Model-driven Development and Evolution of Customized User Interfaces

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

Stefan Wollny
University of Augsburg,
Germany
stefanwollny@googlemail.de

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

ABSTRACT

One of the main benefits of model-driven development of User Interfaces (UIs) is the increase in efficiency and consistency when developing multiple variants of a UI. For instance, multiple UIs for different target users, platforms, devices, or for whole product families can be generated from the same abstract models. However, purely generated UIs are not always sufficient as there is often need for customizing the individual UI variants, e.g., due to usability issues or specific customer requirements.

In this paper we present a model-driven approach for the development of *UI families* with systematic support for customizations. The approach supports customizing all aspects of a UI (UI elements, screens, navigation, etc.) and storing the customizations in specific models. As a result, a UI family can be evolved more efficiently because individual UI variants can be re-generated (after some changes have been applied to the family) without losing any previously made customizations. We demonstrate this by thirty highly customized real-world products from a commercial family of web information systems called *HIS-GX/QIS*.

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

While classic user interface engineering approaches address mainly the quality of the User Interface (UI), other engineering goals, e.g., efficiency, robustness, and maintainability require application of additional software engineering concepts. For instance, concepts from *Model-driven Development* (MDD) [24] have been applied to UIs in various approaches [13, 10]. In MDD, the system to be developed is specified in terms of abstract *models* (e.g., using UML or domain-specific languages), which are transformed stepwise by automated *model transformations* into more concrete models and finally into the implementation code (i.e., code generation).

However, when applying software engineering concepts to UIs, it is important to take the specific characteristics of UIs into account and to ensure that there are concrete benefits of applying these techniques. For instance, MDE supports efficiency and consistency when developing multiple variants of a UIs from the same abstract models. Many existing approaches address the development of UIs for multiple target platforms [4, 9]. Similar scenarios are development of multiple UIs for different users, devices, or contexts of use. Another important scenario is the development of multiple different variants of an application (*product family*) as addressed in software product line engineering [8, 20]. All these scenarios can be addressed with MDD by generating multiple variants of a UI from the same abstract model. To generalize from these concrete scenarios, we introduce in this paper the term *UI family* that refers to multiple different variants of a UI in general and show an MDD approach for UI family development.

Beyond variability in UI families, MDD can also help with *maintenance and evolution* [14], e.g., addressing the need to adapt software over time according to changing market and user requirements. Here, again MDD can help by providing the ability to perform changes on an abstract model level and then to just (re-)generate the new version of the UIs. We demonstrate this here as part of our evaluation.

At the same time, however, the special characteristics of UIs have to be considered. For instance, UIs developed by purely automated approaches are not always optimal in terms of quality [2]. In particular, specific users, customers, or target devices can raise unforeseen requirements on the UI that need to be addressed by manual customizations [18, 19]. Hence, there is a need to support manual customizations within a model-driven UI development process.

In this paper we aim to push the boundaries of model-driven UI development further by tackling these issues: We aim to provide a model-driven development approach for whole families of similar UIs. For this, we introduce a notion of

UI families that generalizes from specific aspects, e.g., multi-platform or multi-user UIs. To support customized high-quality UIs, we introduce a novel concept to integrate manual UI customization into the MDD process. Inspired by stylesheets for HTML UIs, the manual customizations can be stored in separate, modular models. In this way, customizations can be added, combined and reused over multiple products and multiple product versions. In contrast to stylesheets, which cover mainly the visual appearance, our models support customizing *all* aspects of the UI including navigation, layout or the decomposition of the UI into screens.

We evaluate the approach using a commercial web information system for university management *HIS-GX/QIS*[1]. This is a product family where each product (an individual instance of the software for a particular university) is highly customized according to the individual needs of the university – often by third parties or the university itself. We have performed a detailed analysis of the UI customizations in this UI family [19] showing that customizations are spread all over the UI and cover each aspect of the UI. Our evaluation shows for thirty real-world product instances, that all these individually customized UIs can be developed from a single abstract model using our approach. In particular, we demonstrate the benefits of MDD for software evolution: Changes on the UI family, like adding or removing UI elements, need to be performed only on the single abstract model. By regenerating the UIs, the changes can be automatically propagated to all products without loosing any previously specified customizations. We demonstrate this again for all thirty real-world product instances.

The remainder of the paper is structured as follows: The next section introduces the concept of *UI families* that generalizes from multi-target UI development (as in existing model-driven UI development approaches) and product family development (as in software product lines). Next, we introduce our concept for integrating manual customizations into an MDD process for UIs. Subsequently, we show the concrete realization of our approach followed by a comprehensive evaluation using HIS-GX/QIS. Finally, we discuss related work and present conclusions and an outlook.

DEVELOPMENT OF MULTIPLE UIS – UI FAMILIES

Development of different UIs depending on the context of use has become an important issue in UI engineering research, e.g., in the context of ubiquitous computing [15]. For instance, it can be required to vary a UI according to different target devices, user groups, or usage context. The area of *Model-based UI Development* (*MBUID*)provides modeling concepts that can be used to address these challenges. On the other hand, development of families of systems is also an important issue in software engineering as many software products have to be provided in different variations. This is addressed by the area of *Software Product Line Engineering* (*SPLE*). As both concepts require the development of multiple related UIs and can be supported by MDD, we integrate these concepts by introducing *UI families* as a more general term for multiple related UIs.

[1] http://www.his.de/english/organisation

The following sections first introduce MBUID, then SPLE, and finally our concept of UI families.

MBUID Concepts

The area of Model-based UI Development [25] addresses model-based (including model-driven) development of UIs[2]. There have been various approaches using models for different purposes but one of the most important goals has become the development of UIs for multiple contexts of use. For instance, UIs for multiple target platforms, devices, users, or situations are developed from the same abstract UI models (often called multi-platform, multi-user, etc. UIs).

