Concepts for Consistent Variant-Management Tool Integrations

Maria Papendieck, Michael Schulze

pure-systems GmbH
Agnetenstraße 14
D-39106 Magdeburg
{maria.papendieck, michael.schulze}@pure-systems.com

Abstract: Making variability explicit in variable artifacts throughout the product lifecycle is difficult for variant modelers due to the different notations used to express variability. The tool integrations of the variant-management tool pure::variants address this problem, by providing a consistent method for denoting variability in frequently-used development tools. However, differing technical extension mechanisms of the tools lead to inconsistencies between integrations and slow down the development process. To overcome these problems, we proposed a workflow for implementing new variant-management tool integrations based on real-life requirements and tool-independent use cases. The main idea is that integration developers can derive requirements for their specific integration, based on their knowledge about the supported extension mechanisms of the tool. To ensure continued consistency, the requirements document from which specific documents can be derived is continually updated while implementing new integrations. For evaluation, we tested for three exemplary tools whether it is technically feasible to apply the workflow, and argued based on plausibility and an interview with a pure::variants customer whether our goals are met. Although we identified issues for future work, we concluded that the workflow is technically feasible and fulfills our goals.

1 Introduction and Motivation

One of the key tasks of developing a *Software Product Line (SPL)* is to define and manage the variabilities between the variants of an SPL [LSR07, p.7]. To this end, SPL developers typically create a feature model, describing all possible features of the SPL and the relations between them, and annotate all variable artifacts with rules linking them to features. Based on these rules, they can use a variant-management tool to transform the model containing all artifacts – *the master artifact model* – to different variants.

The annotation of artifacts is not trivial, because variabilities can occur in all kinds of artifacts produced during the product lifecycle (e.g., requirements, UML model elements, or paragraphs of a user manual). Due to the heterogeneous nature of the used tools, and since most tools are not designed to express variability natively and in a uniform way, many

different tool-specific notations are used to represent variability [SJ04]. Thus, SPL developers have to learn a new approach for managing variability in each tool. To overcome this problem, pure::variants[1], a leading variant-management tool [BRN+13], integrates a variant-management user interface into several tools that allows users to denote variability always in the same way without having to learn the exact variability notation in the respective tool. However, the extended tools provide different extension mechanisms. For example, it is not always possible to embed a user interface into the tool, and, due to the different variability notations, the mechanisms for editing variability differ in most cases. This introduces two challenges: On the one hand, it leads to inconsistencies between tool integrations, since different functionalities can be implemented in each tool. Thus, it increases the time needed to learn how to use the integration. On the other hand, it slows down the development process, because integration developers have to find new technical solutions for implementing the same functionality.

In this paper, we address these objectives. We aim to propose a workflow that (a) improves the consistency between variant-management tool integrations, and thus reduces the learning curve for users who already know other integrations; (b) reduces the time-to-market for new tool integrations; (c) promotes good usability; and (d) relates to the everyday tasks of variant-management practitioners.

The remainder of the paper is structured as follows: In Section 2, we present the workflow for developing new tool integrations and the use cases and requirements it is based on. Then, we evaluate, in Section 3, whether the workflow fulfills our goals. In Section 4, we give an overview of related work. Finally, we draw conclusions and list future work.

2 Concepts for Consistent Tool Integrations

To increase consistency between integrations and reduce the time-to-market for new integrations, a good approach would be to employ a reuse strategy [LSR07]. Therefore, we first analyze the existing tool integrations to find out where commonalities and variabilities between tool integrations exist. We produce a list of tool-independent use cases from which we derive requirements for variant-management tool integrations.

2.1 Tool-Independent Use Cases

There are two main activities regarding variant-management that tool integrations have to support: Users need to edit variation-point rules and verify whether the edited rules are correct. For these two activities, we compose six use cases. Since a detailed discussion of all use cases would exceed the available space, we summarize them and present the functionality related to the use cases. For the full use cases, refer to [Pap13, Section 3.1]. We illustrate the functionalities using a screenshot of the pure::variants integration to Microsoft Office Word (see Figure 1). It is labeled for better reference.

