possible to add a custom context menu to artifacts, such a menu should be used to trigger the calculation editor.
Rationale	Integration into the context menu of artifacts has the advantage that it is clear to which element the variation point will be attached. Furthermore, users have to execute only one action to trigger the editor.
Use Case	QT3-UC-1, Use Case 3

ID: 4.4.3.2	The calculation editor shall be accessible through a button in the integration's user interface.
Description	Users should trigger the calculation editor for a specific element by selecting the element and pressing a button on the tool integration's user interface.
Rationale	If it is not possible to add a context menu to artifacts, selecting the element and using the button is a good alternative, which should be realizable in most tools.
Use Case	QT3-UC-1, Use Case 3

ID: 4.4.3.3	The restriction editor shall be accessible through a menu of the development tool.
Description	Users should trigger the calculation editor from the main menu of the development tool. The menu entry should be located under an entry 'pure::variants'. When the user selects the entry, the tool integration shows an editor for the currently selected artifacts.
Rationale	If the tool integration's user interface cannot be embedded, we need to trigger pvSCL editors from the development tool, so we can call the tool integration window to front. If it is impossible to add a context menu to artifacts, this requirement is a good alternative.
Use Case	QT3-UC-1, Use Case 3

ID: 4.4.3.4	Triggering the calculation editor for an already defined variation point shall be possible in the same or a similar way as adding a new one.
Description	It should be possible to reaccess the calculation editor for a specific element using the same or a similar method as for adding a calculation (specified either in Requirement 4.4.3.1 or 4.4.3.2). When reopening the editor, it should show the defined rule.
Rationale	Users only have to learn one action if the same action is necessary for creating and reaccessing variation points.
Use Case	QT3-UC-1, Use Case 3
Source	Usability

ID: 4.5	The tool integration shall provide an editor for writing constraints.
Description	Users should be able to write constraints using an editor that enables editing of pvSCL rules in an efficient and error-preventing way. Using the editor and accessing it should happen in the same way as for the restriction editor. However, users should always be able to distinguish whether they are editing a constraint or a restriction. Thus, labels and menus should be different.
Rationale	If the transformation for the extended tool supports constraints, it should be possible to edit them using a pvSCL editor.
Use Case	QT3-UC-1
Possible Realization	Use the same code base for restriction and constraint editor.
Acceptance Criterion	The same implementation is used for restriction and constraint editor. Only the dialog title, labels, and menu entries refer to the constraint editor.

ID: 5	The tool integration shall support users in deleting variation points.
Description	If the development tool does not support a simple option to delete a variation point, the tool integration should provide such an option. It should be consistent with the action to add or edit a variation point.
Rationale	The tool integration eases the way of entering variation points. Therefore, it should also be easy to delete them.
Use Case	QT3-UC-1, Use Case 1
Possible Realization	If, for example, adding and editing variation points is possible through an element's context menu, deleting a variation point should also be possible through such a context menu (see Enterprise Architect Integration).

ID: 6 (6.3.5)	Variant management shall support the visualization of variability-affected model elements.
Description	The variant management is in charge for giving the information what elements belong to what variability information. That means if a user wants to know what elements within a given model are related to certain variability information and the tool is able to visualize them accordingly, the exploration will be much easier than without such capability.
Rationale	E.g. visualizing variability-affected model elements is helpful a) to get an overview and b) to analyze whether all relevant model elements are referring to the correct variability information.
Use Case	QT3-UC-1
Source	SPESXT
Acceptance Criterion	All subrequirements are fulfilled

ID: 6.1	Variation points shall be visibly different from other elements of the master artifact model.
Description	To identify which elements contain variability, these elements should be visibly different from other elements. It should be possible to identify variation points at first glance (preattentively). If this requirement is not met per default by the development tool, the integration should provide a solution for this.
Rationale	When editing model elements, it is important to know which elements already contain variability. If these elements cannot be identified preattentively, users unnecessarily spend time searching for already defined variation points.
Use Case	QT3-UC-1
Possible Realization	Either choose a variation-point element type that is visibly different from other elements, or provide a means to highlight the elements
Source	Usability
Acceptance Criterion	Users are able to look at part of the master artifact model and identify all variation points at first glance.

