Proceedings of the 1st International Workshop on Model-Driven Product Line Engineering (MDPLE'2009)

Mira Mezini, Danilo Beuche, Ana Moreira (Eds.)

CTIT PROCEEDINGS

feasiPLe Project

Organizing committee

- Mira Mezini, Technische Universität Darmstadt, Germany
- Danilo Beuche, pure-systems GmbH, Germany
- Ana Moreira, Universidade Nova de Lisboa, Portugal

General workshop organization

- Tom Dinkelaker, Technische Universität Darmstadt, Germany
- Martin Monperrus, Technische Universität Darmstadt, Germany
- Sebastian Oster, Technische Universität Darmstadt, Germany

Program committee

- Uwe Aßmann, Technische Universität Dresden, Germany
- Danilo Beuche, pure-systems GmbH, Germany
- Krzysztof Czarnecki, University of Waterloo, Canada
- Steffen Göbel, SAP Research, Germany
- Mira Mezini, Technische Universität Darmstadt, Germany (PC Chair)
- Martin Monperrus, Technische Universität Darmstadt, Germany
- Ana Moreira, Universidade Nova de Lisboa, Portugal
- Richard Paige, University of York, United Kingdom
- Klaus Pohl, University Duisburg-Essen, Germany
- Julia Rubin, IBM Research, Haifa, Israel
- Andreas Rummler, SAP Research, Dresden, Germany
- Andy Schürr, Technische Universität Darmstadt, Germany
- Nathan Weston, Lancaster University, United Kingdom
- Steffen Zschaler, Lancaster University, United Kingdom

Additional reviewers

- Tom Dinkelaker
- Florian Heidenreich
- Kim Lauenroth
- Ralf Mitschke
- Sebastian Oster
- Christian Wende

Participant List

- Aksit, Mehmet (University of Twente, The Netherlands)
- Balmelli, Laurent (IBM, USA)
- Beuche, Danilo (pure::systems, Germany)
- Bombino, Massimo (Artisan Software Tools)
- Botterweck, Goetz (LERO, Ireland)
- Buchmann, Thomas (University of Bayreuth, Germany)
- Cancila, Daniela (CEA, France)
- Champeau, Joel (LiSyC-ENSIETA, France)
- Dinkelaker, Tom (Technische Universitaet Darmstadt, Germany)
- Dotor, Alexander (University of Bayreuth, Germany)
- Galvao, Ismenia (University of Twente, The Netherlands)
- Garcés, Kelly (INRIA-EMN, France)
- Gondal, Ali (University of Southampton, UK)
- Hartman, Alan (IBM Research, Israel)
- Kirshin, Andrei (IBM Research, Israel)
- Mansell, Jason (European Software Institute, Spain)
- Monperrus, Martin (Technische Universitaet Darmstadt, Germany)
- Naeem, Muhammad (University of Leicester, UK)
- Olsen, Goran (SINTEF, Norway)
- Paige, Richard (University of York, UK)
- Siikarla, Mika (Tampere University of Technology, Finland)
- Trew, Tim (NXP Semiconductors)
- Trujillo, Salvador (IKERLAN, Spain)
- Van Baelen, Stefan (KU Leuven, Belgium)
- Wende, Christian (TU Dresden, Germany)
- Westfechtel, Bernhard (University of Bayreuth, Germany)

Table of Contents

Summary of the 1st International Workshop on Model-Driven Product Line Engineering

The 1st International Workshop on Model-Driven Product Line Engineering (MDPLE'2009) was held on June, 24th 2009, in the University of Twente (The Netherlands) in conjunction with the European Conference on Model-Driven Architecture (ECMDA'2009).

Model-Driven Software Development (MDD) as well as Software Product Lines (SPL) related concepts, techniques, and tools have without any doubts the potential to increase the productivity and quality of software engineering processes significantly. However, industry is struggling to adopt these ideas on a large scale. And from an academic point of view we still do not know how to integrate these two areas systematically, i.e. (1) how to adapt and apply MDD techniques for the development of software product lines and related engineering tools and (2) how to integrate SPL concepts for the development of families of models, metamodels, model transformations, and modeling languages. The MDPLE'2009 workshop was a place to publish work in this research field and to identify research opportunities.

This workshop brought together researchers, lecturers, graduate and Ph.D. students with industrial practitioners, who are interested in both software product lines (SPLs) and model-driven architecture (MDA). We have discussed the appropriateness of today's approaches, techniques, infrastructures, and language support and to share ideas from both communities. The workshop organization was sponsored by the feasiPLe research project, funded by the German Federal Ministry of Education and Research (BMBF), and the AMPLE project (EU FP6).

There were 26 participants in the morning session and 23 participants in the afternoon session. There were 2 invited talks, given by Danilo Beuche (pure::systems, Germany) and Mehmet Aksit (University of Twente), and 8 presentations of the accepted papers (6 full papers and 2 position papers). In order to identify research issues in software product line and model-driven engineering, there was a collective brainstorming discussion of 1h 30min at the end of the presentations. The process of the discussion was as follows:

- In the morning session, each participant was asked on a voluntary basis, to write down anonymous polemical questions about the state-of-the-art of software product lines.
- At the beginning of the discussion, each participant was asked to introduce himself and to list its main research interests w.r.t. SPLE.
- Then, we collectively clustered the collected questions and research interests as a set of research topics.
- We discussed the topics one-by-one. The participants were asked to give their opinion on each research topics.

The workshop enabled us to establish a list of what are the important and difficult research topics in SPLE:

- There is too much attention given on the technology supporting SPLs (e.g. model-driven) and not on the methods to analyze and identify the business value of SPLs. This was nicely summarized as a need for moving from model-driven to business-driven SPLs. Also, such methods would bridge the gap between the researchers, tool providers, and their customers.
- Research has proposed many feature model extensions. However, from the viewpoint of industry, there is a need for a standardized feature metamodel.
- There is not only variability in products but also in product generators (e.g. DSLs, model transformation and metamodels). The community has to provide concepts and tools to build adaptable and parametrized DSLs and model transformations.
- The space of software products is usually considered as finite and closed. In an open and dynamic context such as a service oriented architecture, the known techniques are not applicable. There is need for studying variable service oriented architectures.
- By marrying formal methods and product generation, it could be possible to achieve "correctness by construction". However, from a pragmatic viewpoint, runtime verification of products is also an open issue.
- There is a need for supporting traceability of design rationales all along the product specification and generation.
- The one to one mapping between features and implementation artifacts is considered as a good design pattern of SPLs. However, the technology rarely supports the vision. There is still room to improve existing techniques (e.g. virtual classes, mixins, aspects, (meta)model composition) to fulfill this vision.
- Important yet unaddressed issues in SPLs are related to usability and cognitive limits of users and tools (real world SPL models can not fit on a single screen). There is need for exploring new visualization of both the SPLs (e.g. visualization of feature models) and the products (e.g. visualization of configurations). Also, it has been recognized that interactive approaches to specifying products is a hot topic.

To conclude MDPLE'2009, we hope this compiled research agenda will help graduate students to identify relevant topics, researchers to write proposals and the the whole community to identify potentials of future collaborations.

Interactive Techniques to Support the Configuration of Complex Feature Models

Goetz Botterweck[1] and Denny Schneeweiss[2] and Andreas Pleuss[1]

[1] Lero, University of Limerick, Ireland,
{goetz.botterweck|andreas.pleuss}@lero.ie
[2] BTU Cottbus, Germany,
denny.schneeweiss@tu-cottbus.de

Abstract. Whenever a software engineer derives a product from a product line, he has to resolve variability by making configuration decisions. This configuration process can become rather complex because of dependencies within the variability model and knock-on effects and dependencies in other related artefacts. Because of the limited cognitive capacity of the human engineer, this complexity limits the ability of handling product lines with large configuration spaces. To address this problem we focus on techniques that support the interactive configuration of larger feature models, including (1) visual interaction with a formal reasoning engine, (2) visual representation of multiple interrelated hierarchies, (3) indicators for configuration progress and (4) filtering of visible nodes. The concepts are demonstrated within S^2T^2 Configurator, an interactive feature configuration tool. The techniques are discussed and evaluated with feature models, however, we believe they can be generalised to other models that describe configuration choices, e.g., variability models and decision models.

1 Introduction

In Software Product Line (SPL) Engineering we are dealing with interrelated, complex models. A common concept for modelling a product line are feature models. They are usually interpreted as a tree structure consisting of feature-subfeature and group-member relations. In addition, there are other relationships across the hierarchy and across different levels (e.g., *requires*, *mutex*). The most common visual representation for such models is a graph [1,2].

When deriving products from a product line, the software engineer has to resolve variability by making configuration decisions. This configuration process and the decisions can become rather complex because of the dependencies within the variability model and knock-on effects and dependencies in other related artefacts that represent the product line. Because of the limited cognitive capacity of the human engineer, this complexity limits the ability of handling product lines with large and complex configuration spaces.

Hence, there is a need for tool support for product configuration. Existing commercial feature modelling software like pure::variants [3] provide basic

graph-based visualisations of feature models. Heidenreich et al. [4] introduce FeatureMapper, which supports mapping features to model elements in arbitrary (EMF-based) models. The mappings are expressed by colouring the elements corresponding to each feature. However, there is still a lack of more advanced visual and interactive tool support for handling product derivation.

We address this problem with S^2T^2 *Configurator* a feature configuration tool which integrates an interactive visual representation of the feature model with a formal reasoning engine, that calculates consequences of the user's decisions and provides formal explanations.

In addition, we support to visualise the relationships between selected features and further aspects of the software product line, like a component model specifying the architecture. This requires to present multiple hierarchies (e.g., features and components) as well as the relationships between them, e.g., which features are implemented by which components.

In earlier work [5] we discussed the visualisation of product lines. In [6] we introduced the software architecture for our Configurator tool, designed as a chain of configurator components, to allow flexible integration of different models and a formal reasoning engine. In this paper we focus on interaction techniques which support the interactive configuration of larger feature models, including the visual interaction with the reasoning engine (section 3.1), visual representation of multiple interrelated feature trees (section 3.2), indicators for configuration progress (section 3.3) and filtering (section 3.4). All described techniques have been implemented in a prototype which is available for download[3].

2 Analysis of Possible Solutions

To address the configuration of complex feature models, we evaluated visualisation techniques with respect to their suitability for typical structural characteristics of such models. Feature models are usually structured as hierarchies, made up of feature-subfeature and group-member relations [1,2]. In addition, depending on the type of feature model, there can also be arbitrary relationships between models elements, independent of the main hierarchy (cross-tree constraints or *intra-model relations*). Further challenges are given by large models (not fitting on the canvas) and multiple, interrelated models with relationships between them (*inter-model relations*).

We classify the evaluated approaches into techniques for visualisation of tree hierarchies (section 2.1), interaction techniques for large data structures (section 2.2), and approaches focussing on multiple hierarchies (section 2.3).

2.1 Visualisation of Tree Hierarchies

The area of Information Visualisation provides various alternative approaches to visualise tree hierarchies:

[3] http://download.lero.ie/spl/s2t2/

Space-filling visualisations such as tree-based maps [7] or Sunbursts [8] provide an insight into the quantitative relations within a hierarchy. Although these types of visualisations are quite effective in this respect, feature model visualisations in general do not benefit from it. It is difficult to enrich them with additional information (i.e., text rendering in small cells is problematic) and displaying relations between elements leads in almost all cases to visual clutter. This problem gets even worse with multiple hierarchies.

Three-dimensional visualisations such as Cone Trees [9] are problematic when applied in interactive configurations since certain elements may be hidden by others. Moreover the navigation and orientation within a 3D environment is challenging for most users.

In general, various user tests on tree visualisations have shown that mature common tree visualisation systems (like file explorers) often perform as good or even better as alternative visualisations (e.g., [10]). Thus, it seems reasonable for feature model visualisation to decide for a conventional 2D tree visualisation and put effort on its optimisation and enhancement. As a first step, techniques from Graph Drawing can be used to reduce the graph's complexity, e.g., by minimising intersections, providing a well-structured spatial layout or reducing the required area.

2.2 Interaction Techniques for Large Data Structures

The second important area arises from the interactive nature of the Configurator tool. Certain interaction techniques enable the user to focus on the information in the centre of his or her interest while muting currently unimportant information. Common techniques in these area are for instance [11]:

The ***overview+detail*** interface design, which is characterised by the use of both an overview and a detail view of the same information space, in *separate* presentation spaces (spatial separation of focus and detail).

focus+context interfaces integrate both details and overview seamlessly into a single display (spatial separation, but with a seamless integration).

A ***zooming*** interface design provides access to overview and detail in temporal separation. Here the users first zoom out to get an overview and then zoom in to see the details.

These are promising approaches and are applied in our tool. To identify which elements are part of the user's focus and which elements are outside of the scope of interest at one point of time it is necessary to consider domain-specific knowledge and the semantics of the models.

2.3 Approaches for Visualising Multiple Hierarchies

Approaches focussing on multiple hierarchies are useful when visualising the relationships between features and other models as explained above. Robertson et al. define polyarchies [12], multiple hierarchies that share nodes. They describe the visualisation design and a system architecture for displaying polyarchy data

from a set of hierarchical databases. They use various animated transitions when switching between the related hierarchies to allow the user to keep context. Polyarchies are somewhat similar our multiple related hierarchies, but lack the intra-model relations and the aspect of progressing configuration.

In later work the authors address visual mappings between two hierarchical schemas [13]. They use auto-scrolling and focusing-techniques to display distant related elements within the boundaries of the screen. Again the data structures are different from feature models (no intra-model relations etc.); nevertheless some of the interaction techniques could be adapted for feature models.

3 Interactive Techniques in the S^2T^2 Configurator

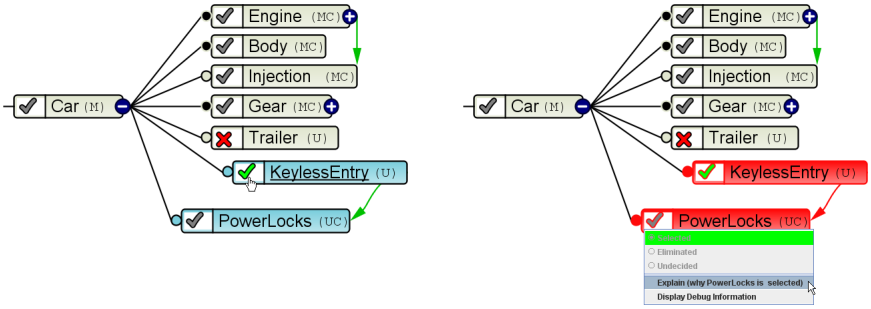
As argued above, we chose a two-dimensional tree hierarchy as the basic visualisation. The configurator supports all basic interaction techniques for tree structures like *collapse/expand functionality*, *panning*, and *seamless zooming*. The following sections describe the more advanced features in greater detail.

3.1 Interaction with Formal Semantics

The S^2T^2 Configurator integrates a reasoning engine that supports interactive functionality such as calculating the consequences of his or her decisions based on the formal semantics of the models [6]. We interpret a feature model as (1) a set of features and (2) a set of constraints over these features. Moreover, (3) we interpret features as variables with value domains limited by the given constraints. For instance, in boolean feature models, there are four potential configuration states: *Undecided* $\{true, false\}$, *Selected* $\{true\}$, *Eliminated* $\{false\}$, and *Unsatisfiable* $\{\}$. During the configuration process, each user decision–and the consequences calculated from it–are interpreted as additional constraints further reducing the available values in the domains. When the configuration process is completed and all variability has been resolved, there is exactly one possible value left for each feature. Similar concepts can be defined for non-boolean feature models by allowing larger value domains (e.g., to specify the maximum price for a car).

Figure 1 shows screenshots from our tool with a small example feature model. The basic visualisation follows common tree-based feature model representations. Additionally, a symbol represents the feature's state: a check mark indicates that the feature is selected, while a cross indicates that the feature is eliminated. The user can modify the feature's state by clicking on this area. Symbols in grey colour indicate that the state is already given by constraints and cannot be changed.

When the model is initially loaded and after each user decision, the reasoning engine automatically calculates the consequences and applies them to the model. In figure 1(a) the Configurator inferred that `Injection` has to be selected as it is required by the mandatory `Engine`. The screenshot shows the moment where the user sets the feature `KeylessEntry` as selected. As it requires the feature



(a) Calculated consequences. (b) Explanations for consequences.

Fig. 1. Configuration with consequences and explanations.

`PowerLocks`, the Configurator automatically sets `PowerLocks` as selected as well. Whenever the Configurator automatically applies changes, this is indicated with an animated blue decoration.

The annotations on the right-hand side in each node denote whether a feature's state has been set by the user (U), is a consequence from a user decision (UC) or is specified by the model (M).

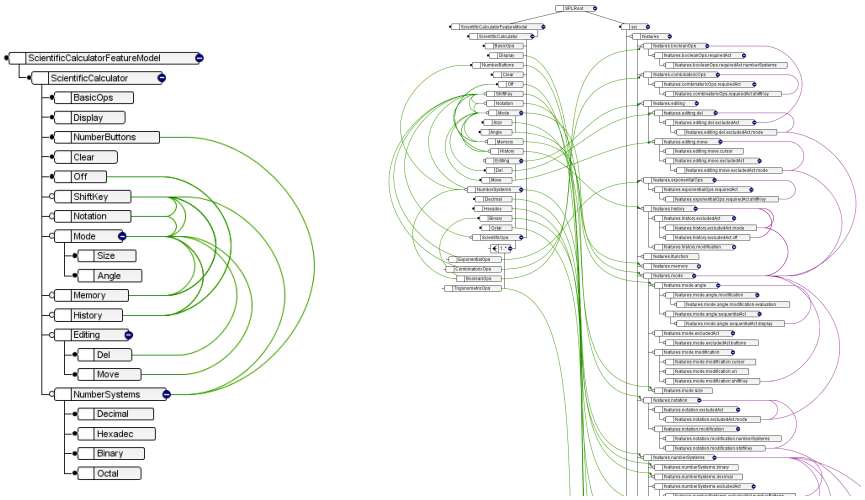
Although in such simple examples the reasons for a consequence are obvious, in more complex feature models the user might not immediately comprehend the situation. Hence, the Configurator supports visual explanations, which are based on formal explanations calculated by the reasoning engine. This enables the user to ask for an explanation (using the context menu) why a certain feature is selected/eliminated. For instance, in figure 1(b) when the user asks why `PowerLocks` is selected automatically, the Configurator will respond by highlighting `KeylessEntry` and the `Requires` edge between `KeylessEntry` and `PowerLocks`.

3.2 Multiple Related Models

Due to its modular design described in [6], S^2T^2 Configurator supports the seamless integration of different kinds of models. As discussed earlier, one goal is to visualise multiple interrelated hierarchies while supporting the user's orientation within the complex model, e.g., by avoiding intersections of edges and similar visual clutter.

The Configurator supports different tree-layouts, e.g., a vertical tree layout with indentations, see figure 2(a). Hierarchy edges are rendered rectilinear while additional relations are shown as semicircles with adjusted diameters. Thus, we avoid crossing feature with edges and two edges only intersect if their nodes are interleaved. Moreover, edges that start and end in similar positions can be easily distinguished in most cases since they usually differ in their horizontal extend.

For the visualisation of two interrelated models we adapted this approach by positioning the trees side-by-side and changing the orientation of the left model



(a) Vertical Tree Layout with semicircles.

(b) Multiple feature hierarchies and their relationships

Fig. 2. Approaches to avoid clutter in hierarchies with cross-model-relations

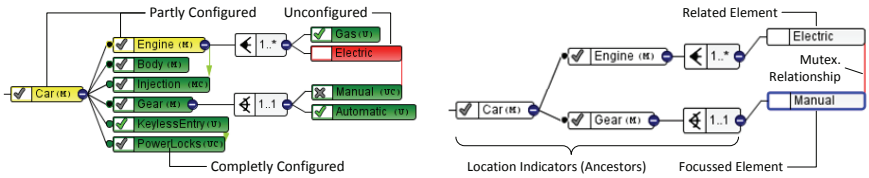
to be rendered with right-aligned nodes, see figure 2(b). Intra-model relations are rendered as semicircles on the outer side of each model, inter-model relations as cubic splines connecting both models. With this layout variation the clutter is kept in reasonable bounds even with two interrelated models.