Figure 1a shows the common concepts for such approaches based on [25, 6]. The left-hand side shows the different models used to specify a UI in MBUID. The right-hand side shows the context of use that influences the UI development. In the following we first explain the models on the left-hand side.

The most abstract models are the *Domain Model* and the *Task Model*. The *Domain Model* is a conventional model used 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 and temporal operators between them (e.g., if two tasks are performed sequentially or concurrently). A concrete approach for task models is, e.g., CTT [16].

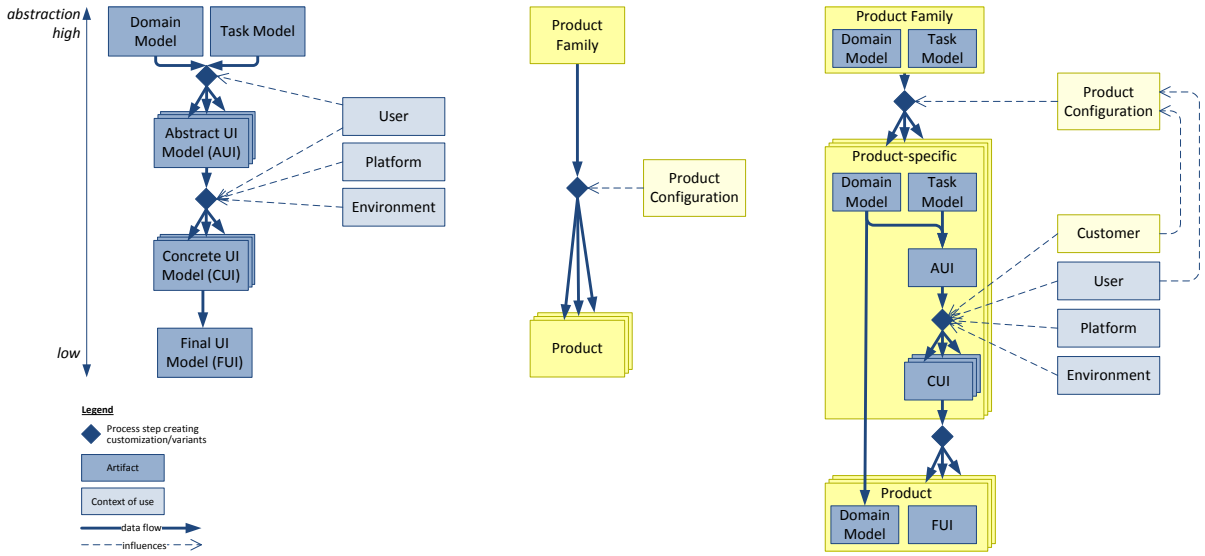
An *Abstract UI Model* describes the UI in terms of abstract UI elements that are platform- and often even modality-independent abstractions of UI widgets, like *input* element, *output* element, *selection* element, or *action* element (abstraction of a button). Each abstract UI element realizes tasks 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, e.g., 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 and concretizes 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 is referred to as the *Final UI*. It can either be a model that represents the final implementation code (potentially interpreted at runtime) or the final code itself.

The context of use for a UI can be defined by *user*, *platform*, and *environment* (models). A UI can be adapted to the context of use either at development time (e.g., developing multiple UIs for different target platforms) or at runtime (i.e., context-adaptive UIs). Here we focus on the former case as runtime adaptation is outside the scope of this paper. The (type of)

[2] "Model-based" usually refers to usage of models in a general sense while "model-driven" more specifically refers to usage of models for (semi-)automated code generation.



(a) MBUID: Multi-UI. (b) SPL: Multi-product (c) UI Family: Multi-product/-UI

Figure 1. Families of related UIs and influencing factors.

user can influence the AUI as, e.g., some tasks are not intended for all users. It can also influence the CUI development as, e.g., an elderly user might require like larger size of UI elements. The platform and environment influence mainly the CUI, like selected widget types and layout.

Basically, the steps from task and domain model to FUI can be performed manually or in an automated way, depending on the particular MBUID approach. In MDD, these steps are performed automatically by model transformations. The context models are then used as additional input for the model transformations, e.g., by parameterizing and/or selecting between different transformation rules.

SPLE Concepts

The area of *Software Product Line Engineering* (*SPLE*) [8, 20] addresses the development of a whole family of similar software products from a common set of shared software assets. Software Product Lines (SPL) have been applied by many companies in various industry domains [23]. A common example is online shop software: As different online shops have large *commonalities* in their functionality they can potentially 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 commonalities and variability in a SPL are usually specified in terms of a variability model. For instance, a variability model for an online shop SPL might specify that each online shop must support at least one payment method (commonality) which can be credit card payment, purchase order, or cash on delivery (variability). A concrete product is then defined by a *product configuration*, i.e., a selection of variants. For instance, there might be an online shop that supports credit card payment only, while another one supports all three payment methods.

Each variant in the product line is associated with an implementation. For instance, each of the payment methods might be associated with a software component that implements the payment method. Using model-driven techniques, it is then possible to automatically generate a product implementation by automated composition of the implementation assets that are associated with selected variants.

Figure 1b summarizes the main concepts: on SPL level, the whole *product family* is specified, i.e., the superset of all implementation assets for all potential variants. A *product configuration* defines which variants have been selected for this product. Different product configurations result in multiple products.

As shown in [18], SPL concepts can also be applied to the UI of an application: On product family level, all UI elements for all potential product variants are specified. A concrete product contains only those UI elements for those variants that have been selected for this product. For instance, UI elements for input of credit card information are only included into the UI if credit card payment has been selected in the product configuration.

