Integrating Automated Product Derivation and Individual User Interface Design

Andreas Pleuss, Goetz Botterweck, Deepak Dhungana
Lero – The Irish Software Engineering Research Centre
University of Limerick, Limerick, Ireland
{andreas.pleuss, goetz-botterweck, deepak.dhungana}@lero.ie

Abstract—Software Product Lines, in conjunction with model-driven product derivation, are successful examples for extensive automation and reuse in software development. However, often each single product requires an individual, tailored user interface of its own to achieve the desired usability. Moreover, in some cases (e.g., online shops, games) it is even mandatory that each product has an individual, unique user interface of its own. Usually, this results in manual user interface design independent from the model-driven product derivation. Consequently, each product configuration has to be mapped manually to a corresponding user interface which can become a tedious and error-prone task for large and complex product lines. This paper addresses this problem by integrating concepts from SPL, product derivation and Model-based User Interface Development. This facilitates both (1) a systematic and semi-automated creation of user interfaces during product derivation while (2) still supporting individual, creative design.

I. INTRODUCTION

In Model-Driven Engineering (MDE) we strive to build software-intensive systems with a high degree of automation and reuse, using models as the primary artifacts of construction. When developing families of such systems [1] we can apply techniques from Software Product Lines (SPL) [2], [3]. An integration of SPL and MDE facilitates the construction of families of systems with strategic reuse and a minimum of technical diversity (SPL) and a high level of automation (MDE). This has been demonstrated in a number of approaches, e.g., [4], [5].

Such highly-automated approaches work well as long as the derived artifacts can be constructed in a mechanized way on a high quality level. For the application’s user interface (UI), however, a fully mechanized construction is not sufficient as it has been shown that the resulting UIs often lack of usability [6], [7], [8]. Hence, to achieve sufficient usability, UI design in practice is usually performed manually with the creativity and intelligence of human designers. The flip side is that this compromises the desired degree of automation. The situation becomes even more complex, if we do not construct one UI, but UIs for a whole product line of software applications.

As running example throughout this paper example we will use a SPL for online shops, called OnlineShopPL. A product of this SPL will be one particular e-commerce system tailored to the specific needs of the customer (i.e., the company running that shop). To capture the available configuration choices we can use a feature model as shown in Fig. 1 (in FODA Notation [9]). To simplify, we focus on the order processing part of the online shop and abstract from other functionality.

The model specifies that a product of the OnlineShopPL, i.e., an online shop, has a mandatory Products section, an optional Shopping Cart and several choices on Transactions and Fulfillment. The feature model is augmented with cross-tree constraints, e.g., the selection of Electronic Delivery excludes the payment method Cash on Delivery.

To derive products from our OnlineShopPL, we could take a the conventional approach for model-driven product line engineering: In Domain Engineering, the product line is described as a feature model (describing the capabilities of the product line) and a related set of components (implementing the capabilities). In Application Engineering, a product is then constructed by first configuring the feature model and then deriving the product’s implementation. This derivation could largely be realized by automated transformations and code generators which assemble the final product.

This approach, however, has a flip side. Such automated techniques are problematic when it comes to the creation of usable and individual UIs. If we look for real world examples of e-commerce software, which can be used to construct online shops, we will find many different installations of the same software platform. On comparing the front-ends (i.e., web sites) of the various online shops, we will find many differences in the UI and interaction design, although these installations were all built based on the same platform [1]. Some variations are of purely visual nature, caused e.g., by the different shop owner’s corporate identities. But there are also lots of variations in

[1] For instance, see the customer reference lists of e-commerce platform providers, such as Intershop on http://www.intershop.de/intershop/references/
the navigation structure, the site layout, the individual UI elements, and the interaction design, depending e.g. on the kind of products presented in the shop and on the individual premises and goals of the shop owner.

Providing such highly customized UIs can be of strong importance for the shop owner as the customer experience strongly influences the online shop’s success. This holds not only for online shops but for most kind of applications which directly target the end-user [10]. Consequently, while the application core (the software components processing customers, products, transactions, payments, etc.) can be generated as described above, the UI is designed manually by UI design experts. The UI designers then have to manually ensure that the UI adheres to a given product configuration and have to manually link it to the generated application core, which are tedious and error prone tasks. Moreover, there is a conceptual gap in the development process at this point: Much effort is invested during Domain Engineering to capture knowledge precisely enough such that automated product derivation becomes possible, but none of the generated artifacts is considered (systematically) for the UI design.

