Applying Visualisation Techniques in Software Product Lines

Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciarán Cawley, Patrick Healy
Lero, the Irish Software Engineering Research Centre
University of Limerick, Ireland
{*daren.nestor, steffen.thiel, goetz.botterweck, ciaran.cawley, patrick.healy*}@lero.ie

Abstract

Software product lines of industrial size can easily incorporate thousands of variation points. This scale of variability can become extremely complex to manage resulting in a product development process that bears significant costs. One technique that can be applied beneficially in this context is visualisation. Visualisation is widely used in software engineering and has proven useful to amplify human cognition in data intensive applications. Adopting this technique in software product line engineering can help stakeholders in supporting essential work tasks and in enhancing their understanding of large and complex product lines.

The research presented in this paper describes an integrated meta-model and research tool that employs visualisation techniques to address significant software product line tasks such as variability management and product derivation. Examples of the tasks are described and the ways in which these tasks can be further supported by utilising visualisation techniques are explained.

CR Categories: D.2.1.3 [Software Engineering]: Reusable Software—Reuse models; D.2.2 [Software Engineering]: Design Tools and Techniques—Computer-aided software engineering (CASE); I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction Techniques;

Keywords: Software product lines, Visualisation, Interaction, Feature configuration

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.

SPL engineering promises benefits such as "order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers" [SEI 2008]. Many of these expected benefits rely on the assumption of additional upfront investment in domain engineering. This is necessary to create the product-line and its core assets, and is expected to pay off in the long run because product derivation based on a product line is (expected to be) more efficient than development from scratch.

SOFTVIS 2008, Herrsching am Ammersee, Germany, September 16–17, 2008.

However, to benefit from these productivity gains we have to ensure that application engineering processes are performed as efficiently as possible. One way of facilitating this is to support the activities by providing visual and interactive tools.

Visualisation is widely used in software engineering and has proven useful to amplify human cognition in data-intensive applications. Call graphs, for example, are used to represent the internal layout of programs and to assist the decomposition of legacy systems into reusable components [Gansner and North 2000]. Adopting visualisation techniques in software product line engineering can help stakeholders in supporting essential work tasks and in enhancing their understanding of large and complex product lines.

This paper introduces software product lines and elaborates on visualisation techniques that support fundamental product line engineering tasks. It advances a research tool that implements various visualisation and interaction techniques that can support stakeholders in the performance of important software product line engineering tasks.

The remainder of the paper is organised as follows. Section 2 briefly gives some background on Software Product Lines, explains difficulties with two of the most error prone processes within software product line engineering and provides an overview of the configuration process. Section 3 introduces the integrated meta-model as a foundation for visual representation of software product lines. In Section 4 we introduce our visual research tool (VISIT-FC) and explain the visualisation techniques used within the tool in more detail. Section 5 explains how VISIT-FC can be used to support specific product line engineering tasks. Section 6 discusses related work, and section 7 outlines future directions for this work. Finally, section 8 concludes the paper.

2 Software Product Lines

A software product line is a set of software intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. This involves strategic, planned reuse that yields predictable results [Clements and Northrop 2002].

The organisation implementing a software product line plans the scope of the product family and designs the products to take advantage of the commonality between the various products. During the design of the product line, the specific differences between products is also planned, and "variation points" are built into the product line artefacts. These are locations where variation between members of the product line will occur. This phase is called *Domain Engineering*. *Application Engineering* refers to the processes involved in creating products from the existing product line. In short, domain engineering is development *for* reuse, application engineering is development *with* reuse.

Two areas within the software product line process that can cause particular difficulties for practitioners are the management of variability, and the process of product derivation.

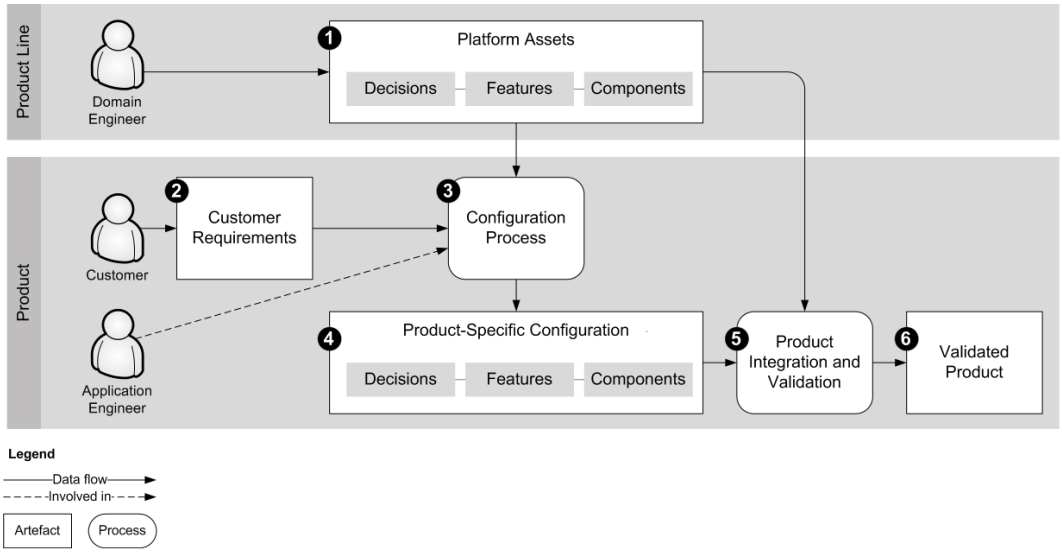


Figure 1: Overview of the product configuration process

2.1 Variability Modelling and Representation

Variability refers to the ability of a software product line development artefact to be configured, customized, extended, or changed for use in a specific context [van Gurp et al. 2001]. It thus provides the required flexibility for product differentiation and diversification within the product line.

Numerous models and approaches have been proposed for representing variability information in various development phases of a software product line approach, especially in requirements engineering [Halmans and Pohl 2004] and architecture design [Thiel and Hein 2002]. Possible relationships between features are usually categorised based on the constraints imposed by the model (for instance, one feature may exclude or require another). A common method of representing product line variability is a feature model [Kang et al. 2002]. A feature diagram is typically represented as a tree where primitive features are leaves and compound features are interior nodes.

