On Controlled Visualisations in Software Product Line Engineering

Florian Heidenreich Ilie Şavga Christian Wende

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany

E-mail: florian.heidenreich@tu-dresden.de

Abstract

Ongoing research in Software Product Line Engineering (SPLE) emphasises the derivation of a concrete product based on a given variant configuration as one of the most promising areas of the field. To allow for (automatic) derivation of products in SPLE, models that describe features and their variability in Software Product Lines (SPLs)—for example feature models—need to be connected with artefacts that are realising the features. It is crucial not only to support the developer in the complex task of defining such connections, but also to provide means to reason and analyse them—for example visualisations. In this paper we present MappingViews*, a novel visualisation technique that provides four different visualisations for connections between variability models and realisation models and show its realisation in our tool* FeatureMapper.

1 Introduction

Software Product Line Engineering (SPLE) abstracts and explicitly represents the notion of variability to ease the development of complex systems. In SPLE, the architecture of a system not only describes one product, but many of the same nature with various variable parts and modules [1, 18]. Feature modelling is a key technique used for expressing the mandatory and variable features in SPLs [5, 14]. One of the challenging issues within SPLE is, that there is a gap from features of feature models to the realisation of the features by specific software artefacts [13].

To bridge this gap, Czarnecki et al. proposed a template-based approach [6, 8] where features from feature models are connected with UML model elements by *Presence Conditions*. *Presence Conditions* are expressions that are directly annotated to the UML model elements by using XPath expressions or UML stereotypes. By introducing this connection between features and model artefacts, an initial step towards automatic product derivation is done. However, this solution leaves out an important concern. As the realisation models grow in size and complexity, it becomes even harder to retrace and understand connections. Visualisation techniques—such as building views on the models—could help, but only one visualisation was provided in [6], that is, the representation of the complete mapping in the realisation models. Additionally, the developer is not in the control of visualisation in their approach and this control is extremely important for understanding and exploring the models. It is hard to impossible to understand a non-trivial system unless you look at it from different points of view [9, page 1].

In many cases developers are interested only in a particular aspect of the connection between a feature model and realising artefacts. How a particular feature is realised? Which artefacts may be effectively used in a variant? In this paper we argue for the need to support different interactive visualisations of mappings between features and realisation artefacts in SPLE that can be *controlled*, i.e. en- and disabled or parameterised, by developers. In Section 2 we present *MappingViews*, a novel visualisation technique that provides four different visualisations that we identified to be useful in tackling the increased complexity in SPLE. Our realisation of the visualisations in our tool *FeatureMapper* [11] is presented in Section 3. Before we conclude in Section 5 we relate our approach to selected existing work in Section 4.

2 Visualisation of Mappings between Features and Software Artefacts

As motivated in Section 1, there is a strong need to provide visualisations that support the developer of a SPL at defining and understanding the complex connections between features from feature models and the software artefacts that are realising those features. We call this con-

nections *Feature Mappings*—or mappings—in our work. These mappings usually do not only contain mappings between features and software artefacts, but also between *feature expressions* and software artefacts, where a feature expression is a logical combination of features (e.g. `FeatureA AND FeatureB`).

In the context of the feasiPLe research project [10], where we have been developing and analysing different exemplary product lines, we noticed that a fixed and static visualisation of the feature mapping, as realised by Czarnecki et al. [6, 8] is too limited in some cases. A single visualisation of the feature mapping as a whole only offers information in one dimension, that is, which feature is realised by which software artefacts. The interpretation and analysis of this mapping is still completely up to the developer.

Motivated by the need for increased support of the developer in SPLE, we propose a novel visualisation technique called *MappingViews* which provides four basic visualisations, called views, that we identified to be helpful in the context of PLE:

- Realisation View,
- Variant View,
- Context View, and
- Property-Changes View.

In the following, to describe these visualisations independently from their concrete technological realisations, we use the term *problem space* instead of a concrete variability management mechanism and the term *solution space* instead of a concrete realisation technique for software artefacts [7].

