Self-Adaptive UIs: Integrated Model-Driven Development of UIs and their Adaptations

Enes Yigitbas, Hagen Stahl, Stefan Sauer and Gregor Engels

Paderborn University, s-lab - Software Quality Lab,
Zukunftsmeile 1, 33102 Paderborn, Germany
{enes.yigitbas,hagen.stahl,sauer,engels}@upb.de

Abstract. Self-adaptive UIs have been promoted as a solution for context variability due to their ability to automatically adapt to the context-of-use at runtime. In classical model-driven UI development (MDUID) approaches, self-adaptivity and context management introduce additional complexity since self-adaptation features are distributed in a cross-cutting manner at various locations in the models. This results in a tightly interwoven model landscape that is hard to understand and maintain. In this paper, we present an integrated model-driven development method where a classical model-driven development of UIs is coupled with a separate model-driven development of UI adaptation rules and context-of-use. We base our approach on the core UI modeling language IFML, and focus on a new modeling language for adaptation rules, called AdaptUI. We show how generated UI code is coupled with adaptation services generated from AdaptUI adaptation rules and integrated in an overall UI framework. This allows runtime UI adaptation realized by an automatic reaction to context-of-use changes. The benefit of our approach is demonstrated by a case study, showing the development of self-adaptive UIs for a university library application, utilizing the Angular 2 JavaScript framework.

Keywords: Model-Driven UI Development, UI Adaptation Rules, Self-Adaptive UIs, Context-Awareness

1 Introduction

The user interface (UI) is a key component of any interactive software application and is crucial for the acceptance of the application as a whole. However, a UI is not independent from its context-of-use, which is defined in terms of the user, platform and environment [1]. As today's user interfaces of interactive systems become increasingly complex since many heterogeneous contexts of use have to be supported, it is no longer sufficient to provide a single "one-size-fits-all" user interface. Building multiple UIs for the same functionality due to context variability is also difficult since context changes can lead to the combinatorial explosion of the number of possible adaptations and there is a high cost incurred by manually developing multiple versions of the UI [2].

In the past, model-driven user interface development (MDUID) approaches were proposed to support the efficient development of UIs. Widely studied approaches are UsiXML [3], MARIA [4], and IFML [5] that support the abstract modeling of user interfaces and their transformation to final user interfaces. However, in classical MDUID approaches, the modeling of self-adaptivity and context management aspects introduce additional complexity since self-adaptation features are distributed in a cross-cutting manner at various locations in the models. This results in a tightly interwoven model landscape that is hard to understand and maintain. Therefore, an integrated model-driven development method is needed where a classical model-driven development of UIs is coupled with a separate model-driven development of UI adaptation rules and context-of-use. In detail, the following challenges have to be addressed to integrate adaptation aspects into MDUID and support the development of self-adaptive UIs in a systematic way:

- *C1: Specification of UI Adaptation Rules*: A language conform to the core UI modeling language IFML, standardized by the Object Management Group (OMG), is required for specifying UI adaptation rules in an abstract manner. With the help of this language, UI designers should be able to separately specify various UI adaptation rules which can adapt the UI at runtime (separation of concerns, abstraction level, extensibility, maintainability).
- *C2: Generation of UI Adaptation Logic*: Based on the specified abstract UI adaptation rules, the adaptation logic needs to be generated for supporting UI adaptation capabilities at runtime.
- *C3: Execution of UI Adaptation at Runtime*: For supporting runtime UI adaptation enabling automatic reaction to dynamic context-of-use changes, the generated adaptation logic needs to be coupled with generated UI code as well as integrated in an overall UI framework.

To address the above described challenges, the contributions of this paper include our vision on enhancing UIs with self-adaptation capabilities in a systematic and model-driven way. Therefore, our contribution covers the following aspects: Firstly, a domain specific language, called AdaptUI, will be presented which supports the specification of abstract UI adaptation rules that cover various adaptation dimensions (e.g. layout, navigation, or task-feature set). Additionally, our approach supports the generation of UI adaptation logic by transforming the abstract UI adaptation rules into an executable representation of the target UI framework. Finally, a rule-based execution engine is integrated in our UI framework for executing the UI adaptations at runtime.