2.2 Product Derivation

Product derivation is the process of creating a specific product from the product line. The process of product derivation can be divided into two phases: the initial phase and the iterative phase [Deelstra et al. 2005]. The initial phase involves using requirements to drive the primary derivation of a product using either assembly from existing components or the selection of the closest existing configuration and adapting it to the requirements of the product being derived. In some cases, an initial configuration sufficiently implements the desired product. In most cases, however, one or more cycles through the iterative phase are required for various reasons, such as requirements changing, the assets having been modified, or to ensure that communication between the components is working as expected.

The iterative phase involves adaptation of the selected assets. The adaptations can be product specific or evolutionary, that is, they can be specific to this instance of the asset, or they can be shared with all the members of the product line.

The iterative phase is a particular source of error. Major factors identified empirically by Deelstra et al. [Deelstra et al. 2005] include amongst others:

- the large number of errors in parameter settings due to large amount of parameters with implicit dependencies,
- false-positives during the compatibility test during component selection,
- the unmanageable number of variation points and variants,
- the lack of hierarchy in the organization of variation points,
- the unpredictable consequences of variant selection.

For instance, in the case study cited above, a business unit identified parameter settings during the configuration of the assets being used in a specific product as accounting for approximately 50 percent of product derivation costs.

Expanding on this, Hotz et al. [Hotz et al. 2006] identified two issues that were at the core of all the problems mentioned. These core issues were:

- The complexity of the product line in terms of variation points, variants and dependencies;
- The large number of implicit properties or dependencies associated with variation points and variants. These tend to be undocumented or only known to experts.

The issues thus identified imply that the product line asset base can suffer from combinatorial complexity - that is, the interactions between assets become extremely complex. This complexity in the area of variability management is a cause of issues in the product derivation process, and is one of the areas where the authors feel that visualisation can be of particular use.

2.3 Process Overview

The benefits of software products lines, e.g., a reduction of costs per product, can only be achieved if we can derive each product from the product line as efficiently as possible. However, as indicated in the preceding section, this process of product derivation is complex and error prone.

With the research presented in this paper we strive to address this challenge by applying visualisation techniques to help the software engineers to handle the complexity inherent in the product line models and, consequently, perform the product derivation process more efficiently.

As a first step towards a solution we will now provide an overview of the configuration process, which is a sub task within product derivation (see Figure 1).

When establishing the product line, the domain engineer creates the product line assets (see ❶ in Figure 1). In our scenario this includes *Features* (which describe the capabilities of the product line from a stakeholder's perspective), *Decisions* (which provide an high-level abstracted view on features and are essentially a combination of features that satisfy a particular need), and *Components* which implement features.

After the product line has been established, products can be derived. This usually starts with some customer initiating a project and providing his *Customer Requirements* ❷.

These requirements are then used in the Configuration Process ❸ to identify matching platform assets and configure these assets. This results in a *Product-Specific Configuration* ❹. Reflecting the various types of platform assets this contains configuration information on the decisions, features, and components.

This configuration process is challenging because it involves a large number of involved elements and dependencies between them. Consequently, it is hard for the application engineer to predict and understand all consequences of his decisions.

Subsequently, the configuration is used as input for the Product Integration and Validation ❺ which will eventually create the *Validated Product* ❻.

As indicated in the product line assets ❶, in our approach we describe the elements which are relevant to this configuration process in terms of *Decisions*, *Features*, *Components*, and the dependencies between them. We will now introduce a meta-model which captures these concepts more formally and provides a foundation for the visualisation techniques which we will present later on.

3 An Integrated SPL Meta-Model

To exploit the benefits of a product line we need to connect the isolated models, for instance, to describe the relationships between features and software components. This inter-connected meta-model forms the basis for the visual representation that we present in our tool. Viewing a visual image as a "sentence in a graphical language" [Mackinlay 1986], the model provides the "syntax" and "semantics" we need to convey the information required.

The meta-model covers three models: First it allows us to describe features with all their usual dependencies and attributes. We extended this towards later phases by adding modelling concepts to describe *components* that implement features. We also added *decisions* which provide a simplified, high-level view onto features and can be used to abstract from details by asking a few major questions which are relevant for a particular stakeholder. This extends the feature model towards earlier phases of the process. A simplified graphical representation of the model elements and the relationships between them is shown in figure 2.

Our meta model supports relationships within a model (between elements in the same model) and relationships between models (between elements in different models). The individual sub-models have been described in detail in earlier work [Botterweck et al. 2007a].

Requires is a directed dependency between elements of the same type, and may have multiple sources and targets. For example, *requires_FF* expresses dependencies between features, and can describe dependencies such as "(Fx *and* Fy) *requires* Fz" or "Fx *requires* (Fy *and* Fz)".

Excludes is an undirected dependency as it describes a *set* of elements that are mutually exclusive (this is not shown in figure 2).

The external relationships are used to describe relationships between the models. Again, we distinguish subtypes for the different elements. *ImplementedBy_DF* expresses that a decision activates certain features, and *ImplementedBy_FC* describes components that implement a feature. This end-to-end traceability is designed to allow configuration decisions in one model to be reflected by changes in another model and to provide the syntax required to represent this graphically

The internal dependencies are similar to existing models. The dependencies *between* models are designed to provide traceability throughout the entire product configuration process. The disconnect that currently exists between the various stages of the configuration process is a barrier to comprehension, understanding of the specific consequences of configuration decisions and overall process efficiency.

4 VISIT-FC and Visualisation Techniques Employed

With the complex interrelated data sets that come with a software product line, comprehension is often difficult as discussed in section 2.2.

In the previous sections, we have discussed the some of the processes involved in software product line engineering, and advanced our meta-model. In this section, we will introduce VISIT-FC, a Visual and Interactive Tool for Feature Configuration. This tool was developed in order to assist product line stakeholders in their comprehension of the product line data, and in understanding the consequences of their configuration decisions. VISIT-FC is based on the integrated meta-model in section 3 and employs visualisation and interaction techniques in an attempt to fulfill MacKinlay's [Mackinlay 1986] expressiveness criteria. VISIT-FC adds interactive functionality for product line engineers, allowing clear exploration and manipulation of the existing product line data. It provides a compact, interactive representation of large decision, feature and component hierarchies, allows configuration with automatic constraint propagation, and provides hints for configuration problems and open decisions.