Our tool supports displaying more than two hierarchies, but then it becomes hard to avoid intersections. Hence, we recommend to avoid such scenarios by letting the user interactively chose two models to be displayed on the left- and right-hand side; similar to common file manager tools like Norton Commander.

3.3 Progress Indicators

With progress indicators we aim to distinguish areas that have been configured from areas that still need attention. A feature node is *configured* if all decisions with respect to this feature have been made. In general, this is the case if the value domain has been reduced to one element; for boolean feature models this means that the feature is either *Selected* or *Eliminated*.

If we aggregate the configuration state of all nodes in a subtree, we can distinguish between three levels of progress for the subtree, *Unconfigured*, *Partly configured*, and *Completely configured*. If we apply results from research on colours and selective attention, we can colour the root of each subtree accordingly and draw the user's attention to those locations where decisions are pending. When the configuration process progresses these colours change and become less emergent. Figure 3(a) shows this in a simple feature model.



(a) Progress indicators. (b) Filtering.

Fig. 3. Progress indicators and filtering.

3.4 Filtering

Another technique to improve the interaction with complex models is to (1) differentiate between elements that are *currently* relevant ("in focus") and those that are not and (2) use this categorisation in appropriate interaction techniques, e.g., by filtering out non-relevant information.

To find relevant nodes, we partitioned the model elements into four sets: *Focussed elements*, *Related elements* (linked directly or indirectly to focussed elements), *Location indicators* (necessary to understand the logical position within the overall model, e.g., all ancestors of focussed elements), and *Other elements*.

Whenever the user focusses on one feature (double click) the Configurator calculates the (directly) related elements. The user can then hide all other elements to concentrate on the current configuration decision (see figure 3(b)). We also display the ancestors of the focussed element to show the relative position within the model.

4 Conclusions and Future Work

In this paper we have discussed techniques that support the interaction with and configuration of complex feature models. The techniques have been demonstrated with the research prototype S^2T^2 Configurator. A beta version of the prototype is available at `http://download.lero.ie/spl/s2t2/`.

After the initial design and realisation of the discussed techniques we gained more insights by experimenting with three test cases: (1) Generated feature models with varying sizes and distributions of element types (2) a product line of parking assistant applications [14] and (3) a calculator case study [15], which contains a feature model and a model of the implementation as well as feature-implementation mappings. In general, this allowed us to perform a first internal evaluation of our approach. In summary, the introduced techniques seemed to improve the situation with respect to the size of models that can reasonably be handled. As next steps we plan to perform user tests and provide a formal evaluation to substantiate this with empirical evidence. Based on that experience we intend to extend the Configurator, e.g., defining additional filtering rules based on common tasks in product-line engineering.

Acknowledgments This work was partly supported by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – The Irish Software Engineering Research Centre (http://www.lero.ie/).

References

1. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature oriented domain analysis (FODA) feasibility study. SEI Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute (1990)
2. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. Software Process Improvement and Practice **10**(1) (2005) 7–29
3. Beuche, D.: Variants and variability management with pure::variants. In: 3rd Software Product Line Conference (SPLC 2004), Workshop on Software Variability Management for Product Derivation, Boston, MA (August 2004)
4. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: ICSE Companion '08. (2008) 943–944
5. Botterweck, G., Thiel, S., Nestor, D., bin Abid, S., Cawley, C.: Visual tool support for configuring and understanding software product lines. In: SPLC 2008, Limerick, Ireland (September 2008)
6. Botterweck, G., Janota, M., Schneeweiss, D.: A design of a configurable feature model configurator. In: VAMOS 2009. (2009)
7. Johnson, B., Shneiderman, B.: Tree-maps: A space-filling approach to the visualisation of hierarchical information structures. IEEE Visualization (October 1991) 189–194
8. Stasko, J., Zhang, E.: Focus+context display and navigation techniques for enhancing radial, space-filling hierarchy visualizations. In: Proc. of IEEE Information Visualization 2000, Salt Lake City, UT (2000) 57–65
9. Robertson, G.G., Mackinlay, J.D., Card, S.K.: Cone trees: animated 3d visualizations of hierarchical information. In: CHI '91. (1991) 189–194
10. Kobsa, A.: User experiments with tree visualization systems. In: INFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization, Washington, DC, USA, IEEE Computer Society (2004) 9–16
11. Cockburn, A., Karlson, A., Bederson, B.B.: A review of overview+detail, zooming, and focus+context interfaces. ACM Comput. Surv. **41**(1) (2008) 1–31
12. Robertson, G., Cameron, K., Czerwinski, M., Robbins, D.: Polyarchy visualization: Visualizing multiple intersecting hierarchies. In: CHI'02. Visualizing Patterns, ACM Press (2002) 423 – 430
13. Robertson, G.G., Czerwinski, M.P., Churchill, J.E.: Visualization of mappings between schemas. In: CHI '05, New York, NY, USA, ACM (2005) 431–439
14. Polzer, A., Kowalewski, S., Botterweck, G.: Applying software product line techniques in model-based embedded systems engineering. In: MOMPES 2009, Workshop at ICSE 2009, Vancouver, Canada (May 2009)
15. Lee, K., Botterweck, G., Thiel, S.: Aspectual separation of feature dependencies for flexible feature composition. In: COMPSAC 2009, Seattle, WA (July 2009)

Constraints for a fine-grained mapping of feature models and executable domain models

Thomas Buchmann and Alexander Dotor

Angewandte Informatik 1, Universität Bayreuth
D-95440 Bayreuth
firstname.lastname@uni-bayreuth.de

Abstract. In the past, several approaches have been made to combine feature models and domain models on the level of class diagrams. But the model-driven development approach also covers models that describe the behavior of a software system. In this paper we will describe a mapping of feature configurations to executable model elements which is one step towards an overall model driven process for product line engineering. We will specify consistency constraints that have to be met to ensure model correctness, and we will discuss the problems that arise during the final model-to-code transformation.

1 Introduction

The term model-driven development [1] of software systems describes the creation of systems by specifying models instead of writing code. Usually these models are created in CASE tools which provide class diagrams to model the static structure of a software system. These kind of diagrams lack the ability to model variability. In the context of software product lines [2], feature models are used to model variability in a family of software systems. Recently some approaches have been made to combine feature models and domain models created with CASE tools [3], [4], [5]. But model-driven development is more than just creating models that describe the static structure of a system - the behavior has to be described as well. In our current work, we develop a model-driven product line for *Software configuration management* (SCM) systems. The benefits of a model driven approach are (1) making the underlying models explicit, rather than having them implicitly defined in the program code, (2) providing reusable modules which can be combined in a flexible way through defining orthogonal components which are loosely coupled and (3) support rapid construction of new systems by providing a product line. The domain model of our product line consists of both class diagrams and behavioral diagrams. In the following we will discuss our approach to map features on elements of both structural and executable behavioral models which is one step towards a well-defined model-driven development process for software product lines.

2 Background

In our work we try to bridge the gap between features in a variability model and model elements of a system family. In a model-driven process a system configuration should

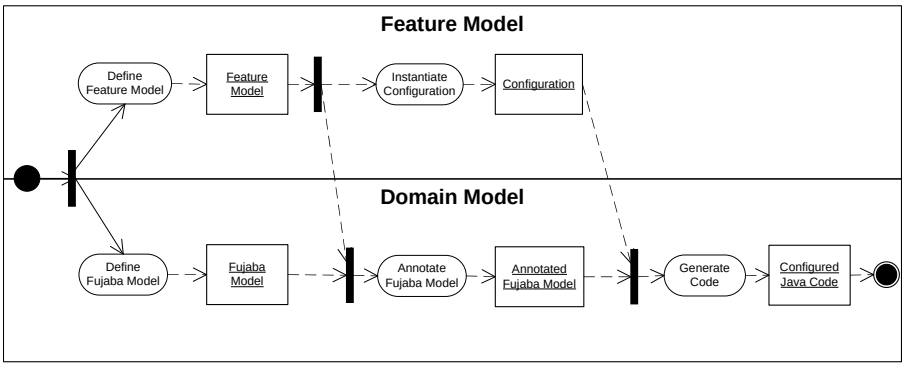


Fig. 1. The MDD process which combines Feature models and domain models using annotations

be mapped automatically to the domain model, and code for the specific feature selection should be generated. We use the CASE tool Fujaba [6] to create the executable domain model for the modular SCM system mainly because of its support for executable models. The added value compared to other CASE tools is provided by *story diagrams* (see p. 6) and their compilation into executable code. In contrast, code generation from class diagrams is supported by a large number of other CASE tools as well, like e.g., in the Eclipse Modeling Framework [7]. Due to space restrictions, we can not provide detailed information about the domain model itself. Information how the domain model has been designed to support orthogonal combination of features and how the behavioral model may be defined in a graphical notation at a level of abstraction above the (generated) program code, can be found in [8], [9] and [10]. Feature modeling was applied to define common and discriminating features of SCM systems. In the context of product line engineering, feature models are widely used. So far, we have defined no constraints on the combination of features (apart from those constraints which are expressed directly in the feature diagram itself). An essential goal of our project is building a product line for SCM systems where features may be combined as freely as possible. However, the major effort has to be invested into the design of a system which actually supports the features defined in the feature model. Defining the feature model itself is fairly easy. In our approach the domain model was annotated manually with features, to establish a mapping between elements of the domain model and the feature model respectively. These annotations can be performed on any level of granularity. On a coarse-grained level, units such as packages, classes and associations are decorated with features, whereas on a more fine-grained level, attributes, methods and even story patterns etc. can be decorated. The coarse-grained approach keeps the multi-variant architecture manageable. But it is up to the modeler's discipline to use the feature annotations carefully. In an extensive way of using feature annotations, the modeler may easily lose track and may face a degree of complexity which cannot be managed anymore. During the code generation process, these feature annotations are evaluated against a given system configuration which is specified in FeaturePlugin [3] and only code for selected elements is generated (see Fig. 1).

3 Technical Problem

To generate code for a system configuration, a mapping has to be established between features of the feature model and the model elements of the domain model. We chose **tagged values containing the name of the features** to realize this mapping. If a feature is selected in a configuration the model elements with its name have to be part of the configured domain model. Each model element can be tagged by multiple features, in which case they are evaluated analogous to a logical *and* (i.e., all features have to be selected). This means also, that untagged model elements are always part of the configured model. The mechanism to tag the model elements varies slightly between different metamodels. In UML, stereotypes with feature names can be used ([11], pp. 651), while e.g., Ecore provides annotations ([7], pp. 119).

Our ultimate goal is to generate executable code from the configured model. Therefore, the transformation of the domain model into a configured domain model can be viewed as analogy to a preprocessing step of a compiler – a **code generation preprocessor**. As a consequence we have to deal with the same problems as compiler preprocessors: it is easy to produce syntactically and semantically wrong code.

Take the following example: Figure 2 shows a class diagram whose class `B` has been tagged by a feature named *sampleFeature*. As long as the sampleFeature is part of the configurations everything runs well, but as soon as it is omitted several problems occur (assuming only class B is omitted from the configured model):

1. Both generalizations have either no target or no source.
2. Both associations have only one member end.
3. Both class X and class Y have a Property of a non-existing type.
4. print_J_from_A in C fails during opaque expression analysis step (e.g., during compile time, see [11], pp. 101), as C is no A anymore, so j cannot be accessed.
5. print_I_from_B in Y fails also, as B is not associated with X anymore, so i cannot be accessed

This example shows that several syntactical constraints have been violated in the configured domain model, e.g., associations and generalization with only one end node. These problems are **violating UML metamodel constraints**. But another kind of problem can only be detected **during compile time**, e.g., the broken inheritance hierarchy.

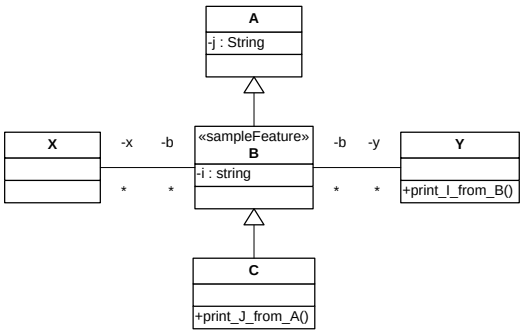


Fig. 2. Example of a tagged class diagram

This leads to the question, if there is a set of consistency rules to maintain syntactical correctness both when configuring a tagged domain model and when generating code for the configured domain model.

4 Solution

If the tagged domain model is syntactically correct all errors in a configured domain model result from the removal of tagged elements (this is quite similar to the deletion of model elements). An untagged element is present in every configured domain model. A tagged element is only present in the subset of configured domain models associated with the conjunction of the features it has been tagged with. So, if another element depends on a tagged element the following rule holds:

> **Rule 1:** If model element A depends on model element B, the set of configured domain models containing A must be a subset of the set of configured domain models containing B.

Or, thinking in terms of tags, the constraints must ensure that every tag on an element is present on its dependent elements, i.e., the tag set on element B must be a subset of element A[1].

4.1 Constraints for UML metamodel violations

This set of constraints is obtained by analyzing the UML Superstructure Specification [11]. All these rules works transitively, i.e., they have to be applied until no further tags are added.

> **Rule 1.1:** If an element is tagged all owned elements are tagged, too.

The basic `Element` of the UML Superstructure Specification introduces an aggregation between an `owner`-Element and its `owned elements` ([11], p. 25). By analyzing the inheritance hierarchy of the UML Superstructure we can define following dependencies (see Table 1).

> **Rule 1.2:** If the target of a directed relationship is tagged the relationship is tagged, too.

Each `DirectedRelationship` must have a source and a target ([11], p. 25). The source of a `DirectedRelationship` is always the owner, so rule 1 insists that a tagged source implies a tagged `DirectedRelationship`. This is also the case if the target is tagged which is demanded by this rule. The dependencies for the concrete elements are shown in Table 2.

> **Rule 1.3:** If the member end of an association is tagged the association is tagged, too.

[1] Please note that the direction of the subset relation has changed, because more tags mean less configured domain models.

Tagged type	Elements to be tagged (type)
Association	owned rules (Constraint), outgoing imports (ElementImport or PackageImport), generalizations to super-associations (Generalization), non-navigable or n-ary roles (Property)
Class	nested classes (Class), owned rules (Constraint), outgoing imports (ElementImport or PackageImport), generalizations to superclasses (Generalization), defined operations (Operation), defined attributes (Property)
Constraint	constraint definition (ValueSpecification)
ElementImport	*none*
Generalization	*none*
Operation	pre-/post and body conditions (Constraint), parameters (Parameter)
Package	contained associations (Association), contained classes (Class), owned rules (Constraint), outgoing imports (ElementImport or PackageImport), sub-packages (Package), outgoing package merges (PackageMerge)
PackageMerge	*none*
PackageImport	*none*
Parameter	default value (ValueSpecification)
Property	default value (ValueSpecification)
ValueSpecifications[2]	*none*

Table 1. Tag propagation Table for concrete UML class diagram elements (Rule 1.1)

Each Association requires at least two member ends. So, in case of tagged properties, the tags have to be propagated to the association that ends at this property.

Rule 1.4: If an association is tagged the member ends are tagged, too.

If an end of an Association is navigable it is owned by the appropriate Class. In this case a tagged association requires its member ends to be tagged.

Rule 1.5: If a type is tagged all typed elements of this type are tagged, too.

If a class is tagged which is a target of an uni-directional Association there is no link to the Property that holds a reference to the source class, because this direction is non-navigable (see [11], pp. 123). So, the Property of the source class that references the tagged class is only linked to the tagged class via the type-Association which is used in this rule.

4.2 Constraints for story diagram metamodel violations

UML provides several behavioral models but – except for state charts – there are no means to generate executable code. Instead, each Operation has a methodbody-string

[2] ValueSpecification is actually an abstract class that represents expressions consisting of various literals and so called *opaque expressions*. These expressions are strings associated with a language, e.g., java source code or OCL expressions that come with their own validator (i.e., java compiler or OCL checker). See [11], pp. 28 and pp. 101 for a complete definition.

Tagged type	Elements to be tagged (type)
`Association`	incoming element import (`ElementImport`), generalization from sub-association (`Generalization`)
`Class`	incoming element import (`ElementImport`), generalization from sub-class (`Generalization`)
`Package`	incoming imports (`ElementImport` or `PackageImport`), incoming package merges (`PackageMerge`)

Table 2. Tag propagation Table for concrete UML class diagram elements (Rule 1.2)

that is validated by a languages specific tool (e.g., a java compiler) [11] (pp. 101). The CASE tool *Fujaba* provides behavioral modeling through **story diagrams**, to specify the body of a method and generate executable code. Story diagrams are activity diagram with two kinds of nodes: Statement activities and story patterns. The first consists of a fragment of Java code, allowing for seamless integration of textual and graphical programming. The latter is a communication diagram composed of objects and links. Furthermore, objects may be decorated with method calls. Elements with dashed lines represent optional parts of story patterns. A crossed element means that the story pattern may be applied only when the respective element does not exist. In addition to method calls, a story pattern may describe structural changes: Objects and links to be created or deleted are decorated with the stereotype `<<create>>` (green color) or `<<destroy>>` (red color), respectively. Furthermore, **:=** and == denote attribute assignments and equality conditions, respectively.

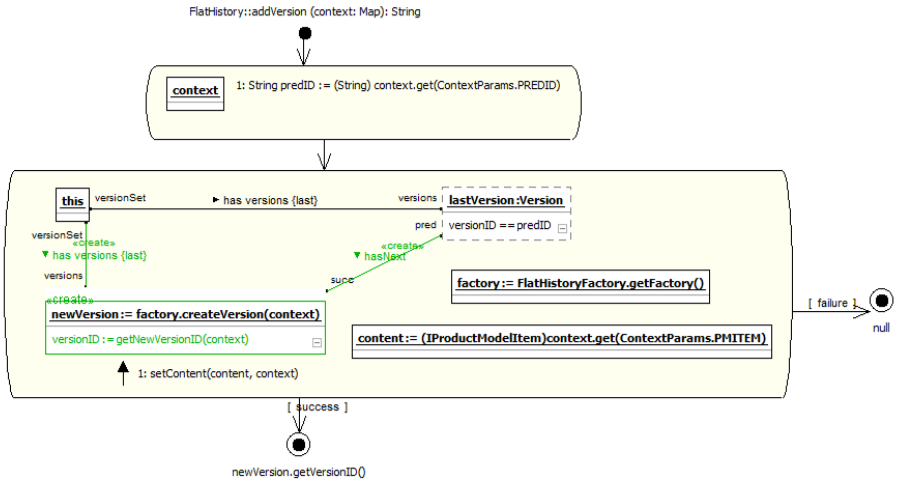


Fig. 3. Example of a Fujaba Story diagram

Figure 3 shows a story diagram that adds a new version to a single-dimensioned version history (i.e., a kind of linked list). The first activity is a collaboration call that retrieves the ID of the previous version (`predID`) out of the `context`-Parameter. The second is a story pattern. It creats a `newVersion` and adds it as last element to the `versionSet`. If a `lastVersion` exists already (note this object is optional) whose

`versionID` equals the `predID` it becomes the predecessor of the `newVersion`. Ultimately the `content` is added via method call to the `newVersion`. Depending on the outcome of this pattern (successful match or failure) the control flow continues to a stop node that either returns the new `versionID` or `null`. In Fujaba each metamodel element has a reference to its instances, e.g., a class to its objects, etc. The class *Version* has two instances in our example (Fig. 3): *lastVersion* and *newVersion*. Instead of passing the validation to a compiler it is now possible to think in terms of behavioral model elements and to define further rules to ensure their correctness.

Rule 1.6: If an UML element is tagged all its instances are tagged, too.

When an element of the class diagram is tagged all its instances in all story diagrams have to be tagged.

Rule 1.7: If a UML Operation is tagged its story diagram is tagged, too.

Each story diagram defines the behavior of a single operation. By tagging an operation its behavioral specification has to be tagged, too.

Rule 1.8: If an UML element is tagged all instances of owned elements of its superclasses are tagged in story patterns of its subclasses, too.

This solves the interrupted inheritance tree problem, by tagging all instances of elements that are not inherited anymore.

