User Interface Engineering for Software Product Lines – The Dilemma between Automation and Usability

Andreas Pleuss
Lero
University of Limerick, Ireland
andreas.pleuss@lero.ie

Benedikt Hauptmann
Technische Universität
München, Germany
benedikt.hauptmann@in.tum.de

Deepak Dhungana
Siemens AG Austria
Corporate Technology
deepak.dhungana@siemens.com

Goetz Botterweck
Lero
University of Limerick, Ireland
goetz.botterweck@lero.ie

ABSTRACT

Software Product Lines (SPL) are systematic approach to develop families of similar software products by explicating their commonalities and variability, e.g., in a feature model. Using techniques from model-driven development, it is then possible to automatically derive a concrete product from a given configuration (i.e., selection of features). However, this is problematic for interactive applications with complex user interfaces (UIs) as automatically derived UIs often provide limited usability. Thus, in practice, the UI is mostly created manually for each product, which results in major drawbacks concerning efficiency and maintenance, e.g., when applying changes that affect the whole product family. This paper investigates these problems based on real-world examples and analyses the development of product families from a UI perspective. To address the underlying challenges, we propose the use of abstract UI models, as used in HCI, to bridge the gap between automated, traceable product derivation and customized, high quality user interfaces. We demonstrate the feasibility of the approach by a concrete example implementation for the suggested model-driven development process.

Author Keywords

User Interface Engineering; Model-driven development; Software Product Lines; Usability Engineering

ACM Classification Keywords

D.2.2 Software Engineering: Design Tools and Techniques—*User interfaces*; D.2.9 Software Engineering: Management—*Software configuration management*; H.5.2 Information Interfaces and Presentation: User Interfaces—*Theory and methods*

INTRODUCTION

A *Software Product Line* (*SPL*) aims for the development of a family of similar software products from a common set of shared assets by making use of the commonalities among them [6, 16]. By applying SPL practices, organizations are able to achieve significant improvement in time-to-market, engineering and maintenance costs, portfolio size, and quality [6]. SPLs have been commercially applied in many industry domains [20] including highly interactive applications like e-commerce software [2] or mobile games [1].

The fundamental premise of an SPL is that the initial investment in a family of products pays off later by allowing systematic, efficient derivation of products. This can be achieved by using techniques from model-driven engineering [21, 23] like automated model transformations to derive the final product from a given product configuration. While this works well for deriving most parts of the product implementation [8, 24, 25], it has limitations for the product's user interface (UI) part: A high quality UI must not only adhere to certain functionality defined by a product configuration (e.g., the presence or absence of UI elements) but also meet usability requirements like adequate layout, composition into screens, and choice of UI element types. This requires to customize the UI beyond purely automated derivation [15, 4]. A simple solution is to design the UI manually for each product [2]. However, as practice shows, this can result in serious drawbacks regarding error rate and maintenance [4].

This paper investigates these problems based on real-world examples and analyses the development of product families from a UI perspective. The paper is structured as follows: We first show the basic SPL concepts followed by their application to UIs. Then we discuss solution alternatives and their benefits and drawbacks. As it turns out, there is a dilemma between support for systematic, automated SPL concepts and usability of the resulting UIs. As we show, this challenge can be addressed by a model-driven SPL process with specific support for UI customization. In the remainder of the paper, we first analyze the different aspects of a UI to be customized and then describe a resulting model-driven SPL process for interactive applications.

GENERAL SPL CONCEPTS

A SPL is used for the efficient development of a family of related products from a shared set of software assets. A common example is online shop software: As different online

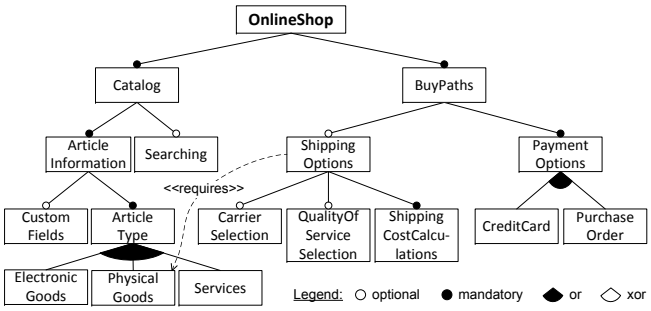


Figure 1. Example feature model for online shops.

shops (as commonly found in the web) have large commonalities in their functionality they can all be built from a common set of software assets. However, there are also variations between them, like the supported payment methods or the way the articles sold in the shop are organized. The core idea of SPL is to systematically manage this variability so that (ideally) a concrete product can be derived by just selecting its desired options.

The commonalities and variability in a SPL are usually specified in terms of a variability model. In this paper we use *feature models* [18] as a very common variability modeling concept but there are several other approaches, e.g., decision models [17] or OVM [16], which are used analogously.

A feature model specifies all features supported by the SPL and the dependencies between them. Figure 1 shows a small example feature model for online shop software[1]. Each node represents a feature that can be supported by the software, like a Catalog, Searching or different ArticleTypes[2]. The relationships between parent features and child features are constrained as *mandatory* (the child feature is always required), *optional* (the child feature is optional), *or-group* (at least one of the child features must be selected), or an *xor-group* (exactly one of the child features must be selected).

For instance, in Figure 1 each online shop must support a Catalog (mandatory feature) which includes ArticleInformation and optionally Searching. ArticleInformation includes the ArticleType from which at least one child has to be selected (or-group). By definition, selecting a child feature requires its parent to be selected as well. In addition, cross-tree constraints can be defined, such as *requires* and *excludes*. For instance, in Figure 1 ShippingOptions requires PhysicalGoods as otherwise there is no need to support shipping.

A feature model allows specifying a concrete product by configuring the product, i.e., selecting and deselecting features according to the constraints in the feature model. Each feature is mapped to SPL assets (the implementation of a feature; depending on the target platform) so that a given feature configuration (ideally) allows direct derivation of the corresponding implementation. For instance, there might be a software component Shipping which implements the different shipping options and is only included into the implementation if the feature Shipping was selected.