2.1 Realisation View

The *Realisation View* is a visualisation that helps at understanding which software artefacts are mapped to a specific feature from the problem space. It hides, removes, or filters out all software artefacts that do not participate in the realisation of the specific feature. This enables the developer to directly see all (and only those) artefacts that participate in the realisation of a specific feature or feature expression. It also provides basic means to analyse and measure the impact of a specific feature in the SPL. That is, the developer can see whether a feature is connected to many artefacts or just to few of them and, thus, an initial understanding of the complexity of the feature can be provided. Additionally, the *Realisation View* can help at analysing feature granularity, that is, analysing whether a feature is realised by coarse-grained modules or by more fine-grained artefacts.

2.2 Variant View

The *Variant View* is a visualisation that hides, removes, or filters out all software artefacts that are not included in a specific variant of the product line. That is, it shows or highlights only those artefacts that are included in a specific configuration of the product line. This view provides basic means to analyse the well-formedness and validity of the resulting concrete variant of the SPL. It is up to the implementer of the visualisation whether artefacts that are common to all products of the SPL, that is, the core, and are *not* mapped to a specific feature are also included in the variant view. This view enables the developer to get a dynamic view on the resulting system. This is particularly helpful in the design phase, where effects of small changes to the mapping between features and artefacts need to be constantly checked to ensure a valid resulting system.

2.3 Context View

It is often necessary to also analyse the communication and interaction among features, between features and the core of the product line or to inspect the realisation of a feature in its context. For this task, the *Context View* visualisation can be used. Each feature or feature expression of interest can be assigned a certain colour. Assuming that it is possible to render the representations of the software artefacts in the same colour (which is possible with state-of-the-art editors for model artefacts), an intuitive way for the visualisation of feature borders and the communication between features is possible. The *Context View* can be seen as a more generalised version of the *Realisation View* as it provides means to visualise the realisation of multiple features or feature expressions. We argue, that it has to be distinguished from the *Realisation View* which only shows the realisation of a single feature or feature expressions, because colouring has a different intent, that is, it reveals feature communication and interaction.

Assigning colours to features also raises some interesting questions. As soon as many colours get assigned (e.g. in a very complex SPL), usability is again decreased because many colours become hardly to distinguish at some point. Thus, we argue, that it only makes sense to enable colouring for a limited set of features. In addition to that, the selection of the colours used should be left to the developer and should not be assigned automatically. Otherwise, issues like red-green colour-blindness become significant.

Another issue is the possibility of defining overlapping feature mappings (e.g., one solution artefact is mapped to more than one feature or feature expression). While mixing of colours looks like an intuitive solution to this problem, our experiments have shown, that it is usually hard to understand those mixed colours and retrace them to the initial

features or feature expressions. Therefore, we recommend to change the mapping and use a corresponding `OR` feature expression, where each feature that was initially mapped to the software artefact is referenced (e.g. `FeatureA OR FeatureB`).

2.4 Property-Changes View

Some features may require changes to the structure and properties of a solution artefact. For example, a feature may require the cardinality of a UML association to be changed (e.g., a feature extends the cardinality from one to one-to-many). These feature-dependent changes—also called *meta expressions* [6] or *property-value mappings* [12]—are often hard to understand and also difficult to visualise. This is mainly due to the fact, that those feature-dependent changes need to be interpreted based on a concrete variant configuration to manifest themselves. A helpful technique for identifying those spots in the software artefacts is to highlight them with an eye-catching colour, for instance red. This way, the developer can further inspect, change, or delete the identified feature-dependent changes. Another important property of the *Property-Changes View* is, that it allows for displaying encapsulated feature-dependent changes, which were separated them from the software artefacts (as opposed, for instance, to XPath expressions [6] embedded into UML models).

3 Visualisation of Feature Mappings in the *FeatureMapper* Tool

Based on our findings regarding necessary and helpful visualisations, we implemented the proposed *MappingViews* approach in our *FeatureMapper* [11] tool.