If we apply the above rules on the example given in Figure 2, we get the following result: Rule 1.1 implies that the properties *x* and *y* are tagged, as well as the generalization to class A. The generalization from class C to class B is tagged according to rule 1.2. Applying rule 1.3 results in tags on both associations. The opposite navigation ends (*b*) are tagged because of rule 1.4.

4.3 Open Problems and Limitations

The current version of the Fujaba story diagram metamodel does not link every story diagram element to its class diagram element, e.g., constraints, transition guards and collaboration statements. And even after a metamodel expansion, Fujaba still allows statement activities which contain arbitrary strings. A way to ensure syntactical correctness of the generated code is to avoid these elements altogether, or to generate and compile the configured model in the background. Another limitation is the inability to specify variants of model elements (both in UML and story diagrams) as both metamodels are not designed for this purpose, e.g., it is not possible to define cardinality variants for association ends or attribute assignment variants for objects.

5 Related Work

Czarnecki et al. describe in their work about Mapping Features to Models [12] a way to establish a bidirectional mapping between feature models and Ecore elements, based on

Ecore class diagrams (see [4]). It uses many of the same notations and display elements as a previous version named *FeaturePlugin* [3]. Unlike FeaturePlugin, which focuses strictly on feature modelling in an isolated context, Ecore.fmp aims to create Ecore compliant class diagrams out of existing feature models and vice-versa. In the current version of Ecore.fmp, the creation of a feature model from an existing Ecore model is not yet supported properly. The creation of Ecore model files out of feature models also still needs to be implemented. Since it is tightly coupled with Ecore, it does not support arbitrary EMF-models or even executable models.

In his work, Florian Heidenreich developed a set of Eclipse plugins that also allows the user to establish a mapping between features and feature realisations (i.e., model elements) [13]. The underlying model (feature realisation) can be defined in arbitrary Ecore-based languages. It provides four different kinds of views, that visualize the current feature selection in different ways [14]. The plugin aims at supporting the developer in the complex task of defining mappings between features / configurations and their realizations. However, support for executable models is very limited due to Ecore. E.g., statecharts, which do not have the same expressive power as Fujaba's story-diagrams, can be used to model the behaviour. Furthermore, both Ecore.fmp and Featuremapper model requirements and components on the same level of abstraction. Ecore.fmp also has no direct support for modeling in the large [15].

There are also some commercial tools, that support modeling a product line by specifying feature models, like pure system's *pure::variants*. These tools do not provide a model-driven process to develop a product line in a model-driven way. They only cover a small part of the product line process - variant management. The configuration of the final product is done during runtime, for example by specifying `defines` which are passed to a C/C++ preprocessor, or by selecting certain sourcecode files. *pure::variants* does not support the mapping of features to model elements.

6 Conclusion

In this paper we presented a novel way to combine feature modeling and model-driven software development, especially when creating executable models. The mapping between features and model elements was done using model annotations. A configuration of selected features is used as an input to a special preprocessor which is started before the actual code generation process. This preprocessor passes model elements with matching feature annotations to the code generator, and drops model elements that do not match. In that way it is possible to establish a mapping between feature model elements to elements in executable models on any level of granularity. The code that is delivered to the customer only contains the fragments which are neccessary for the desired configuration, in contrast to the process of runtime configuration and deploying the complete codebase [10]. We deduced several rules to ensure syntactical correctness of the annotated domain model. These rules have been implemented in a plugin which ensures consistency of the model when generating code. Current work is addressed to integrate the plugin into the Fujaba editor, to allow using our consistency checks during the edit process.

At the moment, our preprocessor only accepts configurations that have been created using the FeaturePlugin [3] plugin. We need to build different import modules to read configurations created with other feature modeling tools like FeatureMapper [5], Ecore.fmp [4] or commercial tools like pure::variants etc. We will also investigate to which level of granularity it is possible to annotate the executable model. It is possible to annotate the static structure at any level of granularity (Classes, methods or even attributes), but does the same also hold for story patterns? E.g., we will try to find out if it is possible to exclude single story patterns of a story diagram by annotations.

References

1. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
2. Clements, P., Northrop, L.: Software product lines: practices and patterns. Volume 0201703327. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2001)
3. Antkiewicz, M., Czarnecki, K.: Featureplugin: Feature modeling plug-in for eclipse. In: OOPSLA '04 Eclipse Technology eXchange (ETX) Workshop, Vancouver, British Columbia, Canada, ACM (Oct. 24-28 2004)
4. Stephan, M., Antkiewicz, M.: Ecore.fmp a tool for editing and instantiating class models as feature models. Technical report, University of Waterloo (2008)
5. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), ACM (May 2008) 943–944
6. Zündorf, A.: Rigorous object oriented software development. Technical report, University of Paderborn, Germany (2001)
7. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework. 2 edn. The Eclipse Series. Addison-Wesley, Boston (2009)
8. Buchmann, T., Dotor, A., Westfechtel, B.: Triple graph grammars or triple graph transformation systems? a case study from software configuration management,1st international workshop on model co-evolution and consistency management, mccm 08, toulouse, france, september 30th, 2008. (2008)
9. Buchmann, T., Dotor, A., Westfechtel, B.: Mod2-scm: Experiences with co-evolving models when designing a modular scm system. In: 1st International Workshop Co-Evolution and Consistency Management (MCCM '08). (2008)
10. Buchmann, T., Dotor, A., Westfechtel, B.: Model-driven development of software configuration management systems - a case study in model-driven engineering. submitted for publication
11. OMG: OMG Unified Modeling Language (OMG UML), Superstructure. OMG. (November 2007) Version 2.1.2.
12. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE 05. (2005)
13. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: 2nd Workshop on Aspect-Oriented Product Line Engineering (AOPLE'07). (2007)
14. Heidenreich, F., Şavga, I., Wende, C.: On controlled visualisations in software product line engineering. In: Proceedings of the 2nd Int. Workshop on Visualisation in Software Product Line Engineering (ViSPLE 2008). (September 2008) To appear.
15. Buchmann, T., Dotor, A., Westfechtel, B.: Experiences with modeling in the large with fujaba. In Assmann, U., Johannes, J., Zündorf, A., eds.: Proceedings of the 6th International Fujaba Days, University of Dresden, University of Dresden (2008)

Feature Composition – Towards product lines of Event-B models

Ali Gondal, Michael Poppleton, Colin Snook

Dependable Systems and Software Engineering Group
University of Southampton, Southampton SO17 1BJ, UK
{aag07r, mrp, cfs}@ecs.soton.ac.uk

Abstract. Event-B is a formal language for modelling reactive systems, based on set theory and first-order logic. The RODIN toolkit provides comprehensive tool support for modelling and refinement in Event-B, analysis and verification using animator/model-checkers and theorem provers. We consider the need to support reuse, in particular product line reuse, in such a formal development method.
Feature modelling is an established technique for reuse in product lines. We introduce concepts of feature modelling and composition in Event-B to support the reuse of formal models and developments. A prototype feature composition tool has been developed (as a RODIN plugin) for Event-B, based on the Eclipse Modelling Framework (EMF). Using an MDD philosophy, the tool extends the Event-B language metamodel to a composition metamodel, and implements prototype composition patterns for Event-B features. Thus, a required composite model can be constructed by selecting, specializing, and composing input features in a defined way. The tool is the first step towards full feature modelling for product line model reuse for Event-B. We describe future work required to meet this goal.

1 Introduction

Formal Methods provide mathematically based languages, tools and techniques for specifying and verifying systems during construction. They allow identification of inconsistencies, ambiguities and defects earlier in the software development life-cycle and reduce the need for unit and integration testing [1]. Verification conditions called *proof obligations* (POs) take the form of mathematical theorems which state correctness properties of models and their refinements. Successful application of formal methods can be seen in aerospace, transportation, defence and medical sectors [1, 2]. Improvements in formal specification languages, verification techniques and robust tools are ongoing, in particular, by the DEPLOY[1] and RODIN[2] projects, including industrial partners such as Bosch,

[1] DEPLOY - Industrial deployment of system engineering methods EU Project IST-214158. http://www.deploy-project.eu
[2] RODIN - Rigorous Open Development Environment for Open Systems: EU Project IST-511599. http://rodin.cs.ncl.ac.uk

SAP, and Siemens and Space Systems Finland. These projects are developing tools, as well as strengthening the theoretical base, for the formal specification language Event-B [3].

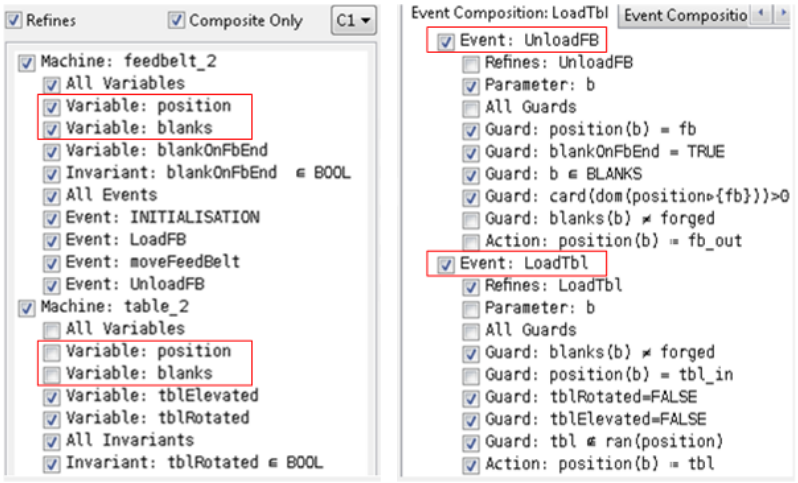
Event-B, based on set theory and first-order logic, is used for modelling and analysing discrete event systems and provides built-in generation and verification of proof obligations. It is a successor of Abrial's B language [4], developed in the RODIN and earlier EU projects. After completion of the RODIN project, the DEPLOY project is now deploying this work into industry. The RODIN toolkit [5] provides support for modelling, animation, model-checking and proof using Event-B. It is an Eclipse based IDE, and easily extensible. Refinement is the core development process of introducing more details in each step from abstract specification to the concrete implementation model in Event-B. Any refined model must be proved to be a true refinement of the abstract model.

A software product line (SPL) refers to a set of related products having a common base and built from a shared set of resources [6]. SPLs focus on the problem of software reuse by providing automated ways to build families of software products sharing commonalities, and differing by variabilities of structure. Feature modelling is a model-driven approach which gives a means to define commonalities and variabilities in terms of atomic requirements *features*. Several tools have been developed for supporting feature modelling for SPL engineering [7]. For SPLs of critical systems, there is a need for the automated verification that formal methods provide. This paper discusses our approach to introduce product line reuse in Event-B using feature modelling concepts and reports on tooling developments towards the full Feature Modelling Tool (FMT) for modelling of SPLs in Event-B. The example used in the paper is discussed in section 2. Section 3 gives a technical overview of the Event-B language. The notion of feature modelling in Event-B is discussed in section 4 followed by the tool discussion, related and future works in sections 5, 6 and 7 respectively.

2 Feature Composition Example

This example has been taken from the Production Cell [8] case study which we have modelled in Event-B. This is a reactive system which has been modelled in a number of formalisms. The purpose of the example here is to demonstrate the prototype tool that we have developed rather than its Event-B modelling. There were multiple features in Production Cell model but we only consider two features here for brevity, i.e. feed belt and table. Metal blanks enter the system through the feed belt and are dropped one by one on to the table from where other components such as a robot may pick them up and deliver them to another component for further processing. It happens in this example that features map to physical components of the system, e.g. table, robot etc. We chose this example to represent a product line of Production Cell systems, where different feature configurations result in different instances of a Production Cell. Variabilities are the number and connectivity of features, e.g. we can build a system with

multiple robots and belts to increase throughput. All the features are modelled generically for use in various configurations, and are verified separately.



(a) Feature Composition (b) Sub-Feature Composition

Fig. 1. Feature Composition Editor

3 Event-B Language

An Event-B model consists of a *machine* - modelling dynamic data and behaviour - and zero or more *contexts* - modelling static data structures or configurations. This separation of behaviour allows the use of different contexts to parametrize the machines. Note that there is no concept of modularization in Event-B, so a model represents a complete system at a particular level of refinement. The state transition mechanism over the machine's *variables* is given by the *event*, which comprises *parameters* (also called arguments or local variables), *guards* and *actions*. The guard is a condition on the event parameters and machine variables that defines the enabledness of the event: the event is only enabled when all of its guards are true. The action is the update operation on a state variable. The syntax of an event e with guards G, variables v and actions A is: $e =$ `when` $G(v)$ `then` $A(v)$. Examples of two events (UnloadFB & LoadTbl) can be seen on Fig. 1(b). The *invariant* is a state predicate specifying correctness properties that must always hold. Variables are typed by invariants. Many POs concern invariant preservation, i.e. correctness of the system is defined and preserved through the invariants. All events must preserve invariants and any violation of invariants will lead to the system being inconsistent. An `INITIALIZATION` event is used to specify the initial values for the variables.

The context (static data) in Event-B contains *sets*, *constants*, *axioms* and *theorems*. Sets are used to define the types and axioms describe the properties of the constants. Theorems must be proved to follow from axioms.

Event-B provides support for refinement where structural and algorithmic detail can be added during each refinement step; new events can be added and existing events can be extended. Variables may be added or transformed. Refinement will usually reduce non-determinism. Similarly, contexts can also be extended to add more details to the model. Refinement proof obligations are generated by the tool to verify that a refined model is a correct refinement of the abstract model. An EMF [9] metamodel for Event-B has been developed as part of the DEPLOY project.

4 Feature Modelling in Event-B

Feature modelling is a well-known technique for reuse in product lines. Methodologically speaking, for its application in an Event-B setting, some form of domain engineering activity comes first. At very least, an instance of the product line should be fully developed, followed by the engineering of a variant system or two. Commonality/variability analysis should be undertaken and followed by the incremental building of a domain feature model and database. Work is under way developing this methodological and domain engineering work, which will be reported elsewhere.

In this paper we introduce a prototype feature composition tool as an initial step towards the development of a more comprehensive tooling for feature modelling. For such a full Feature Modelling Tool (FMT), we will need to define a metamodel for the feature modelling language. The FMT will consist of a feature model editor and a feature instance editor. The feature model editor will be used to build the feature models for different product families and the feature instance editor will provide a configuration mechanism for choosing various features to instantiate a new product line member, somewhat similar to configuration diagram of FeaturePlugin [10]. The feature model will also specify any constraints needed to maintain the correct selection of features in a particular instance. Feature composition rules will respect these constraints. The instance editor will use the feature composition tool described in this paper to compose selected features to build the instance system. We are planning to develop the instance editor in such a way that it can produce an instance using the composition descriptions defined in the composition meta-language as discussed in our earlier work [11]. This framework consists of two layers of metamodelling where a feature model will conform to the feature metamodel and at the same time will serve as a metamodel for the instance model. Validation criteria will be needed to verify the correct instantiation of the metamodels at each level.

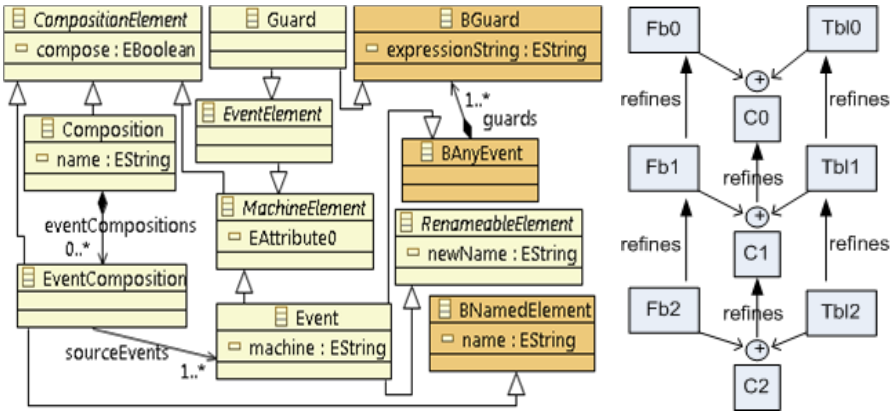
The feature has been defined as "a logical unit of behaviour specified by a set of functional and non-functional requirements" [12]. We define the concepts of "feature" and "sub-feature" in Event-B as atomic units of reuse, specialization and composition. This is in order to preserve the semantics of Event-B and to

formally verify the product-line members. A *feature* is thus a small, coherent and syntactically complete Event-B model which consists of a machine and zero or more seen contexts. This allows a feature and its refinements to be verified using the RODIN provers. A notion of *sub-feature* may be useful: when a feature cannot be reused as a whole, we might be interested in reusing some parts of a feature. Thus, a sub-feature is part of a feature which is syntactically incomplete but can be reused when composed with other sub-features, e.g. in Event-B, an event or a variable with its associated invariant(s) can constitute a sub-feature. The following is our definition of feature and sub-feature in BNF[3].

$$
\begin{aligned}
&Feature ::= Context \mid Machine\ Context^{+}\\
&Machine ::= Name\ Variable^{+}\ Invariant^{+}\ Theorem^{*}\ [Variant]\\
&\qquad\qquad \texttt{INITIALIZATION}\ Init\ Event^{+}\\
&Context ::= Name\ \{Set^{+}\ Axiom^{*}\ Theorem^{*}\ \mid\\
&\qquad\qquad\qquad Set^{*}\ Constant^{+}\ Axiom^{+}\ Theorem^{*}\}\\
&SubFeature ::= EventSF \mid VariableSF \mid InvariantSF \mid ContextSF\\
&EventSF ::= \texttt{INITIALIZATION}\ Init \mid Event\\
&VariableSF ::= Variable^{+}\ InvariantSF\\
&InvariantSF ::= Invariant^{+}\\
&TheoremSF ::= Theorem^{+}\\
&ContextSF ::= Set^{*} \mid Constant^{*} \mid Axiom^{*} \mid Theorem^{*}
\end{aligned}
$$

We start system modelling in Event-B by defining the features at an abstract level and then refine these features gradually by adding further detail at each refinement step. Splitting requirements across features and then modelling and refining feature-wise, separates concerns and should reduce complexity - application experience will tell. We call each feature and its chain of refinements a *feature development*. Developing earlier work [11, 13], we regard these feature developments as units of reuse. They will be specialized - by addition or alteration of information - and composed in various ways in the process of assembling (instantiating) an instance system in the target product line. For example, two *EventSFs* are composed after specialization and conflict resolution into a single *EventSF* named 'LoadTbl' during the composition of two Features i.e. feedbelt_2 & table_2, as shown in Fig. 1. The processes of specialization and composition will occur in general at all refinement levels of the input feature developments. Note that the relation between requirement feature and its implementation is more complex than one to one and feature composition through non-linear refinements will layer in complex, possibly cross-cutting, structure.

[3] The syntax used is: [ ] means optional (0 or 1), { }* means 0 or more occurrences, { }$^{+}$ means 1 or more occurrences and | means OR.



(a) Part of Composition Metamodel (b) Example Structure

Fig. 2.

5 Feature Composition Tool

An initial version of the feature composition tool is available as a plugin to the RODIN platform[4] [5]. It provides a simple structured cut-and-paste composition of features and guides the user in identifying and resolving any conflicts. We are currently working on a number of composition/specialization patterns to maximize automation while not restricting user expressiveness.

The plugin was developed in Eclipse using Java in a model-driven approach. We built a metamodel for the composition model which inherits from the Event-B metamodel. We then used EMF [9] to generate the code from the composition metamodel. The major advantage is in the ease of extension through the metamodel where the code is automatically generated by EMF and then customized. Fig. 2(a) shows part of the composition metamodel for event composition where Event, Guard and EventComposition inherit from Event-B metamodel, i.e. BAnyEvent, BGuard and BNamedElement respectively. Event inherits 'newName' from RenameableElement which allows a new name to be given to an Event, e.g. to resolve naming conflicts. The collection 'sourceEvents' represents the set of events being merged together to form a single 'EventComposition' which inherits 'name' attribute from BNamedElement. All composition elements inherit a 'compose' boolean which represents the selection of that element for composition.

