Visual Tool Support for Configuring and Understanding Software Product Lines

Goetz Botterweck, Steffen Thiel, Daren Nestor, Saad bin Abid, Ciarán Cawley
Lero, The Irish Software Engineering Research Centre
{goetz.botterweck, steffen.thiel, daren.nestor, saad.binabid, ciaran.cawley}@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.

This paper introduces a meta-model and accompanied research tool that employs visualisation techniques to support fundamental product line development tasks. The meta-model specifies major entities such as decisions, features, and components and relationships among them. We discuss which tasks can be supported based on the meta-model and show examples of how these tasks can be further enhanced by utilising interactive visualisation techniques implemented in our tool.

1. Introduction

Software product line (SPL) engineering promises benefits such as “order-of-magnitude improvements in time to market, cost, productivity, quality, and other business drivers” [1]. Many of these expected benefits rely on the assumption that the additional upfront investment in domain engineering, which is necessary to create the product-line and its core assets, pays off in the long run during application engineering, because product derivation based on a product line is (expected to be) more efficient than development from scratch.

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 [2]. Adopting visualisation techniques in SPL engineering can help stakeholders in supporting essential work tasks and in enhancing their understanding of large and complex product lines.

This paper introduces a meta-model and accompanied research tool that employs visualisation techniques to support fundamental product line development tasks. The tool implements various visualisation and interaction techniques that can support stakeholders in the process of product configuration for software product lines.

The remainder of the paper is structured as follows: In Section 2 we present an SPL meta-model that integrates major product line artefacts. In Section 3 we introduce our visual research tool (VISIT-FC) which is based on the meta-model and explain some of the visualisation and interaction techniques implemented. Section 4 provides illustrating examples of four product line tasks and how they can be supported using VISIT-FC. Section 5 discusses related work in visual product configuration and Section 6 outlines future work. Finally, Section 7 concludes the paper.

2. An Integrated SPL Meta-Model

Many SPL research approaches focus on single development artefacts, represented by isolated models, e.g. [3, 4]. However, 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. As an example for such interconnected SPL models we now present a meta-model that links several different types of models.
2.1. Three Models

The meta-model covers three models: First it allows us to describe features with all their usual dependencies and attributes. This part of the meta-model is based on earlier work in [5]. Second, we extended this into one direction, towards later phases, by adding modelling concepts to describe components that implement features. Third, we 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.

Overall, we ended up with five major packages in our meta-model (see Figure 2): Features, Components, and Decisions, as well as Kernel, which describes common structures and Relationships, to describe links among the models and the elements contained within them.

2.2. Relationships within and between Models

Our meta model supports relationships within a model (between elements in the same model) and relationships between models (between elements in different models).

Requires is a directed dependency between elements of the same type. For instance requires_FF expresses dependencies between features (see Figure 1). To include the general case we modelled it to have multiple sources and targets. This way, we are able to describe dependencies such as “(f1 and f2) requires f3” or “f4 requires (f5 and f6)”.

Excludes is an undirected dependencies, since it describes a set of elements that are mutually exclusive (not shown in the diagram).

In addition to these model-internal relationships, we have links between elements in different models, such as ImplementedBy. Again, we distinguish subtypes for the different elements: ImplementedBy_DF expresses that a decision is implemented by certain features and ImplementedBy_FC that a feature is implemented by certain components (see Figure 1).
3. Visual Tool Support

With the complex interrelated data sets that come with a software product line, comprehension is often difficult. Several studies show that a major factor in error in relation to product engineering is the complex implicit knowledge required to understand a product line [6, 7].

We have developed VISIT-FC, a Visual and Interactive Tool for Feature Configuration in order to improve this situation. VISIT-FC is based on the integrated meta-model presented in Section 2 and employs visualisation and interactive techniques to fulfil MacKinlay’s [8] expressiveness criteria. VISIT-FC adds interactive functionality for product line engineers allowing clear exploration and manipulation of the product line data. It provides a compact, interactive representation of large feature hierarchies, allows configuration with automatic constraint propagation, and provide hints for configuration problems and open decisions.

The implementation of VISIT-FC is based on the Eclipse Modelling Framework (EMF), which allows for integration with other tools. For instance, configurations created with VISIT-FC can be used as a starting point for model-driven product derivation. One example of a possible back-end is the derivation of an product-specific architecture [9].

We have used VISIT-FC to support product development activities of an automotive Restraint System Control Unit (RESCU) product line. 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.