Figure 2a shows the basic (model-driven) SPL process (Figures b) and c) shown in comparison will explained later). The upper part shows the *domain engineering* which refers to creation of the whole SPL based on domain knowledge and market analysis. This includes defining the feature model, the SPL assets, and a mapping between them (called *feature mapping*). The lower part shows the *application engineering* which refers to derivation of concrete products based on the SPL. A product is developed by defining a *feature configuration*, i.e., selection of features. From this, the product implementation is derived based on the feature mappings.

The product derivation can be automated using model transformations. Usually, the transformation is performed on the model level, i.e., the implementation is represented by a model ("code model") which can read and write the actual code, to reduce the complexity of the transformation and to ensure traceability of code changes [8, 2]. The derivation itself is performed either by composing the selected SPL assets ("positive variability") or by starting with an implementation of the whole SPL and selectively deleting deselected SPL assets ("negative variability"). Whether an SPL asset is selected or deselected is defined by the feature mapping. However, usually the mapping is not a 1:1 mapping between features and SPL assets but more complex and is specified, e.g., by constraints. We will show a concrete example applied to UIs in the next section.

APPLICATION OF SPL CONCEPTS TO THE UI

The initially most natural way to realize a SPL for interactive applications is a straightforward application of SPL concepts to all its parts, including the UI. This means that the application's UI is just considered as an SPL asset like any other application part. This section shows how this can be realized.

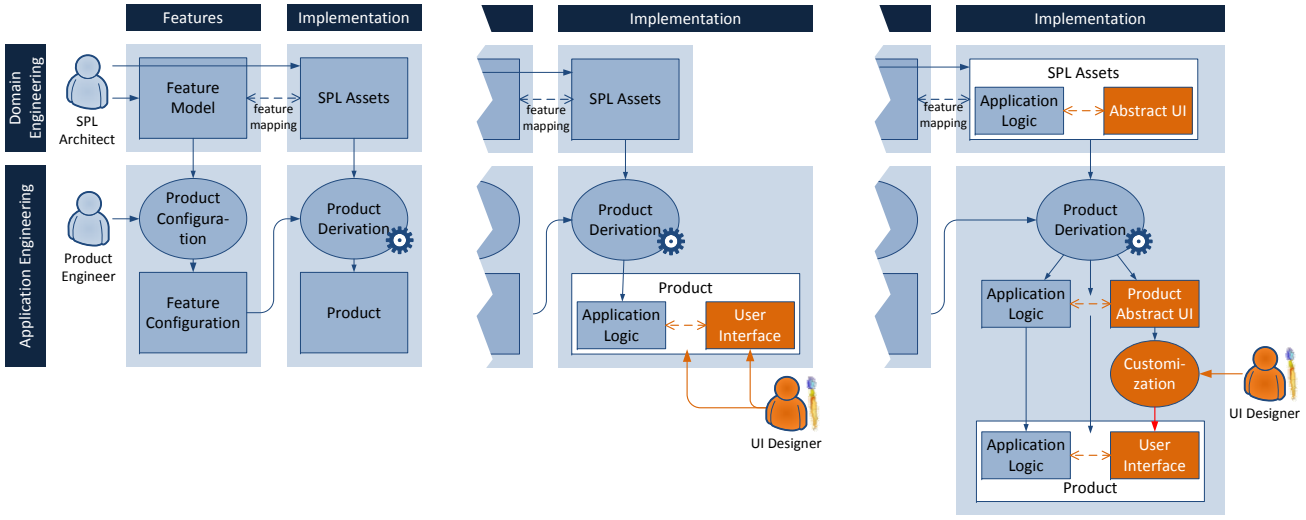
In the following we illustrate how to specify a mapping between UI elements and features which enables product derivation. To this end, we use the approach described in [7]. It uses the principle of "negative variability" [25] which means that product derivation starts from a superimposed model (created manually) which contains the implementation for all features in the whole SPL. The model elements are annotated with *presence conditions* over features. During (automated) product derivation, all model elements are removed whose presence condition is false based on the current feature configuration. Presence conditions can be specified as boolean constraints over features and more complex constraints can be specified using arbitrary XPath[3] expressions. The default present condition is "true", i.e., model elements without an explicit presence condition always remain present in a product.

Figure 3 shows an extract of a potential (manually designed) UI for the example online shop SPL annotated with presence

[1] See, e.g., [12] for a more realistic example with 225 features.

[2] Please note that to avoid misunderstanding we use the term "product" to refer to products developed with a SPL while we use "article" to refers to the items sold in an online shop.

[3] http://www.w3.org/TR/xpath/



(a) Common model-driven SPL approach. (b) Approach with manual UI design. (c) Model-driven approach with UI customization.

Figure 2. Model-driven product derivation in SPLs.

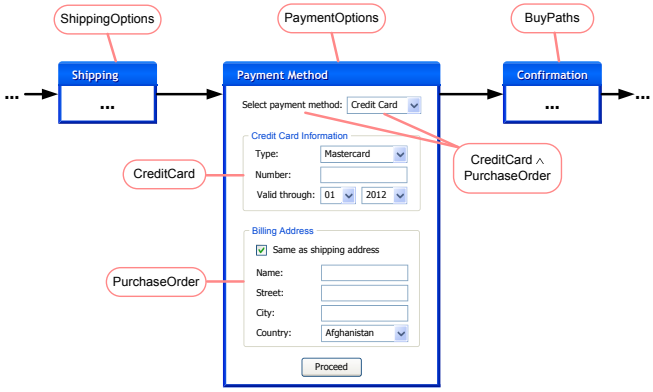


Figure 3. Example for a feature mapping for a UI.

conditions over the features from Figure 1[4]. It shows three screens Shipping, Payment Options, and Confirmation, the basic navigation between them (represented by arrows), and the associated presence conditions. For instance, the screen Shipping is removed from a product (together with all its content) if the feature ShippingOptions is deselected in the feature configuration. A slightly more complex constraint is specified for the the combo box Select payment method: it remains only present in a product if the features CreditCard and PurchaseOrder are both selected as otherwise there is nothing to select for the user.