3.1 The *FeatureMapper* Tool

Figure 1 shows a screenshot of the *FeatureMapper*. It consists of four parts. The tool bar (1) provides means for loading and saving feature mappings and for controlling the different visualisation options. The upper compartment (2) contains the feature model that is associated with the current mapping model. The example describes the variability options in a basic contact management application. Compartment (3) contains the feature or feature expression that is currently active. It can either be changed via double click on a feature in the feature model or via the context menu in this compartment. In the example, the current expression involves the selection of the feature *Relationships*. Compartment (4) contains the feature or feature expression that has already been applied to currently selected model elements of the solution model (which is not depicted in the figure). In the example, the elements are associated with the feature *Addresses*.

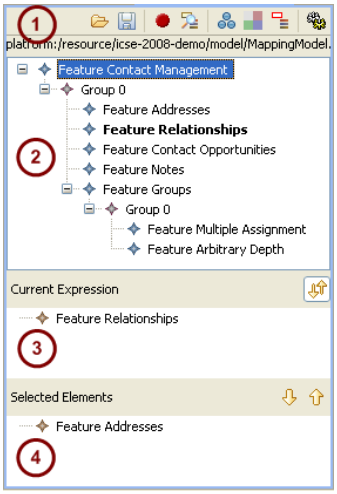

Figure 1. Screenshot of the *FeatureMapper*.

Our tool consists of multiple plug-ins for the Eclipse Platform [20]. It is based on the Eclipse Modelling Framework (EMF) [3] that provides the Ecore metamodelling language which is used to specify the abstract syntax for arbitrary modelling languages. Thus, the modelling of the solution space is not bound to any concrete language and existing EMF-based modelling tools (e.g. TOPCASED [21]) can easily be integrated. As a consequence, software artefacts described by means of other metalanguages (e.g. MOF, source code, or documentation artefacts) are currently not supported by our tool. For creating feature models, we use the feature metamodel developed by the feasiPLe consortium [10].

To create a mapping between features or feature expressions and model elements, the developer first has to select the feature or feature expression from *FeatureMapper's* feature model (cf. Figure 1(2)). Then, the corresponding model elements that realise this feature need to be selected in the solution model so that the active expression can be applied to the model elements via the down-arrow toolbar button in Figure 1(4). Internally, a new mapping element in the mapping model is created, which serves as a link between the feature expression and the model elements. This is later used for visualisation and mapping-based derivation.

3.2 Visualisation in the *FeatureMapper* Tool

Our tool provides different means for visualisation for both graphical and tree-based editors. It works in a *non-invasive* way with any graphical editor that is based on the Graphical Editing Framework (GEF) [19] and editors gen-

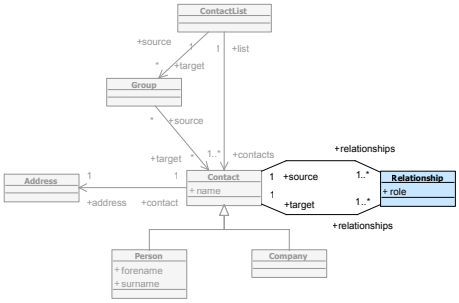

Figure 2. Realisation of the feature *Relationships* rendered by FeatureMapper's *Realisation View* in a GEF-based editor (TOPCASED).

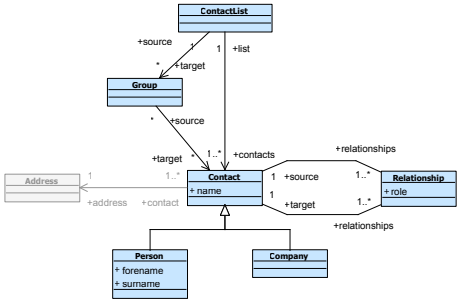

Figure 3. Variant of the product line without the feature *Addresses* rendered by the *Variant View*.

erated by EMF [3]. That means, that the editors do not need to be adjusted to work with our tool.