[1] www.pure-systems.com (last accessed on February 6, 2014)

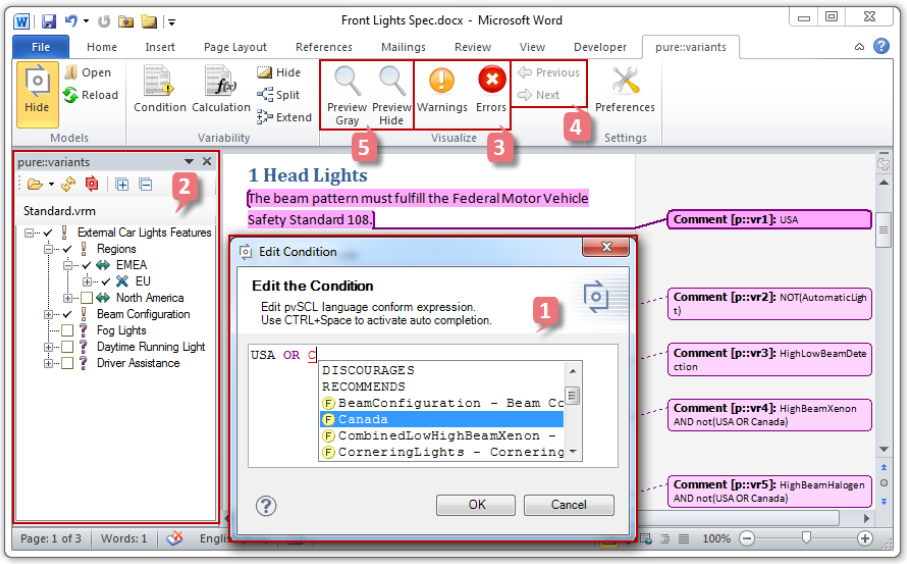


Figure 1: The pure::variants Integration for Microsoft Office Word. The numbers designate key functionalities referred to in the text.

Editing Variation-Point Rules pure::variants supports different types of variation-points: Structural and parametric. For example, *restrictions* or *conditions*, which decide whether or not the annotated element will be part of a variant (structural), and *calculations*, which compute a value that will replace the annotated text in the final variant (parametric) [psG13]. These rules are written in the *pure::variants simple constraint language* (pvSCL). In the use cases, we describe how users work with a pvSCL editor to write new rules, and how being able to load and display pure::variants models in the extended tool helps them. To support these use cases each integration needs to implement the respective pvSCL editors, which prevent errors and speed up the editing process by providing autocompletion, syntax highlighting and error check (see Figure 1, label 1). Furthermore, the integration needs to embed a user-interface into the extended tool that allows loading and displaying pure::variants' feature models and variant result models (see Figure 1, label 2).

Verifying Variation-Point Rules In further use cases, we describe scenarios how users can verify whether the entered rules are correct. We argue that it is easier and more efficient to use different visualizations directly in the extended tool instead of triggering a transformation and looking at the results. We propose two categories of visualizations, which serve different goals: First, two different error visualizations help users find all errors that are automatically detectable (see Figure 1, label 3). A syntax error visualization highlights all variation points that do not comply with pvSCL syntax, whereas a semantic error visualization highlights all variation points that contain references to unknown pure::variants features or attributes. To find errors that are outside the current viewport,

users can jump between errors by using navigation buttons (see Figure 1, label 4). Second, two different preview visualizations enable users to check whether the resulting variant is as expected. One preview visualization grays out all elements that are not relevant for the considered variant, and the other hides these elements (see Figure 1, label 5).