This paper addresses this problem and proposes a solution which facilitates both (1) a systematic and semi-automated creation UI design during product derivation while (2) still supporting for individual, creative design. For this purpose we combine and adapt several concepts from Model-based User Interface Development (MBUID) and integrate them into our SPL product derivation approach. This enables to semi-automatically derive a UI during product derivation, including the connections between UI and the core application. Nevertheless, individual customization and creative design is still fully supported, but at well-defined “injection points”.

The remainder of the paper is structured as follows: We analyze UI development from the viewpoint of automation and discuss how techniques from MBUID can be used for our goals (Sec. II). We then present concepts for the integration of automated product derivation and individual UI design (Sec. III). Subsequently, we show how these theoretical concepts are put into practice in our approach for model-driven UI derivation (Sec. IV). The paper finishes with an overview of related work and conclusions.

II. AUTOMATION IN USER INTERFACE DEVELOPMENT: STATE OF THE ART

As a first step towards a solution we will now analyze available alternatives in UI development. First, we examine the overall spectrum of alternatives from the viewpoint of automation in development. We then take a closer look on the most promising alternatives, the concepts from the area of MBUID.

A. Spectrum of Available Techniques

When discussing UI development under the aspect of automation, we can consider a whole spectrum of approaches (Fig. 2). On the left-hand side we have purely manual UI design without any kind of automation as described in Human-Computer Interaction (HCI) books, e.g., [11]. If applied in the context of model-driven development of the underlying application core, this means that available information is not put to proper use. For instance, the UI designers start from textual requirements, but do not systematically consider, e.g., existing feature models or domain models. The right-hand side represents the opposite extreme: a completely automated process which generates a UI from existing information. For instance, the Janus tool [12] can generate a UI directly from a domain model. Such fully automated approaches can only be used for very specific application domains as they often fail to provide a sufficient UI quality [6], [7], [8].

Approaches from MBUID aim to overcome these problems and to provide systematic and partially automated UI development while preserving usability. In contrast to purely automated approaches they consider additional information about the UI specified by the developers in terms of abstract models.2 The following section gives a more detailed introduction into MBUID concepts, corresponding to the center part of the spectrum in Fig. 2.

B. Model-based User Interface Development (MBUID)

Model-based User Interface Development (MBUID) [6], [13], [14] can roughly be structured according to the process shown in Fig. 3: The most abstract models are a Task Model and Domain Model. The Domain Model specifies the structure of the application logic, e.g., in terms of a conventional UML class diagram.

1) Task Model: Tasks are activities performed by the user or the system to reach the user’s goals. Fig. 4 shows an example task model in ConcurTaskTree (CTT) notation [15], corresponding to the OnlineShopPL (Sec. I): An Application Task is performed by the system (e.g., display available products). An Interaction Task is performed by interaction between the user and the system (e.g., select a payment

2Here we use the term “models” in a broad sense including, e.g., XML-based description languages
method), while an Abstract Task groups different types of subtasks. The horizontal lines express temporal constraints. For instance, a concurrent execution of tasks is shown as \( | | | \), whereas a sequential execution of tasks with information passing is shown as \( >> | | \).  

2) AUI: The Abstract User Interface Model (AUI) is specified based on the Task Model and the Domain Model. It describes the UI in terms Abstract Interaction Objects (AIOs) which are platform- and modality-independent abstractions of UI elements (widgets). (Modality-independent means that the UI is not necessarily graphical but can also be, for instance, speech-based). An example of an AIO is the input element which enables the user to input some data, like e.g., a text field widget. Other examples are the output element which presents some data to the user, the selection element which enables the user to select a value, or the action element which enables the user to trigger some actions like e.g., a button. The data or the operations associated with an AIO can be specified by relationships to elements from the Domain Model. Each AIO is related to a task in the Task Model. For instance, the task Credit Card could be realized by several input elements for the credit card number, card type, etc. AIOs are grouped into Presentation Units (abstractions of windows in a graphical UI) and other container elements to further structure the UI (corresponding to, e.g., Panels in Java). Until now, there is no common standard notation for AUIs models.

3) CUI and Final Implementation: The Concrete User Interface Model (CUI) realizes the AUI for a specific modality in terms of concrete widgets and layout. Like for the AUI, there is no standard CUI notation. Subsequently, the final UI implementation is generated, usually under consideration of information from all other models. Depending on the purpose, many approaches use additional models, e.g., a Context Model for context-sensitive UIs.