It is perhaps important to note that during the design of VISIT-FC the prior existence of a product line was assumed. In the discussion that follows, it is anticipated that an asset base exists, and that the variation points, relationships between them and dependencies are already explicitly in place.

We have used VISIT-FC to support product development activities of an sample Restraint System Control Unit (RESCU) product line, based on industrial experience. This was used for preliminary evaluation of our approach in laboratory conditions. RESCU comprises features such as airbag deployment, seatbelt tensioners, active headrests, and weight sensing that support the protection of vehicle occupants in case of an impact and various other driving situations.

4.1 Explicit Representation

Explicit Representation refers to drawing methods which display the hierarchy as links between nodes. Implicit drawing methods

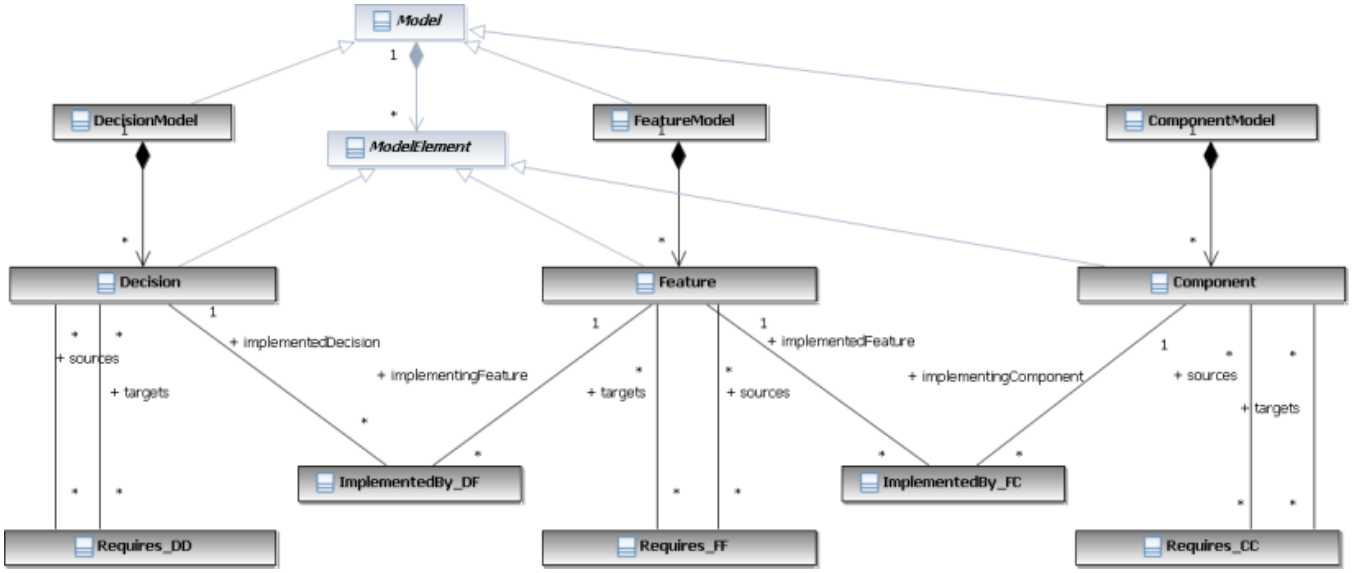


Figure 2: Integrated Software Product Line meta-model (simplified extract)

represent the hierarchy by a special arrangement of nodes, e.g. containment or overlapping. Examples of implicit graph drawing are tree-maps [Johnson and Shneiderman 1991], or the information cube [Sinnema et al. 2004a]. VISIT-FC uses Explicit Representation. Figure 3(a) shows a screenshot of main RESCU product line features in VISIT-FC.

A simple non-radial tree layout was adopted for VISIT-FC. The layout chosen provides the opportunity to encode a significant amount of information on screen utilising the restricted space in an efficient manner. In VISIT-FC the nodes represent features and the edges represent the relationships between those features. Straight edges indicate parent-child relationship and curved edges represent dependency relationships. Figure 3(c) shows a portion of the RESCU feature model.

4.2 Colour Coding

Colour coding of the features adds another layer of information to this basic node link tree structure. The colours indicate the configuration status of the selected features and their sub-features. A FeatureGroup, which is a containter that allows grouping of features, is colour-encoded mandatory but not configured if its sub-features are not resolved.

There are four levels of colour encoding, one for each of the feature states, which are selected (green), eliminated (grey), optional (amber) and mandatory but not configured (red). These colour codes allow a quick overview of the feature model and its state, for instance to see if a valid product configuration exists. Further information is encoded by use of graphical symbols (tick or cross). A tick indicates selection, a cross indicates elimination. Another layer of information is encoded through the use colour coding with the symbols. If the box is shaded, then the feature has been preconfigured and is no longer changeable. If the box is not shaded but the icon is not coloured, then the feature was configured based on a dependency.

Information encoded at this low level of visual representation is processed pre-attentively [Ware 2000] by the human graphical system. Once the colour encoding becomes familiar, a stakeholder should be able to interpret large representations rapidly, while also freeing cognitive resources for other tasks. Even before the specific colour encoding becomes familiar, use of colours with a high degree of contrast can be used to highlight important information, and use of wide-spread colour conventions (such as the use of red to mark a node for attention) can help with basic comprehension.

4.3 Details on Demand

Details on Demand refer to the facility whereby the stakeholder can choose to display additional detailed information at a point where this data would be useful. This point is decided by the user of the system. Information such as cardinalities can be displayed through the use of a “mouse-over” and feature names can be displayed or removed through viewing configuration options.

VISIT-FC also provides the facility to choose a specific feature and show all sub features and dependent features while hiding all other features that are neither sub features nor dependent in any way on the chosen feature. This allows the stakeholder to focus on the relevant data for a particular feature while temporarily hiding data that is not relevant to the task at hand.

