Towards Supporting Feature Configuration by Interactive Visualisation

Goetz Botterweck¹, Daren Nestor¹, André Preußner², Ciarán Cawley¹, Steffen Thiel¹

¹Lero – the Irish Software Engineering Research Centre
University of Limerick
Limerick, Ireland
{ goetz.botterweck | daren.nestor | ciaran.cawley | steffen.thiel }@lero.ie

²Institute of Computer Science
Brandenburg University of Technology at Cottbus
Cottbus, Germany
apreussn@informatik.tu-cottbus.de

Abstract
Adopting a software product line approach allows companies to realise significant improvements in time-to-market, cost, productivity, and system quality. A fundamental problem in software product line engineering is the fact that a product line of industrial size can easily incorporate several thousand variation points. The scale and interdependencies can lead to variability management and product derivation tasks that are extremely complex to manage. This paper presents a metamodel that describes staged feature configuration and introduces a tool that illustrates the advantages of interactive visualisation in managing feature configuration, the first step in a product derivation process.

1. Introduction
In software product line engineering similarities between products are exploited to reduce the amount of work involved in producing a new software product. As a result of dealing with products with similarities software product line engineering has rapidly emerged as an important software development paradigm during the last few years. Developing products based on a product line approach allows companies to build a variety of systems with a minimum of technical diversity and to realise significant improvements in time-to-market, cost, productivity and quality [1].

Industrial scale product lines can easily incorporate thousands of variation points and configuration parameters for product customisation [2]. Managing this amount of variability can be extremely complex. In one case study a business unit identified feature parameter setting alone as accounting for 50 percent of product derivation costs [3]. There is a strong need for appropriate approaches that support different stakeholders in carrying out their development tasks in a software product line environment with a large number of variants [4].

Visualisation has proven useful in enhancing cognition in a number of ways. This is particularly the case in relation to externalising information, thus increasing the "memory" and "processing capacity" available to users; also by supporting the search for information and by encoding the information in a manipulable medium [5]. Visualisation takes abstract data and gives it a form suitable for visual presentation. Such data can be explicitly collected from software or can be codified by software engineers from their own implicit knowledge. With suitable techniques such visualisations can also amplify cognition about the large and complex data sets created and used in industrial software product line engineering.

This paper presents a metamodel describing feature configuration and a prototype tool based on this metamodel. The expressiveness afforded by the combination of the metamodel and visualisation techniques is illustrated by a product line example. The remainder of this paper is organised as follows: Section 2 describes the metamodel; Section 3 introduces a prototype tool to illustrate applications of visualisation techniques to a feature model of a product line for restraint systems. Section 4 discusses related work in visual feature configuration and Section 5 outlines future work. Finally, Section 6 concludes the paper.

2. Feature Modelling
Like conventional software engineering methods, which deal with single product development, Software
Product Line (SPL) approaches use all kinds of models to describe various aspects of the produced software.

However, since there is a whole spectrum of products which can be derived from a product line, SPL approaches have to attach a greater importance to the modeling of the varying features which can be supported by a product line.

During the derivation of one particular application from the product line, the next step after the elicitation of requirements is usually the identification of the right combination of product features which fulfills the given requirements. Such a feature model instance for one particular application is often called a feature configuration.

With the help of a feature model for the product line one can then find out if the desired combination of features can be provided by the product line. This can be determined by checking whether the desired feature configuration conforms to the feature model. Moreover, in addition to the analysis after configuration (“Yes, it does conform to the feature model.”) the information and constraints given by this domain feature model can guide us during the search for a valid, realizable feature configuration.

Before presenting how this configuration process can be performed with an interactive visual tool, we are now going to introduce the metamodel which we use to describe feature models and which serves as the foundation for the tool presented later on.

There are numerous feature modelling languages suggested in the literature, e.g. [6-8]. For our purposes we developed an extended and modified version of Czarnecki’s metamodel presented in [8] (see Figure 1). Adaptations to this metamodel were particularly required for the following reasons:

- In order to improve the support of the feature configuration process we needed more options to describe dependencies between features and relationships between features and architecture elements.
- We needed support for cloning of features within a feature group, which is not possible with the metamodel given in [8].
• The metamodel in [8] explicitly distinguishes between SolitaryFeatures, GroupFeatures and RootFeatures and separates between elements which can be contained by a Feature and a FeatureGroup. We skipped modeling these characteristics as they are not required for our purpose and would make the model more complex.

Beyond that, our metamodel has the following characteristics.

2.1. Staged Configuration

We have decided not to use a static two-level structure of Domain Feature Model and Application Feature Models, but to follow a staged configuration approach similar to [8]. So each model can be loaded, elements can be selected or eliminated, the cardinality constraints can be tightened and then the new model is saved as a more specific configuration of the old model. This can occur many times until no more room for interpretation is left and all variations have been decided upon.

This is reflected in the metamodel by relations between subclasses of ConfigurableElement such as Models or Features. Each ConfigurableElement can have multiple configurations and can in turn be a configuration of one ConfigurableElement. Each configuration has to describe constraints that are stricter than the one described in the related ConfigurableElement.