The remaining sections of this paper are organized as follows: Section 2 presents the conceptual solution of our work. In Section 3, we present the modeling and integration of UI adaptation concerns in MDUID. Section 4 deals with the implementation of our approach. Section 5 shows the benefit and usefulness of our approach based on a case-study from the domain of university library management. Related work is presented in Section 6 and finally Section 7 concludes the paper and gives an outlook on future work.

2 Conceptual Solution

Model-driven User Interface Development (MDUID) is a promising candidate for mastering the complex development task of self-adaptive UIs in a systematic, precise and appropriately formal way. Our model-driven solution architecture for self-adaptive UIs is depicted in Figure 1 and consists of three development paths.

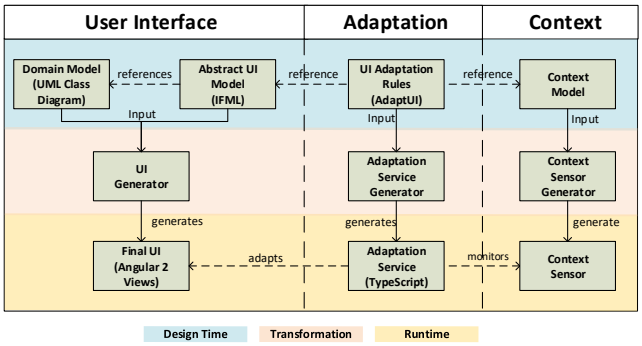

Fig. 1. Model-driven Architecture for Self-adaptive UIs

The first development path (left side of Figure 1) addresses the model-driven development of UIs. This development path makes use of an *Abstract UI Model* and a *Domain Model* which are then transformed by a code generator (*UI Generator*) into a *Final UI*. This development path has been subject of extensive research [6] and we already presented the realization and application of an MDUID approach for different target platforms ([7], [8]) (including smartphone, desktop and self-service systems) based on the OMG standard IFML. The first development path supports efficient development of heterogeneous UIs for different target platforms. However, this development path on its own is not enough to support UI self-adaptation capabilities. Therefore, we extended our existing MDUID solution architecture with parallel development paths which support model-driven development of UI adaptation rules and context-of-use. This way, the model-driven UI development path is complemented by an analog development path which is responsible for the UI adaptation concerns. As the UI adaptation path is also based on the paradigm of model-driven development, the solution preserves various advantages of model-driven software development like Separation of Concerns, Extensibility or Maintainability. In general, the main idea of the model-driven adaptation path (in the middle of Figure 1) is to support the specification of abstract *UI Adaptation Rules* in alignment to the standardized abstract UI modeling language IFML. The specified *UI Adaptation Rules* serve as an input for the *Adaptation Service Generator* which transforms them

into an *Adaptation Service*. The *Adaptation Service* is responsible for adapting the generated *Final UI* at runtime. The third development path (right side of Figure 1) is responsible for characterizing the dynamically changing context-of-use parameters. A *Context Model* that is referenced by the *UI Adaptation Rules*, supports the abstract specification of heterogeneous context-of-use situations. Based on the *Context Model*, the *Context Sensor Generator* allows the generation of various *Context Sensors* like accelerometer, GPS, brightness or noise level. The *Context Sensors* provide context information data that are monitored by the generated *Adaptation Service* to decide on how to adapt the UI at runtime.

In this paper, we are especially focusing on the adaptation path and its integration in the MDUID approach. For illustrating the interplay between the generated final user interface, the *Angular2 Views*, and the *Adaptation Service* as well as to present the effect of specified UI adaptation rules on the final user interface, we elaborate on the adaptation approach. Figure 2 shows a detailed overview of the UI adaptation approach containing the main layers and components for realizing self-adaptive UIs that are able to automatically react to changes in their context-of-use.