In the remainder of this section we discuss specific visualisation techniques that are utilised in the tool. The techniques are explicit representation, horizontal linear tree layout, details on demand, incremental browsing, and focus+context.

3.1. Explicit Representation

VISIT-FC uses Explicit Representation as opposed to Implicit Representation. Explicit Representation refers to drawing methods which display the hierarchy as links between nodes. Implicit drawing methods represent the hierarchy by a special arrangement of nodes, e.g. containment or overlapping. Examples of implicit graph drawing are tree-maps [10], or the information cube [11]. Figure 3a shows a screenshot of main RESCU product line features in VISIT-FC.

3.2. Horizontal Linear Tree Layout

Advanced layouts exist for explicit tree-drawings such as cone-trees or space-trees [12]. However, for the purpose of this prototype, a 2D visualisation was chosen, and therefore a simple non-radial tree layout [13] was adopted. The horizontal orientation is preferable over the vertical orientation although the tool does allow the stakeholder to view the model in vertical tree layout. The non-radial (linear) layout and horizontal orientation combine to provide the optimal use of screen space to allow the display of the kinds of data related to a product line feature model. As an example, displaying the names of features on screen with a radial or vertical tree layout would result either in large amounts of overlapping or a zoomed out view (to avoid overlapping) both of which would obscurely render the visualisation.

The combination of an Explicit Representation and a Horizontal Linear Tree encodes a significant amount of information on screen utilising the restricted space in an efficient manner. VISIT-FC uses an explicit horizontal linear tree layout where 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 3b shows a portion of the RESCU feature model.

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 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), undecided (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 of additional colour coding. If the box is shaded, then the feature has been pre-configured or eliminated 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. Information encoded at this low level of visual representation is processed pre-attentively [12] by the human graphical system. Therefore once the colour encoding becomes familiar, a stakeholder would be able to interpret large representations rapidly.
3.3. 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. 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 removing irrelevant data.

3.4. 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. The stakeholder is thus able to perceive the feature structure step by step, and is not overwhelmed by the complete model.

3.5. Focus+Context

Focus+Context refers to the ability to focus on a particular aspect or portion of the visualisation while not losing the context in which that aspect or portion resides [14]. The advantage of Focus+Context is that the stakeholder does not get lost when zooming into a large structure, or exploring the details of certain features. They are always able to see where they came from, and are not required to keep this in memory. This can be useful, e.g., for the visualisation of search results or to see dependent feature nodes in distant parts of a large model.

Pan, Zoom and Degree of Interest in combination are powerful techniques that allow the stakeholder to move around the visualisation, zoom and highlight a particular area of interest. VISIT-FC provides 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 3c
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 “zoomed in” or “zoomed out” to varying degrees to allow an optimum view for the task at hand.

4. Visual Support of Product Line Tasks

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
- Represent cost driving features
- Represent high-risk features

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.

4.1. 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 4). 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 then highlighted through increased node size. 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 distinctly 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 removing all data from the screen except that which is directly connected to the feature being configured.

4.2. 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 could occur 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 three interconnected models from the RESCU product line for automotive restraint systems (Figure 5).
The SPL engineer starts by selecting the feature “Passenger detection” and the two choices below it, called “Bladder Mat” and “Weight Sensing” (see Figure 5). 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 metamodel (see Section 2).

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 (2). We can also do this in forward direction to identify model elements that are required by or implement the selected element (3). For all model elements identified this way, we also determine excluded elements (see 4).

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 (5) and that it requires a CANBus interface (6) 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.

However, 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. As an example, consider the matrix view shown in Figure 6, which shows the connection between decisions (left side, gray area) and features (right, white area). It corresponds to parts of the RESCU model shown earlier in Figure 5 (left and middle tree). It remains to be seen whether the affinity to tabular overviews expressed by industry practitioners can be substantiated by evidence (showing that tasks can be performed efficiently with these views) or is just caused by habit or lack of better visual tools.

4.3. 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.

Figure 6 shows a visualisation of costs in the RESCU product line with four cost driving features highlighted as enlarged nodes: ActiveHeadRest, DoubleStageFiring, WeightSensing, 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.

4.4. 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 features are usually directly related to the business success of the product line or a particular subset of product line products.

Figure 6 shows the four high-risk feature WindowCurtainAirbags, WeightSensing, ChildSeatDetection, and PedestrianProtection. They are marked in red. We can distinguish different root causes for risks that might influence the business success of a product line. Next we discuss four feature types that could cause increased risks: innovative, routine, 3rd party, and systemic features.