2.2. Basic Structure

As customary for feature models, the basic structure is given as a hierarchy of features and groups of features. This is reflected in the metamodel by containment relationships.

In addition, by using the abstract classes ContainableByFeature and ContainableByFeatureGroup we have set the following constraints:

A Feature can contain Features, FeatureReferences and FeatureGroups. A FeatureGroup cannot contain other FeatureGroups, only Features and FeatureReferences.

2.3. Cardinalities, Selection and Elimination

Features and FeatureReferences can have a minimum and maximum for the number of their occurrence (min and max), which can be written in the form of an interval [min, max].

We decided to model selection and elimination of elements by cardinalities – instead of introducing additional attributes such as isSelected:boolean. By this decision we can avoid unnecessary redundancy and potential inconsistencies. Consequently, optional features are modelled as [0, 1]. Mandatory features are modelled as [1, 1]. Features which have been eliminated are modelled as [0, 0].

FeatureGroups can have a minimum and maximum for the number of elements contained in them (groupMin and groupMax). A set of alternative features is modelled with a FeatureGroup with groupMin=1 and groupMax=1 and the min/max of each feature set to [0, 1].

2.4. Cloning

A feature is cloneable if its max attribute is greater than 1. In that case the attributes clonedFeature and clones of the class Feature are used to relate a cloned feature to its clones and vice-versa.

2.5. Attributes, Values and Types

Following the suggestion given in [8, 9] attributes of features are modelled as subfeatures. To support each feature can have a value Type. Although one can name and use arbitrary types, we currently support values which can be expressed as Boolean, Integer or String.

2.6. Source of Configuration

To support the configuration process it is helpful to remember what was the source for the configuration of a certain element – the machine or the human designer? This information is stored in the configuredBy attribute of each ConfigurableElement. It can be User, Machine or None.

2.7. Dependencies

Our feature metamodel supports UndirectedDependencies, relating a set of Features, and DirectedDependencies, relating a source and a target (see Figure 2).

There are two concrete forms of UndirectedDependencies: MutualExclusion and MutualProblematic.

MutualExclusion expresses that the features are completely incompatible (if one of them is selected, then all others in the set must not be selected). MutualProblematic indicates that the features can be combined, but this should be avoided (if one of the features is selected, then others in the set should not be selected).

There are two DirectedDependencies which describe similar relations from the positive side: Requires expresses that if the source feature is selected then the target feature must be selected, too. The weaker equivalent, Recommends, expresses that if the source feature is selected then the target feature should be selected as well.
There is a third directed dependency, \textit{Influences}, that can be used to describe that the source feature, if selected, has an effect on the target feature. At the moment we do not see a strong need for a generalization-specialization structure, but if necessary this could be modelled as a subclass of \textit{DirectedDependency}.

2.8. Modularisation

In all places where one could use a feature, that is a children of features or FeatureGroups, one can use FeatureReference instead. As these references can point to Features in different models (which are stored in separate files), this mechanism can be used to break down big, complex feature models into more manageable modules.

3. Feature Visualisation and Interactive Configuration

Visualisation has been described as an “adjustable mapping from data to visual form” [5]. The visual form is chosen to present the maximum of information required by the user without causing “map shock”, a phenomenon where someone perceiving an overly complex diagram has an audible reaction to information overload.

To illustrate the use of visualisation techniques in the context of representing and manipulating a feature model for the purpose of product derivation, we have developed VISIT-FC, a Visual and Interactive Tool for Feature Configuration. VISIT-FC is based on the metamodel described in Section 2 and attempts to reach Mackinlay’s [10] expressiveness criteria, which is that a set of facts are expressible if all the facts in the set, and only the facts in the set, are expressed. What this means is that all the information required is displayed, and that there is little or no room for misinterpretation based on unintended associations.

For the purposes of this illustration, we introduce an instantiation of the metamodel, a feature model of the Restrainment System Control Unit (RESCU) product line. The RESCU model includes features of a product line of automotive restraint systems. A simplified version of this model is represented in Figure 3.

3.1. Static Visual Components

In order to support a visual configuration process based on instances of the metamodel presented in Section 2 we can derive some important requirements (see also e.g. [4]).

A visualization tool has to present the overall containment hierarchy of features and feature groups in a clearly arranged way. Visually, the feature model under consideration is represented as a tree structure, which is a useful visual metaphor for representing hierarchical structures. Features are represented as nodes, and the direct relationships between features are shown as edges.

Colour coding of the features adds another level of information to this basic structure. The colours indicate the configuration status of the selected features and their sub-features, that is, a FeatureGroup is colour-encoded mandatory but not selected if its sub-features are not resolved. There are four levels of colour.
encoding, one for each for the feature states, which are selected, eliminated, optional and mandatory but not selected. These colour codes allow a quick overview of the feature model to show its current state, for instance to see if a valid product configuration exists.