4) Automation within MBUID: Within the UI modeling there are still different degrees of automation (see [6], [16]) as illustrated by the center part of Fig. 2. For instance, one can provide model transformations for an automatic transition from Task and Domain Model to the final implementation, like in Trident [17], or require the modeler to manually specify all UI models and the relationships between, like Mastermind [18].

For our purpose, semi-automatic approaches, like Mobi-D [7] seem to be most promising. They aim to provide as much automation as possible while enabling manual customization for critical decisions. Typical critical decisions are (1) the decomposition of the UI into presentation units (e.g., whether to put the shopping cart on a separate screen or joined with the product selection) and (2) the mapping of AIOs to CIOs (e.g., whether a selection element is mapped to a list box or to a drop down list) [6], [7], [16].

III. Integrating Automated Product Derivation and Individual User Interface Design

Based on the preceding analysis, the following section discusses two resulting general principles for derivation of UIs: (1) Derivation on an abstract level and (2) Integration of customization into an automated process.

A. Derivation of Abstract User Interface Models

From a technical point of view, the UI is just another software subsystem. Consequently, a straightforward application of product derivation concepts to the UI would work as follows: During Domain Engineering, the UI for the complete SPL is constructed and the required UI components are implemented. During product derivation, the required components are selected according to the configuration and automatically composed to form the product’s UI. From the viewpoint of

\[ \text{Fig. 4. The Task Model for the OnlineShopPL} \]
UI development, this naive solution would correspond to a purely automated approach without considering human design knowledge or customization, which is insufficient for most cases. Thus, product derivation for UIs requires a different approach, considering the UI-specific body of knowledge from MBUID.

Hence, we propose to perform the UI derivation at a higher level of abstraction using the models from MBUID. Indeed, our example models (Figures 1 and 4) show that a Task Model could be related to a feature model: For instance, the task CreditCard corresponds to the feature Credit Card, the task BillingAddress to the feature PayByBill, the task PaymentMethod depends on the selected Payment features, etc. Of course, one feature can be related to multiple tasks and vice versa; for instance, selecting the feature Electronic Delivery indicates not only that the tasks related to keywdShipping can be omitted but also the task OnStock.

As the AIOs are associated with tasks, they can be (indirectly) related to features as well. Thus, it is possible to automatically determine which AIOs are required for a given product configuration. Other UI decisions should be optimized manually, e.g., the decomposition into Presentation Units and the mapping onto concrete UI elements. These decisions strongly depend on the details of the concrete configuration and product context. For instance in the OnlineShopPL we have to consider the type of items sold in the shop, as this influences form of product presentation and selection (and e.g., the space required on the UI).

By combining SPL and the concepts from MBUID (Fig. 3) we come to the integrated approach we propose for derivation of UIs (see Fig. 5): Domain Engineering processes at an abstract level, i.e., with a model of the Abstract User Interface. During Application Engineering, the product-specific AUI can be calculated automatically from the product’s feature configuration. On that base, the final UI is derived using semi-automatic approaches from MBUID. Some parts of the UI implementation, like the links between UI elements and application logic, can be generated fully automatically. Others, like the concrete layout, the visual appearance, and the selection of concrete UI components are generated automatically, but can be customized if desired.

This general framework introduced here still abstracts from the concrete modeling languages used for its realization. For instance, it is necessary to select adequate UI models and transformations from the various MBUID approaches. We will show a possible realization in Sec. IV.

**B. Systematic Integration of UI Customization Techniques**

So far, we have argued that the step from the AUI model to the final implementation must include the option for manual customization by the designers. We will now analyze potential alternatives and introduce two techniques which contribute to our approach.

In general, customization of models and model transformations is a common task in model-driven engineering. Given one transformation one can customize it for instance by 1) adding additional information to the source model (e.g., tagged values), 2) tuning the transformation itself (e.g., by specifying parameters [19]), or 3) by just post-editing the resulting target model. These basic possibilities are used in MBUID [16] and could be used in our approach.

However, it is desirable to provide additional, more specific customization techniques since 1) during product derivation efficiency is more important than maximal flexibility and 2) the UI designers might not be familiar with generic modeling tools and transformation languages and require more domain-specific (i.e., UI-specific) tool support during customization. An important example in this sense is MOBI-D [7], which provides two concepts for customization. During the transformation, the designer is provided with a dialog where he can adjust parameters in a graphical UI. For customizing the CUI, it provides a specific kind of GUI Builder were the designer can select from those CIOs which correspond to the AIOs from the AUI.