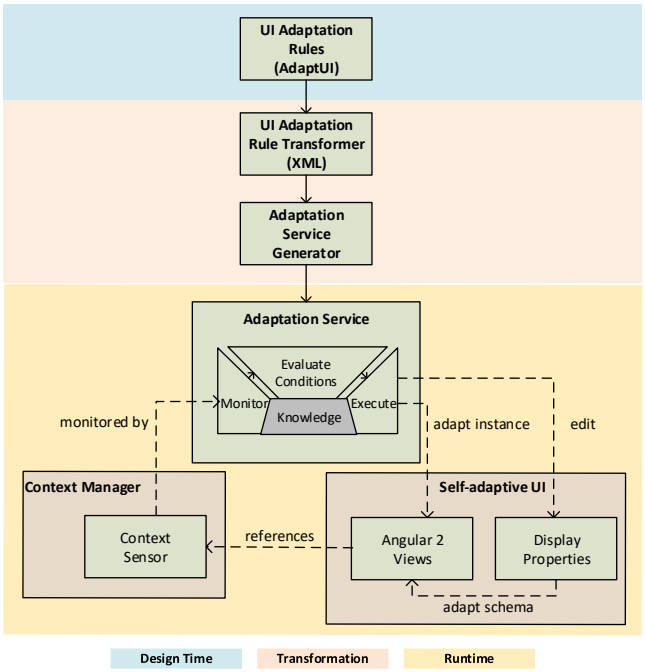

Fig. 2. Model-driven Specification and Adaptation of UIs

The first layer starts with the specification of *UI Adaptation Rules* at design time. The specification language is AdaptUI, a domain-specific language developed in this work which is explained in more detail in the next section. In the second layer, the abstract adaptation rules are transformed into an XML format by the *UI Adaptation Rule Transformer*. The goal of this transformation is to store the adaptation rules in a universal, common file format, that is easily traversable for further transformation processes. The next step in the generation process, is the transformation to an executable *Adaptation Service*. This transformation is done by the *Adaptation Service Generator*. The output of the generator is the *Adaptation Service* which characterizes a runtime component in the third layer. The *Adaptation Service* implements an adaptation loop similar to IBM's MAPE-K loop [11]. Main runtime components besides the *Adaptation Service* are the *Context Manager* and the *Self-adaptive UI*. The *Context Manager* provides the generated context information through *Context Sensors* which are specified in the *Context Model*. The *Self-adaptive UI* consists of two subcomponents: The *Angular 2 Views* which are responsible for representing the UI and *Display Properties* which are affected by the adaptation rules and contain the adaptable schema and type information of the UI. Context information which are generated by *Context Sensors* are monitored by the *Adaptation Service*. Unlike the MAPE-K loop with its analysis and plan phases, the *Adaptation Service* relies on the application of predefined (by the abstract UI Adaptation Rules) conditions and associated actions. Therefore, no planning of actions is necessary. The two phases in the MAPE-K loop are replaced by the *Evaluate Conditions* component. Rules that satisfy the conditions are executed. The rules can modify the UI directly or edit the *Display Properties*. General approach here is that the UI is directly modified, if the change only affects the current view (adaptation of the current instance). If it is, for example, a property change that would affect several pages, it is set in the *Display Properties* (adaptation of schemas). An example for a property could be the layout of tables in the whole UI. The properties are referenced from within the views and thereby can adapt the layout and design. The *Knowledge* component of the MAPE-K loop is not focus of this paper, but logged context information data and stored adaptation states and preferences could be used to infer upcoming UI adaptations.

3 Modeling and Integration of Adaptation Concerns

In this section, we describe our integrated modeling approach for representing UI adaptation rules. Therefore, we present our UI adaptation language AdaptUI and show its coupling to the core UI modeling language IFML and to context modeling.