4.4.1 Critical Innovative Features. 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.

4.4.2 Critical Routine Features. 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.

In our example ChildSeatDetection is such a feature as it standard in most of the regions the product line will be launched (i.e., Europe, USA, and Japan).

4.4.3 Critical 3rd Party Features. 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.

4.4.4 Critical Systemic Features. 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.

5. Related Work

5.1. Product Derivation and Integrated Models

Examples of product derivation processes adopted by software product line organisations are discussed in Deelstra et al. [6] and the ConIPF project [7]. More and more approaches aim to integrate SPL models to support product line development activities, for instance linking feature models to software architectures [15-18]. As another example, Czarnecki and Antkiewicz [19] 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 [9].

Figure 7 – Cost Drivers and High-Risk Features

Other approaches describe the relationships between SPL models with formal techniques. In addition to precise traceability, this allows to formally check properties such as consistency among the integrated models [20, 21]. Lisboa et al. [22] 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. VISIT-FC is based on a meta-model that links decisions, features, and components. This allows for a connection among analysis and solution artefacts.

5.2. Software Product Line Visualisation

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

Decision models tend to be represented as a list of questions, each question being an open decision. Interactive features allow for the filtering of not currently available decisions, such as seen in the DOPLER tool suite [23]. Force directed layouts have also been used to represent the decision tree, for instance in the V-Visualize tool [24].

Feature models are the most prevalent method for modelling a product line. Polyarchies [25] are one method of representing a feature tree. 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 graph. VISIT-FC represents feature trees in this manner by using a coloured graph visualisation.

5.3. Product Line Configuration Tools

FeaturePlugin [26] is an Eclipse plug-in that supports feature modelling. It uses editors generated with the Eclipse Modelling Framework (EMF). Polyarchy layout is the method used to visualise the feature tree. Polyarchies can be difficult to comprehend once they get beyond a certain size even though they allow for collapsible nodes. Iconography is also used, to identify the features, but these icons are small and largely in black and white rather than in colour. This makes them difficult to pick out from the background. It is also difficult to comprehend the dependencies as constraints are shown as unsorted lists written in XPath 2.0, rather than being shown graphically.

pure::variants [27] is commercial feature modelling 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. However, the latter is limited use for larger models as the layout engine is not very sophisticated and the graph is not interactive. It leads to multiple edge crossings, and does not take dependencies into account when laying out the graph. These are displayed by default and, although a different colour, serve more to obscure information than to ease comprehension. The nodes in the graph are not distinguished other than by a thin outline box, meaning it requires extra effort to comprehend. This is in contrast with VISIT-FC’s philosophy of using colour to make nodes easily distinguishable, using interactivity to simplify navigation, and having details available on demand, not imposed on the user.

The ConIPF Variability Modelling Framework (COVAMOF) is augmented by tool support [28] integrated with Visual Studio. The tool provides a polyarchy view of the variability, and has a graph based visual component. The visual component uses iconography, but lacks the use of colour characteristic to VISIT-FC. The overall tool provides many views where a simpler view with contextual information could be more suitable, as it would be less likely to overwhelm the user.

Kumbang [29] provides tool support for integrated feature and component modelling. However, similar to other tools mentioned before, the configuration of features and components are isolated, although Kumbang does link the features to the architecture. Unfortu-

nately, there exists no direct visual, interactive representation of the relationships between these models. The graphical notation used by Kumbang’s visual component is based loosely on E-R diagrams, which have been shown to have problems with scalability and comprehension [30], and does not make use of the visualisation techniques suggested by Moody, to assist in comprehension and navigation.

6. 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 product line based product development can be addressed. Even simple information encoding can provide an increase in the speed at which product configurations can be interpreted. More in depth research into visualisation techniques and their applicability to and usability for tasks such as variability management, product configuration, and evolutionary support is planned.

Development of the tool to implement further functionality described by the meta-model is also planned. This would 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 will also be explored.

We have already explored certain interactive scenarios using the industry based RESCU product line described in the introduction to section 3. The tool has been tested with models of up to 2,500 features without noticeable performance issues. Additional industrial case studies are planned to further investigate the effectiveness of the tool.

7. Conclusions

In this paper we presented a software product line meta-model and introduced a research tool that employs a variety of visualisation and interaction techniques. This aids in improving the understanding of product lines and to support fundamental development tasks.

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 replace the dependence on a small number of experts and allow software product line engineers perform their tasks with much greater autonomy.
8. Acknowledgements

This work is partially supported by Science Foundation Ireland under grant number 03/CE2/1303-1.

9. References