3.2.1 Realisation View

In the *FeatureMapper*, the *Realisation View* helps to understand which parts of a solution model are mapped to a specific feature or feature expression. It greys out all model elements that do not participate in the realisation of the current feature. This is depicted in Figure 2 for the feature *Relationships*. Since the model elements that are not associated with the feature are still shown, the context of interaction between the feature realisation and the rest of the system is preserved. To use this view, the developer first enables the *Realisation View* via the respective button on the *FeatureMapper* toolbar and selects the feature of interest from *FeatureMapper's* feature model.

3.2.2 Variant View

The *Variant View* shows all model elements that are included in a specific variant of the product line. This visualisation can be either parameterised by an existing concrete variant configuration or interactively adjusted by selecting features from the feature model shown in the *FeatureMapper*. Unlike the *Realisation View* it also includes the elements that are common to all products from the product line and are not mapped to a specific feature. Figure 3 shows a variant of the product line that does not include the feature *Addresses*. According to our experience, the *Variant View* is a visualisation that directly resembles the semantics of the transformations used for product derivation. To use this view, the developer first enables the *Variant View* via the respective button on the *FeatureMapper* toolbar and selects all features that should be part of the specific variant of the SPL.

Figure 4. Colours assigned to selected features in the feature model.

3.2.3 Context View

The *Context View* involves the colouring of the features in the feature model as well as the colouring of the model elements accordingly. In Figure 4 a feature model is depicted that has colours assigned to some features[1]. These colours are used by the tool to also colour the model elements that participate in the realisation of the features. In Figure 5 the model elements are shown that are rendered in the colour that is assigned to the corresponding feature. Note, that according to our argumentation in Section 2.3, mixing of

[1]Note that colours in this paper are indexed in order to be readable in black and white printouts.

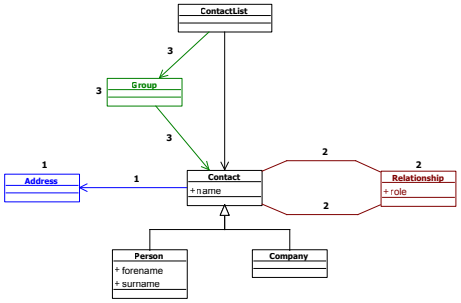

Figure 5. Model elements that are rendered in the colour that is assigned to the corresponding feature.

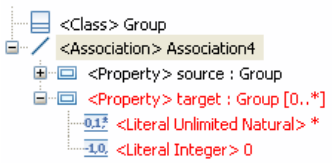

Figure 6. Model elements in a tree-based editor that are highlighted by the *Property-Changes View*.

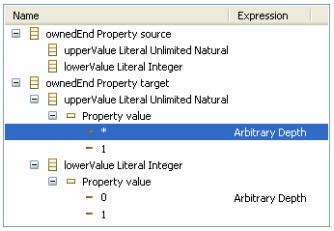

Figure 7. Detailed view on property changes.

colours is not supported. Instead, assigning colours to feature expressions is possible. In the *FeatureMapper* colours can be assigned via a dedicated command from the feature's context menu. An overview dialogue is also provided, which allows for inspection and management of all assigned colours and the corresponding feature expressions.

3.2.4 Property-Changes View

The *FeatureMapper* also allows for defining property-value mappings that change properties of model elements—i.e., cardinalities, names of elements, or initialisation values. Hence, means for highlighting the affected model elements are provided. This is of particular importance, since those changes are not directly visible in the other visualisations and, thus, need specialised handling. In the *Property-Changes View*, model elements with properties that are changed according to feature selection are highlighted. The highlighted elements can then be selected and a dedicated dialogue can be displayed, which gives an overview of the changes associated to the specific feature expressions. An example of the *Property-Changes View* is depicted in Figure 6, where the cardinality of the reflexive association of the class *Group* is highlighted to be subject for feature-dependent changes. Once this model element is identified, it can be inspected further by using a dedicated dialogue as depicted in Figure 7. According to this dialogue, the cardinality of this association is changed from one to many depending on the presence of the feature *Arbitrary Depth*.