We can assume that in frequent cases of cardinalities, such as optional \([0, 1]\), mandatory \([1, 1]\) and eliminated \([0, 0]\), it would simplify and improve the interaction if the tool does not bother the user with cardinalities, but presents graphical symbols (box, check mark, cross). This information is presented in boxes on the right hand side of each feature and allows the selection or elimination of a feature as shown in Figure 4. Simple icons are used to denote state, a cross for elimination and a tick for selection, but in this case colour and highlighting provide another layer of information. If the box is shaded, then the feature has been pre-configured or eliminated (depending on the icon) at an earlier stage of configuration and is no longer changeable. If the box is not shaded but the icon is not coloured, then the feature was selected or eliminated based on a dependency.

All this information is encoded at a low level of visual representation. This level is pre-attentively
processed by the human graphical information processing system [5, 11], meaning that once the colour encodings are familiar, the general information in even quite a large image can be interpreted very quickly.

3.2. Interactive Visual Components

On its own, the image presented is a static visualisation and, as such, is of limited usefulness. The addition of interactivity and automation based on the feature model adds to the power and flexibility of the visualisation [11].

Features with subfeatures are the Features and FeatureGroups mentioned in Section 2.2, and are collapsible. The tool allows for navigation using zooming and panning, with a configurable zoom rate. A text-based search of the feature tree is also allowed, and this feature supports regular expressions, allowing for a powerful and flexible search of the model. Features can be scaled based on user preferences, allowing a choice of emphasizing or marginalizing features for the purposes of clarity.

During the configuration process it has to be perceivable, if not obvious, to the developer which locations in the feature model are related to the feature he or she is currently working on. Hence, the tool has to visualize the relationships given by Dependency and its subclasses. However, showing all dependencies could lead to an incomprehensible blur of intersecting lines. To avoid this, the dependencies are not shown in the default view. However, a feature’s dependencies can be revealed in the standard view and in a contextual view that shows only the selected node and its dependencies. This makes the dependency information available if necessary or desirable, while preventing information overload.

These relationships can be used to filter and hide model elements which are not related to the current feature of interest. This supports the comprehension and handling of large complex feature models. The feature states are modifiable interactively, and dependencies are automatically resolved. This allows the selection or elimination of a feature to select or eliminate other features based on the dependencies of the modified feature. To ensure that this does not result in unwanted changes, any nodes that are automatically changing are brought to the user’s attention by a size increase, and if hidden, scaled down or under a collapsed tree for instance, are revealed. This allows the user to review all implications of the change when the change is made, without intruding with dependency information or forcing the user to work out what the dependencies are from a list of constraints.

4. Related Work

There are many feature modelling, configuration and variability management tools available (e.g., [12], [13], [14], [15]). Only a few tools have a visual component and thus are in scope in the context of this paper. These tools include FeaturePlugin, pure::variants, and COVAMOF.

FeaturePlugin [13] is an Eclipse plug-in for feature modelling. It uses EMF, the Eclipse Modelling Framework, as a generator for its model editors. It supports staged configuration, and a rich variety of modelling functionality. The visual capabilities are supplied by either a nested list structure, or a graphical tree structure based on the FODA style. However, the visual component does not provide effective focus+context displays and navigation of the list can be cumbersome. Dependencies and constraints are shown as an unsorted list that makes it difficult to understand dependency information.

pure::variants [14] is a software package developed by pure-systems GmbH. It provides similar functionality to FeaturePlugin, but lacks support for cardinality. It provides many views supporting various common product line configuration tasks, but the tree layout can be confusing, especially when using the built-in automatic layout, and in larger product lines could easily lead to information overload.

Tool support for COVAMOF, the ConIPF Variability Modelling Framework [15] is provided by a suite of tools that fulfil various requirements outlined by the modelling framework. These tools provide excellent functionality, but the visual components occur in disconnected windows that could hinder comprehension of the overall state. It is also unclear how well the disconnected windows would deal with a very large scale product line.
5. Future Work

The VISIT-FC prototype gives an insight into the benefits of a tool utilizing visual components within the framework of visualisation techniques to provide effective support for the tasks of feature configuration. Use of basic visual comprehension principles and redundant encoding of information using colour, grouping and iconography can improve comprehension and ease-of-use.

VISIT-FC does not yet support the full functionality of the metamodel. In particular, cloning of features is not yet supported, and neither is interactive linking between feature models using the FeatureReference entity. Further development will address the unimplemented functionality. It will particularly cover the linkage between the asset base and the feature model, as well as the linkage between the feature model and the actual implementation. This would provide end-to-end visual support for an interactive product derivation tool.

6. Conclusions

This paper has introduced the product derivation problems posed by large-scale variability and elaborated on the use of visualisation as a solution to some of those problems. Further, a metamodel that describes feature configuration has been presented, a tool that implements some of the functionality described using visual metaphors and interaction has been introduced.

The authors are convinced that visualisation can be of assistance in providing effective support for a variety of product line engineering tasks, including feature configuration as described in this paper. Rather than relying on a paper trail, or on tacit knowledge and experience of a small number of key stakeholders of a product line effort, a visualisation toolkit tailored to the particular needs of product line engineers could lower the complexity involved in managing the documentation, application and reuse of development artefacts.

7. Acknowledgements

This work is partially supported by Science Foundation Ireland (SFI) under grant number 03/CE2/I303-1.

8. References