UI Families

The MBUID and SPLE concepts introduced in the previous sections can both be used for model-driven development of multiple UIs from the same abstract models. The difference between them is that SPLE focuses on the whole application and mainly *functional* differences (e.g., support of credit card payment or not) while MBUID focuses on the UI and mainly *non-functional* differences (e.g., usability when using a particular device). However, one can easily imagine that both concepts in combination can be required in practice: For instance, an online shop cannot only vary in its functionality (SPLE) but also has to vary according to the context of use (MBUID), e.g., provide support for multiple target platforms.

From the viewpoint of a general MDD process there is no need to exclude one of these two possibilities, hence, we combine both concepts. For this we introduce the term *UI family* as a general means to refer to a set of related UIs that vary according to SPLE concepts and/or the context of use.

Figure 1c shows the combined concepts: The family model specifies the UI (and other parts of the application like the domain model) for the whole family. The UI is defined as task model, containing the superset of all tasks supported in the product family. The product derivation influences the functionality of the product (by selecting variants) which results in a selection of tasks to be supported by a concrete product, i.e. a product-specific task model. Hence, in contrast to Figure 1a, adapting the available tasks to different users can be handled by different product configurations as well[3]. Based on the product-specific task model, the common MBUID concepts can be applied like generating multiple UIs for the specific product based on different contexts of use. In summary, this results in multiple products (e.g., different online shops), each with (potentially) multiple UIs (e.g., each shop has multiple UIs for different target devices).

In an SPL scenario, the context of use has to be extended by an additional stakeholder: In addition to the end user there is now also the *customer* who owns a concrete product. Often, this is not the end user itself. For instance, in case of online shops, the customer is the shop owner for which the particular shop has been built for. Often the customer defines not only the product configuration but also influences the UI design. For instance, each UI of an online shop can be strongly influenced by the branding, the business goals, and marketing strategy of the shop owner.

UI CUSTOMIZATION WITHIN MDD

In the previous chapter we have shown a general model-driven process to develop UI families from abstract models. This process can be fully automated, starting from a task model and optional models for the context of use. However, in practice, UIs sometimes require manual customization for two reasons [18]: First, for usability reasons. For instance, automatic distribution of UI elements onto presentation units (e.g., screens) is sometimes difficult and can lead to overfull or too empty presentation units [5]. Second, there can be very specific customer requirements beyond generic rules. For instance, [3] reports that providers of online shops often ask for very specific UI customizations that cannot be foreseen. Hence, we aim to support manual UI customization within our approach.

To support manual UI customization in an efficient way within a model-driven approach we aim to address the following requirements:

- The manual customization is optional only; i.e., there is a default automated UI provided that can be used directly if there is no need for customizations.
- It must be possible to store customizations so that they can be reused, for instance, when re-generating the application during software evolution.
- Customizations should be stored in a modularized way to support easier reuse. For instance, it should be possible to apply a specific customized layout to multiple UIs without modifying their other properties.
- The specification of customizations should be as efficient as possible. For instance, it should not only be possible to customize single elements but also multiple elements that are, e.g., of the same type.

UI Aspect	*Customization Specification*
Tasks and Temporal Operators	Product Derivation
Abstract UI (AUI) Elements	Product Derivation
Relationships to Domain Model	Product Derivation
Presentation Units	AUI Model
Navigation	AUI Model
Layout	AUI Model
Concrete UI (CUI) Elements	AUI to CUI Transformation
Visual Appearance & Adornments	AUI to CUI Transformation OR Stylesheets

Table 1. UI aspects and development step where to specify customizations.

A successful concept are CSS stylesheets as commonly used for HTML UIs. Stylesheets fulfill the requirements above as they can be added, removed, and combined in a flexible way without modifying the HTML code itself. However, stylesheets mainly influence a UI's *visual* appearance while other aspects of the UI cannot be modified, e.g., the distribution of UI elements onto presentation units. Hence, we propose to fill this gap by supporting additional models that can customize *all* aspects of the UI and can be added, removed, and combined within the MDD process similar to stylesheets.

In [18] we classified the properties of a UI into different *UI aspects*, e.g., navigation, layout, visual appearance, etc. These aspects are derived from MBUID concepts and shown in the left column of Table 1. In [19] we perform a empirical case study about which of these aspects are customized in practice using the commercial product family of web applications *HIS QIS/GX* (which we also use for the evaluation in this paper). It shows that customizations for *all* these UI aspects can be found in practice and, hence, need to be supported by a generic model-driven process.[4]

The right column of Table 1 shows for each aspect at which step in a common MBUID process it can be customized. The task model specifies the basic functionality of the UI (*tasks and temporal operators*). According to the general framework in Figure 1 this information can be modified during product derivation (as modifying functionality is considered as creating a different product). The same holds for the *AUI elements* and their *relationships to the domain model* as they are directly related to tasks. The distribution of UI elements onto *presentation units*, the *navigation* between presentation units, and their *layout* are defined on the AUI level and, hence, also customized on this level. *CUI elements* and their *visual*

[3]This can be different for runtime adaptation which is outside the scope of this paper.

[4]We have updated Table 1 accordingly compared to its initial version in [18].

appearance are defined on CUI level. However, customizations on the CUI level often apply to multiple elements. For instance, *all* buttons should have a certain design or *all* selections with less than three choices should be represented by a radio button. Hence, such customizations are best supported by adapting the transformation from the AUI to the CUI. Common model transformation languages like ATL[5] or ETL[6] support defining conditional mapping rules for fine-grained customizations.

In the next section we show a concrete and generic realization of a MDD approach that supports all customizations according to Table 1.

DETAILED APPROACH

This section presents a concrete realization of the concepts discussed so far, i.e., a MDD process for UI families with support for customization. The approach extends and generalizes the process in [17]. Further details on the proposed solution and tool support can be found in [26]. We focus here on manual customizations to adapt the UIs to the context of use; the approach might further be augmented with heuristics [4, 21] to improve the results of the automated transformations but this is not further discussed here. In the following we explain the process as shown in Figure 2.