In the following we propose two customization techniques for our purpose which enables the designer full control about all aspects of the UI while enabling a high degree automated tool support: Tree-based UI Clustering and the UI Placeholder concept.

1) **Tree-based UI Clustering:** Our approach for tree-based UI clustering addresses the problem of decomposing the UI into presentation units. It is based on earlier work [20], we adapt it here to handle specific requirements of product variability. In contrast to other approaches, the approach uses information from both Task Model and AUI and represents them in a common view. Fig. 6 shows this for our OnlineShopPL. The basic tree structure represents the AUI in terms of a hierarchy of AIOs. The AIO types are mainly those mentioned in Sec. II-B with additional support for media.
Fig. 6. The Abstract UI as a tree hierarchy for defining the Presentation Units

Fig. 7 shows the result of the clustering for two different AUIs (AUI+C). Even if the designer wants to manually customize the clusters, the automated heuristic still provides a starting point which is very helpful when dealing with UIs with many elements. In addition, based on the AUI+C representation it is possible to create interactive visual tools which enable the designer to modify the clustering very easily e.g., by drag and drop.

2) The User Interface Placeholder Concept: The placeholder concept, generalized from earlier work [21], addresses the development of the CUI based on the clustered AUI, i.e., the choice of AIOs, their concrete visual appearance, and the layout within the presentation units. The basic idea is to transform the AUI to generate a skeleton implementation, which can be modified in UI-specific visual authoring tools, e.g., the multimedia authoring tool Flash. In particular, the generated UI is composed of placeholders which can be customized and refined.

By using an unique identifier (ID) for each placeholder we can automatically associate it with corresponding elements, e.g., links to the application logic or event handling code generated from the model. Moreover, we can trace placeholders, independent from the designers modifications. As long as the designer does not delete the placeholder itself, all generated information remains untouched while the designer can freely use all the authoring tool’s powerful visual functionalities without any further restrictions.

IV. Detailed Realization

In the preceding section we took first steps towards a solution by developing some principles on a conceptual level. We will now present an approach that integrates this concepts and puts them into practice. Figure 8 shows our detailed approach which we explain in the following step by step using our OnlineShopPL.

A. Domain Engineering

In Domain Engineering (upper layer in Fig. 8) we perform the processes to create and describe the product line in terms of five models, to .

The process starts with Feature Analysis, which analyses the Product Line Requirements and produces the Feature Model capturing the scope and capabilities of the product line. The Feature Model for our OnlineShopPL was shown earlier in Fig. 1.

The next step is Task Analysis, which takes into account the Product Line Requirements and the Feature Model to create a Task Model. The task model for our OnlineShopPL was presented in Fig. 4 in CTT notation.

After completing the Task Model this is turned into an Abstract User Interface (AUI) Model in two steps. First, the
**Task Model** is transformed into the **AUI Model** using an automated Refinement Transformation, which also remembers the links between tasks and AUI elements, which were derived from these tasks, by creating a Mapping Model. Second, the engineer can perform Manual Adjustments to further refine the AUI model.

The final activity within Domain Engineering is **Feature Mapping**, where features are connected to the corresponding elements in the **Task Model** and the **AUI Model**. This information is stored in a second Mapping Model. Hence, as a final result of all these steps we get an abstract UI model, whose elements are mapped onto the corresponding features (via Mapping Model) and tasks (via Mapping Model). The AUI for our **OnlineShopPL** was shown earlier in Fig. 6. The red annotations in square brackets show the mappings to the corresponding features. It should be noted that some features influence multiple locations of the UI, similar to cross-cutting concerns in aspect-oriented programming. For instance, consider the feature **Payment**. If payment is deselected (e.g., customers get individual offers via mail or email), this not only removes elements for specifying the payment method, but also influences the presentation of products and the shopping cart.

### B. Application Engineering

After the SPL has been established we can start Application Engineering (see lower layer in Fig. 8), where we perform the processes to derive products.

The derivation of products from the product line starts with **Feature Configuration** where we try to match **Product Requirements** with the product line’s capabilities as described in the **Feature Model**. This results in a **Product Configuration** (or the insight that some requirements are not covered by the product line). Figure 9 shows an example of a feature configuration for an online shop, with a minimal set of features selected.