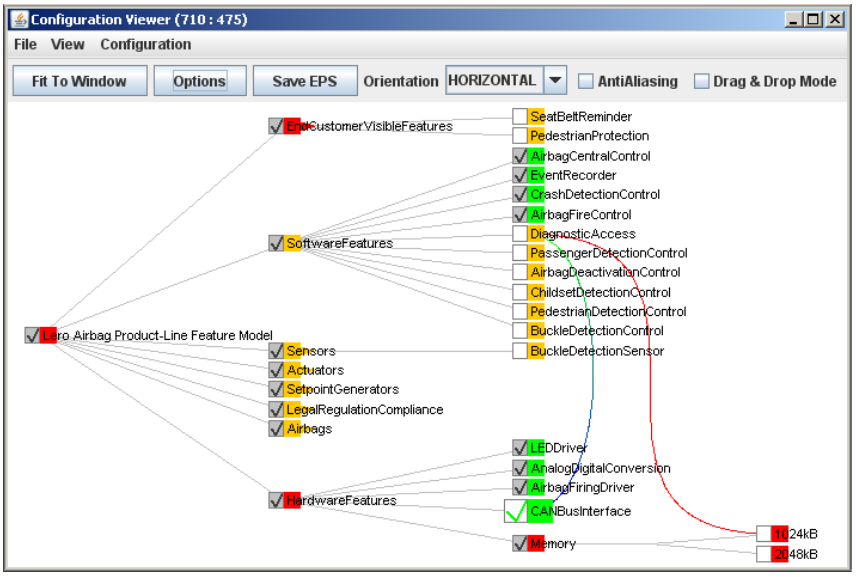
4.4 Incremental Browsing

Incremental browsing is a form of information filtering, where only limited sections of the visualised structure are displayed. The rest is hidden and can be visualised when needed.

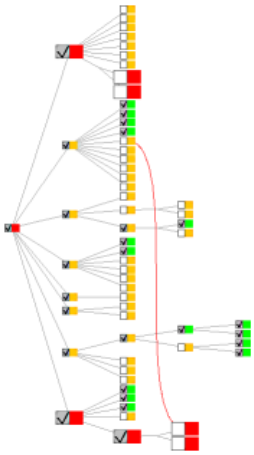
In VISIT-FC the feature model visualisation starts with displaying only the high-level features, and the stakeholder can then explore the feature hierarchy by unfolding the sub-features of features in which the stakeholder is interested in. Further, a node with children is indicated by a small extension to the node. The stakeholder is thus able to perceive the feature structure step by step, and is not overwhelmed by the complete model.

4.5 Focus+Context

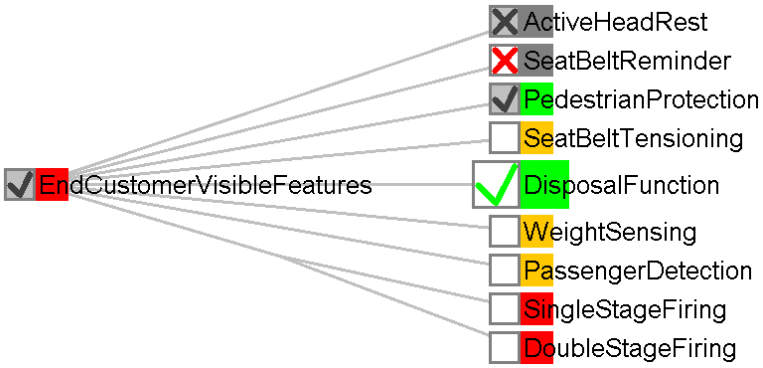
Focus+Context is the name given to techniques that provide for the display of contextual information while preserving the ability of a user to maintain a sense of orientation within the greater display



(a) VISIT-FC Feature Configuration Viewer showing Configuration Example



(b) Overall view of the RESCU Product Line



(c) Detail of Features in RESCU Product Line

Figure 3: VISIT-FC Configuration Viewer showing features of the RESCU product line

space. The advantage of Focus+Context is that the stakeholder does not get disorientated in relation to the overall structure when zooming into a large structure or exploring the details of certain features. They are able to see where they came from, and are not required to keep this in memory. This can be useful, for instance, the visualisation of search results or to see dependent feature nodes in distant parts of a large model.

Pan and Zoom capabilities allow navigation on the virtual canvas, and adjustment of the scale of the view of various sections of the graph. The problem is that on their own these techniques do not provide the context, just the focus. This can be reasonable for small display spaces but can lead to usability problems in larger spaces. Solutions for this include multiple windows, or viewports displaying the location in the overall graph. More effective are distortion techniques. These can either be built directly into the layout, or can provide a distorted view on the layout. Another simple but effective method is to use the Degree of Interest calculation [Furnas 1986] used in the generalized fisheye view. This allows the assignation of an *a-priori* importance to a node. This importance and the distance from the focus are then used to calculate the relative node sizes so more important nodes are highlighted. This is obvious in figure 5, where the larger coloured nodes are the features with higher risk associated with them.

VISIT-FC provides basic versions of these facilities and also allows selective zooming of a specific chosen portion of the feature tree focusing on the area of interest and allowing the non-relevant area to remain in view but to a lesser degree. Figure 3(b) shows a simplified version to illustrate the split zooming facility. It shows certain user selected features that have been "zoomed out" because they are of lesser interest while keeping them in view which maintains the overall context. Different sets of feature nodes can be manually "zoomed in" or "zoomed out" to varying degrees to allow an optimum view for the task at hand.

5 Product Line Tasks Illustrating Visual Tool Support

In this section we discuss four tasks performed by product line engineers during product development and how to support them by employing visualisations:

- Configure a (subsystem of a) product variant
- Understand the consequences of design decisions
- Representing feature attributes

The tasks have been obtained from a case study with one of our industrial partners. We illustrate them based on examples of the RESCU product line modelled in VISIT-FC.

5.1 Product Configuration

This subsection describes an example that a stakeholder would undertake to configure the diagnosis interface of the RESCU product line.

In this scenario, the stakeholder is interested in configuring "Diagnostic Access" (see the corresponding green feature in Figure 3(a)). By clicking on the Diagnostic Access node, the stakeholder can select this feature for the product being derived. Because of existing dependencies the application then automatically configures two other features in the product line by selecting the feature "CAN Bus Interface" (a sub-feature of "Hardware Features") and eliminating the "1024KB Memory" variant. These dependent features are highlighted through increased node size notifying the stakeholder of the automatic actions. If a dependent node is not currently displayed at the point of automatic selection/elimination, then it is made visible at that time. The stakeholder can explicitly display the dependencies using curved colour coded links.