Product Derivation

The process starts with product derivation (A to B in Figure 2) as in SPLE: First, the models on the level of the whole *product family* A are created. They specify a superset of all potential product variants, i.e., all model elements that can appear in any of the products (except additional customizations). We focus here on the UI part of the products, while all other models (e.g., domain models and other models to specify an application) are summarized as *application logic* and not discussed further. Regarding the UI, the tasks and temporal operators, the AUI elements, and their relationships to the domain model are customized during product derivation (see Table 1). However, as AUI elements are a more concrete representation of tasks, the tasks can be omitted here and product derivation can be performed directly on the AUI elements. However, we still require the temporal operators from the task model for the further steps of the process (e.g., calculating the presentation units for the specific product). Hence, we use a specific AUI model for the product derivation that contains AUI elements, their relationships to the domain model, and temporal operators.

Figure 3 shows a sample AUI model for an example product family of online shop applications. Analogous to task models like CTT [16], the model consists of a tree structure; siblings are connected by temporal operators. The leaf nodes are AUI elements, like *input*, *output*, or *selection*. The non-leaf nodes are UI containers. An exception are selection elements which can be used as non-leaf nodes to represent selections of complex objects. For instance in the example, articleSelection enables to select one or more articles. It is also possible to

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

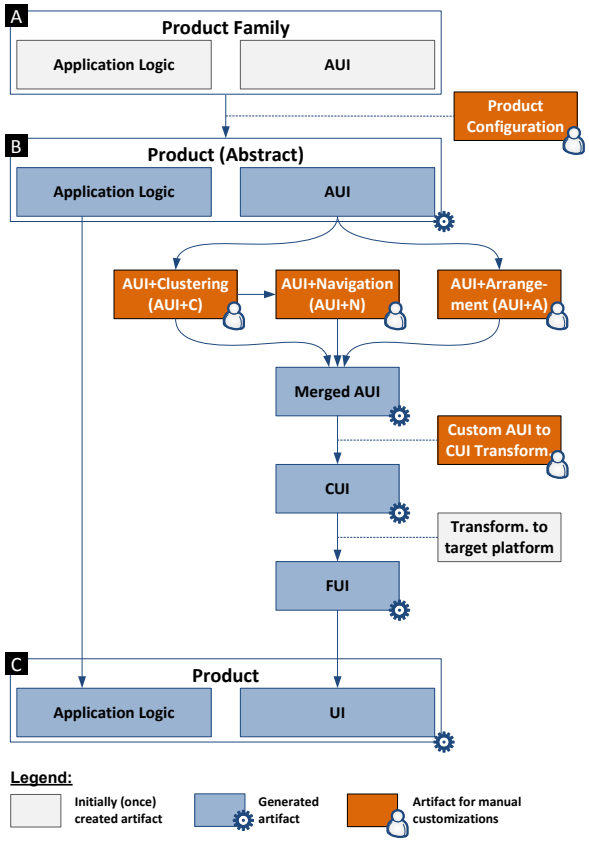


Figure 2. Detailed model-driven development process with customizations.

specify 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 articleSelection). In addition, it is possible to reuse a container multiple times within the model. For instance, shippingAddress and billingAddress are both copies of a container *Address* (which is defined elsewhere). The relationships to the domain model are stored with the AUI elements but not shown in the diagram.

Once the product family model is defined, concrete products are derived by defining *product configurations*. A product configuration specifies which elements are present in a concrete product. For instance, if a concrete online shop does not support credit card payment, all corresponding AUI elements (here the UI container creditCard) are deleted from the model (see [18] for details and how to ensure consistency). The result of this process step is a product-specific model B that includes a product-specific AUI. An example is shown later in Figure 4, where, e.g., the UI container credit card was removed. The next process steps address customizing the product-specific AUI (according to Table 1).

AUI+Clustering Model

The *AUI+Clustering* model (AUI+C) is used to customize the decomposition of the UI into presentation units. This is specified by clustering AUI elements. A cluster contains AUI elements and can either be an *AUICluster* an *AUIFragment*. An AUICluster represents a presentation unit. An AUIFragment

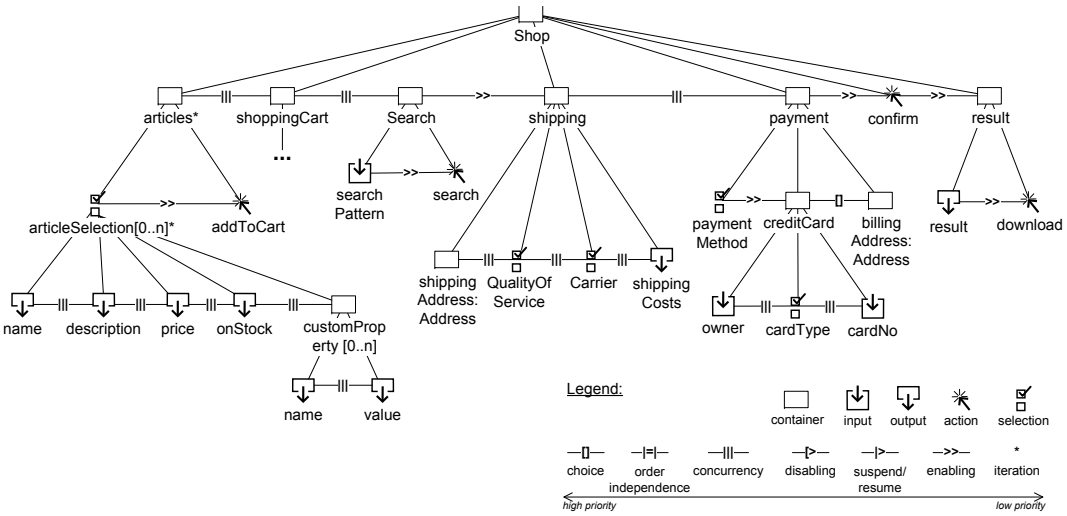
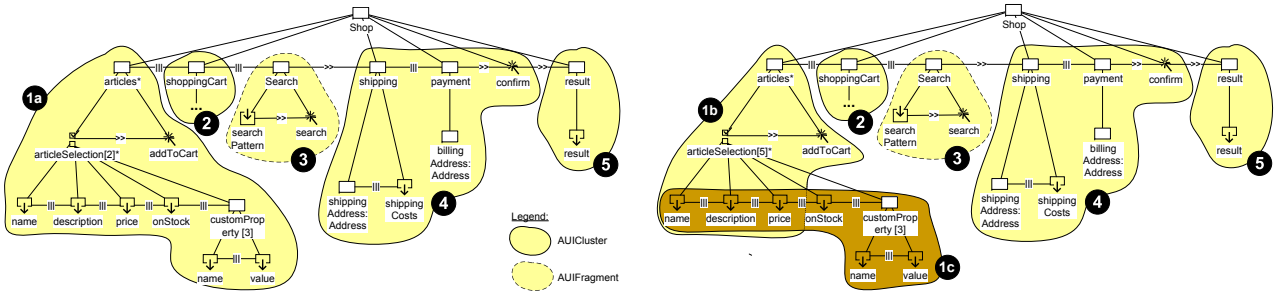