The tool provides a composition editor[5](Fig. 1) to select the features (in a similar manner to existing feature modelling tools, e.g. [10]) that need to be

[4] See `http://sourceforge.net/projects/rodin-b-sharp/` and wiki entry at `http://wiki.event-b.org/index.php/Feature_Composition_Plugin`

[5] The earlier prototype was contributed by Christopher Franklin (a University of Southampton Intern)

composed resulting in the composition model. This model is then used by the tool to transform/compose the input features into a composite Event-B feature. The composition model is serialized in the RODIN model repository for replaying the composition later. The editor provides conflict resolution functionality and highlights any conflicts such as multiple declarations of *variables* or *events* with the same name in different input features. It provides facilities for making the input models disjoint before composition to automatically resolve any conflicting element names. It can also automatically resolve conflicting elements by deselecting the repeating/redundant information in different models (see variables 'position' and 'blanks' on Fig. 1(a)). The editor also provides an option for composing sub-features such as *events* (Fig. 1(b) shows the composition of two events). The tool is also capable of composing features at different refinement levels. The composite feature is a typical Event-B model and is automatically checked by the RODIN static checker for any errors. Similarly, proof obligations (POs) for the composite model are generated in normal fashion and the RODIN provers discharge any POs automatically if they can. Fig. 2(b) shows the structure of our example features refined up to two levels and then composed using the tool. The composition of features at each level needed some extra invariants, guards and merging of events. Fig. 1(b) shows the composition of two events into one and de-selection of redundant elements to resolve conflicts.

6 Related Work

To our knowledge, there is no tool support for feature modelling within the formal methods domain, although, there are tools which support feature modelling for product lines such as XFeature[6], FeaturePlugin [10] etc. The underlying concepts discussed in [14] are quite similar to what we want to achieve in the domain of formal product line development. Another area that is closely related to our work is the definition of composition patterns. These patterns will enable us to write composition rules when composing Event-B features. The RODIN toolkit is already in the phase of adopting the model transformation and code generation facilities of EMF[7].

7 Conclusion & Future Work

We have given an overview of our approach to introduce the concepts of feature modelling within formal methods using Event-B. Our prototype feature composition tool is an initial step towards the development of a feature modelling tool for configuring Event-B features and composing them to instantiate software product line systems by reusing and extending existing features as mentioned in section 4. This is required because existing feature modelling tools don't provide

[6] Feature Modelling Tool `http://www.pnp-software.com/XFeature/Home.html`
[7] See `http://wiki.event-b.org/index.php/EMF_framework_for_Event-B`

enough facilities to give semantics to features and the resulting formal verification capabilities such as offered by Event-B. Our prototype tool will enable us to experiment with different case studies and to improve the tool requirements and underlying notations for feature modelling. This remains work in progress.

The example of section 2 revealed additional tool requirements and research questions.We will require a composition management capability to record, manage and replay the sequences of compositions and specializations of features required by a target system. This should substantially increase productivity. Another area to explore is reusing proofs. When we compose Event-B features into a composite feature, the tool generates proof obligations (POs) for consistency and refinement checking. Most of the POs for the input features still exist for the composite feature, and may have already been discharged interactively. Hence, it would be useful if the tool could reuse interactively discharged POs to save user time and effort. This might be achieved through the composition of proof trees while composing their associated features.

References

1. Abrial, J.R.: Formal methods in industry: Achievements, problems, future. In: ICSE '06: Proceedings of the 28th ICSE, NY, USA, ACM (2008) 761–768
2. Bowen, J.P., Hinchey, M.G.: The use of industrial-strength formal methods. In: COMPSAC '97, Washington, DC, USA, IEEE Computer Society (1997) 332–337
3. Metayer, C., Abrial, J.R., Voisin, L.: Event-B language. Rodin deliverable 3.2, EU Project IST-511599 -RODIN (May 2005)
4. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, New York, NY, USA (1996)
5. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An open extensible tool environment for Event-B. In: ICFEM. (2006) 588–605
6. Clements, P., Northrop, L., Northrop, L.M.: Software Product Lines : Practices and Patterns. Addison-Wesley Professional (August 2001)
7. Lee, K., Kang, K.C., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In: ICSR-7, UK, Springer-Verlag (2002) 62–77
8. Lindner, T.: Task description. In Lewerentz, C., Lindner, T., eds.: Formal Development of Reactive Systems. Volume 891 of LNCS., Springer (1995)
9. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework. 2nd edn. The Eclipse Series. Addison-Wesley Professional (December 2008)
10. Antkiewicz, M., Czarnecki, K.: Featureplugin: Feature modeling plug-in for Eclipse. In: Eclipse '04: Proceedings of the 2004 OOPSLA, NY, USA, ACM (2004) 67–72
11. Poppleton, M., Fischer, B., Franklin, C., Gondal, A., Snook, C., Sorge, J.: Towards reuse with "Feature-oriented Event-B", Nashville, TN, In McGPLE (October 2008)
12. Bosch, J.: Design and Use of Software Architectures: Adopting and Evolving a Product-line Approach. ACM Press/Addison-Wesley, NY, USA (2000)
13. Poppleton, M.: Towards feature-oriented specification and development with Event-B. In: Proc. REFSQ. LNCS, Trondheim, Norway, Springer (2007) 367–381
14. Cechticky, V., Pasetti, A., Rohlik, O., Schaufelberger, W.: Xml-based feature modelling. (2004) 101–114

Functional Hazard Assessment in Product-Lines – A Model-Based Approach

Ibrahim Habli, Tim Kelly, Richard Paige

Department of Computer Science, University of York, York, United Kingdom
{Ibrahim.Habli, Tim.Kelly, Richard.Paige}@cs.york.ac.uk

Abstract. A product-line can offer the reuse of complete lifecycle assets, comprising planning, development and assessment artefacts. In safety-critical systems engineering, safety assessment artefacts are indispensable assets which are expensive to generate. For safety-critical product-lines, it would be cost-effective to encapsulate these assessment artefacts as reusable assets, i.e. to provide the capability of reusing a product-line function or component in addition to its safety assessment artefacts. In this paper, we focus on Functional Hazard Assessment of a product-line's functions. We propose a product-line functional hazard model which is integrated with the product-line context and domain models. We also show how this proposed product-line functional hazard model fits within the product-line processes.

Keywords: Product-lines, model-driven development, safety-critical systems, functional hazard assessment

1 Introduction

Reuse is a recognised approach to reducing the development cost of software-intensive systems. Instead of analysing, designing, implementing and verifying a function or a component from scratch every time, it would be more cost-effective to develop it once and reuse it in multiple systems. Specifically, reuse could be more cost-effective if it is managed according to predefined contextual and architectural constraints such as in product-line development. Product-line development is an approach to large-scale and holistic reuse. A product-line can offer the reuse of complete lifecycle artefacts comprising planning, development and assessment artefacts. In safety-critical product-lines, safety assessment artefacts generated from Functional Hazard Assessment (FHA), Fault Tree Analysis (FTA) and Failure Mode and Effect Analysis (FMEA) [1] produce the main evidence supporting claims that the system is acceptably safe to operate within a specific environment. These safety assessment artefacts are indispensable assets which are expensive to generate. To this end, it would be advantageous to encapsulate these artefacts as reusable product-line assets, i.e. to provide the capability of reusing a product-line function or component in addition to its safety assessment artefacts (e.g. its failure mode assessment data). However, to be able to reuse safety assessment artefacts in a trustworthy manner, the relationship between a reusable function or component and its associated safety

assessment artefacts should be explicitly and unambiguously modelled. In other words, it is not enough to emphasise the importance of the *process* by which the safety assessment information is generated, but also the importance of the *model* against which the structures, relationships and assumptions underlying the generated assessment information are captured. The safety assessment process is carried out once whereas the generated assessment information, identified and specified against the model, is reused multiple times. In this paper, we focus on Functional Hazard Assessment of a product-line's functions. We propose a product-line functional hazard model that is integrated with the product-line context and domain models. We also show how this proposed product-line functional hazard model fits within the product-line processes.

This paper is structured as follows. Section 2 presents a brief introduction of FHA, followed by an overview of the modelling approach proposed in this paper (Section 3). Section 4 proposes a product-line functional hazard model and how it interfaces with product-line context and domain models. Section 5 shows how the proposed product-line functional hazard model fits within the product-line's domain and derivation processes. The paper concludes with a summary in Section 6.

2 Functional Hazard Assessment

Functional hazard assessment (FHA) is an inductive technique which examines the way in which system functions contribute to system safety [1]. FHA identifies *failure conditions* associated with system functions and the *effects* these failure conditions can have on overall safety. Failure conditions are typically identified by considering three hypothetical deviations: (1) function not provided, (2) function provided when not required and (3) function provided incorrectly. The effects of each failure condition are then identified, specifically those affecting the intended behaviours of the system, its environment and users. FHA then *classifies* each function based on the severity of the effects of the function's failure conditions. Failure conditions leading to deaths or injuries are typically classified as '*Catastrophic*' or '*Hazardous*'. Less severe failure conditions are typically classified as '*Major*' or '*Minor*'. Finally, *system safety requirements* are defined, addressing each failure condition in accordance with the severity of its classification (i.e. rigour in specifying and meeting the safety requirements of a function is proportionate to the criticality of the function).

3 Approach Overview

System safety aspects and the way in which they are identified and assessed cannot be considered in isolation from the system's context, functions and dependencies. This is mainly because safety is a *consequential* attribute; i.e. safety conditions such as hazards, failure conditions and failure modes are an outcome of certain system functionalities and behaviours in a given environment. This is why system hazards are typically identified given the safety analysts' understanding of the system functions and their environment. For example, safety conditions are stated given certain

assumptions about data sampling rate, level of independence, expected operation modes, maintenance procedures and end-user training. In this paper, we focus on one particular class of safety conditions: *Functional Hazards*. Functional hazards are functional failure conditions with hazardous consequences, e.g. loss of braking capability or inadvertent engine thrust control. These failure conditions cannot be identified and analysed without a sufficient understanding of the system functions and the way in which they interact with one another and with their environment. Specifically, in a product-line, functions are typically captured in the domain model as high-level *features*. The product-line environment, on the other hand, is defined in the context model. As such, any product-line functional hazard model should interface with the product-line context and domain models.

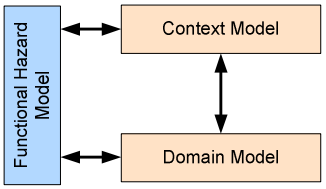


Fig. 1. Overall Structure of the Product-Line Models

An overview of the interrelationships between the functional hazard, context and domain models is depicted in Figure 1 (a more detailed description of these models is presented in the next section). The context model captures information regarding the structure of the physical, operating, support, maintenance and regulatory environment of the product-line systems. Broadly speaking, the context model defines external constraints that these systems need to respect. Without a clear definition of these, implicit contracts and dependencies between these systems and their support environment could be violated, leading to uncertainties regarding the assumed safe behaviours of the systems. The domain model, on the other hand, considers the structures of, and interactions between, *features* provided by systems in a domain. A feature is a set of related capabilities and characteristics that represent a logical unit of functionality for the stakeholders of the product [2] [3]. Features in a domain model have explicit associations with one another and with environmental factors (e.g. particular functions deployed only during certain operational phases). Finally, the functional hazard model captures failure conditions, effects, severity classification and safety requirements which relate to specific functional and environmental configurations, as defined in the context and domain models.

One special characteristic of product-line development is variation management. The product-line context and domain models capture how products, derived from the product-line's assets, are permitted to vary from one another. These products vary in terms of the functions they provide or the environment in which they can be deployed. Because of the interdependency between the product-line's functions, environment and functional hazards, functional hazards are not immune from the impact of functional and contextual variation. The nature and severity of functional hazards assumed to be associated with a certain reusable function may change due to certain permitted configurations of the variation points defined for that function.

Uncertainties in the traceability between a function and the hazards that may be posed by that function, due to certain product-line variations, are dangerous and unacceptable in the safety domain. Seemingly simple variations in the original and new deployment of reusable functions may contribute to the occurrence of hazards or failure modes previously assumed to be irrelevant to the system [5].

To this end, the management of variations and the way in which they affect safety, however minor, is a prerequisite for trusted reuse. We believe that the impact of variation on safety can be identified, traced and controlled by integrating the product-line functional hazard model with the context and domain models. By adopting a model-based approach to managing the safety impact of product-line variation, product-line functions, along with their associated functional hazard information, can be reused with greater confidence and without the need for the reassessment of these functions whenever reused in the derivation of new products.

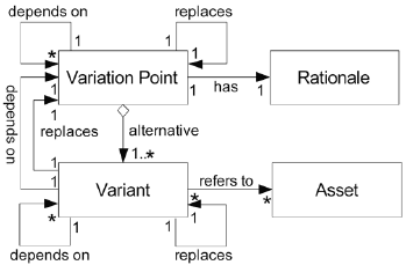


Fig. 2. Example Generic Variability Metamodel [5]

Many models for describing variability have been proposed in the product-line literature [5] [6] [7] [8] [9] [10]. In essence, a variability metamodel comprises *variability points* (figure 2). Variability points specify places in an artefact where variability can arise [11]. Variability points are bound by engineers to create concrete *variants* of variable artefacts or attributes. Variability points are often interdependent in that the binding of one variability point may restrict the binding of other variability points. For example, for safety, a choice of relevant operation modes and their criticality in a context model may dictate the binding of certain variability points regarding the level of partitioning in the design model. This in turn may have impact on how the bound design artefact could contribute to system hazards identified in the safety models. In this report, we do not prescribe a specific structure for the variability metamodels. The context, domain and functional hazard models defined in the next section are generic and can easily be interfaced with variability models which comprise, as a minimum, explicit variability points and the resulting variants.

4 Product-Line Functional Hazard Model

In this section, we propose a functional hazard model for product-line development. This model defines the relationships between, on the one hand, product-line

functional and environmental variants and, on the other hand, their corresponding failure conditions, effects, classifications and safety requirements.

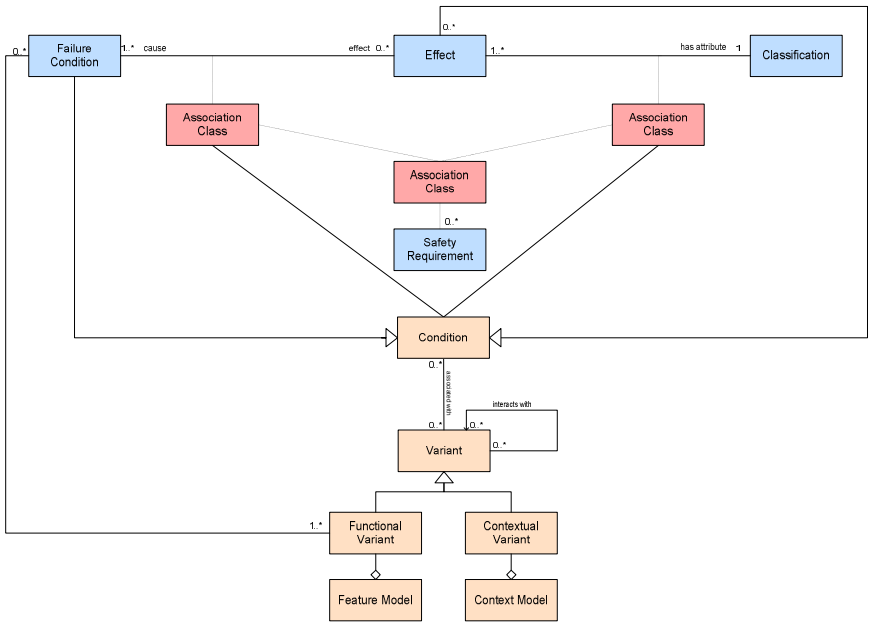


Fig. 3. Product-Line Functional Hazard Model

The product-line functional hazard model is depicted in Figure 3 (please note that, for presentation purposes, the depicted model is an abstraction of the original model which is created in the form of an Ecore model [12]). In the model in Figure 3, a *'failure condition'*, which is a subtype of a '*condition*' and associated with a '*functional variant*', can lead to one or more *'effects'* (an effect is also a subtype of a '*condition*'). Each effect is associated with a '*classification*' based on the severity of this effect from the safety perspective. It is important to note that the two elements '*Functional Variant'* and '*Contextual Variant'* are the main source of the variation. The *'failure condition'*, '*effect'*, *'classification'* and *'safety requirement'* model elements can also vary, though in a *subsequential* manner. That is, variation in failure conditions, effects, classifications and safety requirements, within the scope of the FHA, is not indigenous and is a consequent of the variation in functional specifications and the environment (e.g. operation phase, temperature and behaviour of other internal and external functions). A failure condition occurs and leads to certain effects due to, and in the context of, a certain configuration of functional variants and contextual variants. When this configuration varies, the failure condition and its associated effects may also vary. Similarly, the classification of the same failure condition effect in a product-line may vary. For example, in-flight engine shutdown may be classified as '*Minor*' for one aircraft whereas classified as '*Hazardous'* for another. This depends on the various ways in which the aircraft and engine product-lines are configured (e.g. variation in the number of engines fitted). In

particular, the impact of variation in functions and external conditions is captured in Figure 3 in the following *association classes*:

Failure Condition and Effect Association Class: The relationship between a failure condition and its effects is governed by specific conditions which are associated with certain configurations of functional and contextual variants. These configurations may vary, and as a result, the causal relationship between failure conditions and effects may vary accordingly. For example, the same failure condition of a function may lead to different effects in a product-line FHA depending on the various permitted configurations of this function within certain environments.

Effect and Classification Association Class: A failure condition in a product-line FHA may lead to one or more effects. However, once an effect occurs, its classification may depend on functional and contextual variants other than those affecting the occurrence of the failure condition(s). Therefore, it is important to explicitly capture separately the functional and contextual variants that affect classification.

Safety Requirement Association Class: Derived safety requirements are one of the outputs of an FHA – "*Assignment of requirements to the failure conditions to be considered at the lower level*" [1]. Safety requirements are not only associated with failure conditions, but also with the classification of these failure conditions. Safety requirements for the same failure condition in a product-line can differ when the failure condition has more than one classification – i.e. due to permitted variation that can affect classification. To this end, a safety requirement in the FHA model in Figure 3 is linked with the *association* that relates a failure condition to its effect. Similarly, a safety requirement in the FHA model is linked with the *association* that relates the effects of a failure condition to its classification. In this way, we can ensure that the integrity and rigour associated with defining and meeting the safety requirement is proportionate to the classification of the failure condition addressed by that safety requirement. In short, variation in safety requirements is a function of variation in (1) the association leading to failure conditions and (2) the association influencing the classification of these failure conditions.

5 Product-Line Functional Hazard Assessment Process

In this section, we present an approach to integrating FHA into the product-line's domain engineering and application engineering phases. The role of the FHA in the domain engineering phase is to examine the failure conditions associated with each variant of a function addressed in the feature model by producing the function's effects, classification and associated safety requirements. These product-line failure conditions, effects, classifications and safety requirements are captured against the product-line functional hazard model defined in the previous section. This enables functional hazard data to be reused whenever a function in the feature model is selected in the derivation of a new product as part of the product-line. As shown in Figure 4, the FHA is the first safety assessment method applied in the product-line domain engineering phase and is highly associated with the product-line domain and

context analysis. The relationship between the FHA activity and the context and domain modelling activity is bidirectional. When a function is captured as a feature in the feature model, and its environment defined in the context model, each variant of the function, given its specified context, is examined in the FHA to determine its failure conditions, effects, classifications and safety requirements.

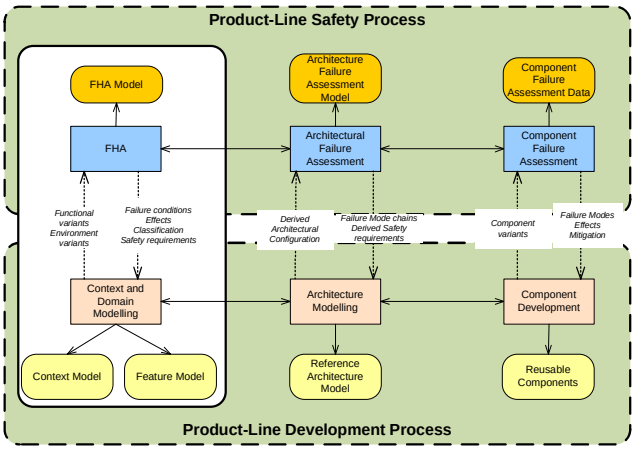


Fig. 4. FHA in the Product-Line Domain Engineering Phase

The FHA results also play a role in constraining certain configurations of features or contextual assumptions which may pose unacceptable safety conditions. For example, the functional hazard data may show that the severity of certain failure condition effects of a function (represented as a feature) are unacceptable within the agreed scope of a product-line and may therefore result in imposing constraints on the selection of that function (e.g. it may be within the scope of the product-line to address *hazardous* but not *catastrophic* events). Furthermore, as part of defining the functional variants of the product-line's feature model, decisions and assumptions can be made regarding independence between the product-line's functions (e.g. to avoid common mode failures). Decisions and assumptions concerning independence, affecting safety, should be explicitly captured and managed as part of the product-line FHA. In the product-line feature model, constraints should be established, based on the product-line FHA results, that restrict feature compositions that can violate assumed independence between functions, e.g. independence between functions and their monitors.