2.2 Requirements for variant-management tool integrations

To identify more detailed commonalities and variabilities between integrations, we compile requirements for variant-management tool integrations. We derive them from (a) the tool-independent use cases, (b) requirements for variant-management tool integrations already existing in the context of the SPES$_{XT}$[2] project, (c) our experience with technical constraints of previously extended tools, and (d) usability guidelines published in [Nie93].

We organize the requirements in hierarchical fashion with the following top-level requirements. The letter at the end of each requirement refers to the source it is based on:

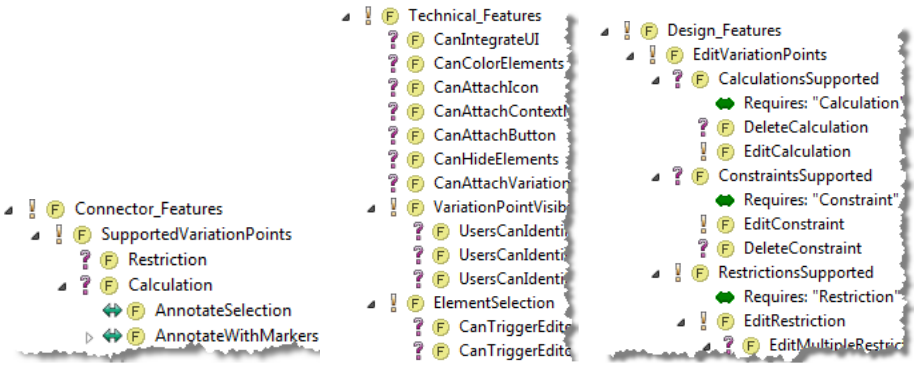
1. Variant management shall always be done in the same way for the user independent of the used development tool (b).
2. The tool integration shall comply with usability heuristics (d).
3. Variant management shall seamlessly integrate itself into the development tools by providing a user interface (b).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

Below these top-level requirements, 62 child requirements exist, which describe in detail the functions that each integration should support. Due to technical and functional differences between the to be extended tools, not all of the requirements apply to all integrations. For some functionalities alternative or optional requirements exist, depending on which extension mechanisms the respective tool supports.

2.3 Managing Variability within Tool-Integration Requirements

To reduce the time-to-market for new tool integrations, we suggest to use pure::variants to manage the variabilities between requirements, so that integration developers can generate the final requirements document for the tool they want to extend. Therefore, we compile all possible functionalities of a tool integration, and all relevant extension mechanisms of tools into three feature models (see Figure 2). The model *Connector Features* contains all relevant features related to the underlying pure::variants transformation, which ships with a pure::variants connector. *Technical Features* lists all technical details that influence the

[2] Extended Software Plattform Embedded Systems research project



(a) Connector Features (b) Technical Features (c) Design Features

Figure 2: The feature models for managing variability within tool-integration requirements [Pap13].

final requirements document, whereas *Design Features* summarizes all functionalities that may be supported. Additionally, the elements of the *Design Features* model are linked to the previous two models, such that only those features that the extended tool technically supports can be selected in a variant. To manage the requirements in a requirements master artifact model, we add pure::variants restrictions to all alternative or optional requirements.

2.4 Workflow for Developing Variant-Management Tool Integrations

Based on these feature models and the requirements master artifact model, we propose a workflow for developing variant-management tool integrations, which fulfills our goals. This workflow may also be used for the development of variant-management tool integrations for other variant-management tools like BigLever Gears[3].

In Figure 3, we show the steps of the workflow: First, developers need to create a variant model based on the three feature models presented in Figure 2. To this end, they specify which variation-point types the respective pure::variants transformation can compute and which extension mechanisms the tool provides. Based on these selections, they can choose which functionalities the integration should support. If pure::variants reports errors for the selection, a feature cannot be selected due to technical constraints. In this case, there are two options, either developers create an alternative requirement and adapt the feature models accordingly, or the feature is not supported in the respective tool. When no errors are reported, pure::variants is used to generate a variant of the tool-integration requirements. In Figure 3, a variant for an integration to Microsoft Excel is created.