In this approach, presence conditions are evaluated according to the containment hierarchy, i.e., if a container is removed (e.g., a screen or a grouping) all contained elements are removed as well. For instance, if the feature PaymentOptions is deselected the screen Payment Method is removed with all its contents.

In addition, [7] allows specifying rules for post-processing steps. An example is the navigation (represented by arrows in Figure 3): if a screen is removed, incoming and outgoing arrows are merged to close the navigation flow. Of course, this can be customized by attaching presence constraints to navigation flows.

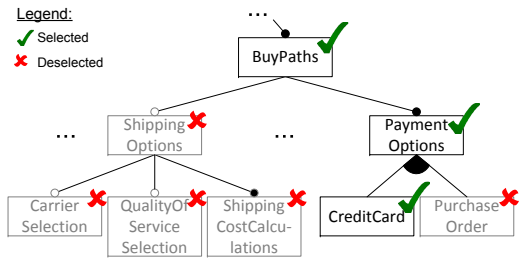
In this way, the whole UI for a product can be derived from a feature configuration including its behavior specification. For instance, [7] shows derivation of Activity Diagrams, which might be used to specify the behavior of a UI. Figure 4 illustrates an example. The upper part in Figure 4a shows a feature configuration based on the feature model from Figure 1 (only those parts of the feature model relevant for the example). In this example, ShippingOptions are deselected and CreditCard is the only payment option. The lower part in Figure 4b shows the resulting UI, which was derived based on the feature mapping (from Figure 3) and the derivation rules explained above.

THE DILEMMA BETWEEN AUTOMATION AND USABILITY – SOLUTION ALTERNATIVES

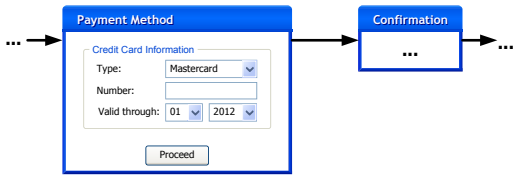
In the preceding sections we introduced the general concepts of SPLs and their potential applications to the UI. While SPL concepts work very well in practice (see, e.g., [20]), the specific challenges of the UI in SPLs for interactive applications have mostly been neglected in software engineering community. In fact, the UI aspect differs from other SPL assets insofar as a high quality UI must not only provide a certain functionality but also must provide high usability which is much more difficult to satisfy with automated techniques.

In this section, we discuss the challenges when applying SPL concepts to the UI. There are two experience reports from in-

[4]For the product derivation, the information in the figure needs to be specified as a model using an appropriate modeling language but this is not important at this stage. We will discuss appropriate modeling approaches later in this paper.



(a) Feature configuration.



(b) Resulting UI.

Figure 4. Example product for the online shop SPL.

dustry that mention the UI aspect and which we hence use for our discussion: [2] presents the experience from a company called *SystemForge* with their model-driven SPL for online shops for small and medium businesses. In [4], the company *HIS* reports on challenges arising with their product *HISinOne*, a web-based university management software supporting, e.g., management of students, human resources, and financial accounting. HISinOne is not a SPL in the strict sense (i.e., there is no SPL approach fully applied) but is still a highly variable ecosystem providing different customized products for their customers.

In the remainder of this section, we will first discuss the benefits and problems of existing solution alternatives and then we present a third solution alternatives to address the identified problems.

On the Need for UI Customization

The approach described in the previous section (Figure 3) directly applies the common SPL concepts (as shown in Figure 2a) to the application's UI part. This means that the UI is considered like any other software artifact as part of the SPL assets in Figure 2a. The main advantage is that in this way all existing strengths of SPLs – like efficiency, reuse, and a high degree of automation – apply to the UI part as well. Moreover, existing further SPL concepts can be applied, like concepts for SPL validation or maintenance.

However, this approach has limitations. The quality of a UI is not only determined by functional requirements (i.e., presence or absence of UI elements and behavior) but also by its usability which is influenced by many other factors like the decomposition into screens, the layout, and the detailed visual appearance [19]. All such UI properties must hence be customized to the specific needs and requirements of a specific product.

In context of a SPL, two levels of UI customization can be distinguished, which we discuss in the following.

Customization according to general usability requirements: In real SPLs, feature models become usually large and complex and consist of many hundreds or even thousands of features. Hence, it is often impossible to foresee the consequences of all different combinations of feature configurations in advance. For the UI this can mean that usability issues arise after automated product derivation. For instance, removing elements from a screen (as in negative variability) can lead to screens which are almost empty and are no longer useful as screens of their own [4]. To some extent this might be addressed by additional rules or heuristics during the product derivation, e.g., that a screen with very few content is merged with other screens. In practice, it depends on the complexity of the UI and the desired degree of usability whether such heuristics are sufficient enough.

Customization according to product-specific UI requirements: The most obvious part of an UI which often needs to be customized is its visual appearance. For instance, in case of an online shop, each shop's visual appearance should reflect the shop owner's corporate identity. Customizing the visual appearance is often unproblematic as this can be achieved without changing the derived UI implementation, e.g., by using stylesheets and dynamic loading of custom text and graphics. However, other UI properties like its general structure (e.g., decomposition into screens and screen layout) cannot be customized in this way. For instance, HIS reports that universities want to have input fields in the same order as they appear on their paper forms to increase the input speed [4].

To some extent, such product-specific customization might be captured by extending the feature model with UI specific options, e.g., to provide several standard layouts to choose between [2]. However, as pointed out by [2], this alone is not sufficient as all their customers want more control over their UI. Similarly, [4] clearly states that, according to their experience, the complexity of UI customizations and their interdependencies cannot be captured in a feature model.

Summary: In summary, customization needs of the UI within a SPL can be addressed to some extent by applying heuristics or capturing UI-specific options in the feature model. However, for interactive applications where the UI is important and highly variable, this is often not sufficient in practice.