By use of split zooming and panning, the stakeholder modifies the view for even further clarity. If desired the stakeholder can display all dependent features providing a useful view of connected parts of the product being derived. Moreover, he or she can switch the view to the dependency context mode temporarily re-moving all data from the screen except that which is directly connected to the feature being configured.

5.2 Understanding Consequences of Decisions

One typical challenge when dealing with groups of larger SPL models is the understanding of a cluster of model elements, which are connected via various dependencies. Such a situation occurs for instance when the SPL engineer is preparing a configuration decision and wants to understand the consequences of that decision. We will illustrate this by using a representation of the instantiated decision-feature-component model describing the RESCU product line for automotive restraint systems (Figure 4).

The SPL engineer starts by selecting the feature "Passenger detection" and the two choices below it, called "Bladder Mat" and "Weight Sensing" (see ❶ in Figure 4). He then activates the traceability function to all related model elements. To enable this functionality we have to define and calculate what is "related". We can do this based on concepts defined in the meta-model.

For instance, we can follow all directed dependencies (requires, implemented-by) to calculate their transitive closure. We can do this in backward direction to identify model elements that require or are implemented by the selected elements (❷). We can also do this in forward direction to identify model elements that are required by or implement the selected element (❸). For all model elements identified this way, we also determine excluded elements (see ❹).

By these means, the SPL engineer can quickly identify all model elements that are related to the selected "Passenger detection" feature. For instance he could see, that for "Weight Sensing" passenger detection there is a problem with the low memory configuration (❹) and that it requires a CANBus interface (❺) to allow for the integration of the advanced sensor.

There are several other scenarios that can be supported by this functionality. For instance, imagine a SPL platform developer wants to replace a component. Using the illustrated traceability function can help him to identify features and decisions which potentially are affected by this modification.

5.3 Representing Feature Attributes

It is important that the maximum amount of information is extractable from each feature. Additional information can be stored with the feature, but this information is of little use if it cannot be made available easily and in a comprehensible manner. One way in which the importance of a feature attribute can be made comprehensible is to make it relative to the same attribute in other features. Thus, it becomes a visual comparison task.

In the following scenarios, we will look at two of these attributes, "cost driver" and "risk level".

5.3.1 Representing Cost Drivers

Cost driving features are those with a high relative contribution to the products' total development costs and thus are critical to the gross profit margin of the product line. One typical reason for this is that the feature's implementation relies on specific software or hardware components which are expensive to acquire, develop or integrate. It is important to know the cost-driving features in a product line, for instance during product configuration, as they usually significantly influence price negotiations with the customer.

Cost driving features can be highlighted as enlarged nodes: *WindowCurtainAirbags*, *WeightSensing*, *ChildSeatDetection*, and *PedestrianProtection*. In this example PedestrianProtection is shown as the most expensive feature (largest node). This feature provides functionality to protect a pedestrian during a collision with a road vehicle in a low speed city traffic situation. Acceleration sensors in the vehicle's bumpers enable to detect a collision with a pedestrian and trigger an actuation system which raises the vehicle's engine hood. In this way the risk of serious head injuries is reduced. Figure 5 shows how cost drivers could be identified.

For example, *PedestrianProtection* is a cost driver because its implementation relies on expensive hardware and hydraulic components that must be integrated in the final product variant. Thus the relative cost contribution of this feature to the product is disproportionately high. The same principles apply for the three other cost-driving features.

5.3.2 Representing High-Risk Features

Another important type of features that must be made explicit is high-risk features. These features represent critical capabilities of the product line that are essential to provide to customers but that are also difficult to achieve. High-risk feature are usually directly related to the business success of the product line or a particular subset of product line products.

Figure 5 shows the four high-risk features: *WindowCurtainAirbags*, *WeightSensing*, *ChildSeatDetection*, and *PedestrianProtection*.