To ensure that all requirements can be met, developers now analyze based on test implementations and experience whether all requirements can be satisfied. If a requirement

[3] www.biglever.com/solution/product.html (last accessed on February 6, 2014)

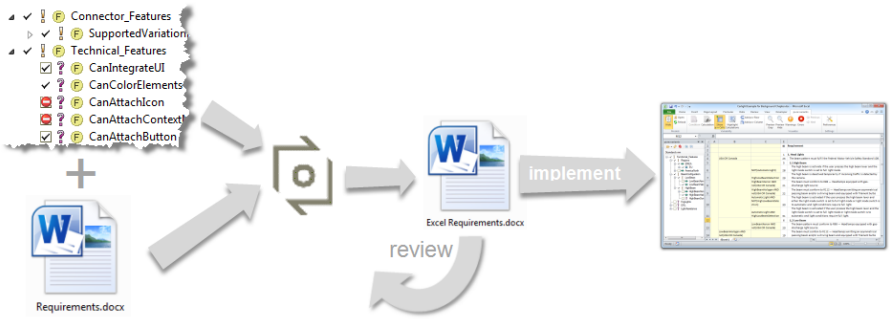


Figure 3: The workflow for developing variant-management tool integrations [Pap13]

cannot be fulfilled, the feature models and requirements may be incorrect or incomplete. In this case, developers update the faulty or incomplete parts and generate a new variant. Finally, they implement all requirements.

To ensure that integrations stay consistent even after the initial implementation, all integrations should be updated in case the requirements master artifact model changes. We choose to base this workflow rather on reusing requirements than reusing code, because different technical constraints prevent using the same code for implementing the same functions. However, we employ the requirements and the knowledge of each tool's extension mechanisms to reuse code whenever applicable.

3 Evaluation

For evaluation, we tested whether it is technically feasible to produce consistent tool integrations using the workflow and analyzed whether we fulfilled our goals.

3.1 Technical Evaluation

For evaluating the workflow with respect to technical feasibility, we applied it to three different tools: The requirements-management tool IBM Rational DOORS[4], the spreadsheet application Microsoft Office Excel[5], and the UML-modeling tool Enterprise Architect by Sparx Systems[6]. We chose these applications, since they are used by a large number of people [HJD11, Mic] and typically apply to different phases of the product lifecycle.

In general, the technical evaluation was successful. We could satisfy most requirements. Eight requirements were not yet fulfilled since there was not enough time (e.g., Require-

[4] www.ibm.com/software/products/en/ratidoor (last accessed on February 6, 2014)
[5] office.microsoft.com/en-us/excel (last accessed on February 6, 2014)
[6] www.sparxsystems.com (last accessed on February 6, 2014)

ment 1.1 *Tool integrations shall use the same or similar icons for the same functions*). They will be part of future work. Furthermore, we could not meet four requirements due to technical constraints of some extended tools and did not find alternative solutions for the requirements. These concern the undo and redo functionality in Excel and Enterprise Architect, the batch editing of pvSCL rules in DOORS, and the hide preview visualization in Enterprise Architect. We classified the latter two requirements as uncritical. However, the lack of undo and redo support is a usability problem, for which we should try to find a workaround in future releases. For more details refer to [Pap13, Section 4.4].

3.2 Evaluation of Goals

Our goals are to propose a workflow for developing new tool integrations that (a) improves the consistency between variant-management tool integrations, (b) reduces the time-to-market, (c) promotes good usability, and (d) relates to the everyday tasks of variant-management practitioners. To assess whether the proposed workflow meets these goals, we argue whether the first two goals (main goals) are satisfied, and conduct an interview with a pure::variants customer regarding the last two goals (quality goals).