On the Need for Automation

The preceding section has shown that interactive applications often need individual UI customization. On the other hand, the application logic of, e.g., online shops is well understood and can be derived very efficiently using model-driven SPL techniques. One way to solve this conflict is a two-fold development approach: The application logic is generated using a SPL while the corresponding UI is developed manually. Figure 2b illustrates this alternative in comparison to the basic SPL approach in Figure 2a. For instance, this solution is applied by SystemForge [2] by generating only UI templates (providing variables to access the application logic) while the UI is developed manually by the customer herself

on that base. Similarly, HIS allows the customers unrestricted manual modifications of the UI code [4].

However, this solution results in new problems. While the manual design allows unrestricted customization according to the product-specific usability needs, it does no longer comply to the overall model-driven SPL approach. This means that systematic SPL concepts can no longer be applied to the application's UI part. For instance, the feature configuration still contains lots of information which is relevant for the UI, but this information is no longer used in a purely manual UI design approach. Moreover, the UI's compliance to the feature model as well as the correct linkage between UI and application logic now has to be ensured manually which can be tedious and error-prone.

The most important drawback is in evolution and maintenance. Concepts from model-driven SPL, like traceability, can no longer be applied to a manually designed UI. Updates on the software which require changes on the UI can no longer be automatically applied. For instance, HIS reports on these difficulties when, e.g., adding new input fields to the UI of the whole SPL, e.g., due to new laws for Universities [4]. In a manually designed UI, each change has to be integrated manually to each single product which is error-prone and lacks of efficiency.

In addition, other benefits of model-driven SPLs, like support for multiple target platforms, can no longer be applied to the UI. For instance, an online shop might be provided in several versions for the desktop and for different mobile devices. In a model-driven SPL, all these versions can be derived consistently from a given feature configuration. Having purely manual UI design, each version must be created and maintained separately and consistency between them must be ensured manually.

In summary, there is a dilemma between usability and automation. It is desired to have an approach which provides full support for UI customization but still integrates with the systematic model-driven SPL concepts. We propose such an approach in the next section.

A Model-driven Approach Supporting UI Customization

We propose to use abstract UI models which allow all required manual customization on the model level and fully integrate into a model-driven SPL approach. For this, we make use of the existing concepts defined in the research area of *model-based user interface development* (*MBUID*) [22, 5, 10]. These approaches provide models on different abstraction levels to specify the UIs and to support multi-platform development. They do not address SPLs but we can reuse their basic concepts of abstract UI modeling for our purpose.

The types of models typically used in MBUID are Task Model, Domain Model, Abstract UI Model, and Concrete UI Model (see, e.g., [5]). We briefly introduce them in the following[5].

[5]Please note that advanced properties of UIs, like context-sensitive behavior or multimodality, as discussed in [5], are beyond the scope of this paper and not further considered.

The most abstract models are the *Task Model* and the *Domain Model*. The *Domain Model* is a conventional model to describe domain concepts and the corresponding application structure, e.g., in terms of a UML class diagram. A *Task Model* describes the user tasks to be supported by the application (like "Choose Articles" in an online shop application) and the temporal relationships between them. A concrete approach for task models is, e.g., CTT [14].

An *Abstract UI Model* describes the UI in terms of abstract UI elements, which are platform-independent abstractions of UI widgets, like *input* element, *output* element, *selection* element, or *action* element (abstraction of a button). Each abstract UI element realizes tasks from the *Task Model* and is associated with properties or operations from the Domain Model. Abstract UI Elements are contained in *Presentation Units*, which are top-level containers like Windows/Frames, and other *UI containers* (abstractions of, e.g., panels). The *Abstract UI Model* also describes the navigation between the *Presentation Units* and an (abstract) layout.

A *Concrete User Interface Model* refines the *Abstract UI Model* by specifying concrete UI elements, i.e., concrete UI widgets, and their layout. It can still abstract from a specific GUI API (e.g., providing a generalized "List Box" widget).

The final implementation, sometimes represented as a model as well, is referred to as the *Final UI Model*.

To solve the dilemma described in the previous sections, we propose to perform product derivation for UIs on an appropriate level of abstraction (i.e., derivation of more abstract UI models instead of a final UI) which allows to specify all required UI customizations using the models. From these abstract UI models it is then possible to move down to the final implementation using a model-driven process. The approach is shown in Figure 2c.

In this way, it is possible to overcome all drawbacks described in the previous sections: On the one hand, there is now support for full UI customization provided, on the other hand all advantages of a model-driven approach still apply like efficiency, consistency, traceability, support for maintenance, and even support for multiple target platforms. Due to the automated derivation of the (abstract) UI, also the consistency with the feature configuration and the correct linkage of the UI with the application logic is automatically ensured.

In the next section, we analyze which is the right abstraction level, which UI elements need to be potentially customized and which can always be derived. Afterwards, we present on that base the details of the proposed approach.

ANALYSIS OF UI CUSTOMIZATION NEEDS

In this section we discuss the different aspects which make up a UI and analyze them regarding their potential need for customization within an SPL. This is necessary for our approach to identify the right abstraction level for the UI models to be used and, in particular, which aspects of a UI can be directly derived from a feature configuration and for which aspects we need to support customization.

Development / *UI Aspect*	Customization	Customization Specification
Tasks and Temporal Operators	No	-
Abstract UI (AUI) Elements	No	-
Relationships to Domain Model	No	-
Presentation Units	Yes	AUI Model
Navigation	No	-
Concrete UI (CUI) Elements	Yes	AUI to CUI Transformation
Layout	Yes	AUI Model or AUI to CUI Transformation
Visual Appearance & Adornments	Yes	Stylesheets
Other CUI properties	Yes	AUI to CUI Transformation

Table 1. UI aspects and their need for customization.