Figure 3. Example AUI for a product family of online shops.



(a) Single cluster for article list and article details

(b) Article details in a separate cluster.

Figure 4. Two alternative AUI+C models for an example product-specific UI.

represents a container which is embedded into multiple other presentation units (which is calculated based on the temporal operators). For instance, a search bar might be embedded into multiple presentation units and, hence, clustered into an AUIFragment.

Figure 4 gives an example for two alternative clusterings of a product-specific AUI: The cluster ❸ is an AUIFragment (here the search bar) and will hence be embedded into other presentation units. All other clusters are AUIClusters which will result in presentation units. The two alternatives differ in the clustering of articles: in Figure 4a, article information is displayed on a single presentation unit ❶ₐ while in Figure 4b one presentation unit is used for the list of articles ❶ᵦ while article details are presented in a separate presentation unit ❶꜀. Customization of the clustering is an essential step when adapting to various platforms, e.g., due to varying display sizes.

AUI+Navigation Model

The *AUI+Navigation* model (*AUI+N*) specifies the navigation between presentation units. It is automatically calculated based on the clusterings (from the AUI+C) and the temporal operators (from the AUI) using an algorithm presented in [12]. The resulting AUI+N model contains the *navigation links* between the presentation units and *fragment inclusion dependencies*, which specify the embedding of AUIFragments into presentation units.

The AUI+N model can be customized by adding or removing links or dependencies (while presentation units and fragments itself are customized in the AUI+C model only).

AUI+Arrangement

The *AUI+Arrangement* model (*AUI+A*) specifies the abstract layout with respect to how AUI elements are arranged within a presentation unit (the size of UI elements is defined later on CUI level). We use the concepts from [11]: The abstract layout is defined by an *order* of elements and an *orientation* constraint. The orientation constraint defines the relative placement to the previous element and is either "*horizontal-to*" or "*vertical-to*".

To keep the customization models (AUI+C, AUI+N, and AUI+A) modular, they do not contain AUI elements directly but only references to the AUI elements in the AUI model. In this way, they can be edited independently from the AUI model and each other. It is also possible to reuse them by applying them to other product-specific AUI models.

Merged AUI and Transformation to CUI

The *Merged AUI* model integrates the information from the AUI+C, AUI+N, and AUI+A models into a single model.

This is not necessary in a technical sense but considered as helpful for the developer to get an overview on the resulting AUI. It is used as starting point for the AUI to CUI transformation.

The *AUI to CUI transformation* maps the presentation units and AUI elements to CUI elements (e.g., a selection element becomes a list box). It also sets the properties of CUI elements like size, default values, or a reference to a style definition in a stylesheet. The transformation is currently specified using the declarative model transformation language *Epsilon Transformation Language (ETL)*[7]. The advantage of using a transformation language instead of a simple mapping model is the support for defining complex generic rules and conditions. For instance, it can be specified that selections should be mapped differently depending on certain conditions. The transformation language also allows to manipulate the mapping of individual elements (by specifying a condition over a name). The transformation is customized by adding rules. Transformation languages like ETL also support mechanisms like inheritance and modularization which eases structuring and reuse of the transformation rules.

Final Code Generation and Tool Support

Finally, the final UI code (*FUI* and resulting final product C) is generated. Here, the developer has to select a transformation to the desired target platform. Currently we have implemented a transformation which generates HTML5 code. Feasibility of similar transformations for other platforms has been shown, e.g., in [4].

The whole process has been implemented and tool-supported based on the *Eclipse Modeling Framework* (*EMF*)[8]. All models are specified as metamodels compliant to *Ecore*, an Eclipse-based implementation of the *MOF* standard[9]. All model transformation are implemented using the model transformation language ETL.

We also implemented a first version of a modeling tool for all models (based on Eclipse). Currently, the tool only provides tree-oriented representations of the models (no graph-oriented representation yet), but eases the creation and management of the models and guides the development process. For instance, it provides a screen showing an overview over all models and supports to run the model transformations. Figure 5 shows a screenshot. The main screen consists of multiple tabs, one for each model. The screenshot shows the tab for the AUI+C model which supports creation and management of AUIClusters and AUIFragments and the assignment of AUI elements to them.