Specifying sound UI adaptation rules is a challenging task which should be supported by a dedicated domain specific language. Based on OMG's core UI modeling language IFML, we developed a new modeling language for UI adaptation rules, the language AdaptUI. AdaptUI allows domain experts, for example web designers, to model adaptation concerns by specifying the conditions and

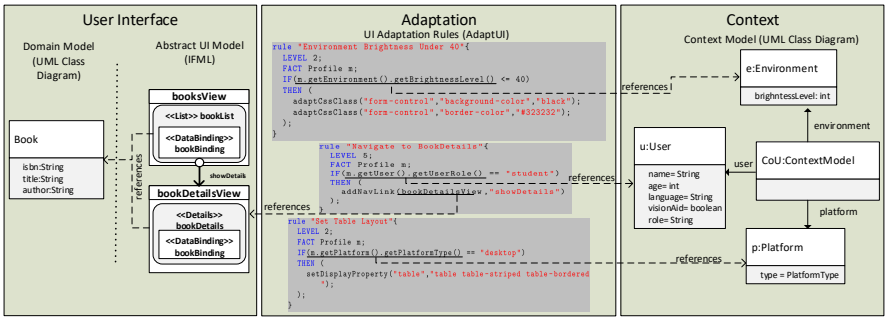

Fig. 3. Specification of AdaptUI adaptation rules

actions for UI adaptations. To support various adaptation techniques for devising self-adaptive model-driven UIs, AdaptUI enables specification of different UI adaptation rules. The following main categories of UI adaptation types are supported by AdaptUI: task-feature-set, navigation, and layout adaptation. Task-feature-set adaptation supports UI adaptation by flexibly showing and hiding UI interaction elements like tables, buttons, text-fields etc. Navigation adaptation means that the navigation flow of the UI can be flexibly adapted based on the contextual parameters by adding, deleting or redirecting links between user interface flows. Finally, layout adaptation deals with adaptation rules that support layout optimization like changing font size, colors or splitting screens to divide a complex UI view into multiple views so that for example small screen sizes are satisfied. Figure 3 shows a modeling example of UI adaptation rules based on our language AdaptUI. On the left side of this figure, small excerpts of the core UI models are depicted. There is an abstract UI model based on IFML which shows the representation of two UI view containers *booksView* and *bookDetailsView* which are connected by a navigation edge *showDetails*. To enable the specification of data bindings in IFML, the corresponding classes from the domain model are referenced, in our case the class *Book*. To support the separate specification of UI adaptation rules in addition to the IFML model in a comfortable way, AdaptUI allows to specify and bind different adaptation rules to the IFML modeling elements. In the center of Figure 3, an example specification of an AdaptUI navigation adaptation rule is shown, which is called "'Navigate to BookDetails"'. This AdaptUI rule defines that the specific view *bookDetailsView* can be only reached, if a specific user context is satisfied. For defining this rule, AdaptUI rules are referencing a context model where relevant contextual parameters are described. In the case of our example, the user role student has to be satisfied so that the *bookDetailsView* can be reached. In a similar way, various other UI adaptation rules like adapt brightness or set table layout (see Figure 3) can be specified to react to potential context-of-use changes.

An overview of the general structure of the AdaptUI language is shown in Figure 4. The root element of all elements in AdaptUI is the *AdaptUI-Model*. It contains the definition of a *Flow* (chosen to be conform to the terminology of used rule engine Nools[1]) containing arbitrary number of *AdaptationRule* elements. An *AdaptationRule* consists of the *RuleName*, *FactDefinition*, a *PriorityLevel*, *Conditions* and *Actions*. The *FactDefinition* is given as the class name in the final Angular 2 application and an identifier by which it is referred to within the rule. To decide in which order rules are executed if more than one satisfies all conditions, the *PriorityLevel* is used as indicator for priority. Higher level means that the rule is executed before rules with lower level.