Table 1 lists the different aspects of UIs as they typically appear in MBUID models. *Other CUI properties* refers to element-specific properties of concrete UI elements like if a text field provides word-wrapping or a if list box allows multi-selections. The second column of Table 1 specifies whether there is a need for product-specific customization in a SPL (in contrast to purely automated derivation from a feature configuration). The third column specifies on which level of abstraction the customization should be supported within a model-driven process. In the following, we discuss each row in detail.

Tasks and temporal operators: The tasks are directly related to the functionality of an application. Customizing the tasks within a SPL would contradict the SPL approach where the functionality of a product is specified by a feature configuration. Hence, there is no need for customization. (Of course, it is possible that a product of a SPL is extended with new custom functionality but this means to change the application logic and goes beyond the scope of UI customization.) The temporal operators between tasks are directly associated with them and there is no need to customize them as long as the tasks do not change.

Abstract UI elements: The abstract UI elements are still on a very generic level (such as input, output, selection, action) and, hence, determined by the tasks. For instance, a task "select payment method" will always be realized by a selection element while "input credit card number" will be realized by an input element. Thus, as there is no need for customizing the tasks, the abstract UI elements do not need to be customized.

Relationships to the domain model: Abstract UI elements are associated with properties or operations from the domain model to specify the information which they represent or the operation they trigger. Therefore, these mappings to the domain model are directly related to the corresponding tasks and do not require customization.

Presentation Units: The decomposition of the UI into Presentation Units (i.e., top-level UI containers that cluster other UI elements into logical groups) needs to consider product-specific constraints. As described above, product derivation can lead to almost empty (or overfull) presentation units due to removal (or addition) of UI elements. Moreover, product-specific layout and space constraints can require customization of presentation units as well. For instance, in some online shops the description of articles might require extra space or a specific layout which influences the distribution of UI elements to the presentation units. Related to that, there might be product-specific requirements on which presentation unit a specific information should be presented. An example is the decision, which information about the articles to show on an "Article Details" page only, which information in the "List of Articles", and which in the "Shopping Cart". For instance, in an online shop selling business software, the number of licenses to buy and the resulting prices can be very important. In contrast, in a shop selling computer games, most customers will buy only a single license, so that the input field to set the number of licenses might be displayed on a less prominent place (e.g., in the "Shopping Cart" only but not in the "List of Articles").

The product-specific customization of Presentation Units should be supported early on abstract level (i.e., abstract UI model) as many further development steps, like navigation and layout specification, rely on the definition of the Presentation Units.

Navigation: Once the Presentation Units are defined, the navigation between them can be derived based on the temporal operators from the task model. Thus, in most cases the navigation does not need to be customized itself.

Concrete UI elements: The choice of concrete UI elements implementing the abstract UI requires customizations as well. On the one hand the choice of concrete UI elements is often considered by the user as part of the visual design and thus subject to customer requirements, e.g., when using tabbed menus instead of classical menu bars. On the other hand, the choice of a concrete UI element depends on the actual content (e.g., the articles in an online shop). For instance, for a small number of choices, radio buttons might be desired while a large number of choices might be represented by a list or by even more sophisticated custom widgets, as used in Rich Internet Applications.

The choice of concrete UI elements can be customized efficiently by adapting the model transformation which defines the mappings from abstract UI Elements to concrete UI Elements. In SPL context, this transformation is defined once for the SPL and provides the default mapping. There are several ways to adapt a model transformation, either just by extension, or by using an additional "mapping model" to define custom mappings, or by other mechanisms provided by current model transformation languages (like ATL[6] or ETL[7]) like parameterization or rule inheritance.

Layout: As the Presentation Units and the concrete UI elements can require customizations, the layout need to be customized as well. This can either be performed in the abstract UI model to specify general layout rules on abstract UI level or in the model transformation from abstract UI model to concrete UI model (using the same techniques like for the con-

[6] http://www.eclipse.org/atl/
[7] http://www.eclipse.org/gmt/epsilon/doc/etl/

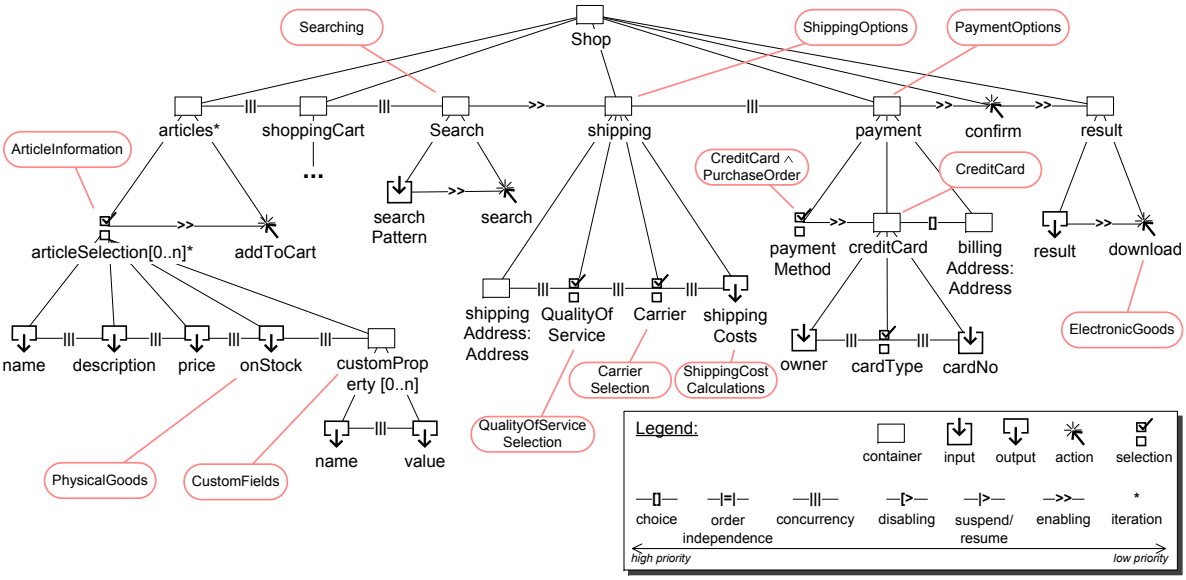


Figure 5. Abstract UI model for a online shop product line including presence conditions.

crete UI elements). Fine-tuning of single layouts can be specified in the concrete UI model.