In the product-line's application engineering phase, the functional hazards posed by each function, reused in the derivation of a new product, should be identified and analysed. However, this analysis is not carried out from scratch. Each product function, along with its associated contextual conditions, are fed into the *product* FHA activity, which in turn checks for potential matches with functional and contextual variants whose failure conditions have been previously captured in the *product-line* functional hazard model. Whenever there is a match, relevant failure condition data should be added to the *product* functional hazard model and linked with the product's requirements and contextual conditions. In the case of mismatches,

the safety analysts need to examine the sources of these mismatches. For example, mismatches can due to product-specific functions or product-specific contextual assumptions. Safety analysts should analyse these new functions or contextual assumptions and generate their corresponding failure conditions, effects, classifications and safety requirements. These failure conditions, effects, classifications and safety requirements should not only be added to the *product* FHA model, but should also be added to the *product-line* FHA model and therefore should be available for reuse, where relevant, in the analysis of future products.

6 Summary

We have proposed a product-line functional hazard model, which is integrated with the product-line context and domain models. We have also proposed a way in which this functional hazard model fits within the product-line processes. We are currently validating the functional hazard model against a number of safety assessment processes as defined in some aerospace and automotive standards. We are also working on a generic safety assessment metamodel for product-lines which could be integrated with a product-line's context, domain and reference architecture models.

References

1. Society of Automotive Engineers (SAE): ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment, SAE 400 Commonwealth Drive, USA (1994)
2. Bosch, J.: Design and Use of Industrial Software Architectures, Addison Wesley (2000)
3. Clements, P., Northrop, L.: Software Product Lines, Practices and Patterns, Addison Wesley (2002)
4. Redmill, F.: Safety integrity levels – theory and problems, lessons in system safety. Proceedings of the Eighth Safety-Critical Systems Symposium, 1-20 Redmill F and Anderson A (eds), Southampton, UK, Springer Verlag (2000)
5. Bachmann, F., Goedicke, M., Sampaio do Prado Leite, Nord, R., Pohl, K., Ramesh, B., Vilbig, A.: A Meta-model for Representing Variability in Product Family Development. Software Product-Family Engineering, 5th International Workshop (2003)
6. Thiel, S., Hein, A.: Systematic Integration of Variability into Product-line Architecture Design, Proceedings of the 2nd International Conference on Software Product-lines, (2002)
7. Gomaa, H.: Designing Software Product-lines with UML. Addison Wesley (2005)
8. Kang K., Cohen, S., Hess, J., Novak, W.: A. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study. (CMU/SEI-90-TR-021), SEI (1990)
9. Becker, M., Towards a General Model of Variability in Product Families, Proceedings of the 1st Workshop on Software Variability Management, Groningen, Netherlands (2003)
10. Weiss, D. M., Chi Tau Robert Lai: Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley Professional (1999)
11. Jacobson, I., Griss, M., Patrik, J.: Software Reuse: Architecture, Process, and Organisation for Business Success, Addison-Wesley-Longman (1997)
12. Eclipse, Eclipse Modelling Framework Project (EMF), EMF Core, http://www.eclipse.org/modeling/emf/?project=emf

Flexible Service Specification and Matching Based on Feature Models

Muhammad Naeem and Reiko Heckel

University of Leicester {mn105, reiko}@mcs.le.ac.uk

Abstract. We propose to use variability techniques from the realm of product lines to help make service specifications more flexible. Feature diagrams provide a high-level model of the essential and optional aspects of services in combination with detailed models of service's semantics based on visual contracts specified by graph transformation rules. In this way we hope to provide a precise, yet flexible specification of requirements towards as well as provided services, which is amenable to automation while being visual and user-friendly.

1 Introduction

The use of the web as a platform for application integration provides the technology to react to today's rapidly changing business climate [1]. However, while technically web services can be discovered, selected and bound to at runtime [2], the necessary automation of these tasks continues to pose major challenges. One of them is the level of flexibility required to find a service satisfying the specified requirements. Human programmers, if they cannot find exactly what they asked for, will be happy enough to adapt their demands to a sufficiently similar service and change their implementation accordingly. But any automated selection and binding requires a detailed specification not just of the signatures and data types of the services required but also of their actions and protocols. Such a detailed description will, however, be less likely to be matched by any existing service.

Let us consider a scenario where a requester (potential client) finds a service satisfying most of the requirements. For example, while requesting a service for hotel and flight reservation, requester may also be interested in transport from the airport to the hotel, a guide to visit holy or historical places, while having a preference to pay via bank transfer. The provider may offer to book flight, hotel and transport, and allow payment via bank transfer or credit card. Comparing our requirements with the offer we find a mismatch in the fact that no guide services are provided. To resolve this mismatch we need information about the relevance of the missing feature, e.g., whether it is optional or compulsory for the success of our application. In addition, there will be dependencies between features as well as with other elements of our models.

In this paper we are proposing to combine visual techniques for the modelling of product lines, specifically *feature models* [3–5], with a modelling approach using *visual contracts* (graphical pre- and post conditions for operations [8, 9]) for

specifying and matching service descriptions as produced by the service provider against requirements as expressed by the requester. The approach is based on an easy integration with mainstream software UML notations, provides the level of formality and flexibility required to realistically allow automated matching of services, and supports the adaptation of provider and requester interfaces and implementations to the chosen variant.

The remainder of the paper is organised as follows: Section 2 discusses some related work, Section 3 introduces the specification and matching of services by visual contracts using a small case study. Section 4 extends this approach by feature models and Section 5 describes the matching and adaptation of these extended models. Section 6 concludes the paper.

2 Related Work

A number of approaches address flexible matching of services with semantic descriptions. Paolucci et al. in [11] have described an engine that allows matching of advertisements and requests on the bases of the capabilities. Several matchmaking frameworks are developed in [12–16], which operate on service descriptions written in RDF, DAML+OIL or DAML-S. Wu in [17] proposes a similarity-based approach, which grounds the matching process on a comparison of signature specifications in WSDL, but not on semantic descriptions. Matchmaking approaches ranking the similarity between advertisement and request are developed in [18] and [19]. In [20], Yongley adopted the ranking technique to service matching based on semantic descriptions.

We believe that variability needs to be built into the service specification, so that automated decisions can be made in the case of imperfect matches. This is different from ranking matches according to their degree of similarity. Moreover, our approach, while being inspired by standard methods for specifying and matching service semantics, does so on the basis of a visual notations which allows integration into mainstream software modelling languages.

3 Specifying and Matching of Services

In this section we describe the basic approach to visual semantic modelling and matching of services by means of a case study of an online travel agent. We use graph transformation rules to represent visual contracts.

Graph Transformation Systems (GTS) provide a modelling language where graphs model (data) states and rules specify state changing operations. The class of admissible states is specified by a *type graph* [9]. Fig. 1 shows a possible state graph of the case study where bookings have been generated by a client for hotel, flight, and transport. The types of nodes and their associations are represented by the type graph in the left of Fig. 1.

In the context of semantic web services we can think of the type graph as a representation of an ontology and of the instance graph as a data confirming

to it. Transformations of instance graphs are due to the application of graph transformation rules, shown in Fig. 2.

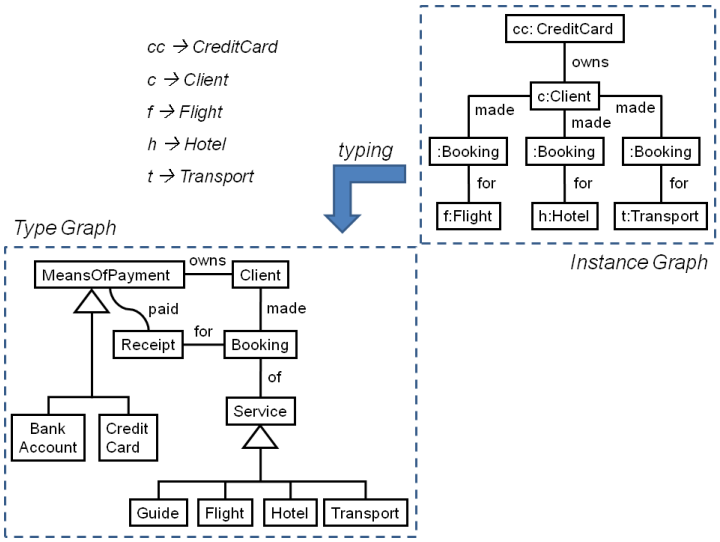


Fig. 1. Type and instance graphs

Service models are given in two version: from the requester's point of view describing desired functionality and from the provider's perspective specifying the services that are actually implemented.

The rule in the top part of Fig. 2 represents the requirements of a requester who is looking to reserve guide services and is willing to pay by credit card. The left side of the rule shows the preconditions of the operation while the right-hand side shows its postcondition / effect [9]. The rule expresses the demand for an operation which allows a client to book a guide payed for by a credit card owned by the client. Similar operations could be requested for the booking of hotel, flight, and transport, or to allow payment via bank transfer.

A rule describing the operation as implemented by the provider is in the bottom of Fig. 2. It offers to book any service using any available payment method.

It should be clear that, in this case the requirements of the requester are satisfied by the operations as specified by the provider because of the subtyping relation between *MeansOfPayment* and *CreditCard* as well as *Service* and *Guide*. These relations are captured in the ontology that both rules are based on, and which we assume to be standardised.

However, while we clearly have to rely on such standardisation to allow matching of services at all, we would like to have the possibility for example

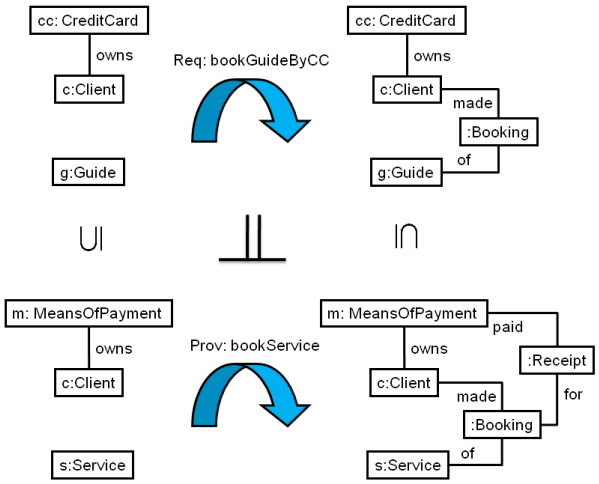


Fig. 2. Visual contracts specifying booking operations from requester (top) and provider (bottom) point of view

to specify a provider that does not offer guide services. We could do so at a moment by replacing the generic rule shown in the bottom of Fig. 2 by a number of more specific ones for booking hotel, transport and flight. In this case, the requester's requirement would not be satisfied by this service. In the next sections we will introduce a more economical way of representing this and other provider models based on features. We will also have to extend our notion.

Formally, a (provider) rule satisfies another (requester) rule if the preconditions of the first entail the preconditions of the second and the postconditions / effects of the second entail those of the first. Entailment in this case boils down to subgraph matching, allowing for the specialisation of types, i.e., the right-hand side of the provider rule entails the right-hand side of the requester rule because the former is a supergraph of the latter with more general types.

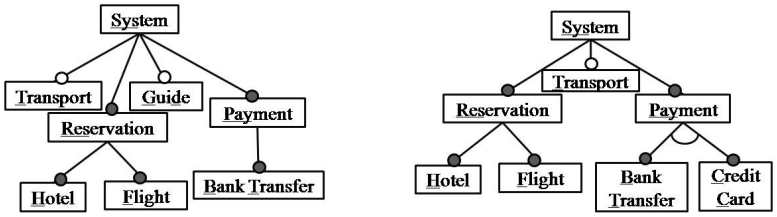
4 Visual Contracts with Features

In order to allow for a more fine-grained specification of the functionality on offer as well the desired flexibility in the matching of provisions to requests, we propose to use feature models. A feature is "a distinguishable characteristic of a concept that is relevant to some stakeholders" [4] while feature diagrams can be used to show the variability of features in a hierarchical form, including different types of features (such as optional, mandatory, alternative, etc.) and their interdependencies [3–7]. A feature model consists of a feature diagram and

other associated information, in our case given by the type graph (ontology) and visual rules (visual contracts) of requester and provider models.

Semantically, a feature diagram describes a set of instances, each representing a permissable subset of features. By taking the intersection between the sets of subsets on the requester and provider side we can identify the feature sets agreeable to both parties. Each such set describes a particular selection of features which can be used to derive a corresponding variant of the underlying service models.

Feature diagrams for requester and provider are shown in Fig. 3(a), 3(b). For example, the requester diagram declares Transport and Guide as optional features whereas Hotel and Flight Reservation and Payment by Bank Transfer as mandatory features. We have used *System* for the concept node of both requester and provider feature trees for ease in comparison.



(a) Requester (b) Provider

Fig. 3. Feature Diagrams of Requester and Provider

The connection between feature diagrams and visual contract models is provided by labelling the model elements (node types, rules, etc.) by the features they are part of. This is shown for our example in the type graph in Fig. 4 by small gray boxes with dashed borders placed at the corners of classes. Semantically, this means that since the Guide is not available as a feature of the provider (as seen from the provider's feature diagram), the corresponding class in the type graph is projected from the provider's view. In this way, the provider's booking rule in Fig. 2 describing the booking for all services does not promise subsume the booking of Guide services. For brevity we have used underlined characters of feature names (from Fig. 3) as labels in Fig. 4.

Similarly, operations and their visual contracts are labelled, e.g., rule *Req :: bookGuideByCC* would be labelled *Guide* and *CC* while *Prov :: bookService* would carry all labels except for *Guide*.

5 Matchmaking and Adaptation

In order to find out if a provider description matches the requirements expressed by a requester model, we proceed in three steps.

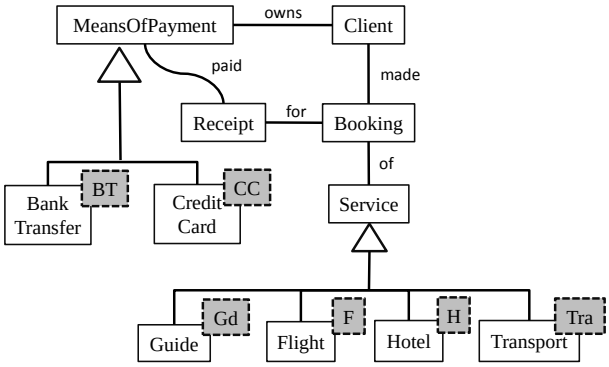


Fig. 4. Type Graph labelled by features: BT for Bank Transfer, CC for Credit Card, Gd for Guide, F for Flight, H for Hotel, Tra for Transport

1. Compute intersection of feature sets of requester and provider feature diagrams.
2. For each feature set in the intersection, derive the corresponding variant of provider and requester model.
3. Check compatibility of each derived pair of models.

For Step 1, feature trees can be converted into propositional formulas [3, 6, 7]. The set of solutions of the conjunction of the propositional formulas derived from the two feature models provides us with the desired intersection, i.e., the set of all subsets of features that are admissible according to both models. The result can be visualised as a feature diagram again, such as in Fig. 5 representing the intersection of feature diagrams of Fig. 3(a), 3(b). The largest admissible features set is $\{Sys, Res, F, H, Tra, Pay, BT\}$, but also $\{Sys, Res, F, H, Pay, BT\}$ is in the intersection.

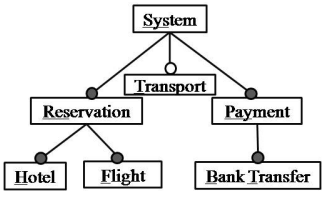


Fig. 5. Intersection of feature diagrams of Figures 3(a), 3(b)

In order to define the variant of requester and provider models in Step 2, we have to delete from these models all elements labelled by features not in the relevant feature set, and then recursively all the elements dependent on those deleted. For example, deleting the *Guide* class results in deleting the corresponding subtyping relation as well as all the operations and rules containing instances of this class. They are therefore disregarded in the next step. Further, if a subclass is removed from a superclass which occurs in a rule, this rule is removed as well and replaced by all its specialisations where the superclass is replaced by all permissable subclasses. In this way, from rule *Prov :: bookService* in Fig. 2 we obtain three specialisations, one of which (for Service → Transport) is shown in Fig. 6 together with the variant of the type graph.

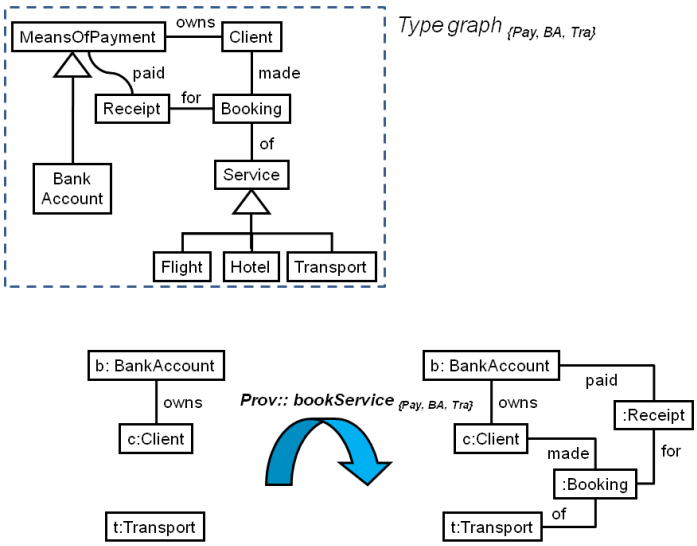


Fig. 6. Variant of type graph and rule

Since as a result of Step 2 we will obtain a pair of ordinary models (without features) for each subset of features selected in Step 1, we can apply the standard notion of matching as explained in Section 3 to check that they are indeed compatible.

Thus, there could be two reasons for a requirement not to be matched by a service description: An empty intersection of their feature diagrams (e.g., if a mandatory feature of the requester is not provided), or an incompatibility in the semantics of the actual operations. For the latter, consider a rule like the one in Fig. 2 but without the links between *c:Client* and *cc:CreditCard* in the left- and right-hand sides. In this case, the client would try to pay with a credit card not

owned by the person who has made the booking, which would contradict the requirement of the provider rule.

Notice that we take for granted here the fact that all models are specified over a shared ontology. Thus rules use the same classes for the same concepts and naming of features is consistent.

6 Conclusion

We proposed an extension by feature models of a visual approach to semantic web services based on graph transformation. As feature models can be rephrased in terms of propositional logic and visual contracts map to simple description logics, the entire approach could be handled in a purely logical framework. However, we believe that the visual presentation of models is as important for their usability as the explanation of the matching procedure at the same level.

Future work will focus on evaluating the approach, developing tool support, and extending it towards prioritising of features to be able to rank different feature sets for their level of satisfaction before going on to check the consistency of the semantic descriptions.