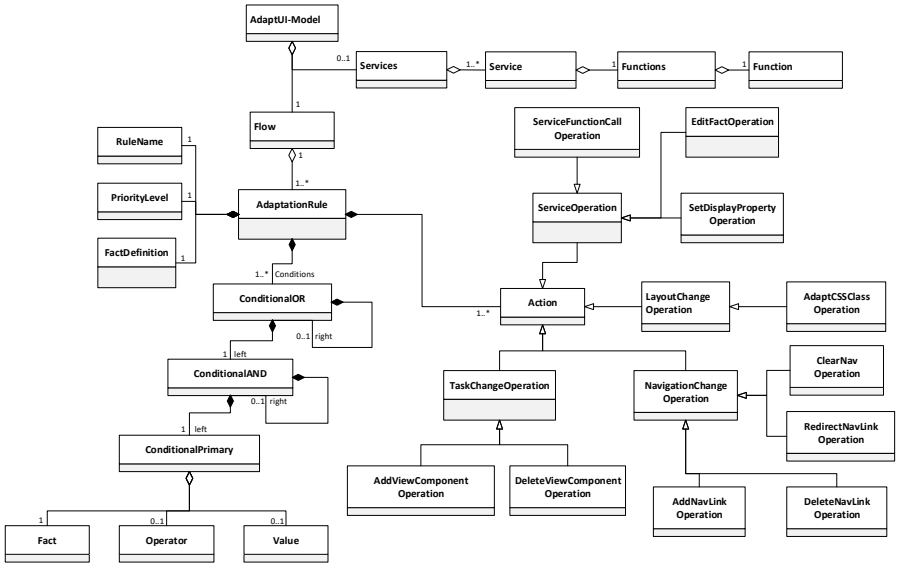

Fig. 4. Structure of AdaptUI-DSL

Conditional expressions can be used to check if the fact satisfies certain conditions. The condition can be a combination of boolean expressions concatenated by OR-operators and AND-operators. For this, AdaptUI provides several elements to build such an expression. The *ConditionalOR* elements are connected by OR-operators. The left side of such a *ConditionalOR* expression is a *ConditionalAND* expression. The right side of a *ConditionalOR* expression can be empty or be another *ConditionalOR* expression. The elements in a *ConditionalAND* expression are concatenated by an AND-Operator. The left side of *ConditionalAND* is the *ConditionalPrimary*, which is a boolean expression made up

[1] https://github.com/C2FO/nools

of just a fact or a combination of fact, operator and value. The right side of a *ConditionalAND* can be either another *ConditionalAND* or be empty.

The *Actions*, which were introduced in the beginning of this paragraph, are executed if the conditions are satisfied. The supported *Actions* of the AdaptUI language are based on the action categories defined in our previous work [8] and cover the adaptation operations *TaskChangeOperation*, *NavigationChangeOperation* and *LayoutChangeOperation*. A fourth type proposed in our prior previous work is a *ComposedAction* which combines multiple actions of the first three categories. In AdaptUI, the *ComposedAction* type is implicitly modelled by the composition relation between *AdaptationRule* and *Action*.

Beside a *Flow* an *AdaptUI-Model* can also contain *Services* in the target language of the UI. This means, the services referenced here are existing Angular 2 services that are used within the web application. The definition of these services enables the user of the language to use them later on in the rule specification. A *Service* is defined by its name and relative location to the *Services* folder of the Angular 2 implementation. A *Service* can contain interfaces to *Functions* which also have a *Function* with its name attribute. Both, *Functions* and *Services* are referred to by their respective ID. To allow editing facts or call Angular 2 services through an *AdaptationRule*, an additional category, *ServiceOperation*, is included in the Actions.

4 Implementation

We implemented an *IFML2NG2* generator to support the utilization of our modeling and development approach for devising self-adaptive UIs. The realized generator automatically creates Angular 2 views, based on the IFML model and domain model, and the adaptation service, based on the AdaptUI rule specification. In the following, we focus on and briefly describe the implementation of *AdaptUI*, the *Adaptation Service Generator* and the *Runtime Components* to support UI adaptation at runtime (see Figure 2).