3.3 Summary

The presented visualisations support the developer in understanding the mapping between features from a feature model and their realisation in solution models. Each visualisation provides a different view on the SPL and reveals properties of the system that are not visible in standard tools. In particular, each view helps at answering specific questions a developer may be interested in during development of an SPL:

- Which artefacts realise a specific feature? (*Realisation View*)
- Which artefacts are used in a specific variant of a SPL? (*Variant View*)
- Is the resulting variant still a valid model? (*Variant View*)
- Which features are interacting or communicating with each other? (*Context View*)
- Which features require property-value changes? (*Property-Changes View*)

Most importantly, the views can be interactively enabled and disabled in the *FeatureMapper* and, thus, visualisation can be controlled by the developer.

4 Related Work

As mentioned in Section 1, the work by Czarnecki and Antkiewicz [6] presents an approach, where features are mapped to UML models by *Presence Conditions* that are annotated to the model elements. This offers a basic means for visualisation, namely the mapping as a whole. However, *controlled visualisation*, to which we argue for in this paper, is not provided. It is still fully up to the developer to interpret and analyse the mapping, which can be a daunting and difficult task if models grow in size and complexity.

The approach presented by Botterweck et al. [2] provides various means to support the developer during product configuration by visualising dependencies between features and by colouring to indicate feature states. Additionally, features are decorated by iconic representations of common cardinalities. However, the approach does not include the visualisation of the mapping between the problem and solution spaces.

Kästner et al. [15, 16] provide a Colored Integrated Development Environment (CIDE) for mapping features to source code artefacts. They use a mapping to map features to elements of arbitrary abstract syntax trees. Similar to our tool, they provide colouring of software artefacts (source code snippets in this case) based on the feature mapping. Colouring also involves mixing of different colours, because the CIDE tool does not allow for feature expressions. However, property-value mappings are not supported. Most important, since current source code editors do not support changing the background colour of specific source code statements, the editors need to be adopted to provide the functionality.

In [17], Nestor et al. present a research agenda for the visualisation of variability in SPLE. They introduce a visualisation reference model that is a slightly adapted version of the reference model presented in [4]. As part of this model, a catalogue of visualisation interaction tasks is introduced which includes, for example, the tasks of filtering items, providing details on-demand, and viewing relationships among items. Our work contributes to this work by proposing concrete visualisations for its tasks and by providing an implementation of these visualisations.

5 Conclusion

We motivated the need for visualisations by the fact, that mappings between features and software artefacts become complex rapidly. We argued for *controlled* visualisations, because visualisations are according to Card et al. "*adjustable mappings from data to visual form*" [4, page 17] and, thus, need to be enabled, disabled, or parameterised by the developer. In this paper we presented two contributions to the area of visualisation of software product lines.

First, we presented *MappingViews*, a novel visualisation technique. It includes four basic visualisations which we found to be useful for tackling the increasing complexity in defining, understanding, and analysing mappings between features and software artefacts that realise those features in an SPL. The *Realisation View* allows for viewing all software artefacts that participate in the realisation of a feature. The *Variant View* provides a view on a concrete variant of the SPL based on a variant configuration. The *Context View* helps at understanding feature communication and feature interaction, while the *Property-Changes View* offers means to identify and change feature-dependent changes in software artefacts.

Second, we presented an implementation of the proposed visualisations in our tool *FeatureMapper*. The tool allows for mapping features to modelling artefacts that are expressed by Ecore-based languages. Furthermore, it provides means for mapping-based transformation of models (not covered in this paper). A screencast of the tool in action and related work can be viewed at `http://featuremapper.org`.