ID: 6.1.1	The tool integration shall provide a means to highlight restrictions.
Description	Since restrictions and the related artifacts are not visibly different from other elements, the integration needs to provide a means to highlight restrictions and the related elements.
Rationale	Necessary for fulfilling Requirement 6.1
Use Case	QT3-UC-1
Possible Realization	Provide a toggle button that highlights all restrictions and related elements when selected.
Acceptance Criterion	Restrictions can be or are always highlighted.

ID: 6.1.2	The tool integration shall provide a means to highlight calculations.
Description	Since calculations and the related artifacts are not visibly different from other elements, the integration needs to provide a means to highlight calculations and the related elements.
Rationale	Necessary for fulfilling Requirement 6.1
Use Case	QT3-UC-1
Possible Realization	For example, provide a toggle button that highlights all calculations and related elements when selected.
Acceptance Criterion	Calculations can be or are always highlighted.

ID: 6.1.3	The tool integration shall provide a means to highlight constraints.
Description	Since constraints and the related artifacts are not visibly different from other elements, the integration needs to provide a means to highlight constraints and the related elements.
Rationale	Necessary for fulfilling Requirement 6.1
Use Case	QT3-UC-1
Possible Realization	For example, provide a toggle button that highlights all constraints and related elements when selected.
Acceptance Criterion	Constraints can be or are always highlighted.

ID: 6.2	The tool integration shall provide a means to quickly find all model elements related to a specific variation point.
Description	The tool integration could, for example, provide a list of all variation points. When clicking on a variation point, all related elements should be highlighted and users should be able to navigate between variation points. This could be further refined by a filter function, so that only those rules were shown that contain a search string specified by the user.
Rationale	The described functions increase the speed with which users can edit the variability of a master artifact model, since they do not need to search for existing rules manually. Furthermore, it is useful when analyzing whether all relevant model elements are referring to the correct variability information.
Use Case	QT3-UC-1
Acceptance Criterion	The integration provides a view that shows all variation points of the master artifact model. Clicking on a link entry highlights all elements that are annotated with this variation point.

ID: 7	The tool integration shall support users in finding variation-point errors.
Description	Variation-point errors that can be detected automatically include pvSCL syntax errors and unknown feature names. The integration should help users in finding these errors.
Rationale	Helping users to find errors in variation points increases the efficiency of the error finding process and leads to less errors in master artifact models.
Use Case	QT3-UC-1, Use Case 4
Source	[Pap11]
Acceptance Criterion	All subrequirements are fulfilled

ID: 7.1	The tool integration shall provide a means to highlight pvSCL syntax errors and unknown names.
Description	The tool integration should provide visualizations that highlight pvSCL syntax errors and variation points in which unknown feature names or attribute names are referenced.
Rationale	Using visualizations, users do not have to search for errors, but can identify the respective elements preattentively.
Use Case	QT3-UC-1, Use Case 4

ID: 7.2	When searching variation-point errors, users shall be able to distinguish between syntax errors and unknown names.
Description	Distinguishing syntax errors from unknown feature or attribute names is important, because both errors have different consequences.
Rationale	For example, pvSCL syntax errors have to be always corrected in the variation-point definition. However, an unknown feature or attribute name can also result from an error in the feature-model definition.
Use Case	QT3-UC-1, Use Case 4
Acceptance Criterion	

ID: 7.2.1	Syntax errors and unknown names shall be distinguished by different colors.
Description	The integration should provide two different visualizations. The syntax error visualization highlights all variation points containing syntax errors in red. The semantic error visualization highlights all variation points containing unknown feature or attribute names in yellow.
Rationale	Necessary to fulfill Requirement 7.2
Use Case	QT3-UC-1, Use Case 4

ID: 7.2.2	Syntax errors and unknown names shall be distinguished by icons.
Description	Syntax errors and unknown feature or attribute names should be distinguished by icons. Users will need to trigger only one error visualization to show both types of errors. During visualization, an Eclipse error icon should be visibly attached to elements containing syntax errors, and an Eclipse warning icon should be shown for variation points in which unknown names are referenced.
Rationale	pure::variants users already know Eclipse's error and warning icons.
Use Case	QT3-UC-1, Use Case 4
Possible Realization	In column- and row-based development tools an extra column could be added that displays error or warning icons.
Source	Consistency with pure::variants
Acceptance Criterion	See Description

ID: 7.3	The tool integration shall provide a means to navigate between errors.
Description	When errors are highlighted the tool integration should show previous and next buttons that enable users to jump between errors.
Rationale	In large models, not all faulty elements are visible in the current viewport. Therefore, users need a means to jump to the next error.
Use Case	QT3-UC-1, Use Case 4
Acceptance Criterion	When error visualization is enabled, a previous and next button is also enabled. Using the buttons, users can jump to the next/previous faulty element of the entire master artifact model.