4.1 AdaptUI

For specifying abstract UI adaptation rules, the described UI adaptation language *AdaptUI* is used. Foundation of *AdaptUI* is the open-source framework Xtext[2] for development of programming languages and domain-specific languages. The defined language also comes with support of an infrastructure integrated in the Eclipse IDE. Features include syntax highlighting and code completion as useful tools for the user of *AdaptUI*.

4.2 Adaptation Service Generator

The goal of the *Adaptation Service Generator* is the automated creation of an Angular 2 service that allows the adaptation of the UI at runtime. The adaptations to the UI are expressed in a rule-based form in an XML format. Based on

[2] http://www.eclipse.org/Xtext

this input file, the *Adaptation Service Generator* generates an Angular 2 service containing the JavaScript rules engine Nools. Nools is an efficient RETE-based rule engine written in JavaScript and provides an API for specifying fact and rules. The *Adaptation Service Generator* is implemented with Xtend[3] and receives the UI adaptation rules in an XML format as input. Structurally, it consists of the components *NoolsServiceGenerator*, *NoolsRuleGenerator*, *NoolsConditionGenerator* and *NoolsActionGenerator* (see Figure 5). These components are responsible for creating an injectable Angular 2 service for monitoring the context model and executing adaptation operations.

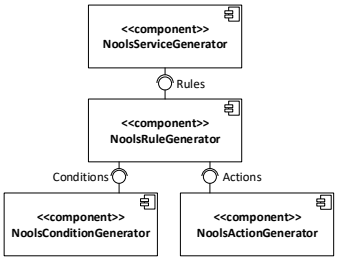

Fig. 5. Structure of the Adaptation Service Generator

The base structure of the Angular 2 service, generated by the *NoolsServiceGenerator*, consists of the required Angular 2 imports, the class declaration of the service and the implementation of the Nools flow. The flow is composed of all the rules defined in the abstract UI adaptation rules. For each rule it is defined under which conditions the rule actions are executed. The generation of the individual rules is delegated to the *NoolsRuleGenerator*. For each adaptation rule the name is the name of the abstract UI adaptation rule. The salience of the rule is the priority level of the rule and corresponds to the level defined in the *AdaptUI* rule specification. In addition to that, the rule fact is defined by the factType and factName attributes. The generation of the conditions and adaptation operations of the rule is delegated to the *NoolsConditionGenerator* and the *NoolsActionGenerator* respectively.

The *NoolsConditionGenerator* is responsible for creating the rule conditions. All child elements of the conditions element are combined with the OR-operator. If there is a conditionGroup element, all child elements of the conditionGroup are combined with the AND-operator. The result is a string of concatenated conditions with operators. Likewise, to generate the actions that the rule should execute when the conditions are satisfied, the *NoolsActionGenerator* is called with the actions element as parameter and, additionally, the mapping of services and functions defined in the abstract UI adaptation rule specification. However,

[3] http://www.eclipse.org/xtend

there is a defined set of actions. If the action element is unknown, there is no code created. This means, that if there are new possible actions added to the schema definition, they also need to be implemented in the *NoolsActionGenerator*.

4.3 Runtime Components: Adaptation Service, Self-adaptive UI and Context Manager

At runtime, we have the components *Adaptation Service*, *Self-adaptive UI* and *Context Manager*. The *Self-adaptive UI* is generated by our *IFML2NG2* generator. Its Angular 2 views consist of an HTML template, which is used to render the UI in the browser, and an Angular 2 component, which is implemented in TypeScript and manages the view. Likewise, the *Adaptation Service* is generated as Angular 2 service and is also implemented in TypeScript. As described in the earlier section, the *Adaptation Service* uses Nools, a JavaScript based rule engine, for monitoring the context information provided by the *Context Manager*. In our current implementation, the *Context Manager* and the *Display Properties* (see Figure 2) are implemented manually and independently from the generation pipeline in TypeScript. However, to ensure the integration of the adaptation loop, they are referenced within the *AdaptUI* specification. The facts of the *AdaptUI* rule specification reference the different context-of-use information stored in the context model of the *Context Manager*. Furthermore, it is possible to define UI adaptation operations that should change the schema used by the view elements of the UI.