References

1. Leymann, F.: Choreography for the Grid: Towards Fitting BPEL to the Resource Framework. Journal of Concurrency and Computation: Practice and Experience. **17** (2005).
2. Papazoglou, P.: Service-Oriented Computing: Concepts, Characteristics and Directions. In Proceedings of the Fourth International Conference on Web Information Systems Engineering. IEEE Computer Society Washington, DC, USA. (2003) 3–12
3. Czarnecki, K., Eisenecker, U.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley, Boston, MA. (2000)
4. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-Oriented Domain Analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213. (1990)
5. Kang, K., Kima, S., Lee, J., Kim, K., Shin E., Huh, M.: FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures. Annals of Software Engineering. **5** (1998) 143–168
6. Batory D.: Feature Models, Grammars, and Propositional Formulas. In Proceedings of Software Product Lines Conference 2005. Springer Berlin/Heidelberg. LNCS **3714** (2005) 7–20
7. Jong, M., Visser, J.: Grammars as Feature Diagrams. In Proceedings of ICSR7 Workshop on Generative Programming. (2002) 23–24
8. Hausmann, J., Heckel, R., Lohmann, M.: Model-based Development of Web Service Descriptions: Enabling a Precise Matching Concept. International Journal of Web Services Research. **2** (2005) 67–84
9. Engels G., Heckel, R.: Graph Transformation as a Conceptual and Formal Framework for System Modelling and Model Evolution. In Proceedings of the 27th International Colloquium on Automata, Languages and Programming. Springer Berlin/Heidelberg. LNCS **1853** (2000) 127–150

10. Simos, M., Creps, R., Klingler, C., Lavine, L., UNISYS DEFENSE SYSTEMS RESTON VA: Software Technology for Adaptable Reliable System (STARS) Organization Domain Modeling (ODM) Guidebook. Technical Report STARS-VC-A025/001/00, Lockheed Martin Tactical Defense Systems, Manassas, VA. **2** (1996)
11. Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic Matching of Web Services Capabilities. In Proceedings of the 1st International Semantic Web Conference. LNCS **2342** (2002) 333–348
12. Chiat, L., Huang, L., Xie, J.: Matchmaking for Semantic Web Services. In Proceedings of IEEE International Conference on Services Computing. IEEE SCC. (2004) 455–458
13. Trastour, D., Bartolini, C., Priest, C.: Semantic Web Support for the Business-to-Business E-Commerce Lifecycle. In Proceedings of Eleventh Conference on World Wide Web. ACM New York, NY, USA. (2002) 89–98
14. Gonzales-Castillo, J., Trastour, D., Bartolini, C.: Description Logics for Matchmaking of Services. Proceedings of the KI-2001 Workshop on Applications of Description Logics, Aachen: CEUR Workshop Proceedings. **44** (2001) 89–126
15. Li, L., Horrocks, I.: A Software Framework for Matchmaking Based on Semantic Web Technology. In Proceedings of the Twelfth International Conference on World Wide Web. ACM New York, NY, USA. (2003) 331–339
16. Paolucci, M., Kawamura, T., Payne, T., Sycara, K.: Semantic Matching of Web Services Capabilities. In Proceedings of the 1st International Semantic Web Conference. Springer-Verlag London, UK. LNCS **2342** (2002) 334–347
17. Wu, J., Wu, Z.: Similarity-based Web Service Matchmaking. In Proceedings IEEE International Conference on Services Computing. IEEE Computer Society Washington, DC, USA. SCC **1** (2005) 287–294
18. Noia, T., Sciascio, E., Donini, F., Mongiello, M.: A System for Principled Matchmaking in an Electronic Marketplace. In Proceedings of the Twelfth International Conference on World Wide Web. ACM New York, NY, USA. (2003) 321–330
19. Jaeger, M., Tang, S.: Ranked Matching for Service Descriptions using DAML-S. In Proceedings of the Open InterOp Workshop on Enterprise Modelling and Ontologies for Interoperability Co-located with CaiSE'04 Conference. (2004) 217–228
20. Yao, Y., Su, S., Yang, F.: Service Matching Based on Semantic Descriptions. In Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services. IEEE Computer Society Washington, DC, USA. (2006) 126–131

Is Model Variability Enough?

Salvador Trujillo, Ander Zubizarreta, Josune De Sosa, Xabier Mendialdua

IKERLAN Research Centre, Spain
{strujillo, ander.zubizarreta, jdesosa, xmendialdua}@ikerlan.es

Abstract. Done well, the combined use of Model Driven Development (MDD) and Software Product Lines (SPL) offers a promising paradigm for software engineering. Such combination of abstraction from MDD and variability from SPL is particularly powerful in the field of software-intensive systems. Although this field has flourished recently, the focus so far has been mostly on how to cope with the variability of models. This focus on model variability has limited however the extension of variability to other artifacts apart from models such as metamodels and model transformations, that may cope with variability too. This position paper discusses whether model variability is enough for realizing the Model-Driven Product-Line dream. We shift our attention from the variability of models to a more general situation where the variability may embrace models, metamodels and model transformations.

Introduction

Modeling is essential to cope with the increasing complexity of current software systems. Models assist developers during the entire development life cycle to precisely capture and represent relevant aspects of a system from a given perspective and at an appropriate level of abstraction.

MDD is a paradigm to automate the generation of boiler-plate code. Raising the abstraction level enables to focus on the domain details and separate the implementation details. This brings a number of specific benefits such as productivity, reduced cost, portability, drops in time-to-market, and improved quality. Overall, the main economic driver is the productivity gain achieved, which is reported by some studies [11,16].

A key artifact in MDD is a model transformation that defines the mappings between a model and other model or between a model and a code artifact. Although MDD was initially aimed at the generation of an individual program, shortly after appeared the need for families of programs.

Researchers and practitioners have realized the necessity for modeling variability of software systems where software product line engineering poses major challenges [20]. A software product line is a set of software intensive systems that are tailored to a specific domain or market segment and that share a common set of features [3,17].

For example, in industrial software systems the presence of different types of subsystems (e.g., exclusive subsystems from different providers) implies that

each is controlled in a similar though different way. This is typically achieved by defining two features that are not necessarily present in all possible systems. A feature is an end-user visible behavior of a software systems, and features are used to distinguish different software systems or variants of a software product line [13].

This impacts not only on the implementation, but on the modeling level. The modeling used in software product lines can be twofold. First, there are approaches for describing the variability of a software product line, e.g, there are feature models that specify which feature combinations produce valid variants [13]. Second, all variants in the product line may have models that describe their structure, behavior, etc.

However, when dealing with variability in an MDD scenario, there are further artifacts apart from models that may need to cope with such variability (e.g. model transformations may have to cope with variability imposed by the software product line). Hence, this paper takes a step back to study such impact into a broader perspective by analyzing the scenarios for Model-Driven Product lines. We shift our attention from the variability of models to a more general situation where the variability may embrace models, metamodels and model transformations. Hence, a realization of one feature may consist of variations of such artifacts.

Our contribution is to elaborate on motivating scenarios where model variability is not enough, illustrating this with a simple example and pointing to the need for extending variability beyond models.

Motivating Scenario

There are different scenarios when combining MDD and SPL, with differences depending on the modeling language used. It is not the same to use UML or to work with Domain Specific Languages (DSL).

The motivating scenario where model variability shows enough may happen in situations where:

1. The used metamodel is standard and so it is not subject to variability. For instance, when using a UML class diagram, it seems hard to make its metamodel variable since it is somehow standardized. This may apply generally to the metamodels of the UML.
2. The model transformations come from a common library of model transformations that are shared. For instance, consider the dozens of model transformations expressed in ATL (Atlas Transformation Language) that are available online[1].

Therefore, in scenarios with standard metamodels and shared transformations, the use of model variability seems enough. Indeed, at this point we wondered: what if not? Some motivating scenarios where model variability does not seem enough follows.

[1] http://www.eclipse.org/m2m/atl/atlTransformations/

1. There are different target models and model transformations may need to be customized for different targets. Model transformations share a significant common part while differ in some variable parts. For instance, consider the case where different implementation code is generated from the same source model. The target code is expected to be executed in different operating systems that could be defined well as features. Hence, there is a large proportion of shared code and some particularities bound to each operative system. In this situation, the application of variability to model transformations may enable to handle those differences in a unique model transformation.
2. In the previous situation, the source model conforms to a single metamodel while the target conforms to different metamodels. Depending on the proportion between common and variable parts of such metamodels, it may be of interest to apply variability as well to those metamodels. For instance, in the earlier example, the code for each operative system may conform to a specific metamodel. Although different, it may share a significant common part and so handling the metamodel variability may be required.
3. Model transformations define mappings between source and target metamodels. The variability in the target metamodel may happen as well in the source model. In general, it could be restricted to either the source, the target or both. Ultimately, since model transformations are chained, this need may propagate along the chain of model transformations.

Next, we illustrate this motivating scenario with a simplistic example.

Example

Although the motivating scenario is realistically more likely to occur within a larger system, we illustrate our ideas with a family of academic applications for managing libraries developed following MDD. Our motivating scenario demands to cope with the variability of models, metamodels and model transformations.

Consider a model of an academic application for managing libraries. An example library called `myLibraryModel` having many Books with their attributes Title, Author, and Category. The structure to which the model conforms is defined by a metamodel called `myAcademicMetaModel`. There is a family of library applications where the SPL paradigm is used.

The focus shifts from the development of an individual program to the development of reusable assets that are reused across the family. Consider a feature that extends `myLibraryModel` with magazines in addition to books. Each magazine will have a reference number, a title and the number of the issue. This may require the introduction of some variability to `myLibraryModel`. Books can also have other features apart from those already presented (e.g. including the year of publication, the publisher, or a small abstract of the book).

When composing features to get a product out of the product family, it might happen that the resulting composed model does not conform to the `myAcademicMetaModel` introduced earlier. Such metamodel may need to be re-

fined together with the model. Similarly, this may apply to the model transformations.

In general, different library products can be created, each one with the selected features. Adding features implies that the model, metamodel and model transformations may need to be refined to deal with variability.

Variability Beyond Models

Extending variability beyond models may initially impact on metamodels and model transformations.

A model is an instance conforming to a metamodel. More generally, a metamodel is a model of a modeling language where such language is specified [14]. In other words, the metamodel describes the elements of the domain and their relationships. Further work may address the variability of metamodels and its relationship to that of models.

Model transformations play a pivotal role in MDD because they turn the use of models for drawing into a more extensive model-driven usage where implementations can be directly obtained [19]. When talking about model transformations, two different approaches must be distinguished: model-to-model transformations and model-to-text transformations. Model-to-model transformations usually make use of rules that are defined as mappings between input and output metamodels. Model-to-text transformations combine rules with text templates that define the form of the output text.

This position paper outlines the need for extending variability beyond models. Further work may address the variability of model transformations and particularly its relationship to that of metamodels and models. Our current efforts are geared towards the step-wise refinement of models, metamodels and model transformations [21].

Related Work

Merging MDD and product lines is not new, we know of few examples that explicitly use features in MDD [6,5,7,8,9,18]. One is BoldStroke: a product-line for supporting a family of mission computing avionics for military aircraft [9]. Czarnecki introduces super-imposed variants and model templates to map features to models [5]. Weber et al. introduce the Variation Point Model that models variation points at the design level [23].

There is a line of work on feature-based composition of models. Feature-oriented model-driven development is an approach that ties feature composition to model-driven development [20]. Recent work by Apel describes superimposition as a model composition technique to support variability of product lines [1]. FeatureMapper is a tool that supports mapping features from feature models to solution artifacts [10]. These works do not yet consider the composition of model transformations or metamodels. Azanza introduces a metamodel-guided composition algorithm [2].

Sanchez-Cuadrado presents an approach for the reuse of model transformations in RubyTL by using an idea reminiscent of libraries in programming [4]. This first step towards model transformation reuse does not still incorporate the notion of product family. The superimposed modules of ATL language can be composed into different transformation definitions. This is not related to features, neither to the notion of composition demanded in a product family scenario [12]. Oldevik proposes an aspect-based extension of the MOFScript model-to-text transformation language, which is called a Higher Order Transformations (HOT) [15].

Aspect-Oriented MDD applies cross-cutting aspects to model artifacts. Particularly, there is recent work dealing with generators and model transformations using openArchitectureWare [22].

These works provide a strong foundation for current research efforts on model-driven product-lines such as AMPLE (http://ample.holos.pt/) and feasiPLe projects (http://feasiple.de).

Conclusions

This position paper discussed whether model variability is enough for realizing the Model-Driven Product-Line dream. We claim the need to shift our research attention from the variability of models to a broader perspective embracing the variability of models, metamodels and model transformations.

This is indeed the direction of our current research efforts where we are addressing the variability of models, metamodels, and model transformations and their relationships, since in our cases they appear to be often closely interrelated.

Acknowledgments. This work was co-supported by the Spanish Ministry of Science & Innovation under contract TIN2008-06507-C02-02.

References

1. S. Apel, F. Janda, S. Trujillo, and C. Kaestner. Model Superimposition in Software Product Lines. In *2nd International Conference on Model Transformations (ICMT 2009), Zurich, Switzerland, June*, 2009.
2. M. Azanza, D. Batory, O. Diaz, and S. Trujillo. Metamodel-guided Composition in Model Driven Product Lines. In *Draft under Review*, 2009.
3. P. Clements and L.M. Northrop. *Software Product Lines - Practices and Patterns*. Addison-Wesley, 2001.
4. J. Sanchez Cuadrado and J. Garcia Molina. Approaches for Model Transformation Reuse: Factorization and Composition. In *International Conference of Model Transformations (ICMT)*, 2008.
5. K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *4th International Conference on Generative Programming and Component Engineering (GPCE 2005), Tallinn, Estonia, Sep 29 - Oct 1*, 2005.

6. K. Czarnecki and M. Antkiewicz. Model-Driven Software Product-Lines. In *20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2005), San Diego, CA, USA, Oct 16-20*, 2005.
7. S. Deelstra, M. Sinnema, J. van Gurp, and J. Bosch. Model Driven Architecture as Approach to Manage Variability in Software Product Families. In *Workshop on Model Driven Architecture: Foundations and Applications (MDAFA), Enschede, The Netherlands, June 26-27*, 2003.
8. B. Gonzalez-Baixauli, M.A. Laguna, and Y. Crespo. Product Lines, Features, and MDD. In *1st Europeean Workshop on Model Transformation (SPLC-EWMT'05), Rennes, France, Sep 25*, 2005.
9. J. Gray and et al. Model Driven Program Transformation of a Large Avionics Framework. In *3th International Conference on Generative Programming and Component Engineering (GPCE 2004), Vancouver, Canada, Oct 24-28*, 2004.
10. F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *30th International Conference on Software Engineering (ICSE 2008), Companion*, pages 943–944, New York, NY, USA, may 2008. ACM.
11. D. Herst and E. Roman. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) - Approach: A Productivity Analysis. Technical report, TMC Research Report, 2003.
12. F. Jouault and I. Kurtev. Transforming Models with the ATL. In *International Conference on Model Driven Engineering Languages and Systems (MODELS 2005)*, 2005.
13. K. C. Kang and et al. Feature Oriented Domain Analysis Feasability Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, November 1990.
14. I. Kurtev. *Adaptability of Model Transformations.* PhD thesis, University of Twente, 2005.
15. J. Oldevik and O. Haugen. Higher-order transformations for product lines. *Software Product Line Conference, International*, 0:243–254, 2007.
16. OMG. MDA Success Stories. http://www.omg.org/mda/products_success.htm.
17. K. Pohl, G. Bockle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles and Techniques.* Springer, 2006.
18. D. Schmidt, A. Nechypurenko, and E. Wuchner. MDD for Software Product-Lines: Fact or Fiction. In *Workshop at 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2005), Montego Bay, Jamaica, Oct 2-7, 2005*, 2005.
19. S. Sendall and W. Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42–45, 2003.
20. S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May*, 2007.
21. S. Trujillo and A. Zubizarreta. Lock-Step Refinement of Models, Metamodels and Model Transformations in Model-Driven Product-Lines. In *Draft under Review*, 2009.
22. M. Voelter and I. Groher. Handling Variability in Model Transformations and Generators. In *Proc of the DSM Workshop at OOPLSA*, 2007.
23. D.L. Webber and H. Gomaa. Modeling Variability in Software Product Lines with the Variation Point Model. *Science of Computer Programming*, 53, 2004.

A Model-based Product-Line for Scalable Ontology Languages*

Christian Wende and Florian Heidenreich

Technische Universität Dresden
Institut für Software- und Multimediatechnik
D-01062, Dresden, Germany
{c.wende|florian.heidenreich}@tu-dresden.de

Abstract. Research in the area of semantic web brought up a plethora of languages to represent ontologies. They all differ in expressiveness and reasoning efficiency. Thus, the choice of a specific language means a trade-off between reasoning capabilities and performance. This paper outlines how techniques from product-line engineering can be combined with model-based language engineering to allow for organising ontology languages in a language family and configuring them for concrete use cases.

1 Introduction

Ontologies provide means for encoding knowledge about specific domains and often include reasoning rules that allow for deriving implicit knowledge. The manifold of domains that ontology languages are applied to led to a plethora of languages to represent ontologies. In this paper we focus on a subset of ontology languages based on the OWL2 standard [25] that provides an implementation for Description Logics (DLs) [1]. They share the common approach of representing knowledge using hierarchies of unary atomic *concepts* that are augmented with binary logical operators or *roles* to describe concept relationships. Expressiveness and reasoning efficiency is directly determined by the concrete binary operators a language provides [6]. Thus, the choice of a specific language means a trade-off between performance and reasoning capabilities. To achieve scalability of ontology languages both performance and functional requirements of a specific use case need to be consider.

Customising ontology languages from a language family has been identified to be a promising approach to address specifics of the use case they are applied to [27]. In addition, it has a number of other benefits: (1) Common language features can be reused among the language family members. (2) The family members are organised in a systematic way. (3) Specific expressiveness and reasoning requirements can be addressed by recombining existing language features. (4)

* This research has been co-funded by the German Ministry of Education and Research (BMBF) within the project feasiPLe and by the European Commission within the FP7 project MOST contract number 216691.

Ontology language evolution can be realised by contributing new features to the language family. (5) Language tooling (e.g., dedicated parsers, printers, editors, and reasoners) can be generated. To support a systematic approach for language customisation, we argue for combining techniques from product-line engineering and model-based language engineering. This paper contributes a systematic classification of ontology languages using the the paradigm of feature modelling. This enables feature-based customisation of OWL2 w.r.t. a specific use case. Second, we introduce a model-driven process to generate the tool infrastructure that is crucial for the adoption of the language variant in knowledge modelling.

The rest of this paper is structured as follows: Section 2 discusses the application of feature modelling to specify commonalities and variability among a family of OWL2-based ontology languages and explains how to use this variability information to select a language variant matching the needs of a specific use case. Section 3 discusses the generation of the tool infrastructure that is crucial for the application of a custom language variant. We introduce a model-driven process to automatically generate a parser, printer, and editor for a given language variant. For the sake of complexity we consider the generation of semantic tooling (e.g. reasoners) out of the scope of this paper. However, even with existing reasoners reasoning efficiency can be scaled by purely syntactic language adaptation [8, 13, 23]. We conclude and present future work in Section 5.

2 Feature-Based Ontology Language Configuration

The syntactic and semantic expressiveness of ontology languages is described inductively by the DLs [1] constructors they provide for knowledge representation [6]. The connection between reasoning characteristics of an ontology language and its logical constructors motivates a guidance of language configuration by means of singular logical constructors. Since constructors are interdependent and interact, it is necessary to specify dependencies and relations between them. This can be done using the paradigm of feature modelling.

The feature model depicted in Fig. 1 is based on the constructors used in DLs to describe the expressiveness of ontology languages. It describes commonalities and variability in our OWL2-based family of ontology languages. Every feature represents a single constructor by its textual name and the letter used in the usual naming conventions for DLs (for an overview see [1]). In addition, a cardinality is given for every language feature. A feature connected by a line ending with a filled circle represents a mandatory feature and is used in every OWL2 language variant. Empty circles identify optional features.