ID: 8 (6.3.6)	Variant management shall support the previewing of variants.
Description	The variant management is in charge of giving the information what elements belong to a variant to allow tools visualizing them perhaps by highlighting them. Another possibility would be graying out those elements that are not part of a concrete variant. This functionality helps the user to get an impression of how that variant will look like when finalized.
Rationale	Previewing means visualizing those elements that belong to a variant and which do not.
Use Case	QT3-UC-1
Source	SPESXT
Acceptance Criterion	All subrequirements are fulfilled

ID: 8.1	The tool integration shall provide a preview that is faster than a transformation of the same master artifact model.
Description	A preview should always be faster than a transformation of the same master artifact model.
Rationale	The main value of a preview visualization is that no time-consuming transformation is necessary. Therefore, it should be faster than the transformation under comparable conditions.
Use Case	QT3-UC-1, Use Case 5
Acceptance Criterion	The preview of a master artifact model is faster than the transformation of the same model. The test model should encompass the different functionalities of the development tool (e.g., in text processing tool do not only use text, but also embed pictures, use tables of contents, etc.).

ID: 8.2	The tool integration shall provide a preview that grays out elements not included in the variant.
Description	The tool integration should provide a visualization that grays out all elements that would not be included in a variant. To ensure readability, the text of a grayed-out element should be darker than its background color. The exact coloring should be adapted based on the default coloring of elements. If a color already has another meaning, an alternative color should be used.
Rationale	Graying out elements has the advantage that elements are still visible, but do not attract as much attention as other elements. Thus, verifying that variants do not contain falsely excluded elements is easier than in a real variant.
Use Case	QT3-UC-1, Use Case 5
Source	[Pap11]
Acceptance Criterion	Users can trigger a visualization that causes all elements not included in the variant to be colored in shades of gray. The text of each element is still readable, and each element can still be distinguished from the background.

ID: 8.3	The tool integration shall provide a preview that hides elements not included in the variant.
Description	The tool integration should provide a visualization that hides all elements that would not be included in a variant.
Rationale	For getting an overview of a variant it may be better to hide elements that are not included in a variant. Furthermore, it may not be possible to color elements in some development tools.
Use Case	QT3-UC-1, Use Case 5
Acceptance Criterion	Users can trigger a visualization that hides elements not included in the variant.

ID: 8.4	The tool integration shall provide a preview that highlights all variability-affected elements that are included in the variant.
Description	The tool integration should provide a visualization that highlights all elements that are affected by a variation point and would be contained in a variant (based on the currently loaded variant description model). To ensure readability, the text of a highlighted element should be darker than its background color. Highlight color green should be preferred. If it already has another meaning, an alternative color should be used. The exact coloring should be adapted based on the default coloring of elements.
Rationale	In combination with the gray-out visualization, it helps users validate if (a) all variable elements are attached to variation points and (b) the variation points lead to the correct transformed variant.
Use Case	QT3-UC-1, Use Case 5
Source	[Pap11]
Acceptance Criterion	Only variability-affected elements are highlighted. The highlight color or a similar color is not used as default color for non-highlighted elements.

ID: 8.5	Preview visualizations shall replace calculation values.
Description	Calculations shall be substituted in the same way as during transformation.
Rationale	The preview visualization should be as close to the transformation as possible, so that users can check the entered pvSCL rules without transforming a variant. Therefore, the visualization should also include calculation replacement.
Use Case	QT3-UC-1, Use Case 5
Possible Realization	If possible, use the same base class for transformation and for preview visualization.
Acceptance Criterion	Calculation values are replaced during all preview visualizations.