At runtime, the *Adaptation Service* monitors the context information and executes the adaptation rules whose conditions are satisfied. To adapt the UI view elements on instance level, JQuery is used to directly manipulate the DOM tree of the view. Changes only affect the current UI view element and do not persist on other UI views. When changing the schema for a group of view elements in the *Display Properties*, the adaptation affects the properties of all view elements of this type. This also includes instances of this view element type on subsequently visited views. This is done by binding the layout class of the view elements of this type, represented by CSS classes, to the properties stored within the *Display Properties*.

5 Case study

The case study setting is based on an example scenario which is derived from the university library management domain (see Figure 6). The scenario setting is a library web application for universities which is called "LibSoft". LibSoft provides core library management functionality like searching, reserving and lending books. LibSoft's UI can be accessed by heterogeneous users and user roles (like student or staff member) through a broad range of networked interaction devices (e.g. smartphones, tablets, terminals etc.) which are used in various environmental contexts (e.g. brightness, loudness, while moving etc.). Depending on the situation, users are able to access their library services where, when and how

it suits them best. For example, if the user wants to pursue a self-determined cross-channel book lending process, she can begin an interaction using one channel (search and reserve a book with her laptop at home), modify the transaction on her way using a mobile channel, and finalize the book lending process at the university library via self-check-out terminal or at the staff desk. In the example scenario described above, each channel has its own special context-of-use and eventually the contextual parameters regarding user, platform and environment can dynamically change. Figure 7 shows such a context-of-use (CoU) change from CoU2 to CoU4 (compare Figure 6). The depicted context-of-use object model excerpts in Figure 7 illustrate how different contextual parameters regarding user, platform and environment can change. Therefore, it is important to continuously monitor the context-of-use parameters and react to possible changes by automatically adapting the UI for the new context-of-use situation.

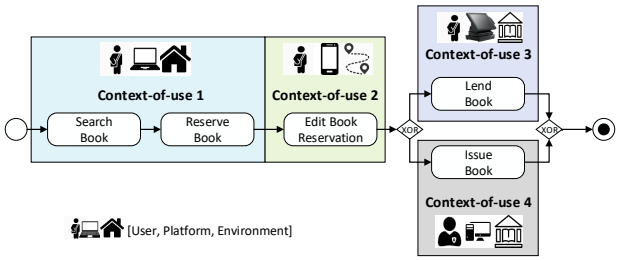

Fig. 6. Example scenario: UIs in dynamically changing context-of-use situations.

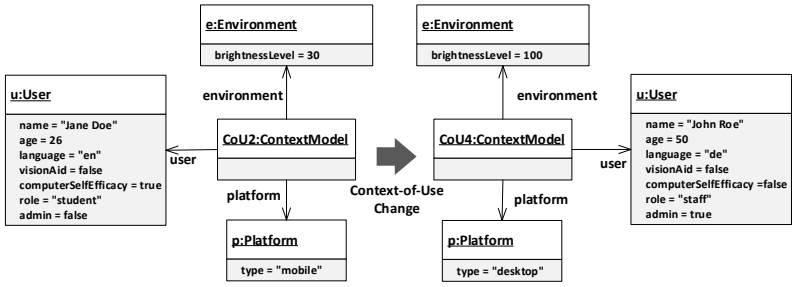

Fig. 7. Context-of-use object model excerpts

Already a small set of contextual parameters can highly influence the UI since lots of context situations can occur if the context-of-use parameters dynamically change. Based on the different context dimensions, various adaptations to the UI

can be specified and integrated in the web application. The integration happens, as explained within the earlier sections, by specifying the adaptation rules with the help of AdaptUI and using the specification as input for the generator. For utilizing our approach in the case study setting, an IFML model, representing the views and navigational flows of the UI, a domain model and a set of UI adaptation rules were created as described in Section 3. The specified models were transformed into final user interfaces using our *IFML2NG2* generator.