3.2.1 Main Goals

Consistency We argue that the workflow improves the consistency between new tool integrations for the following reasons: First, Requirement 1 of the requirements master artifact model directly addresses consistency. Thus, developers implementing the requirements should focus more on consistency. Furthermore, in Requirement 1, we suggest to use the same code base for common user-interface elements. This should also improve consistency. Second, the workflow indirectly supports consistency by reusing requirements wherever possible. If the requirement's text is the same, the implemented functions should also be the same or similar.

Reduced Time-to-Market We reason that using the workflow reduces time-to-market, because developers generate the respective requirements document instead of writing it anew. Furthermore, the *technical features* model provides a list of extension mechanisms, which developers need to test. This supports systematic testing, and thus saves time.

3.2.2 Quality Goals

To evaluate whether the workflow promotes good usability and relates to the everyday tasks of variant-management practitioners, we prepared an interview with pure::variants customers experienced with at least one pure::variants tool integration. Since such participants are difficult to find (probably due to their limited time budget), we conducted only one valid interview with a pure::variants customer. Although this reduces the scientific value of the interview, we still think describing the interview is worthwhile, since the

participant made relevant suggestions for improvements. Hence, we next summarize the interview's settings and results.

During the interview, we watched the participant execute three simple tasks using the integration for IBM Rational DOORS, and asked him for feedback on the integration. The interview was planned to last 30 to 45 minutes and took place in form of a web meeting.

In general, the participant found the integration useful. Nevertheless, he criticized the feature-search capabilities when writing pvSCL rules based on large feature models with many similarly-named features. Since industrial practitioners often use large feature models [BRN+13], we addressed this issue by modifying the autocompletion of the pvSCL editor to present more helpful proposals.

The interview indicates that the workflow promotes good usability and relates to the everyday tasks of variant-management practitioners. However, generalizability of the interview results is limited, since we had only one participant [Fly06]. Nevertheless, we argue that the workflow fulfills the quality goals, because: First, explicit requirements regarding usability guidelines exist, which should prevent common usability problems if they are fulfilled in new tool integrations. Second, the requirements are based on existing tool integrations. Therefore, usability should not decrease if the new workflow is used. Finally, five of the eight main requirements explicitly reflect the demands of the industrial partners, since they are taken from requirements of the SPES$_{XT}$ project. Thus, they relate to the everyday tasks of variant-management practitioners. In future, we will verify our arguments by conducting more interviews.

4 Related Work

We structure the presentation of related work according to the three main topics our work is concerned with: Making variability explicit, reviewing variability using visualizations, and managing variability throughout the product lifecycle. For each category we only present selected work. For our complete findings refer to [Pap13, Chapter 6].

Making Variability Explicit In literature, many different representations of variability have been proposed [HSS+10]. However, only few provide tool support. For example, CZARNECKI AND ANTKIEWICZ provide the prototype *fmp2rsm*, which embeds their *feature modeling plugin* into IBM Rational Software Developer [AC04]. It enables users to add *presence conditions* to artifacts. Similar to pure::variants restrictions, they enable the decision whether an artifact will be part of a variant.

Furthermore, the Eclipse plugin *FeatureMapper*, which was developed in the context of the research project FeasiPLe in which pure-systems is also involved, enables users to denote variability in all models based on the Eclipse Modeling Framework [HcW08, HKW08]. It also allows to use pure::variants feature models. Different to our approach, users can select the features related to an artifact directly in the feature model.

Visualizations Like our integrations, both of the presented tools support visualizations for reviewing the entered variability information. E.g., *fmp2rsm* enables automatic coloring, which maps a different color to each presence condition. Using *FeatureMapper* users can assign colors to features, and thus also color all related artifacts. *FeatureMapper* supports gray-out preview as we do. To help users find changes that are not visible in the diagram, it also enables highlighting all property changes related to the feature-selection.