EVALUATION

We have evaluated our approach using the UIs from a commercial web information system *HIS-GX/QIS* by the company *HIS*[10]. It is a system for the management of universities currently used by 145 German universities. We focus on a specific part of the application, the web-based *online application*, where prospective students can apply for a course of study. The system guides the user through various forms to enter detailed information like personal data, previous education, and decisions about the course of study to apply for. It is a purely HTML-based UI that consists by default of 13 screens with 111 UI elements. By "UI element" we refer to interactive elements like input fields or buttons while static content like text and graphic is not further considered in the study.

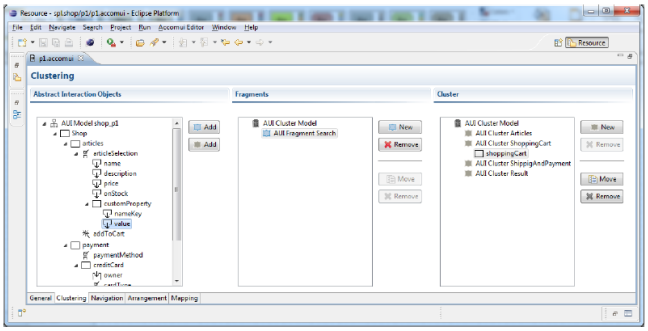


Figure 5. Screenshot from our modeling tool showing an AUI+C model.

HIS-GX/QIS can be considered as a family of products. There is a basic product version that contains all common functionality for applying at a university. However, all product instances which are in use online[11] are customized to a wide variety of usage contexts: On the one hand there are different types of universities, from full universities to small art schools. On the other hand, universities in Germany are subject to the different laws by the 16 federal states in Germany. For this reasons, HIS supports the universities in customizing their products, including the UI. Customizations are performed by HIS itself, by the universities, and sometimes also by third party companies.

The study in [19] analyses the UI customizations in detail based on 30 products (running at 30 different universities). It turns out that all products are highly customized. Moreover, customizations are spread over most parts of the UI (77.5 % of the UI elements) and over *all* UI aspects (those listed in Table 1). This includes both, functional customizations like in product derivation (adding and removing UI elements) but also UI-specific customizations (see [19]). This includes complex customizations like changing types of UI elements (e.g., from list box to radio buttons), changing the order of UI elements, merging and splitting screens, or changing the navigation path.

To evaluate our approach using real-world UIs, we extracted the HTML UIs from the existing HIS-GX/QIS products using an extraction tool from our previous study [19]. Based on the extracted data, we aim to address the following research questions:

[7] http://www.eclipse.org/gmt/epsilon/doc/etl/
[8] http://www.eclipse.org/modeling/emf/
[9] http://www.omg.org/mof/
[10] http://www.his.de/english/organisation

[11] See, for instance, https://qisweb.hispro.de/fab/rds?stg=n&state=wimma&imma=einl and https://sbservice.tu-chemnitz.de/qisserver2/rds?state=wimma&stg=f&imma=einl for two examples.

1. *RQ1 – Development of Customized UIs:* Can all of these customized UIs be generated using our approach?
2. *RQ2 – Evolution of Customized UIs:* Can all of these customized UIs be modified by making changes on the abstract product family model without loosing any customizations?

The next section describes the general setup for the study, followed by two sections on these two research questions.

Evaluation Approach

Figure 6 shows the approach for the evaluation. Starting point are the HTML UIs ❶ from the HIS-GX/QIS online application system running at thirty universities. We created a tool based on *Selenium*[12], a framework for browser automation, to extract all relevant information about the UIs. Our tool traverses the HTML forms at a given URL and extracts ❷ the desired data from the HTML and CSS code into data tables. For each screen (presentation unit) we extract its name and its position in the navigation path and its contained UI elements. For each UI element we identified its name and type and all other properties like size, style or default values. Extracting the layout is more difficult as it can be influenced by multiple factors like the stylesheets and the screen size. Hence, we divide the screen into a virtual grid and defined the relative position of each element by a row and a column value. Static content like text and images is not extracted but just represented by a placeholder. Our tool does not interpret JavaScript. This does not limit the results as JavaScript is *not* used in HIS-GX/QIS due to German accessibility laws for universities. A little restriction is that we cannot ensure that there are dynamic branches in the navigation path depending on the input; our tool just follows the sequential standard navigation path that results from the default values which we use as input to the forms. As a result we get a data table for each product containing the extracted UI information ❸.

The next step is to automatically create the models for our MDD process from the extracted UI specifications ❹. All identified UI elements are mapped to a AUI element in the product family model ❺. As all UI elements in HIS-GX/QIS have a unique id that is consistent over all products, we can identify identical UI elements within different products. The resulting product family AUI model is the superset of all different UI elements extracted. Note that the generated AUI model is less structured than a manually created one as all UI elements are just placed as children of the root node and connected by the *concurrency* operator which is the most generic temporal operator, but this is sufficient for our purpose.

In addition, for each product we generate the product-specific customization models ❻ from the extracted data: For each product we store the information which AUI elements are present in the product (product configuration). We generate a product-specific AUI+C model that specifies the presentation units and the AUI elements they contain. We generate an AUI+N model to define the product-specific navigation. The navigation in HIS-GX/QIS is sequential in general, but the order of screens is sometimes customized so this is stored in the

[12] http://seleniumhq.org/

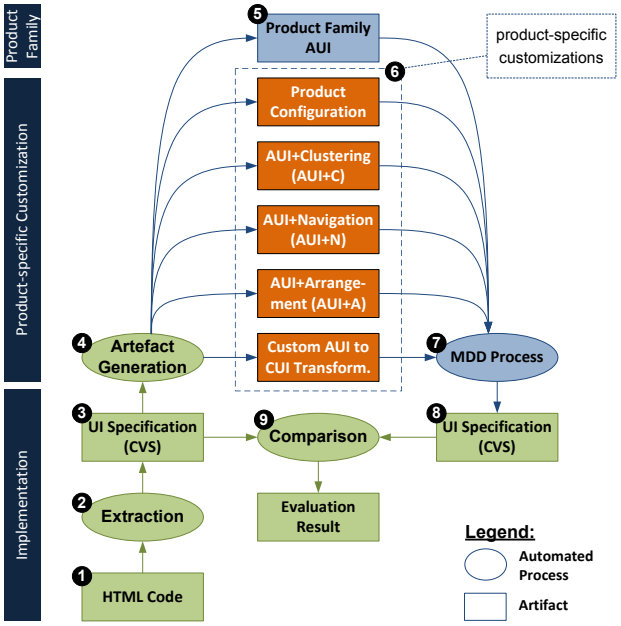


Figure 6. Approach for the evaluation.