All possible feature combinations span the variation space for our language family. A concrete selection of features from this variation space describes a specific ontology language variant. Its simplest member can be built using only the mandatory features (Concepts, Top, Bottom, Intersection, AtomicNegation and ValueRestriction). It corresponds to the minimal Attributive Language $\mathcal{AL}$ that is considered the base of our desciption logics language family. By including for example the optional features $\mathcal{R}+$, $\mathcal{C}$, $\mathcal{H}$, $\mathcal{O}$, $\mathcal{I}$, $\mathcal{N}$, and $(\mathcal{D}+)$ one could

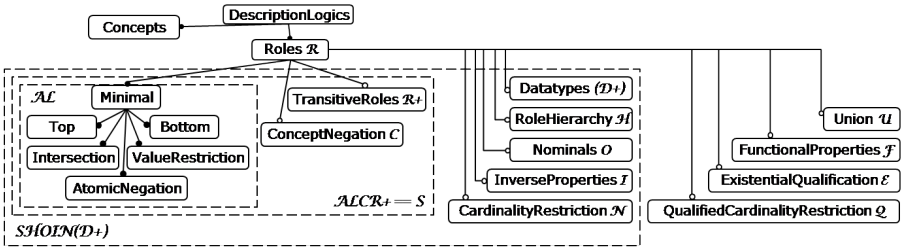


Fig. 1. Feature model that describes the variability space of ontology languages

configure the language variant $\mathcal{SHOIN}(\mathcal{D}+)$ with the expressiveness of OWL DL [21]. When features or feature combinations are annotated with their implications on reasoning efficiency (as studied in [6]), this feature model can be used to guide the customisation process regarding language expressiveness and efficiency. In addition, one can identify gaps in the language hierarchy and fill them by configuring constructors or by adding new logical constructors.

This feature-based modularisation of languages introduces a foundation for making reasoning technology more scalable: Optimised language variants can be configured that take the concrete expressiveness and efficiency requirements of a specific use case into account.

3 Model-based Ontology Language Engineering

There are three driving forces that made us address the problem of building the language family in a model-based way. First, modelling techniques offer support for transformation of models into other representations, e.g., reduced models or more specific models. This is particularly needed when deriving a concrete language from the language product-line. Second, models can be easily used for code generation, which is needed to automatically generate tool support for the concrete language. This is an important point, since building tools by hand is an expensive task. Third, models usually share a common metamodel which directly supports interoperability between different tools that are based on the metamodelling technology at hand. We decided to use Eclipse and the Eclipse Modelling Framework (EMF)[1] because of the variety of tools that exist to create, edit, and transform models based on EMF.

3.1 Language Family Development Process

The initial starting point for developing the ontology-language product line is modelling the problem space by means of a feature model (cf. Fig. 1). We used a lightweight version of the EMF/Ecore-based feature metamodel developed in the feasiPLe project[2].

[1] `http://www.eclipse.org/modeling/emf/`
[2] `http://feasiple.de`

Since we want to (1) transform the description of the solution space, that is, the realisation of the language syntax features and (2) generate tooling (e.g., parsers, printers, editors, ...) out of the specification, we used Ecore to build an OWL metamodel and EMFText[3] [11]—a model-based tool for defining textual concrete syntax for models. EMFText offers a dedicated language for specifying text syntax for models called CS which is similar to Extended Backus-Naur Form (EBNF). With CS, rules are defined which specify textual syntax for metaclasses of a given Ecore-based metamodel. In our case, we first modelled the OWL metamodel (based on [3]) which defines the abstract syntax of the various language features. Then we derived CS rules that describe the textual syntax for the concepts corresponding to OWL Manchester syntax [14].

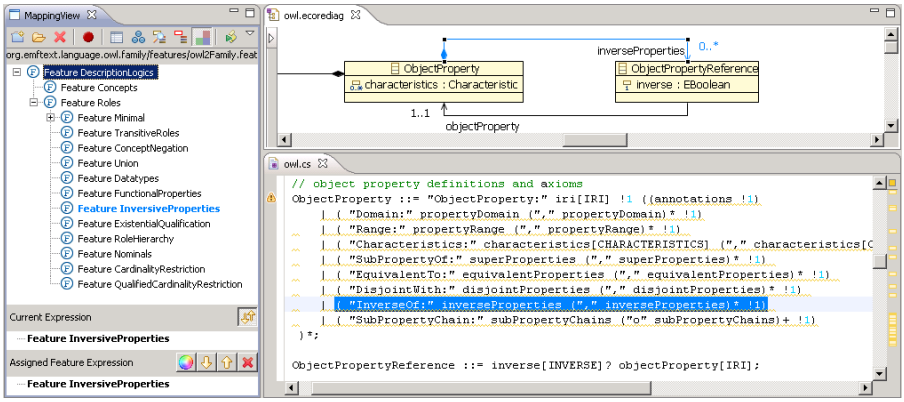


Fig. 2. Using the FeatureMapper to map OWL features to specific parts of OWL2 abstract and concrete syntax

One observation we made in previous work [10, 12] is that it is crucial to have a mapping between the problem space (i.e., a variability description of language features in a feature model) and the solution space (i.e., a concrete realisation of specific language features in EMFText's CS language). This mapping can then be used both for visualising dependencies between features and realisation artefacts and for automating the product-instantiation process. An overview of the models used to specify the the ontology language product line and their relationships is depicted on the right part of Fig 3.

The FeatureMapper[4] is a tool that was specifically developed to tackle that problem and allows for creating a mapping between feature models and EMF-/Ecore-based models. Since EMFText is built with itself, the CS language is again a model-based language which can be used in combination with FeatureMapper. We extended the FeatureMapper to also support mapping and

[3] `http://www.emftext.org`
[4] `http://www.featuremapper.org`

visualisation of textual languages that are created by EMFText and used it to create a mapping between the language features in the feature model and the realisation of those features in the CS specification. Both tools are depicted in Fig. 2 where the view on the left side contains the FeatureMapper with the feature model for the ontology-language product line. The editors on the right side show a part of the OWL metamodel and CS specification for existing languages in the OWL language family. To present the concrete mapping to the language developer the FeatureMapper colours the elements in the CS specification and the OWL metamodel in accordance to the colour of the feature in the feature model they are mapped to. The example depicts the mapping used to define the syntactic realisation of the feature `InverseProperties`. For that purpose the association `inverseProperties` between `ObjectProperty` and `ObjectPropertyReference` that is used in the abstract syntax to represent inverse properties and the corresponding fragment of concrete syntax are mapped to the feature `InverseProperties`.

3.2 Language Derivation Process

After we defined the scope of the language product line, we are able to derive concrete instances (i.e., languages) from that definition. To do so, the FeatureMapper provides means to transform models according to a given feature selection (a variant model) by interpreting the mapping model that contains the various mappings between features and model elements. After this transformation step, a reduced CS specification is produced that only contains the rules that are needed for the selected language features. This reduced CS specification is then used by EMFText to generate a dedicated parser, printer, and editor. The process of feature-based language derivation is depicted in Fig. 3.

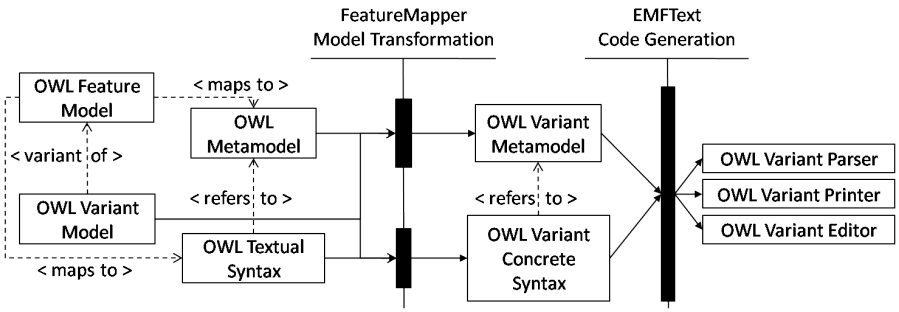


Fig. 3. Model-based Ontology Language Derivation Process

4 Related Work

Scalability is a widely discussed and very prominent topic that constitutes a main challenge for the application of ontology languages [6, 19]. The objective of efficient reasoning has led to a manifold of languages with specific reasoning characteristics [5, 2, 25, 20, 22]. The idea that scalability can be achieved by selecting a language from this manifold that is appropriate for the requirements of a specific use case is not new. In [17] the authors provide a comprehensive comparison of nine DLs-based ontology languages w.r.t. their syntactic features and reasoning efficiency. The results of this survey are envisaged as guideline for matching ontology languages to use cases. The fact that the OWL2 standard [25] introduces three languages with different expressiveness and reasoning characteristics illustrates that nowadays ontology languages are already designed with that idea in mind. Our work picks up this idea and presents a methodical approach and a technological infrastructure to get from language features selected for a use case to the actual implementation of the language variant and the corresponding tool infrastructure. In addition, the presented model-driven approach is suited to deal with the proceeding evolution of ontology languages by supporting the introduction of new language features.

A second branch of work addressing the scalability issue deals with the development of more efficient reasoners or the enhancement of existing reasoning techniques. This led to numerous highly optimised native ontology reasoners [9, 26, 28] that perform well even for expressive ontology languages but only for reasonable sized ontologies. Large amounts of facts often result in poor response times that impede applicability in practice [17]. Approaches presented in [4, 7, 29] store ontologies in relational databases to use the optimised database query engines for ontology reasoning. As discussed in [19] this leads to increased load-time but more efficient reasoning compared to native ontology systems. In [16] ontologies are represented in disjunctive datalog programs. Additional algorithmic optimisation can be applied on the datalog facts to enhance reasoning efficiency. Other approaches [8, 13, 23, 24] enhance reasoning efficiency by approximating more expressive ontology languages to less expressive ones. The reduction of complexity leads to better reasoning performance while preserving the completeness and soundness of the reasoning results. Reasoners and approximation approaches are designed for a very a specific subset of DLs features. Using a generic tool like Sesame [4] that allows for exchanging the reasoning back-end they can be combined with our feature-based ontology language configuration. Thus, we could provide appropriate (semantic) reasoning infrastructure w.r.t. a specific language variant.

5 Conclusion

The contribution of this paper is twofold: First, we transferred the existing DL-based classification of ontology languages to the paradigm of feature modelling. This enables feature-based customisation of OWL2 w.r.t. a specific use case.

Future work needs to enrich the current classification with further metadata (e.g., efficiency annotations) that can be used to guide language configuration. In addition to that, other ontology language extensions—like rule extensions [15] or probabilistic extensions [18]—should be included in such classification.

Second, we presented a model-driven process to generate a dedicated parser, printer, and editor from a given variant specification. This infrastructure is crucial for the application of the language variant in knowledge modelling. In addition, the effort to provide new language extensions or language adaptations can be reduced by using the introduced model-driven process.

The solution presented in this paper only tackles syntactic language variations. To advance the impact on reasoning efficiency and language applicability, future work needs to investigate the possibilities of deriving language specific infrastructure w.r.t. language semantics. This relates to topics as semantic approximation of OWL [24], and composition of language semantics [30].

The introduced approach for applying techniques of product-line engineering for the systematic development of language families is not limited to ontology languages. Thus, future work will also address its extension to other languages.

References

1. F. Baader. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.
2. D. Berardi, A. Cali, D. Calvanese, and G. D. Giacomo. Reasoning on UML Class Diagrams. *Artificial Intelligence*, 168, 2003.
3. S. Brockmans, P. Haase, and B. Motik. OWL 2 Web Ontology Language MOF-Based Metamodel. Available at http://www.w3.org/2007/OWL/wiki/MOF-Based_Metamodel, 2007.
4. J. Broekstra, A. Kampman, and F. Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. *The Semantic Web ISWC 2002*, pages 54–68, 2002.
5. D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Tractable reasoning and efficient query answering in description logics: The DL-Lite family. *Journal of Automated Reasoning*, 39(3):385–429, 2007.
6. F. Donini, M. Lenzerini, D. Nardi, and W. Nutt. The complexity of concept languages. *Information and Computation*, 134(1):1–58, 1997.
7. Q. Fang, Y. Zhao, G. Yang, and W. Zheng. Scalable Distributed Ontology Reasoning Using DHT-Based Partitioning. *Proceedings of the 3rd Asian Semantic Web Conference on The Semantic Web*, pages 91–105, 2008.
8. P. Groot, H. Stuckenschmidt, and H. Wache. Approximating Description Logic Classification for Semantic Web Reasoning. 2005.
9. V. Haarslev and R. Moller. RACER system description. *Automated Reasoning - Lecture Notes in Computer Science*, pages 701–706, 2001.
10. F. Heidenreich, I. Şavga, and C. Wende. On Controlled Visualisations in Software Product Line Engineering. 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)*, Sept. 2008.

11. F. Heidenreich, J. Johannes, S. Karol, M. Seifert, and C. Wende. Derivation and Refinement of Textual Syntax for Models. In *Proceedings of the 5th European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009)*, June 2009. To appear.
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. P. Hitzler and D. Vrandecic. Resolution-based Approximate reasoning for OWL DL. *The Semantic Web ISWC 2005*, 3729, 2005.
14. M. Horridge and P. F. Patel-Schneider. OWL 2 Web Ontology Language: Manchester Syntax. Available at http://www.w3.org/TR/2008/WD-owl2-manchester-syntax-20081202/, 2008.
15. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. *W3C Member Submission*, 2004.
16. U. Hustadt, B. Motik, and U. Sattler. Reducing SHIQ- description logic to disjunctive datalog programs. *Proceedings of Principles of Knowledge Representation and Reasoning*, pages 152–162, 2004.
17. C. Keet and M. Rodriguez. Comprehensiveness versus Scalability: Guidelines for choosing an appropriate knowledge representation language for bio-ontologies. *KRDB Research Centre Technical Report, KRDB07-5*, 2007.
18. T. Lukasiewicz. Probabilistic Deduction with Conditional Constraints over Basic Events. *Journal of Artificial Inteligence Research*, 1999.
19. L. Ma, Y. Yang, Z. Qiu, G. Xie, Y. Pan, and S. Liu. Towards a complete OWL ontology benchmark. *The Semantic Web: Research and Applications*, 2006.
20. D. McGuinness, R. Fikes, J. Hendler, and L. Stein. DAML+ OIL: an ontology language for the Semantic Web. *IEEE Intelligent Systems*, 17(5):72–80, 2002.
21. D. L. McGuinness and F. van Harmelen. OWL Web Ontology Language Overview. Available at http://www.w3.org/TR/owl-features/, 2004.
22. J. Pan and I. Horrocks. RDFS (FA): connecting RDF (S) and OWL DL. *IEEE Transactions on Knowledge and Data Engineering*, 19:192.
23. J. Pan and E. Thomas. Approximating OWL-DL Ontologies. Proceedings of the 22nd National Conference on Artificial Intelligence (AAAI-07), 2007.
24. J. Pan, E. Thomas, D., and Sleeman. Ontosearch2: Searching and querying web ontologies. *Proceedings of WWW/Internet*, 2006.
25. P. F. Patel-Schneider, P. P. Hayes, and I. Horrocks. OWL 2 Web Ontology Language: Profiles: OWL-R. W3C Working Draft. Available at http://www.w3.org/TR/owl2-profiles/♯OWL-R, 2008.
26. E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: Science, Services and Agents on the World Wide Web*, 5(2):51–53, 2007.
27. H. Stuckenschmidt. Statement of Interest: Towards Ontology Language Customization. *Ontologies and Information Sharing*, 2001.
28. D. Tsarkov and I. Horrocks. FaCT++ description logic reasoner: System description. *Automated Reasoning - Lecture Notes in Computer Science*, 4130:292, 2006.
29. J. Zhou, L. Ma, Q. Liu, L. Zhang, Y. Yu, and Y. Pan. Minerva: A scalable OWL ontology storage and inference system. *The Semantic Web ASWC 2006*, 4185:429, 2006.
30. S. Zschaler and C. Wende. Collaborating Languages and Tools - A Study in Feasibility. *Technical Report TU Dresden - TUD-FI08-06*, 2008.

Multi-Variant Modeling*

Concepts, Issues, and Challenges

Bernhard Westfechtel[1] and Reidar Conradi[2]

[1] Angewandte Informatik 1, Universität Bayreuth
D-95440 Bayreuth
Bernhard.Westfechtel@uni-bayreuth.de
[2] Norwegian University of Science and Technology (NTNU)
N-7491 Trondheim, Norway
Reidar.Conradi@idi.ntnu.no

Abstract. When applying model-driven engineering to a product line, there is a need to deal with multi-variant models. So far, in industry software product line engineering has primarily been applied to data represented in (text) files and directories. Applying variation to non-textual models is more difficult, since models are complex structured objects. This paper presents basic concepts and discusses some issues and challenges with respect to multi-variant modeling.

1 Introduction

Model-driven engineering (MDE) denotes an approach to software development which strongly emphasizes the use of explicit and formal models throughout the whole software lifecycle. Models do not merely serve as documentation. Rather, MDE aims at developing executable models, eliminating the need for manual programming.

According to [1], *software product line engineering* (SPLE) is a paradigm to develop software applications using platforms and mass customization. In this context, the variability model plays a central role because it defines the dimensions of variation supported by the product line.

So far, SPLE has been applied primarily to data represented as (text) files and directories. This picture is beginning to change. *Model-driven product line engineering* (MPLE) has been addressed in several research projects [2–4], and initial tool support has been provided by industry [5].

Nevertheless, the integration of model-driven and product line engineering still has to be explored further; the field has not yet reached maturity. In this paper, we focus on *multi-variant modeling*, i.e., the construction and representation of models incorporating multiple variants. We present concepts, issues, and challenges of multi-variant modeling, and investigate the state of tool support.

* 1st International Workshop on Model-Driven Product Line Engineering (MDPLE 2009), Twente, The Netherlands, June 2009

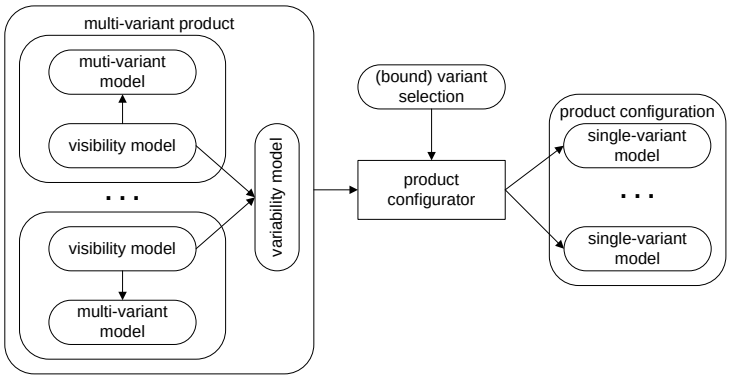


Fig. 1. Framework for multi-variant modeling

2 Multi-Variant Modeling

2.1 Conceptual Framework

Figure 1 introduces a simple *conceptual framework* for multi-variant modeling and product configuration. A *multi-variant software product* is the *union of all variants* of the software product. It is composed of a set of interrelated *multi-variant models*. A global *variability model* defines the variation points and variants the product line is required to support. For each multi-variant model, a *visibility model* defines in which variants the elements of the model are contained. To obtain a specific product variant, a bound *variant selection* is defined. Thus, for each variation point a specific variant is selected. The *product configurator* evaluates the visibilities of the elements of multi-variant models against the variant selection. It selects only those elements which are visible under the current selection. Thus, the product configurator produces a set of *single-variant models*, which are subsets of the underlying multi-variant models.

2.2 Variability Model

A *variability model* describes the variation points along which a software product may vary, as well as supported combinations of variants (e.g., Feature-Oriented Domain Analysis [6] or the Orthogonal Variability Model [1]). In the context of this paper, we define a variability model as a relation $vm \subseteq vp_1 \times \ldots \times vp_m$, where each vp_i represents a variation point[3]. The relation vm could be defined *extensionally*, i.e., by enumerating all tuples in vm. Usually, vm is defined *intensionally* by some predicate vp such that $vm = \{(v_1, \ldots v_m) | vp(v_1, \ldots v_m)\}$. A *variant selection* is a subset $vs \subseteq vm$. The selection is *bound* if $|vs| = 1$; a bound selection corresponds to a single *product variant*.

[3] For feature models, it would be more appropriate to define a variant as a set rather than a tuple of features and the variability model as a set of variants. However, this difference is immaterial to the discussion below.