Bibliography

[AC04] Michał Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: Feature Modeling Plug-in for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, Eclipse '04, pages 67–72. ACM Press, 2004. (cited on Page 81)

[AM04] Ian Alexander and Neil Maiden. *Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*. Wiley Publishing, 1st edition, 2004. (cited on Page 17)

[BFG+02] Jan Bosch, Gert Florijn, Danny Greefhorst, Juha Kuusela, J. Henk Obbink, and Klaus Pohl. Variability Issues in Software Product Lines. In *Software Product-Family Engineering*, volume 2290 of *Lecture Notes in Computer Science*, pages 13–21. Springer, 2002. (cited on Page 10)

[BGH+13] Frank Bodmann, Ömer Gürsoy, Stefan Henkler, Alexander Nyßen, Michael Schulze, Carsten Strobel, and Bastian Tenbergen. Requirements for the SPES Reference Technology Platform (SPES_RTP) – QT3.AP1.D1. Unpublished requirements document, 2013. (cited on Page 8 and 34)

[Böh] Wolfgang Böhm. *SPES 2020 - Software Plattform Embedded Systems*. http://spes2020.informatik.tu-muenchen.de/home.html. Last accessed on May 30, 2013. (cited on Page 8)

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the 7th International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '13, page 7:1–7:8. ACM Press, 2013. (cited on Page 2, 11, and 79)

[CA05] Krzysztof Czarnecki and Michał Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of the 4th international conference on Generative Programming and Component Engineering*, GPCE '05, pages 422–437. Springer, 2005. (cited on Page 81)

[CAK+05] Krzysztof Czarnecki, Michał Antkiewicz, Chang Hwan Peter Kim, Sean Lau, and Krzysztof Pietroszek. fmp and fmp2rsm: Eclipse Plug-Ins for Modeling Features Using Model Templates. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 200–201. ACM Press, 2005. (cited on Page 83)

[CB10] Lianping Chen and Muhammad Ali Babar. Variability Management in Software Product Lines: An Investigation of Contemporary Industrial Challenges. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC '10, pages 166–180. Springer, 2010. (cited on Page 1 and 2)

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. (cited on Page 10)

[CN01] Paul C. Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 3rd edition, 2001. (cited on Page 1 and 10)

[CRM91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The Information Visualizer, an Information Workspace. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Reaching through Technology*, CHI '91, pages 181–186. ACM Press, 1991. (cited on Page 25)

[Cza11] Krzysztof Czarnecki. Understanding Variability Abstraction and Realization. In *Top Productivity through Software Reuse*, volume 6727 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 3rd edition, 2011. (cited on Page 2)

[DB07] Mark Dalgarno and Danilo Beuche. Variant Management. 2007. *Paper presented at the 3rd British Computer Society Configuration Management Specialist Group Conference*. (cited on Page 3, 10, and 11)

[DZ03] Daniela E. Damian and Didar Zowghi. Requirements Engineering Challenges in Multi-Site Software Development Organisations. *Requirements Engineering*, 8(3):149–160, 2003. (cited on Page 18)

[Fly06] Bent Flyvbjerg. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry*, 12(2):219–245, 2006. (cited on Page 78)

[FMV] Federal Motor Vehicle Safety Standard 108 - Lamps, Reflective Devices, and Associated Equipment. http://www.fmcsa.dot.gov/rules-regulations/administration/fmcsr/fmcsrruletext.aspx?reg=r49CFR571.108. Last accessed on May 30, 2013. (cited on Page 8)

[Gar11] Janel Garvin. Users Choice: 2011 Software Development Platforms – A Comprehensive User Satisfaction Survey of Over 1200 Software Developers. Technical report, Evans Data Corporation, 2011. (cited on Page 11)

[GM10] Yaser Ghanam and Frank Maurer. Linking Feature Models to Code Artifacts Using Executable Acceptance Tests. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC '10, pages 211–225. Springer, 2010. (cited on Page 82)

[Gol10] E. Bruce Goldstein. *Sensation and Perception*. Cengage Learning, 8th edition, 2010. (cited on Page 32)

[GP96] Jill Gerhardt-Powals. Cognitive Engineering Principles for Enhancing Human-Computer Performance. *International Journal of Human-Computer Interaction*, 8(2):189–211, 1996. (cited on Page 25)

[Gru89] Jonathan Grudin. The Case Against User Interface Consistency. *Communications of the ACM*, 32(10):1164–1173, 1989. (cited on Page 4, 21, and 74)

[HcW08] Florian Heidenreich, Ilie Şavga, and Christian Wende. On Controlled Visualisations in Software Product Line Engineering. In *Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering*, ViSPLE '08, pages 335–341. Lero Int. Science Centre, 2008. (cited on Page 82 and 83)

[HHPK05] Imed Hammouda, Juha Hautamäki, Mika Pussinen, and Kai Koskimies. Managing Variability Using Heterogeneous Feature Variation Patterns. In *Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering*, FASE '05, pages 145–159. Springer, 2005. (cited on Page 83)