Visual appearance: The visual appearance has often to be customized, e.g., to make different applications look less uniform or according to concrete customer requirements like the corporate identity. Customizing the visual appearance can be performed without changing the UI itself using, e.g., stylesheets and dynamic loading of text and images at runtime.

Other CUI properties: Properties of UI elements which are directly related to functionality, e.g., whether a list allows multiple selections, should be derived automatically during product derivation. Other properties are set during the transformation from abstract to concrete UI and can be customized there in the same way as described for the concrete UI elements.

PROPOSED DOMAIN ENGINEERING CONCEPTS

In the remainder of the paper we show how the proposed process from Figure 2c can be realized in detail based on the analysis results in the previous section. We have implemented the complete proposed approach to ensure the feasibility of the concepts proposed. Due to space limits we will focus in this paper on the most important concepts only. Interested readers can find all implementation details in [9].

In this section we explain the domain engineering (see Figure 2c), i.e. definition of an appropriate abstract UI model and a feature mapping. Thereby, according to our approach, we need to distinguish between the application logic and the (abstract) UI.

Abstract User Interface Modeling: According to the analysis in the previous section, tasks, temporal operators, abstract UI elements, and their relationships to the domain model do not require customization. Hence, they need to be specified just once and can afterwards automatically be derived using the common SPL product derivation concepts. Moreover, as in MBUID the abstract UI elements are a more concrete representation of tasks, the tasks can be omitted here as they contain no extra information. This means that for the product derivation we need a model which contains the abstract UI elements, their relationships to the domain model, and the temporal operators. In the following, we introduce such a model.

Figure 5 shows our abstract UI model for the online shop example SPL. It is annotated with exemplary presence conditions analogous to Figure 3. The nodes in this model are the abstract UI elements. They are structured using the concepts from CTT task models [14], i.e., in a tree hierarchy and with temporal operators between them. The abstract UI elements supported are the same like in common model-based UI development approaches, like *input*, *output*, *selection*, and *action*, which must be leaves in the tree. All non-leaf nodes are UI containers. An exception is the selection element which can be either used as simple abstract UI element or as special container. In the former case, it is used for a simple value selection, in the latter case it is used to select from a list of objects represented by its children. For instance in Figure 5, the selection element articleSelection allows to select from a list of articles, which are represented by multiple children.

The abstract UI model also supports specification of multiplicities for elements whose number is not specified yet as it is either product-specific or calculated at runtime (like the number of articles in the articleSelection). It is also possible to reuse a container multiple times by defining multiple copies. For instance, shippingAddress and billingAddress are both copies of the container Address (defined elsewhere) which is denoted by a colon after the container name followed by the name of the copied container.

The semantics of the temporal operators refers to the task associated with the abstract UI Element, i.e., input of data for

input elements, output of data for output elements, etc. The available operators are the same like in CTT [14] (see legend in Figure 5)[8]. Abstract UI elements required for the navigation (like a "Submit" button in a web application) are not specified in the abstract UI model here as those have to be generated based on the navigation which is specified later.

Each leaf is associated with a property or operation from the application logic (not visualized in the figure). This can also be non-persistent helper classes used only for return values or for database queries, etc. We briefly introduce a model for the application logic in the next section.

Application Logic: The models and model transformation used to specify and generate the application logic in a SPL are often specific to the domain and the target platforms. For instance, a SPL for web applications uses a different approach than a SPL for infotainment systems in a car.

We use here an existing model-driven approach for web applications as example, called *UWE* (*UML-based Web Engineering*) [11]. UWE supports, in conjunction with its extension UWE4JSF[9], model-driven development of Rich Internet Applications based on *Java Server Faces* (*JSF*).

UWE provides five kinds of platform-independent models, three of which can be considered as application logic in terms of Figure 2c: The UWE *Content Model* and the UWE *User Model* describe the application structure in terms of extended UML class diagrams, whereas the UWE *Process Model* specifies the application logic in terms of extended UML activity diagrams.

As UWE supports generation of complete web applications, it comes also with its own models to describe the application's UI. These are the UWE *Navigation Model* (describes the navigation structure between web pages) and the UWE *Presentation Model* (specifies the UI of pages). This means that our model-driven process for the UI has to end up with these two models (instead of generating a UI implementation directly) to be able to leverage the UWE approach.

Feature Mapping and Derivation Rules: For the feature mapping and the resulting derivation, we use the "negative variability" approach based on [7] as previously explained by Figure 3. Each abstract UI element can be mapped to a feature. If containers are removed their content is removed as well.

For the temporal operators we use the following derivation rule: If a deleted abstract UI Element has two neighbor siblings (in the tree structure) then the temporal relationship with the *higher* priority (see legend in Figure 5) is deleted and the remaining one is used to connect the two siblings. If a deleted abstract UI element has only one neighbor sibling (i.e., it is a leftmost or rightmost sibling in the abstract UI tree), then its temporal relationship to its neighbor is deleted as well.