AUI+N. We generate an AUI+A model that defines the order and relative position of each UI element based on the virtual grid we used for the layout extraction. Finally, we generate an AUI to CUI transformation for each product which is a simple mapping for each AUI element to a CUI element assigning the product-specific properties to the CUI element (the extracted properties like size, style or default value). Again, the generated transformation has a somewhat trivial structure and is less readable than more generic transformation rules created by a human developer. Nevertheless the result is sufficient for our study. Altogether the artifact generation step results in a single product family AUI model ❺ and a set of product-specific customization models ❻ for each product.

RQ1: Development of Customized UIs

The result of the previous steps are a AUI model for the HIS-GX/QIS product family and a set of customization models for each product. We now aim to show that this information is correct and sufficient to generate the product-specific UIs. Hence, we perform the UI generation ❼ based on the extracted models using our MDD process as shown in Figure 2. We can use our HTML code generator to generate real executable HTML implementations from the models. However, based on the HTML code it is not possible to automatically compare the generated UIs with the initial real-world UIs as the generated code differs syntactically from the manually written code in the real-world applications. Hence, we generate the UIs in the UI specification format that we used for the extraction ❽ as this enables automated comparison. This is sufficient for our evaluation assuming that the extracted UI Specification is an exact representation of the HTML UIs.

The last step is to compare the generated UI specifications with the extracted ones ❾. This is performed using a file comparison tool. It turns out that there are no differences between them. Consequently, we have shown that our approach

is sufficient to generate a family of real-world customized UIs based on a single abstract product family model.

RQ2: Evolution of Customized UIs

In a second part of our evaluation we study the support for evolution. We aim to perform changes on the product family model (e.g., adding or removing UI elements) and to re-generate the product-specific UIs. We want to show that the changes are reflected in the re-generated UIs while all product-specific UI customizations remain. We aim to perform three evolution steps in which different modifications should be performed on the whole product family. For this, we specified fictitious evolution scenarios:

Scenario 1: Due to a new law, it is no longer allowed to ask for the date of birth and gender of a student. Hence, the corresponding UI elements are removed. Instead, for administrate reasons a social insurance number should be requested now. So a corresponding text input field should be added on the same screen where users put their name.

Scenario 2: The universities now aim to support multiple alternative email addresses. So new input fields for a second and a third email address should be added on the same screen where the primary email is requested. In addition, the previous input field for email has to be renamed to primary email. Moreover, the universities now want to support social media like Facebook, Twitter, and Skype. So a new UI container should be added containing corresponding input fields.

Scenario 3: The law from scenario 1 is withdrawn and date of birth and gender input fields should be restored on the same screen where they were located before. Moreover, two output fields used to show some extra information should be added to the same screen.

The scenarios should be realized in three evolution steps. For each step, the corresponding changes are performed on the product family AUI (❺ in Figure 6). By default, our MDD process puts new AUI elements (which have not been manually clustered into a presentation unit) into the same presentation unit as their previous sibling in the AUI model. This means, e.g., in scenario 1, that the new AUI element social insurance number must be inserted after the AUI element name in the product family AUI to be automatically put into the same presentation unit.

Once the product family AUI has been modified according to a scenario, the corresponding modified UIs are automatically generated ❼ without any further manual intervention. Like in the first part of the evaluation, we generate the UI in our UI specification format ❽ to enable easy automated comparison of the updated UIs with the original extracted one ❾. In case of evolution, the modified UIs should be identical with the initial ones except the modifications exactly as specified in the scenarios.

As a result all thirty generated product-specific UIs showed the expected structure for all three evolution steps: they were exactly identical to the original ones (including all product-specific customizations) except the modifications specified in the scenarios. Also, all expected modifications were found in all the generated UIs.

As an additional test, we also generated HTML UIs for the modified UIs. This worked as well and the updated UIs contained all changes defined on the product family AUI. Of course, the generated HTML UIs look different as we did not store static content (text/graphics) and the original stylesheets during the extraction. However, this could easily be added (by adding model elements for static content) if desired. This means that our approach enables to modify a whole family of tens (or hundreds) of UIs by just modifying a single product family AUI.

Discussion

The evaluation has shown some clear benefits of MDD for UIs: Modifications need to be performed just once on the abstract product family model and can be automatically propagated to tens or hundreds of customized real-world UI without the need for any manual intervention. But there are of course also some limitations and open issues of our approach.

First, the evaluation is currently restricted to purely HTML-based UIs. This was helpful for the automated extraction of the real-world UIs as reverse engineering of dynamic JavaScript-based UIs is difficult. In general, the specification of dynamic behaviour in our MDD approach is currently restricted to temporal operators and the corresponding navigation between presentation units. In the future we plan to extend our approach towards more dynamic UIs like Java-based UIs and to gain further experience by performing case studies with other UI families.