[HJD11] Elizabeth Hull, Ken Jackson, and Jeremy Dick. DOORS: A Tool to Manage Requirements. In *Requirements Engineering*, pages 181–198. Springer London, 2011. (cited on Page 17)

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering*, pages 943–944. ACM Press, 2008. (cited on Page 82)

[HLMS10] André Heuer, Kim Lauenroth, Marco Müller, and Jan-Nils Scheele. Towards an Effective Visual Modeling of Complex Software Product Lines. In *Proceedings of the 3rd International Workshop on Visualisation in Software Product Line Engineering*, ViSPLE '10, pages 229–237, 2010. (cited on Page 82 and 83)

[HSS+10] Florian Heidenreich, Pablo Sánchez, João Santos, Steffen Zschaler, Mauricio Alférez, João Araújo, Lidia Fuentes, Uirá Kulesza, Ana Moreira, and Awais Rashid. Transactions on Aspect-Oriented Software Development VII. chapter Relating Feature Models to Other Models of a Software Product Line: a Comparative Study of FeatureMapper and VML, pages 69–114. Springer-Verlag, 2010. (cited on Page 14 and 81)

[ISO98] Ergonomic Requirements for Office Work with Visual Display Terminals (VDTs) - Part 11: Guidance on Usability. Technical Report 9241-11, International Organization for Standardization, 1998. (cited on Page 24)

[JDB07] Hans Peter Jepsen, Jan Gaardsted Dall, and Danilo Beuche. Minimally Invasive Migration to Software Product Lines. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 203–211. IEEE Computer Society, 2007. (cited on Page 1)

[Joh10] Jeff Johnson. *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*. Morgan Kaufmann, 2010. (cited on Page 25 and 37)

[KB09] Charles W. Krueger and Martin Bakal. Systems and Software Product Line Engineering with SysML, UML and the IBM Rational Rhapsody BigLever Gears Bridge. Whitepaper, 2009. (cited on Page 83)

[KCH+90] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, 1990. (cited on Page 11)

[KJ09] Charles W. Krueger and Ken Jackson. Requirements Engineering for Systems and Software Product Lines. Whitepaper, 2009. (cited on Page 83)

[Lie] Josh Lieberman. *The Number One Test Management Tool is Excel and We Find That Unacceptable*. http://www.qasymphony.com/quality-assurance-management-tools-access-to-excel-%E2%80%93-are-we-digressing.html. Last accessed on May 30, 2013. (cited on Page 18)

[Lin94] Gitte Lindgaard. *Usability Testing and System Evaluation: A Guide for Designing Useful Computing Systems*. Chapman & Hall, 1st edition, 1994. (cited on Page 24)

[LKL12] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A Survey on Software Product Line Testing. In *Proceedings of the 16th International Software Product Line Conference*, SPLC '12, pages 31–40. ACM Press, 2012. (cited on Page 1)

[LSR07] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer, 1st edition, 2007. (cited on Page 1, 2, 10, 11, and 79)

[Men10] Jeremy Mendel. The Effect of Interface Consistency and Cognitive Load on User Performance in an Information Search Task. Master's thesis, Clemson University, 2010. (cited on Page 21)

[Mic] Microsoft News Center. *Microsoft Sees Big Opportunities for Partners With Upcoming Wave of New Products and Services*. http://www.microsoft.com/en-us/news/press/2012/jul12/07-09WPCDay1PR.aspx. Last accessed on May 30, 2013. (cited on Page 17)

[Mil56] George Miller. The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information. *Psychological Review*, 63(2):81–97, 1956. (cited on Page 26 and 37)

[Mye85] Brad A. Myers. The Importance of Percent-Done Progress Indicators for Computer-Human Interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '85, pages 11–17. ACM Press, 1985. (cited on Page 37)

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, 1993. (cited on Page 21, 24, 25, 37, 38, and 75)

[Nie94] Jakob Nielsen. Enhancing the Explanatory Power of Usability Heuristics. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems: Celebrating Interdependence*, CHI '94, pages 152–158. ACM Press, 1994. (cited on Page 25 and 26)

[NL06] Jakob Nielsen and Hoa Loranger. *Prioritizing Web Usability*. New Riders, 1st edition, 2006. (cited on Page 24)