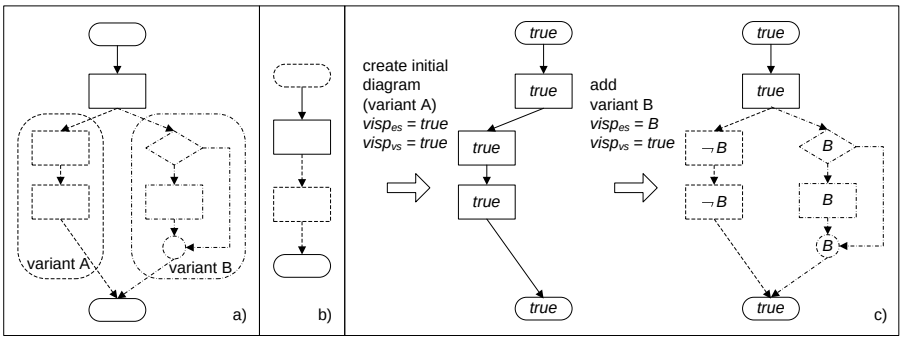


Fig. 2. Example: flow diagrams

2.3 Single- and Multi-Variant Models

In the context of MDE, a *model* is an abstraction of the system to be built. A model m is represented as a pair (E, S), where E denotes a set of a elements and S is a structure defined on these elements. A model is an instance of some *metamodel*, which defines the types of available model elements (e.g., data objects, composition relations) and the constraints imposed on their combinations (e.g., by class diagrams and OCL constraints). We may view a metamodel mm as a set of (potential) models conforming to this metamodel, i.e., $mm = \{m | mp(m)\}$, where mp denotes a (complex) predicate to be satisfied.

The simplest way to represent a *multi-variant model* is to reuse the metamodel of *single-variant models*, i.e., single- and multi-variant models are instances of the same metamodel mm. This approach assumes that the metamodel mm has been defined in such a way that variation may be expressed under the constraints of mm. Consider, e.g., class diagrams, where variation may be expressed by disjoint subclassing or by realizing interfaces by alternative classes. This kind of variability is called *intrinsic*. In fact, many approaches to SPLE tacitly assume that it suffices to exploit intrinsic variability [4, 2, 7, 1]. Unfortunately, not all metamodels support intrinsic variability:

Example 1 (Flow diagrams). A (sequential) *flow diagram* is a connected graph which consists of a single start node (no incoming, one outgoing control flow, cardinality 0/1), as well as multiple activity nodes (1/1), binary decision nodes (1/2), join nodes (+/1), and end nodes (1/0). A multi-variant control flow diagram is given in Figure 2a. Here, solid and dashed lines denote universal (mandatory) and variant-specific (optional) parts, respectively. The multi-variant control flow diagram is not a valid control flow diagram: The first activity node has more than one outgoing edge, and the end node has more than one incoming edge. □

A multi-variant model is a *union of single-variant models*. In general, we cannot expect that such a union is a valid instance of a single-variant metamodel. In particular, cardinality constraints for single-variant models may not hold for their superimposition.

In *universal multi-variant modeling*, all model elements should be allowed to vary (regardless of whether the modeler will apply variation universally or only with certain methodological constraints). This can be achieved only by *extrinsic variability*, i.e., by a "meta-level" mechanism universally applied to any kind of model. Such mechanisms are provided e.g. by *software configuration management systems*, which add version control to software objects.

2.4 Visibility Models

Each multi-variant model is associated with a *visibility model* which defines the visibilities of the model elements. We may consider a multi-variant model as a set of pairs $(e, vis(e))$, where e denotes an element and $vis(e)$ its visibility. Visibilities may be represented in different ways. For example, in the UML stereotypes, tagged values, or annotations may be used to express visibilities. Formally, the visibility model may be defined as a function vis which maps each model element onto the set of variants in which it is visible: $vis : E \rightarrow 2^{vm}$, where $vis(e) = vs$ is some (not necessarily bound) variant selection. The visibility of a product element e may be presented by some *visibility predicate* $visp(e)$ such that $vis(e) = \{(v_1, \ldots, v_m) | visp(e)(v_1, \ldots, v_m)\}$.

Let e_1 and e_2 denote two variants of the "same" element (which assumes some sameness criterion, e.g., element identifiers). The visibilities of different elements must be disjoint:

$$e_1 \neq e_2 \Rightarrow vis(e_1) \cap vis(e_2) = \emptyset \tag{1}$$

2.5 Product Configuration

The *product configurator* takes a set of multi-variant models decorated with visibilities and a variant selection. The variant selection should be consistent, i.e., $vs \subseteq vm$ or (in terms of predicates vsp for the variant selection and vp for the variability model, as defined in Subsection 2.2):

$$vsp \Rightarrow vp \tag{2}$$

The product configurator returns a set of models whose elements are *visible* under the current selection. Thus, for each selected element $vs \subseteq vis(e)$. In terms of predicates, this may be rephrased as follows:

$$vsp \Rightarrow visp(e) \tag{3}$$

If the variant selection is bound, for each element at most one variant will be selected. In this case (which we will assume in the following), the product configurator returns a set of *single-variant models*.

The configured models should be *valid* instances of the corresponding single-variant metamodels. That is, the product configurator should produce a *syntactically consistent result* (which is a necessary, yet not sufficient condition that the result is usable as it stands). Unfortunately, this requirement is hard to guarantee in general. Counterexamples may be constructed easily:

Example 2 (Inconsistencies in configured models). Three errors are contained in the multi-variant flow diagram of Figure 2b:

1. The mandatory start node has been marked as optional (variant specific).
2. Its outgoing control flow has been marked as mandatory, even though the start node is optional (this may give rise to a dangling control flow).
3. If the second activity node is omitted (with its adjacent control flows), the diagram is no longer connected. □

Thus, *constraints* must be imposed on the combination of visibilities, e.g., to preserve product properties (well-formedness rules). For example, the first error in Example 2 may be avoided by requiring that a mandatory element must not be marked as optional. More precisely, the visibility of a mandatory component e_1 (the start node) must be implied by the visibility of the enclosing composite e_2 (the flow diagram):

$$visp(e_2) \Rightarrow visp(e_1) \tag{4}$$

The second error may be avoided by a constraint referring to the *dependencies* between model elements. For some model $m = (E, S)$, the structure S on its elements E induces a dependency relation $D \subseteq E \times E$, where $d(e_1, e_2)$ holds if and only if e_1 depends on e_2. The visibility of a dependent element e_1 must not exceed the visibility of its master e_2:

$$visp(e_1) \Rightarrow visp(e_2) \tag{5}$$

Since a mandatory component existentially depends on its enclosing composite, the equations above jointly imply that their visibilities must be equal.

The *product constraints* defined above are necessary, yet not sufficient conditions for product consistency. In general, arbitrarily complex constraints may be defined in the metamodel. This is exemplified by the third error (the diagram is not connected).

2.6 Multi-Variant Editing

In the previous subsection, we have discussed product configuration, tacitly assuming that a set of multi-variant models has already been created. In the following, we will address the question how multi-variant models come into being. This requires some way of *multi-variant editing*.

In the case of *unfiltered editing*, the user edits a multi-variant model, where all variants are displayed and modified simultaneously. This corresponds to editing a program file with conditional compilation statements. The advantage of unfiltered editing consists in the fact that the user sees *all variants simultaneously* and therefore may assess the impact of changes better than in the case of filtered editing to be explained below. On the other hand, the information overload incurred by the overlay of multiple variants may be difficult to manage.

In the case of *filtered editing*, the user edits a single variant called *view*. Usually, it is desired that editing may affect more than just the variant displayed in the view. One way to deal with this problem is to define an *edit set* which defines the scope of a change [8]. A view is a bound variant selection vs, and the edit set is a set of variants es containing vs. All elements are affected whose visibilities overlap with the edit set.

An alternative approach of filtered editing makes use of *layers* or *change sets* [9]. The user defines a single-variant view by a sequence of layers. Editing operations refer to some designated layer, into which the performed changes are aggregated. The scope of the changes is confined to the designated layer, which can be used to construct multiple variants.

3 The UVM Approach

3.1 Background

This section briefly presents the UVM approach as an example for multi-variant management. UVM, which stands for *Uniform Version Model* [10], provides for extrinsic variability. UVM offers a basic way to express variant handling, independent of (orthogonal to) the data or product model and the application of such models in a user context. The approach has been applied to both textual and non-textual data (e.g., entity-relationship diagrams [11]).

UVM was developed in the context of *software configuration management* (SCM). The SCM landscape is dominated by systems which focus on temporal and cooperative versioning (usually based on version graphs, see e.g. Subversion [12], CVS [13] or ClearCase [14]) and provide only limited support for logical versioning (variants). For this reason, SCM and SPLE are often perceived as more or less disjoint disciplines. For example, BigLever Software Gears covers only product line variants and assumes that revisions of the product line are managed by an SCM system [15].

However, work on multi-dimensional variation has been performed in several SCM research prototypes, as well [16, 17]. Moreover, a few research projects in SCM were dedicated to the development of a *uniform version model*, supporting all dimensions of evolution through a single base mechanism (ICE [18] and UVM [10]).

3.2 Basic Versioning

In UVM, each stored and versioned information *item* (with application-specific granularity and interpretation) has a *visibility* tag called vis. There are as many pairs of (visibility-value,item-value) as there are versions of this item. The visibility is a logical expression in 3-nary logic over an open set of global versioning attributes.

UVM is based on *filtered editing*. The view is defined by a *version choice*, i.e. a bound combination of versioning attributes. Depending on some version choice vc, the visibilities will evaluate to $false$ or $true$. This assumes, however, that all attributes are completely bound, i.e., no unbound version settings left. The single-variant view is then the (sequential) union of item-values with visibilities $vis = true$, and application-specific tools will only see this visible subset.

The edit set is defined by an *ambition*, i.e., a logical expression with a partial binding of versioning attributes. The version choice must lie inside the ambition:

$$vc \Rightarrow amb \tag{6}$$

The edit set defines the scope of the change. The visibilities of elements are managed automatically. All new elements e inherit their visibilities from the ambition:

$$vis_{new}(e) = amb \tag{7}$$

Deletion of an element does not mean that the element is physically removed. Rather, all versions of the element are no longer visible under the edit set (ambition). This is achieved by constraining the visibilities:

$$vis_{new}(e) = vis_{old}(e) \wedge \neg amb \tag{8}$$

Example 3 (Multi-variant editing).

An example is given in Figure 2c. In the first step, the predicates for the edit set and the view are both set to $true$. This results in an initial variant of the flow diagram, where all elements carry the visibility $true$. For the nodes, the visibility is shown inside the node. Edges carry visibilities, as well, but their visibilities are not displayed. The flow diagram created so far corresponds to variant A in Figure 2a. The second step is performed under the visibility B both for the view and the edit set. All elements created so far qualify for the view. In the view, two activities (on the left) are deleted, and a new branch of the control flow diagram is inserted (on the right). The new elements get visibility B, the visibility of the deleted elements is constrained to $\neg B$. After the second step, we obtain the multi-variant flow diagram of Figure 2a. Variant A is represented implicitly (as $\neg B$). □

3.3 Consistency Control

UVM supports consistency control through the following mechanisms:

Automated management of visibilities All performed changes are tagged consistently with visibilities according to the rules given above. Thus, many errors can be avoided which the user may introduce by defining visibilities individually and manually.

Enforcement of product constraints In a data model layer above the core, product constraints may be implemented. For example, the visibility of an association may be narrowed down automatically (i.e., it is not visible if one of its ends is not visible) [11].

Enforcement of version constraints The user may define constraints on the combination of versioning attributes (e.g., the variants $Windows$ and $Linux$ are mutually exclusive). These constraints are exploited by a versioning assistant which supports the user in consistent version selections.

However, UVM cannot guarantee in general that a new version choice gives us a single-variant that is consistent, either syntactically or wrt. static or dynamic semantics. So a *merge-edit operation* may be needed when two attribute settings are combined in a new version choice. In this way, a *feedback cycle* is realized: By editing a configured variant, the capabilities of the product line are enhanced.

Example 4 (Merge-edit to reconcile mutually conflicting items from overlapping versions).

Consider a class diagram containing some class C, where two users have both added some method called M in parallel. This results in two method variants M_1 and M_2 with visibilities V_1 and V_2, respectively. Now, both variants are combined (automatic *merge step*). This is achieved by the version choice $V_1 \wedge V_2$, which is also the ambition. Since the version choice implies both visibilities (e.g., $V_1 \wedge V_2 \Rightarrow V_1$), both method variants M_1 and M_2 are selected simultaneously. This results in a name clash: Method M is declared twice.

The user may now resolve the conflict in any desired way (e.g., by deleting or renaming one of the method variants, or by merging both variants manually into a single combined variant). This *manual edit step* will eventually result in a consistent class diagram (and associated implementation). When the changes are committed, the visibilities are updated for the changed elements. For example, in the case of a manual merge, a new method variant M_3 is created with visibility $V_1 \wedge V_2$, and the visibilities of the old variants are narrowed down. For example, the new visibility of M_1 will be $V_1 \wedge \neg(V_1 \wedge V_2) = V_1 \wedge \neg V_2$. Thus, the old variants will not be selected any more under the combination $V_1 \wedge V_2$. □

---

4 Related Work

Table 1 compares UVM against a few tools for multi-variant modeling, all of which rely on internal rather than external variability.

Feature Mapper [19, 4] is based on feature modeling and supports the annotation of EMF models with visibilities. The user may switch between single- and multi-variant views. Visibilities may be defined individually for each model element. Feature Mapper also offers a recording mode where a visibility is defined beforehand and all elements created are decorated automatically with this visibility. Consistency control is not addressed; Feature Mapper cannot guarantee (or check) that a configured model variant is syntactically consistent.

Feature-based model templates were introduced in [7]. Cardinality-based feature models [20] are used for the variability model. The approach refers to models whose metamodels are defined with MOF and OCL. Product consistency of configured variants may be checked automatically [21]: By an abstract interpretation of well-formedness rules given in OCL for the respective metamodel, it can be checked whether each variant which may be configured according to a given feature model is consistent with respect

	Feature Mapper	**Feature-based model templates**	**EASEL**	**UVM**
Domain	EMF models	MOF- and OCL-defined models	Class diagrams	(generic)
Variability	internal	internal	internal	external
Variability model	feature model	feature model	layers	three-valued logic
Variability constraints	not supported	excludes and requires	implications	logical expression
Multi-variant representation	interleaved deltas	interleaved deltas	directed deltas	interleaved deltas
Visibilities	logical expression	logical expression	defined by layers	logical expression
Visibility management	manual or automatic	manual	automatic	automatic
Multi-variant editing	filtered or unfiltered	unfiltered	filtered	filtered
Product consistency	not addressed	verifiable	checked, inconsistencies tolerated	(in data model layer)

Table 1. Classification of tools for multi-variant modeling

to these rules. In this respect, feature-based model templates go beyond the capabilities of competing approaches.

EASEL [22] supports multi-variant editing for class diagrams. In contrast to the other approaches compared here, EASEL relies on layers, i.e., directed deltas rather than interleaved deltas. Instead of decorating model elements with visibilities, changes (addition, deletion, and modification of model elements) are aggregated into layers which may be composed dynamically. EASEL detects contradictory or inconsistent combinations of layers automatically by analyzing the change operations contained in the layers. Inconsistencies are tolerated, and may be fixed by the user in a similar way as in UVM (merge-edit).

5 Conclusion

In this paper, we have examined multi-variant modeling, and we have briefly discussed how an approach from SCM (the Uniform Version Model) may be applied to multi-variant modeling. From our examination, we draw the following conclusions:

– Multi-variant modeling should be supported in a universal and uniform way. *Universal* means that each item of a model (elements, attributes, references, etc.) should be allowed to vary. *Uniform* means that multi-variant modeling should be supported by one generic base mechanism which may be applied to any kind of model, independently of the underlying metamodel. In a layered architecture, we may push this

argument even further and strive to provide a base layer which is even independent of the metametamodel (and corresponding data model) [10].
- A multi-variant model is not necessarily just an ordinary model. Annotating ordinary models with visibilities is limited to exploiting *intrinsic variability*. For universal multi-variant modeling, *extrinsic variability* is required.
- In the case of extrinsic variability, the *union of single-variant models* results in a multi-variant model for which *single-variant constraints* might be *violated*. That is, a multi-variant model is not necessarily a valid instance of a single-variant metamodel.
- No matter whether intrinsic or extrinsic variability is applied: In general, it is hard to guarantee even the *syntactic consistency* of *configured single-variant models*. Some errors may be eliminated by appropriate management of visibilities, but this is unlikely to work in all cases. As a consequence, modeling tools should be more "forgiving", i.e., they should be capable of processing inconsistent models; otherwise, the user cannot conveniently check and fix the result produced by the configurator.
- *Simple and intuitive tools* are required for assisting the user in managing the complexities of multi-variant modeling. Unfiltered editing raises the risk of information overload. However, filtered editing exhibits some pitfalls, as well (e.g., adequate definition of the scope of a change). More research is required on adequate user interfaces.
- Even the best tools will not eliminate the complexity of multi-variant modeling. Thus, a *process* is required which ensures disciplined use of the concepts.

Acknowledgments The authors gratefully acknowledge the constructive comments of the unknown reviewers.

References

1. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Berlin, Germany (2005)
2. Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.P., Widen, T.: Consolidated product line variability modeling. In Käköla, T., Dueñas, J.C., eds.: Software Product Lines: Research Issues in Engineering and Management. Springer, Berlin (2006) 195–241
3. Stephan, M., Antkiewicz, M.: Ecore.fmp: A tool for editing and instantiating class models as feature models. Technical Report 2008-08, University of Waterloo, Waterloo, Canada (2008)
4. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: Companion Proceedings of the 30th International Conference on Software Engineering (ICSE'08), Leipzig, Germany, ACM Press (May 2008) 943–944
5. Krueger, C.W.: Leveraging the synergy of model-driven development and software product line engineering. Technical Report #200710311, BigLever Software, Austin, TX (October 2007)
6. Chang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania (November 1990)

7. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In Glück, R., Lowry, M.R., eds.: 4th International Conference on Generative Programming and Component Engineering (GPCE 2005). LNCS 3676, Tallinn, Estonia, Springer (October 2005) 422–437
8. Sarnak, N., Bernstein, R., Kruskal, V.: Creation and maintenance of multiple versions. In Winkler, J.F.H., ed.: Proceedings of the International Workshop on Software Version and Configuration Control, Grassau, Germany, Teubner Verlag (1988) 264–275
9. Goldstein, I.P., Bobrow, D.G.: A layered approach to software design. Technical Report CSL-80-5, XEROX PARC, Palo Alto, California (1980)
10. Westfechtel, B., Munch, B.P., Conradi, R.: A layered architecture for uniform version management. IEEE Transactions on Software Engineering **27**(12) (December 2001) 1111–1133
11. Munch, B.P.: Versioning in a software engineering database — the change oriented way. PhD thesis, NTNU Trondheim, Norway (1993) IDT-Report 1993:4, 265 p.
12. Collins-Sussman, B., Fitzpatrick, B.W., Pilato, C.M.: Version Control with Subversion. O'Reilly & Associates, Sebastopol, California (2004)
13. Vesperman, J.: Essential CVS. O'Reilly & Associates, Sebastopol, California (2006)
14. White, B.A.: Software Configuration Management Strategies and Rational ClearCase. Object Technology Series. Addison-Wesley, Reading, Massachusetts (2003)
15. Krueger, C.W.: The software product line lifecycle framework. Technical Report #200805071r1, BigLever Software, Austin, TX (December 2008)
16. Mahler, A.: Variants: Keeping things together and telling them apart. In Tichy, W.F., ed.: Configuration Management. Volume 2 of Trends in Software. John Wiley & Sons, New York (1994) 73–98
17. Tryggeseth, E., Gulla, B., Conradi, R.: Modelling systems with variability using the PROTEUS configuration language. In Estublier, J., ed.: Software Configuration Management: Selected Papers SCM-4 and SCM-5. LNCS 1005, Seattle, WA, Springer (April 1995) 216–240
18. Zeller, A., Snelting, G.: Unified versioning through feature logic. ACM Transactions on Software Engineering and Methodology **6**(4) (October 1997) 397–440
19. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: Proceedings of the Second Workshop on Aspect-Oriented Product Line Engineering (AOPLE 07)
20. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Formalizing cardinality-based feature models and their specialization. Software Process: Improvement and Practice **10**(1) (2005) 7–29
21. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L., eds.: Proceedings 5th International Conference on Generative Programming and Component Engineering (GPCE 2006), Portland, Oregon, ACM Press (October 2006) 211–220
22. Hendrickson, S.A., Jett, B., van der Hoek, A.: Layered class diagrams: Supporting the design process. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proceedings 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2006). LNCS 4199, Genova, Italy, Springer (October 2006) 722–736