[8]In CTT, additional operators allow to specify whether information is passed between two tasks. We do not need this distinction here, as information passing is managed by the application logic anyway.
[9]http://uwe.pst.ifi.lmu.de/toolUWE4JSF.html

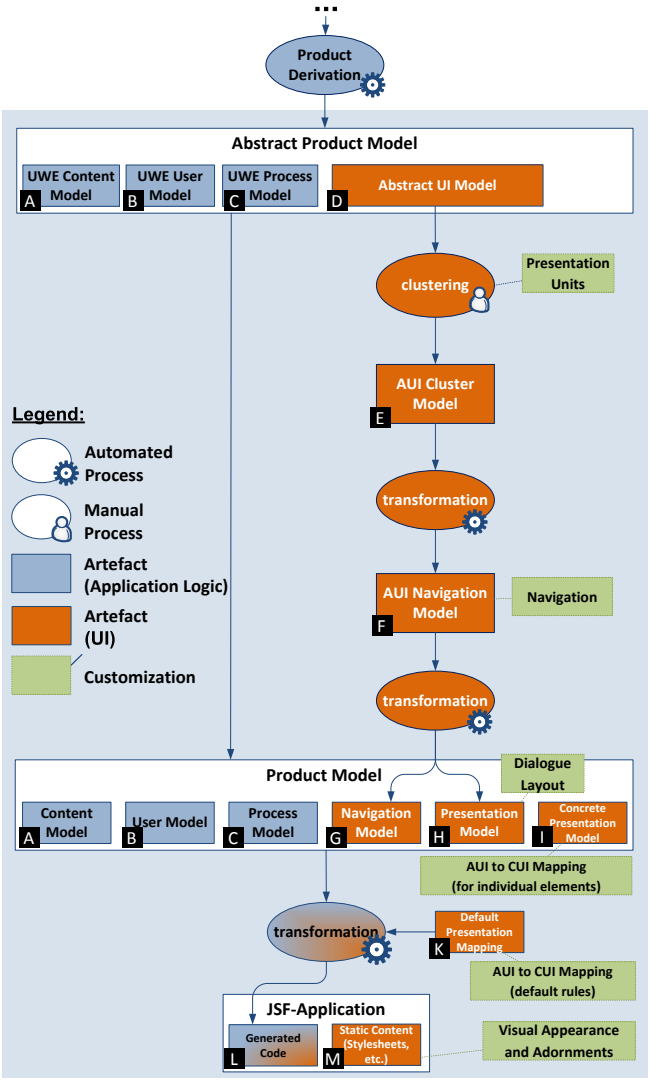
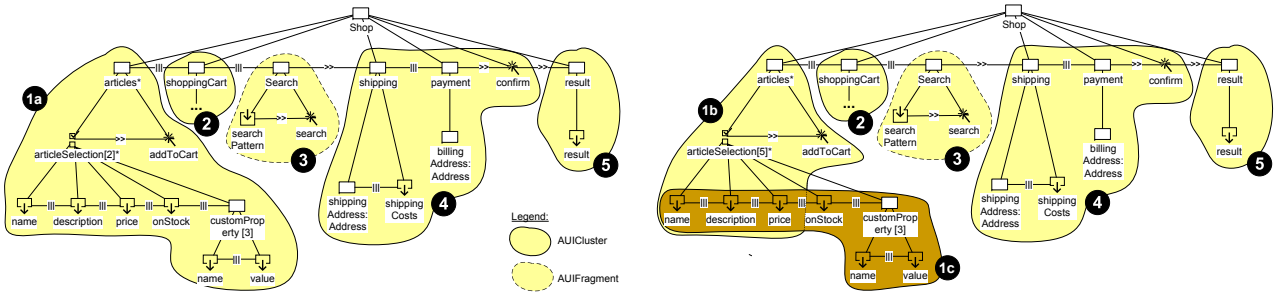


Figure 6. UI Model transformation process.

For the application logic, the negative variability approach is applied as well (derivation of UML class and UML activity diagrams in case of UWE) but this is not further discussed here.

PROPOSED APPLICATION ENGINEERING CONCEPTS

This section shows the application engineering within our model-driven process (see Figure 2c), i.e. the derivation of concrete products with support for UI customization on model level. This process is shown in more detail in Figure 6. It starts with the derivation of the product-specific abstract model UI model based on the feature mapping and the derivation rules described in the previous section. The product derivation performed so far resulted in an abstract, product-specific UI model (D in Figure 6) and an incomplete UWE model containing only the application logic A B C for the configured application. The result of the process should be the complete UWE model including the UI specification, i.e., UWE navigation G and UWE presentation model H I.



(a) Single cluster for article list and article details

(b) Article details in a separate cluster.

Figure 7. Two alternative clusterings for a product-specific UI.

The steps in between should support customization of the UI for all aspects identified in Table 1. We describe them in the following.

Presentation Units and Navigation: As discussed before, the decomposition of the UI into Presentation Units (i.e, the single windows or web pages) often needs to customized due to spatial and other product-specific constraints. We support this on the level of the abstract UI model by specifying clusters which results in a so-called *Abstract UI Cluster Models* E (inspired by [3]) shown in Figure 7. Abstract UI Cluster Models support two cluster types: *AUI-Clusters* are the basic cluster type to group several abstract UI elements. They will become a single Presentation Unit later on. *AUI-Fragments* are also used to group various abstract UI elements but will not become independent Presentation Units in the later UI. Instead, they will become reusable components that can be embedded into AUI-Clusters, to realize (sub-)views which are available within multiple other views, like a "search bar" available on multiple web pages.

Figure 7 gives an example for two alternative clusterings of a product-specific abstract UI. For instance, the UI containers shoppingCart ❷ and result ❺ (and their children) are clustered into AUI-Clusters of their own. The UI containers shipping and payment (and their children) are clustered together with the action element confirm into a single AUI-Cluster ❹. The container Search and its children are clustered as an AUI-Fragment ❸ which means that it will be embedded into other clusters. The two alternatives (Figure a) and b)) differ in the clusters used to present the products in the shop (UI container products and its children): In Figure 7a there is a single cluster ❶ᵃ to display the list of products, including product details. The multiplicity of productSelection is set to "2" which means that two products are displayed at once. Figure 7b shows an alternative decision where the product list is more condensed (showing only the product's name, description, price) so that five products are displayed on a single view ❶ᵇ. Instead, an additional view (dark colored cluster ❶ᶜ in Figure 7b) is used to show the product details.