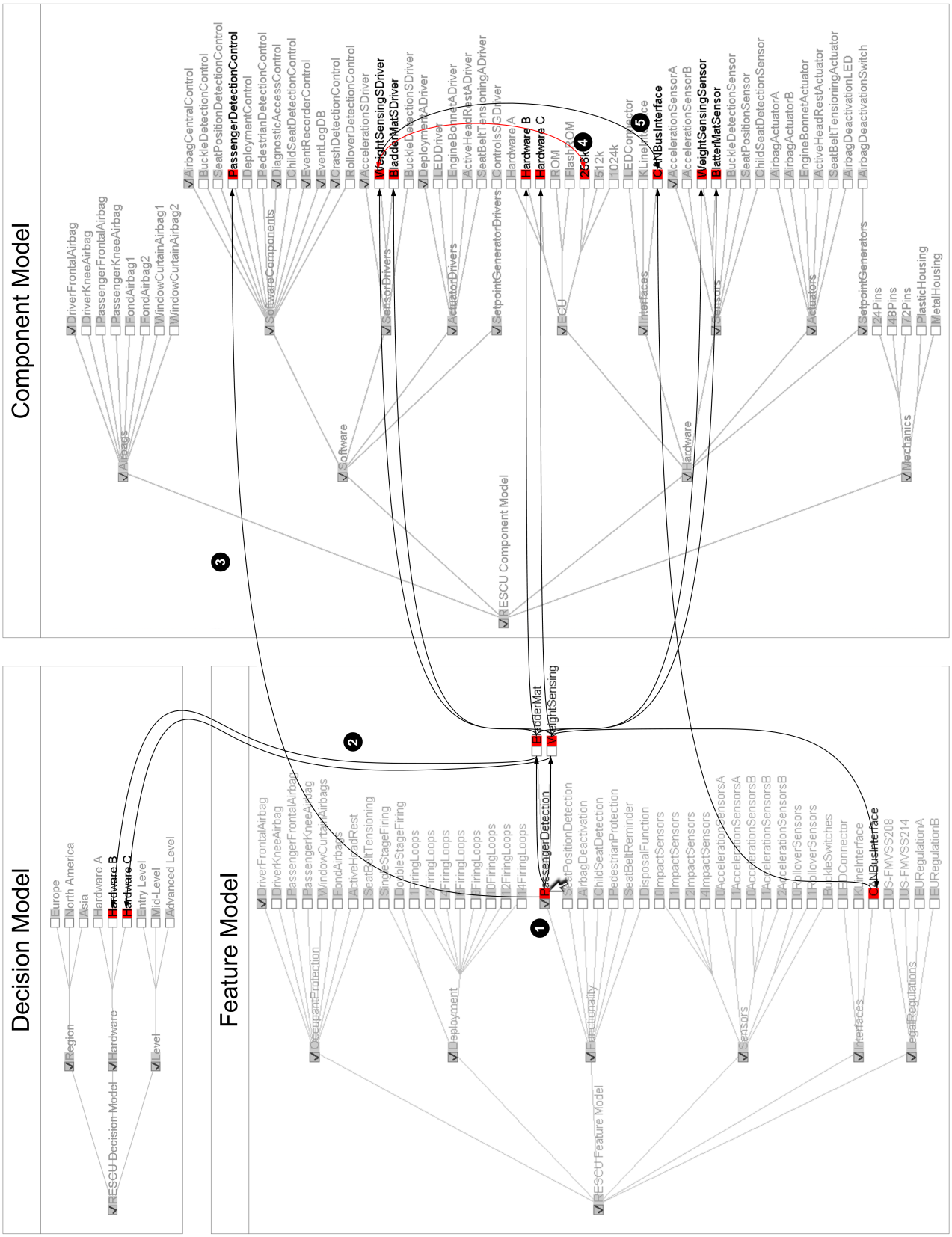


Figure 4: Linking three model hierarchies (decisions, features and components)

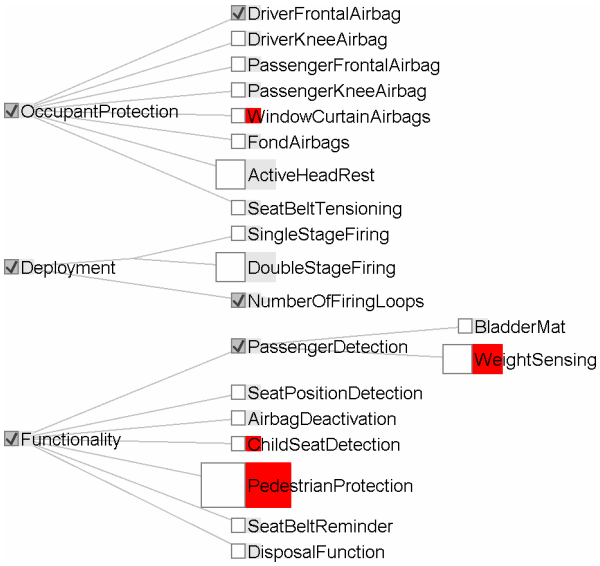


Figure 5: The risk associated with each Feature (stored as an attribute) highlighted

They are marked in red. We can distinguish different root causes for risks that might influence the business success of a product line.

A higher risk might be involved in innovative features. Innovative features usually make use of the newest available technology. This technology is often less mature as long-term tests are usually lacking and field experience is limited. In our example *WindowCurtainAirbags* for occupant protection represents such a feature. The failure probability of a product variant which includes this feature might be higher than one without.

Routine features are features that are absolutely essential for a set of products in the product line. If the completion of particular routine features is late then these products cannot be introduced into the market as planned. This might be of high risk for an organization since the revenues that were planned for these products cannot be realized in time.

If the product line relies on particular components that are developed by a 3rd party vendor then the associated features might be high-risk features. In our example *WeightSensing* is such a feature. This feature provides passenger weight information to the restraint system which is then used to optimise several deployment strategies (e.g., seatbelt tensioning). In RESCU *WeightSensing* requires smart weight sensors which are developed by a 3rd party supplier. These sensors must be integrated into the system and the possible product variants that include the *WeightSensing* feature must be tested carefully. If some of the tests fail then this might result in longer redevelopment and test cycles which can compromise the timely market introduction of the product line.

Systemic features are features that rely on a combination of software, hardware, and other components such mechanic and hydraulic components. They are usually cost drivers as well. Systemic features often require hardware-software co-development and involve complex integration tasks. In our example *PedestrianProtection* is such a feature.

6 Related Work

6.1 Layout Techniques

Advanced layouts exist for explicit tree-drawings such as cone-trees or space-trees [Ware 2000]. Cone-trees are a 3D layout that viewed from directly above resemble a radial tree. 3D visualisation introduces problems such as occlusion of information and difficulty of navigation and comprehension [Cockburn and McKenzie 2000] so for the purposes of VISIT-FC a 2D visualisation was chosen.

Space-trees [Plaisant et al. 2002] are a two dimensional layout that attempts to overcome the problem of information overload (where the sheer amount of information displayed causes comprehension to be hindered by the graphical "noise") by collapsing the nodes that are not currently in use and providing a graphical notation to indicate that there is a sub-tree of the node. VISIT-FC makes some use of this concept - the sub-trees can be collapsed and there is a small indicator. However, VISIT-FC extends this concept to include more than simple node-leaf relationships, and uses other visualisation techniques throughout to highlight information.

6.2 Product Derivation and Integrated Models

Examples of product derivation processes adopted by software product line organisations are discussed in Deelstra et al. [Deelstra et al. 2005] and the ConIPF project [Hotz et al. 2006]. More and more approaches aim to integrate software product line models to support product line development activities, for instance linking feature models to software architectures [Siegmund et al. 2008][Liu and Mei 2003][Sochos et al. 2004].

Czarnecki and Antkiewicz [Czarnecki and Antkiewicz 2005] use OCL-based model templates to describe consequences of feature configurations in other models. Similar approaches use declarative model-transformation languages to describe how a feature decision is reflected in the architecture [Botterweck et al. 2007b]. Lisboa et al. [Lisboa et al. 2007] present a tool set for domain analysis and generate a product model by specifying a scope, generating a domain model for a scope and selecting configurable features from a domain feature model. Other approaches describe the relationships between SPL models with formal techniques. In addition to precise traceability, this allows formal checking properties such as consistency among the integrated models [Janota and Botterweck 2008][Satyananda et al. 2007].