Managing Variability Throughout the Product Lifecycle To ease the transition from single-system development to product-line development, SCHMID AND JOHN present a concept for managing variability in a homogenous way throughout the product lifecycle [SJ04]. The basic idea is to provide an approach for modeling variability, and a method for mapping variability to the different notations of artifact models. Similar to pure::variants, only the mapping should be tool-specific. However, SCHMID AND JOHN focus more on customizable approaches to support the mapping instead of consistent tool integrations.

5 Conclusion and Future Work

Variant-management tool integrations help users to denote variability always in the same way without having to learn the exact variability notation in each tool used during the product lifecycle. However, extending these tools in a consistent way is not trivial, since each tool provides different extension mechanisms and different technical constraints. This introduces inconsistencies between tool integrations and slows down the development process, since developers need to find new solutions for implementing the same functions.

To overcome these problems, we proposed a new workflow for developing tool integrations. The workflow is based on requirements, which we compiled from our own tool-independent use cases, industrial requirements of the SPES$_{XT}$ project, usability guidelines, and our experience with technical constraints of previously extended tools. Its main idea was to reuse requirements between tool integrations, and thus increase consistency and reduce time-to-market for new integrations.

For evaluation, we concluded that the workflow is technically feasible, and argued based on plausibility and an interview with a pure::variants customer that our goals are fulfilled.

However, we still need to complete several tasks: First, we need to implement all requirements that are not yet fulfilled due to time constraints. Second, we need to interview more pure::variants customers to complete the evaluation of quality goals, and to continually adapt the requirements to the special use cases of customers. This also includes addressing the suggestions for improvement that resulted from the interview with a pure::variants customer. Third, we will implement integrations for more tools used during the development of software-intensive systems. With each new integration, we will adapt the requirements master artifact model further and propagate the necessary changes to all tool integrations.

6 Acknowledgements

This work is partly based on documents of the SPES$_{XT}$ project. It has been funded by the German Ministry for Education and Research (BMBF) under the funding ID 01IS12005. The responsibility for the contents rests with the authors.

References

[AC04] Michał Antkiewicz and Krzysztof Czarnecki. FeaturePlugin: Feature Modeling Plug-in for Eclipse. In *Proc. 2004 OOPSLA Workshop on Eclipse Technology eXchange*, Eclipse '04, pages 67–72. ACM Press, 2004.

[BRN+13] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. A Survey of Variability Modeling in Industrial Practice. In *Proc. 7^{th} Int. Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS '13, page 7:1–7:8. ACM Press, 2013.

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

[HcW08] Florian Heidenreich, Ilie Şavga, and Christian Wende. On Controlled Visualisations in Software Product Line Engineering. In *Proc. 2^{nd} Int. Workshop on Visualisation in Software Product Line Engineering*, ViSPLE '08, pages 335–341, 2008.

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

[HKW08] Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Companion Proc. 30^{th} Int. Conf. on Software Engineering*, pages 943–944. ACM Press, 2008.

[HSS+10] Florian Heidenreich, Pablo Sánchez, João Pedro Santos, Steffen Zschaler, Mauricio Alférez, João Araújo, Lidia Fuentes, Uirá Kulesza, Ana Moreira, and Awais Rashid. Relating Feature Models to Other Models of a Software Product Line - A Comparative Study of FeatureMapper and VML*. *Transactions on Aspect-Oriented Software Development*, 7:69–114, 2010.

[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, 2007.

[Mic] Microsoft News Center. *Microsoft Sees Big Opportunities for Partners With Upcoming Wave of New Products and Services*. www.microsoft.com/en-us/news/press/2012/jul12/07-09WPCDay1PR.aspx. Last accessed on Feb. 6, 2014.

[Nie93] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, 1993.

[Pap13] Maria Papendieck. Consistent Concepts for Variant-Management Tool Integrations during the Complete Product Lifecycle. Master's thesis, Otto-von-Guericke University Magdeburg, 2013.

[psG13] pure-systems GmbH. *pure::variants User's Guide*, 2013.

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