Another important aspect is the usability and efficiency of the proposed models. This has not been addressed by the evaluation yet. As described in our detailed approach, we have implemented a basic Eclipse-based tool to create, manage, and apply the models and the model transformations. While a tool like this seems to be usable for developers with background in modeling, it seems not sufficient for UI designers who are used to visual design tools. Hence, in the future there is need for visual tool support that hides the models in the background and provides direct visual feedback about the customizations specified (e.g., an immediate preview). Related to this, there is also a need to get more experience with developers in practice to evaluate whether the customization models are efficient to specify and use, and to which extent they can be reused across different products (e.g., which granularity is most appropriate for reuse). The models and the corresponding tool support have then to be refined accordingly.

CONCLUSIONS AND OUTLOOK

In this paper, we have presented an MDD approach for families of customized UIs. We have provided a generic notion of UI families, have presented tool support, and have evaluated our approach by application to a commercial real-world UI family. In particular, we have demonstrated the benefits for evolution of UI families: Modifications need to be performed *just once* on the abstract product family model and can be automatically propagated to numerous individually customized products without loss of customizations.

The presented approach continues and extends our previous work in [17, 18]. To the best of our knowledge, supporting families of customized UIs has not been addressed in related work yet. Also practical evaluation of MDD for UIs is addressed only in a few publications: [7, 1, 2, 21] evaluate the quality of UIs developed using MDD. While they show that it is possible to generate acceptable results there is an agreement that manual customization is often still desirable. An important approach besides our work to support such manual customization is [22]: It presents visual tool support to specify presentation units and their content within a MDD process. However, other UI aspects like the navigation are not considered yet in this approach.

In future work, we aim to address the issues discussed in the previous sections: enhanced visual tool support (e.g., like in [22]), more dynamic UIs beyond HTML including the specification of UI behavior, and management and reuse of customization models.

Acknowledgments

This work was supported, in part, by Science Foundation Ireland grant 10/CE/I1855 to Lero – the Irish Software Engineering Research Centre http://www.lero.ie/.

REFERENCES

1. S. Abrahão, E. Iborra, and J. Vanderdonckt. Usability evaluation of user interfaces generated with a model-driven architecture tool. In *Maturing Usability*, pages 3–32. Springer, 2008.
2. N. Aquino, J. Vanderdonckt, N. Condori-Fernández, O. Dieste, and O. Pastor. Usability evaluation of multi-device/platform user interfaces generated by model-driven engineering. In *ESEM 2010*, pages 30:1–30:10. ACM, 2010.
3. P. Bell. A practical high volume software product line. In *OOPSLA'07*, pages 994–1003. ACM, 2007.
4. G. Botterweck. A model-driven approach to the engineering of multiple user interfaces. In *MoDELS Workshops*, pages 106–115. Springer, 2006.
5. H. Brummermann, M. Keunecke, and K. Schmid. Variability issues in the evolution of information system ecosystems. In *VaMoS'11*, pages 159–164, 2011.
6. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. A unifying reference framework for multi-target user interfaces. *Interacting with Computers*, 15(3):289–308, 2003.
7. C. Chesta, F. Paternò, and C. Santoro. Methods and tools for designing and developing usable multi-platform interactive applications. *PsychNology Journal*, 2(1):123 – 139, 2004.
8. P. Clements and L. M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.
9. B. Collignon, J. Vanderdonckt, and G. Calvary. Model-driven engineering of multi-target plastic user interfaces. In *ICAS'08*, pages 7–14. IEEE, 2008.
10. J. V. den Bergh, S. Sauer, K. Breiner, H. Hussmann, G. Meixner, and A. Pleuss, editors. *Proceedings of the 5th International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI 2010): Bridging between User Experience and UI Engineering*, volume 617. CEUR Proceedings, 2010.
11. S. Feuerstack, M. Blumendorf, V. Schwartze, and S. Albayrak. Model-based layout generation. In *AVI '08*, pages 217–224. ACM, 2008.
12. 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, Germany, 2010.
13. H. Hussmann, G. Meixner, and D. Zuehlke, editors. *Model-Driven Development of Advanced User Interfaces*. Springer, 2011.
14. T. Mens and S. Demeyer, editors. *Software Evolution*. Springer, 2008.
15. B. A. Myers, S. E. Hudson, and R. F. Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000.
16. F. Paternò. *Model-Based Design and Evaluation of Interactive Applications*. Springer, 2000.
17. A. Pleuss, G. Botterweck, and D. Dhungana. Integrating automated product derivation and individual user interface design. In *VAMOS'10*, pages 69–76, 2010.
18. A. Pleuss, B. Hauptmann, D. Dhungana, and G. Botterweck. User interface engineering for software product lines – the dilemma between automation and usability. In *EICS 2012*, pages 25–34, 2012.
19. A. Pleuss, B. Hauptmann, M. Keunecke, and G. Botterweck. A case study on variability in user interfaces. In *SPLC 2012*, pages 6–10. ACM, 2012.
20. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering*. Springer, 2005.
21. D. Raneburger, R. Popp, H. Kaindl, J. Falb, and D. Ertl. Automated generation of device-specific WIMP UIs: weaving of structural and behavioral models. In *EICS 2011*, pages 41–46. ACM, 2011.
22. A. Schramm, A. Preußner, M. Heinrich, and L. Vogel. Rapid UI development for enterprise applications: combining manual and model-driven techniques. In *MODELS'10*, pages 271–285. Springer, 2010.
23. Software Engineering Institute. SPL Hall of Fame, 2008. http://splc.net/fame.html.
24. T. Stahl and M. Voelter. *Model-driven software development : technology, engineering, management*. John Wiley, 2006.
25. P. A. Szekely. Retrospective and challenges for model-based interface development. In *DSV-IS*, pages 1–27. Springer, 1996.
26. S. Wollny. Model-driven support for user interface evolution in software product lines. Master's thesis, University of Augsburg, Germany, 2013.