Based on the clustering and the temporal operators, the Presentation Units and the navigation between them are calculated in a model transformation. The result is stored as an *Abstract UI Navigation Model* F which specifies the navigation between Presentation Units as a simple kind of state diagram and allows the developers to perform manual refinements, if desired. The calculation extends the algorithms used in [14, 13] to calculate Presentation Units based on the temporal operators. For instance, if multiple clusters are defined as concurrent (e.g., because they are connected by a concurrency operator) they must be accessible at the same time. Therefore, the UI must allow the user to switch between the resulting Presentation Units for example by providing links for navigating between them. AUI-Fragments are a special case in this calculation: An AUI-Fragment will never become an independent Presentation Unit, instead it is included into all concurrent AUI-Clusters. The details of the transformation are described in [9].

Concrete UI Elements: While the basic UWE models are platform-independent, UWE provides transformations to different target platforms like JSF. The concrete UI elements are hence generated by this transformation. Therefore, UWE defines a *Default Presentation Mapping* K which maps UWE UI elements to JSF elements. In addition, UWE provides a so called *Concrete Presentation Model* I containing mappings for individual UI elements. Both mappings can be adapted to customize either the default mapping or the mapping for individual UI elements.

Layout: The layout of presentation units is specified in the UWE Presentation Model H by assigning *UWE presentation groups*. By hierarchical nesting, it is possible to equip several web pages at once with an equal layout or to define templates for the general site layout.

Visual Appearance and Adornments: The generated JSF application L has to be extended with static, not generated artifacts like images, stylesheets and property files M which allow to easily customize the final visual appearance and adornments.

CONCLUSIONS AND OUTLOOK

Model-driven SPLs are highly efficient and have strong industrial relevance when developing a family of products. However, although SPL concepts have been applied to various types of interactive applications, like online shops, the specific challenges caused by the UI have been neglected so far. In this paper we provide the following contributions:

1. We demonstrate how SPL concepts can be applied to the UI (p.2/3).

2. We expose (and classify) general practical problems of applying SPL concepts to UIs (p.4). This analysis is based on reports from industry like [4].
3. We point out the problems of purely manual UI design within an SPL (p.4/5) based on reports from industry.
4. We develop a general model-driven SPL approach for UIs which overcomes the problems identified in 2) and 3) (p.5–9).

We demonstrate the feasibility of the approach by an example implementation based on UWE which is described in more detail in [9].

To our knowledge, UI development within SPLs has not been addressed by existing work so far (except our first work in [15]). In turn, a SPL approach differs from existing generative approaches for UIs (like MBUID) as the initial UI (for the whole SPL) is designed manually which prospectively results in higher quality UIs than purely generated UIs.

A more detailed evaluation – like empirical studies on the impact of the process proposed – is planned for future projects. For this, future work includes the development of more advanced tool support to create and manage the models and, ideally, to provide early visual feedback on the resulting UIs. In addition, SPLs in practice will use their own modeling languages instead of UWE as used in our implementation so the overall process has to be adapted accordingly. This requires mainly to adapt the final transformations from the AUI Navigation model to the final UI specification.

Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero, http://www.lero.ie/ and by Siemens Corporate Technology CT T CEE.

REFERENCES

1. V. Alves, P. M. Jr., L. Cole, P. Borba, and G. Ramalho. Extracting and evolving mobile games product lines. In *SPLC'05*, 2005.
2. P. Bell. A practical high volume software product line. In *OOPSLA'07*, 2007.
3. G. Botterweck. A model-driven approach to the engineering of multiple user interfaces. In *MoDELS Workshops*. 2006.
4. H. Brummermann, M. Keunecke, and K. Schmid. Variability issues in the evolution of information system ecosystems. In *VaMoS'11*, 2011.
5. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, M. Florins, and J. Vanderdonckt. Plasticity of user interfaces: A revised reference framework. In *TAMODIA'02*, 2002.
6. P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
7. K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE'05*, 2005.
8. J. Greenfield, K. Short, S. Cook, and S. Kent. *Software factories: assembling applications with patterns, models, frameworks and tools*. Wiley, 2004.
9. B. Hauptmann. Supporting derivation and customization of user interfaces in software product lines using the example of web applications. Master's thesis, Technische Universität München, October 2010. http://www4.in.tum.de/~hauptmab/pub/Hauptmann2010.pdf.
10. H. Hussmann, G. Meixner, and D. Zuehlke, editors. *Model-Driven Development of Advanced User Interfaces*. Springer, 2011.
11. N. Koch, A. Knapp, G. Zhang, and H. Baumeister. Uml-based web engineering: An approach based on standards. In *Web Engineering: Modelling and Implementing Web Applications*. Springer, 2007.
12. S. Q. Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, University of Waterloo, 2006 2006.
13. K. Luyten. *Dynamic User Interface Generation for Mobile and Embedded Systems with Model-Based User Interface Development*. Phd, Transnationale Universiteit Limburg, Belgium, 2004.
14. F. Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, London, UK, 2000.
15. A. Pleuss, G. Botterweck, and D. Dhungana. Integrating automated product derivation and individual user interface design. In *VAMOS'10*, 2010.
16. K. Pohl, G. Boeckle, and F. van der Linden. *Software Product Line Engineering*. Springer, 2005.
17. K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *VaMoS'11*. ACM, 2011.
18. P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature diagrams: A survey and a formal semantics. In *RE'06*, pages 136–145, 2006.
19. B. Shneiderman and C. Plaisant. *Designing the User Interface*. Addison Wesley, 4th edition, 2004.
20. Software Engineering Institute. SPL Hall of Fame. Web site, 2008. http://splc.net/fame.html.
21. T. Stahl and M. Voelter. *Model-driven software development : technology, engineering, management*. John Wiley, 2006.
22. P. A. Szekely. Retrospective and challenges for model-based interface development. In *DSV-IS*, 1996.
23. J.-P. Tolvanen and S. Kelly. Defining domain-specific modeling languages to automate product derivation: Collected experiences. In *SPLC'05*, 2005.
24. M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. In *SPLC'07*, 2007.
25. M. Voelter and E. Visser. Product line engineering using domain-specific languages. In *SPLC'11*, 2011.