[Pap11] Maria Papendieck. Improving Usability of UML Modeling Tools for Feature-Based Product Line Development. Bachelor's thesis, University of Magdeburg, 2011. (cited on Page 30, 32, 33, 38, 46, 121, 124, and 125)

[PBL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1st edition, 2005. (cited on Page 1, 2, 40, and 85)

[PD10] Bernhard Preim and Raimund Dachselt. *Interaktive Systeme – Band 1: Grundlagen, Graphical User Interfaces, Informationsvisualisierung*. Springer, 2nd edition, 2010. (cited on Page 21)

[PHAB12] Klaus Pohl, Harald Hönninger, Reinhold Achatz, and Manfred Broy. *Model-Based Engineering of Embedded Systems: The SPES 2020 Methodology*. Springer, 1st edition, 2012. (cited on Page 1 and 8)

[PME86] P. G. Polson, E. Muncher, and G. Engelbeck. A Test of a Common Elements Theory of Transfer. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '86, pages 78–83. ACM Press, 1986. (cited on Page 21)

[PTG07] Tauhida Parveen, Scott Tilley, and George Gonzalez. A Case Study in Test Management. In *Proceedings of the 45th Annual Southeast Regional Conference*, ACMSE '45, pages 82–87. ACM Press, 2007. (cited on Page 18)

[pur] pure-systems GmbH. *pure::variants User's Guide – Version 3.2.4 for pure::variants 3.2*. (cited on Page 12, 13, and 14)

[pur04] pure-systems GmbH. Variant Management with pure::variants. Technical White Paper, 2004. (cited on Page 11)

[RC02] Mary Beth Rosson and John M. Carroll. *Usability Engineering: Scenario-Based Development of Human-Computer Interaction*. Morgan Kaufmann, 1st edition, 2002. (cited on Page 21 and 74)

[RJB04] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2nd edition, 2004. (cited on Page 19)

[RMC06] Cheul Rhee, Junghoon Moon, and Young Chan Choe. Web Interface Consistency in E-Learning. *Online Information Review*, 30(1):53–69, 2006. (cited on Page 21)

[Sha91] Brian Shackel. Usability – Context, Framework, Definition, Design and Evaluation. In Brian Shackel and S. J. Richardson, editors, *Human Factors for Informatics Usability*, pages 21–37. Cambridge University Press, 1991. (cited on Page 24)

[Shn97] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 3rd edition, 1997. (cited on Page 25)

[SJ04] Klaus Schmid and Isabel John. A Customizable Approach to Full Lifecycle Variability Management. *Science of Computer Programming*, 53(3):259–284, 2004. (cited on Page 2 and 83)

[Som10] Ian Sommerville. *Software Engineering*. Addison-Wesley, 9th edition, 2010. (cited on Page 15)

[SP04] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley, 4th edition, 2004. (cited on Page 24, 25, 26, and 30)

[Spa] Sparx Systems. *Enterprise Architect*. http://www.sparxsystems.com/. Last accessed on May 30, 2013. (cited on Page 17)

[Ste06] Friedrich Steimann. The Paradoxical Success of Aspect-Oriented Programming. In *Proceedings of the 21st annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 481–497. ACM Press, 2006. (cited on Page 2)

[TAB12] Paolo Tell and Muhammad Ali Babar. A Systematic Mapping Study of Tools for Distributed Software Development Teams. Technical report, IT University of Copenhagen, 2012. (cited on Page 15 and 18)

[TJ] Ha Duy Trung and Stan Jarzabek. DME: Documentation Management Environment for Software Product Lines–Tool Demo Proposal. http://www.comp.nus.edu.sg/~stan/DME.pdf. Last accessed on May 30, 2013. (cited on Page 81)

[UNR] UN Regulation 112 - Headlamps Emitting an Asymmetrical Passing Beam and/or a Driving Beam and Equipped With Filament Bulbs. http://www.unece.org/fileadmin/DAM/trans/main/wp29/wp29regs/R112rev2_e.pdf. (cited on Page 8)

[WHE+08] Dindin Wahyudin, Matthias Heindl, Benedikt Eckhard, Alexander Schatten, and Stefan Biffl. Balancing Agility and Formalism in Software Engineering. chapter In-Time Role-Specific Notification as Formal Means to Balance Agile Practices in Global Software Development Settings, pages 208–222. Springer, 2008. (cited on Page 18)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 31. Mai 2013