Our future work focuses on developing a visualisation framework for the *FeatureMapper*, where the presented visualisations adjust automatically when different transformational semantics are used in product derivation. At the moment, the visualisation assumes that each artefact that is assigned to a specific feature is simply removed if the feature is not present in a concrete variant of the SPL. This is true for the standard transformation that is included in the *FeatureMapper*. But since the *FeatureMapper* allows for using arbitrary interpretations of the mapping model by an Eclipse extension point, other transformational semantics are possible. For example, a more specific transformation may remove additional depended artefacts based on a mapping to a single artefact or may apply automatic repair actions to ensure well-formedness of the resulting model. These issues will be addressed by the envisioned visualisation framework.

We are currently also investigating additional views that reveal other interesting properties of an SPL. For example, a dedicated view for visualisation of feature granularity–as the *Realisation View* basically does–seems to be promising. This view could visualise the system depending on the granularity of the changes that are used to realise a feature. Since systems with many fine-grained changes tend to become complicated and decrease maintainability [15], hotspots of a system with many fine-grained changes need to be identified so that possibilities for refactoring towards more coarse-grained changes can be evaluated.

Additionally, we plan to further evaluate the approach by using the *MappingViews* visualisations with our *FeatureMapper* tool in a real-world case study which is currently under development within the feasiPLe project.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valueable comments and Jan Kopcsek for implementing *MappingViews* in the *FeatureMapper* tool.

This research has been co-funded by the German Ministry of Education and Research (BMBF) within the project feasiPLe (cf. `http://www.feasiple.de`).

References

[1] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 2003.

[2] G. Botterweck, D. Nestor, C. Cawley, and S. Thiel. Towards Supporting Feature Configuration by Interactive Visualisation. In *Proceedings of the 1st International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2007), collocated with the 11th International Software Product Line Conference (SPLC 2007)*, 2007.

[3] F. Budinsky, S. A. Brodsky, and E. Merks. *Eclipse Modeling Framework*. Pearson Education, 2003.

[4] S. L. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann Publishers, 1999.

[5] K. Czarnecki. Overview of Generative Software Development. In *Proceedings of the Unconventional Programming Paradigm*, volume 3566 of *LNCS*, pages 326–341. Springer, 2005.

[6] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In R. Glück and M. Lowry, editors, *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE'05)*, volume 3676 of *LNCS*, pages 422–437. Springer, 2005.

[7] K. Czarnecki and U. W. Eisenecker. *Generative Programming – Methods, Tools, and Applications*. Addison-Wesley, June 2000.

[8] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering (GPCE'06)*, pages 211–220, New York, NY, USA, 2006. ACM.

[9] S. Diehl. *Software Visualization*. Springer, 2007.

[10] feasiPLe Consortium. feasiPLe Research Project, July 2008. URL http://feasiple.de.

[11] FeatureMapper Project Team. FeatureMapper, July 2008. URL http://featuremapper.org.

[12] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 943–944, New York, NY, USA, May 2008. ACM.

[13] F. Heidenreich and C. Wende. Bridging the gap between features and models. In *2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07) co-located with the 6th International Conference on Generative Programming and Component Engineering (GPCE'07)*. Online Proceedings, 2007. URL http://www.softeng.ox.ac.uk/aople/.

[14] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented Domain Analysis (FODA) Feasibility Study. *Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA*, 1990.

[15] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE 2008)*, pages 311–320, May 2008.

[16] C. Kästner, S. Trujillo, and S. Apel. Visualizing Software Product Line Variabilities in Source Code. In *Proceedings of the 2nd International Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008), collocated with the 12th International Software Product Line Conference (SPLC 2008)*, 2008.

[17] D. Nestor, L. O'Malley, A. Quigley, E. Sikora, and S. Thiel. Visualisation of Variability in Software Product Line Engineering. In *Proceedings of the 1st International Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2007)*, Jan. 2007.

[18] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[19] The Eclipse Foundation. Graphical Editing Framework, July 2008. URL http://www.eclipse.org/gef/.

[20] The Eclipse Foundation. The Eclipse Platform, July 2008. URL http://www.eclipse.org.

[21] The Topcased Project Team. TOPCASED, July 2008. URL http://www.topcased.org.