With the decisions captured in the **Feature Configuration** we can now perform **Product Derivation**, where we use a technique called **Negative Variability** [5], which works as follows: The models on product-line level and **AUI Model** contain the union of all potential model variations. Based on the **Feature Model Configuration** and the feature-implementation mappings we can filter out elements (i.e., tasks and AUI elements) which are related to eliminated features and, hence, are not required for this particular product. As a result we get the product-specific **Task Model** and **AUI Model**, as well as **Mappings** between them. These are subsets of the product-line models shown earlier (see figures 4 and 6), with all elements removed that correspond to eliminated features.

Given the derived task and AUI model we can perform **Clustering** to determine Presentation Units, which later will become, e.g., screens or forms. For this step we use the clustering technique from Sec. III-B. The designer can either customize the clustering based on the tree hierarchy or proceed with the generated model and, if required, modify the Presentation Units later in the authoring tool (step ). Figure 7 discussed earlier, represents the clustered AUI (AUI+C) for the minimal configuration in Fig. 9.

Finally, we apply a model transformation **AUI-to-CUI** to turn the AUI+C model into an platform-specific CUI Model, which is then fed into an CUI Generator to create the CUI Implementation. We are experimenting with multiple AUI-to-CUI generators for different platforms (e.g., GUI, Web,
mobile). These have been omitted from the illustration in Sec. IV. As an example, Fig. 10 shows the generated UI skeleton for the Adobe Flash UI platform.

The CUI generators use the placeholder concept from Sec. III-B. This means that (1) the UI structure and the glue code for the integration with other subsystems are completely generated from the model and (2) design related artifacts (which determine visual elements, layout, appearance) are generated as placeholders. In Manual Design these placeholders can be directly loaded into authoring tools, such as Adobe Flash, where the UI designer can use the functionality of professional design software and all possibilities of their human creativity.

V. Conclusion

For many applications, an individually designed, usable, and esthetic user interface is a key success factor. On the other hand, automated approaches with techniques from model-driven development and software product lines promise improvements in time to market, cost, productivity and quality [2].

In this paper we address the integration of these, initially contradictory, goals. To this end, we have carefully considered the related work and approaches from UI development, MBUID in particular, and gathered a systematic overview from the viewpoint of automation (Sec. II). Regarding SPL, we build up on the state-of-the-art for model-driven product derivation, as discussed in Sec. I. Until now, the few existing approaches which try to integrate UI construction with product derivation (e.g., [22]), use the straightforward derivation approach described in Sec. III-A, which is only suitable for very specific UIs without customization. To the best of our knowledge, there is no approach yet that integrates automated product derivation and individual UI design as addressed in this paper.

We have elaborated two general concepts, the derivation on the level of abstract UI models, and the systematic inte-
of UI customization techniques (Sec. III). Moreover, we have shown a concrete realization of these concepts by a concrete, detailed process illustrated by the online shop example (Sec. IV).

The different steps within our process are supported by prototypical tools and model transformations. We use various Eclipse-based frameworks like the Eclipse Modeling Framework (EMF), GMF (Graphical Modeling Framework), oAW (openArchitectureWare), and ATL (ATLAS Transformation Language). Based on these we implemented a tool chain for feature configuration ([23], [24]) and product derivation (using negative variability, see Sec. IV-B). The clustering and further processing of interaction elements is implemented as an ATL transformation adapted from earlier work [20]. The code generation including the placeholder concept reuses the ATL transformations from [21].

Using our concepts, large parts of UI development are performed automatically: The selection of the (abstract) UI elements according to the product configuration, the implementation of the UI’s overall structure, and the implementation of relationships of UI elements is derived automatically from the product configuration. For all parts with need for custom design (like the selection of the concrete UI elements, UI layout, and the concrete visual appearance) we provide systematic support by generating a consistent starting point (including all required relationships) which can then be customized and refined visually.

Future work includes a more detailed evaluation and the gathering of experience with the approach, which is currently still on a conceptual prototype level. In particular, we intend to explore support for traceability and (iterative) product evolution. Another potential research direction is the combination with MBUID approaches for context-sensitive UIs (e.g., [25]) by considering the product configuration as a specific kind of context. Finally, we plan to analyze variability in UIs more systematically. A question is, for instance, which variations between UIs are rather accidental or caused by the designer’s personal preference and which cause a measurable difference in usability.

ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre, http://www.lero.ie/.

REFERENCES