6.3 Product Line Configuration Tools

There are various modelling tools that employ a visual component to aid product configuration and variability management.

FeaturePlugin [Antkiewicz and Czarnecki 2004] is an Eclipse plug-in that supports feature modelling. It uses editors generated with the Eclipse Modelling Framework (EMF). Nested lists and a tree layout are employed as techniques to support product configuration. One drawback is that the lists can be difficult to navigate as the focus+context display implementation is not very effective. It is also difficult to comprehend the dependencies as constraints are shown as unsorted lists.

pure::variants [Beuche 2004] is a commercial feature modeling software. It supports various views which provide different approaches for different stakeholder tasks but does not support cardinality. It supports a graph-based view with an automatic layout function. However, the graph view is a static layout, and the visualisation of dependencies serves to obscure information as they are shown as straight line relationships, irrespective of node placement.

The ConIPF Variability Modelling Framework (COVAMOF) is augmented by tool support [Sinnema et al. 2004b]. Even though significant functionality exists, understanding of the overall state of the configuration can be difficult due to the separate and disconnected window views.

Kumbang [Asikainen et al. 2007] provides tool support for integrated feature and component modelling. However, similar to other tools mentioned before, the views for the configuration of features and components are isolated. There exists no direct visual, interactive representation of the relationships between these models.

6.4 Software Product Line Visualisation

Software product lines are often represented as a hierarchical structure. The two most widely used structures are decision trees or feature trees. While conceptually quite different, from a graphical perspective they are both simply tree-like structures.

Decision models tend to be represented as a list of questions, each question being an open decision. Interactive features allow for filtering, such as seen in the DOPLER tool suite [Rabiser et al. 2007]. Force directed layouts have also been used to represent the decision tree, for instance in the V-Visualize tool [Sellier and Mannion 2007]. The question lists can be filtered on access permissions, as well as having decisions that require higher-level decisions to be made filtered out. Particularly in the DOPLER tool suite, tests indicated that the users preferred the simple "list of questions" over the graphical alternatives presented - however this could be directly related to the style of the visualisation used. This type of model allows for the abstraction of technical information as the questions can be very general. In the case of the integrated model mentioned in section 3, it allows a high level configuration which can be passed on to the feature model for more specific configuration.

Feature models are the prevalent method for modeling a product line. Polyarchies [Robertson et al. 2002] are one method of representing a feature tree, as used in pure::variants [Beuche 2004]. This common graphical notation is most frequently seen used to represent directory structures, such as the structure displayed by Windows Explorer. The simplest method of displaying the feature tree, however, is simply to draw it as a tree. There are various notations used, but the authors have chosen a cardinality-based approach as being the most flexible.

The notable difference between the tools mentioned above and VISIT-FC is that VISIT-FC integrates the various stages of the process. It provides clear visual cues to draw attention to the consequences of decisions across multiple stages of the process, not just within one stage of the process.

7 Future Work

The development of the VISIT-FC research tool is based on the utilisation of well understood but non-complex visualisation and interaction techniques. It has shown an avenue down which the challenges faced by stakeholders during software product line engineering can be addressed. Even simple information encoding through colour schemes suggests an increase in the speed at which product configurations can be interpreted. More research into visualisation techniques and their applicability to and usability for tasks such as variability management, product configuration, and evolution support is planned.

Further functionality provided by the meta-model and extensions to it will also be implemented. This will allow for an improved end-to-end visual support for interactive product derivation. The possibility of providing this prototype tool as an Eclipse plug-in to allow tighter integration with the underlying EMF models will also be explored.

Further application of visualisation techniques to the existing layouts and capabilities is also desirable. In particular, the use of layout techniques that provide distortion, such as hyperbolic trees, or using a distorted view of the linear layout with a fisheye view are interesting directions in which progress could be made.

Even though we are interested in sophisticated graph-based visualisations, we should take into account that sometimes visually simpler techniques have to be considered as well. For instance, discussions during research workshops with industry partners indicate that in industrial practice tabular overviews are prevalent. We hope to discover whether the affinity to tabular overviews expressed by industry practitioners can be substantiated by evidence or is caused by lack of better visual tools.

The next step is experimental evaluation of the tool, in conjunction with our industrial partners. A limited participant empirical case study is planned to give an initial indication of the effectiveness of the tool. It is intended that this will be followed by an action research study within product configuration groups within an industrial partner.

8 Conclusions

In this paper we have described an integrated meta-model and research tool that employs visualisation techniques to address significant software product line tasks such as variability management and product derivation. Examples of these tasks were described and the methods by which stakeholders can be further supported in the performance of these tasks by utilising visualisation techniques were explained.

In the authors' opinion, further research into the applicability of various visualisation and interaction techniques could help to address the challenges faced by stakeholders and significantly increase the efficiency of common product line engineering tasks.

Furthermore, a configurable visualisation toolkit could reduce the dependence on a small number of experts and allow software product line engineers perform their tasks with much greater autonomy.

Acknowledgments

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

References

ANTKIEWICZ, M., AND CZARNECKI, K. 2004. FeaturePlugin : Feature modeling plug-in for eclipse. In *Proc. OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, ACM, Vancouver, British Columbia, Canada., 1–6.

ASIKAINEN, T., MANNISTO, T., AND SOININEN, T. 2007. Kumbang: A domain ontology for modelling variability in software product families. *Adv. Eng. Inform. 21*, 1, 23–40.

BEUCHE, D. 2004. Variants and variability management with pure::variants. In *Proc. 3rd Software Product Line Conference (SPLC 2004), Workshop on Software Variability Management for Product Derivation*.

BOTTERWECK, G., NESTOR, D., PREUSSNER, A., CAWLEY, C., AND THIEL, S. 2007. Towards supporting feature configuration

by interactive visualisation. In *Proc. 1st Int'l Workshop on Visualisation in Software Product Line Engineering (ViSPLE2007)*.

BOTTERWECK, G., O'BRIEN, L., AND THIEL, S. 2007. Model-driven derivation of product architectures. In *Proc. 22nd IEEE/ACM Int'l Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, ACM, R. E. K. Stirewalt, A. Egyed, and B. Fischer, Eds., 469–472.

CARD, S. K., NATION, D., AND UI, D. V. 2002. Degree-of-interest trees:. In *Proc. Advanced Visual Interfaces 2002*, 231–245.

CLEMENTS, P., AND NORTHROP, L. 2002. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA.

COCKBURN, A., AND MCKENZIE, B. 2000. An evaluation of cone trees. In *Proc. of the HCI'00 Conference on People and Computers XIV*, Usability and System Evaluation, 425–436.

CZARNECKI, K., AND ANTKIEWICZ, M. 2005. Mapping features to models: A template approach based on superimposed variants. In *Proc GPCE 2005*, Springer, R. Glück and M. R. Lowry, Eds., vol. 3676 of *Lecture Notes in Computer Science*, 422–437.

CZARNECKI, K., HELSEN, S., AND EISENECKER, U. W. 2004. Staged configuration using feature models. In *Proc. of the Third Software Product Line Conference (SPLC 2004)*, 266–283.

DEELSTRA, S., SINNEMA, M., AND BOSCH, J. 2005. Product derivation in software product families: a case study. *The Journal of Systems and Software 74*, 2 (Jan.), 173–194.

FURNAS, G. W. 1986. Generalized fisheye views. In *Proc. of the ACM Conference on Human Factors in Computer Systems*, M. M. Mantei and P. Orbeton, Eds., The SIGCHI Bulletin. Association for Computer Machinery, New York, U.S.A., 16–23.

GANSNER, E. R., AND NORTH, S. C. 2000. An open graph visualization system and its applications to software engineering. *Software — Practice and Experience 30*, 11, 1203–1234.

HALMANS, G., AND POHL, K. 2004. Communicating the variability of a software-product family to customers. *Inform, Forsch. Entwickl 18*, 3-4, 113–131.

HOTZ, L., WOLTER, K., KREBS, T., NIJHUIS, J., DEELSTRA, S., SINNEMA, M., AND MACGREGOR, J. 2006. *Configuration in Industrial Product Families - The ConIPF Methodology*. IOS Press.

JANOTA, M., AND BOTTERWECK, G. 2008. Formal approach to integrating feature and architecture models. In *Proc. Fundamental Approaches to Software Engineering (FASE 2008)*.

JOHNSON, B., AND SHNEIDERMAN, B. 1991. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proc. IEEE Conf. Visualization*, IEEE Computer Society, A. E. Kaufman and G. M. Neilson, Eds., 284–291.

KANG, K., COHEN, S., HESS, J., NOVAK, W., AND PETERSON, S. 1990. Feature oriented domain analysis (foda) feasibility study. Tech. rep.

KANG, K. C., LEE, J., AND DONOHOE, P. 2002. Feature-oriented product line engineering. *IEEE Software 19*, 4 (July/Aug.), 58–65.

LISBOA, L. B., GARCIA, V. C., ALMEIDA, E. S., AND MEIRA, S. R. L., 2007. Toolday a process-centered domain analysis tool.

LIU, D., AND MEI, H. 2003. Mapping requirements to software architecture by feature-orientation. In *Proc. 2nd Int'l Software Requirements to Architectures Workshop*, 69–76.

MACKINLAY, J. 1986. Automating the design of graphical presentations of relational information. *ACM Trans. Graph. 5*, 2, 110–141.

PLAISANT, C., GROSJEAN, J., AND BEDERSON, B. B. 2002. Spacetree: Supporting exploration in large node link tree, design evolution and empirical evaluation. In *Proc. INFOVIS*, IEEE Computer Society, 57.

RABISER, R., DHUNGANA, D., AND GRÜNBACHER, P. 2007. Tool support for product derivation in large-scale product lines: A wizard-based approach. In *Proc. 1st International Workshop on Visualization in Software Product Line Engineering (ViSPLE 2007)*.

ROBERTSON, G., CAMERON, K., CZERWINSKI, M., AND ROBBINS, D. 2002. Polyarchy visualization: Visualizing multiple intersecting hierarchies. In *Proc. ACM CHI 2002 Conference on Human Factors in Computing Systems*, ACM Press, Visualizing Patterns, 423 – 430.

SATYANANDA, T. K., LEE, D., AND KANG, S. 2007. Identifying traceability between feature model and software architecture in software product line using formal concept analysis. In *Proc Fifth International Conference on Computational Science and Applications*.

SEI, 2008. Software product lines. online, http://www.sei.cmu.edu/productlines/.

SELLIER, D., AND MANNION, M. 2007. Visualizing product line requirement selection decisions. In *Proc. 1st International Workshop on Visualization in Software Product Line Engineering (ViSPLE 2007)*.

SIEGMUND, N., KUHLEMANN, M., ROSENMULLER, M., KAESTNER, C., AND SAAKE, G. 2008. Integrated product line model for semi-automated product derivation using non-functional properties. In *Proc. 2nd Int'l Workshop on Variability Modelling of Software-intensive Systems (VAMOS)*.

SINNEMA, M., DE GRAAF, O., AND BOSCH, J. 2004. Tool support for COVAMOF. In *Workshop on Software Variability Management for Product Derivation*.

SINNEMA, M., DEELSTRA, S., NIJHUIS, J., AND BOSCH, J. 2004. Covamof: A framework for modeling variability in software product families. In *Proc. Third Software Product Line Conference*, 197–213.

SOCHOS, P., PHILIPPOW, I., AND RIEBISCH, M. 2004. Feature-oriented development of software product lines: Mapping feature models to the architecture. In *Proc. Net.ObjectDays*, Springer, M. Weske and P. Liggesmeyer, Eds., vol. 3263 of *Lecture Notes in Computer Science*, 138–152.

THIEL, S., AND HEIN, A. 2002. Modeling and using product line variability in automotive systems. *IEEE Software 19*, 4, 66–72.

VAN GURP, J., BOSCH, J., AND SVAHNBERG, M. 2001. On the notion of variability in software product lines. In *Proc. WICSA*, IEEE Computer Society, 45–54.

WARE, C. 2000. *Information Visualization: Perception for Design*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.