Screenshots of the resulting self-adaptive UI are depicted in Figure 8. According to the monitored context information for CoU2, the layout for the UI is optimized for a mobile device used in a darker environment, because the user Jane is editing her book reservation while travelling to the library and it is already quite dark outside (see left side of Figure 8). Also, the UI is adapted to the user properties by enabling access to the functions and navigation available to students. The UI language is set to English as it is preferred by the user Jane. Since Jane is recognized as a self-efficacious user with the application, she gets extended functionalities, like a more complex search and filter mechanism for the list view of the books. When the context changes from CoU2 to CoU4, the generated self-adaptive UI adapts itself automatically to the new contextual parameters. In this case, the staff members view on a desktop device with a wider and brighter layout is shown, displaying the list of reserved books, because in CoU4 a staff member, John Roe, uses his desktop computer to issue the book to Jane. Additionally, to the functionalities and functions available to staff members, John is provided with a link to the administration interface, because he is granted access to the administration interface. The UI Language is set to German and the search and filter mechanisms of the list are simplified, because he just started using LibSoft and is, therefore, not yet self-efficacious. Since the location is a well-lit library, the brightness of the environment is high.

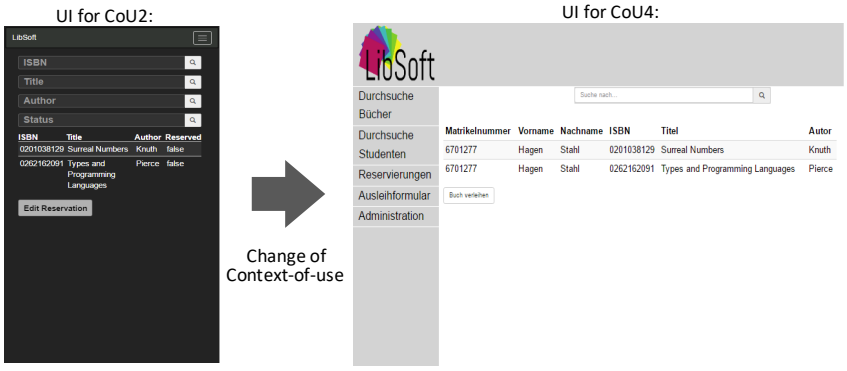

Fig. 8. UI adaptation according to different contexts-of-use

The case study demonstrates the benefit of our approach for supporting the development of self-adaptive UIs and showcases our solution approach for addressing the introduced challenges C1-C3. Through the separate specification of abstract AdaptUI rules the modeling of adaptation concerns is supported in a comfortable way. The case study also shows how generated UI code is coupled with adaptation services generated from AdaptUI adaptation rules and integrated in an overall UI framework. As shown in the example scenario, this allows runtime UI adaptation realized by an automatic reaction to context-of-use changes.

6 Related Work

Recent research provides various approaches that support the model-based and model-driven development of UIs and their adaptations.

Model-based and model-driven development methods have been discussed in the past for various individual aspects of a software system and for different application domains. This applies to the development of the data management layer, the application layer or the user interface layer. The CAMELEON Reference Framework (CRF) [1] provides a unified framework for model-based and model-driven development of UIs. UIs are represented in CRF on the following levels of abstraction: Tasks and Domain Models, Abstract User Interface (AUI) Model, Concrete User Interface (CUI) Model and Final User Interface (FUI). UsiXML [3], MARIA [4] and IFML [5] are widely studied approaches for model-driven UI development which were applied in various domains. However, these approaches do not explicitly cover the specification and integration of UI adaptation aspects in the development process by providing a UI adaptation language that enables the generation of adaptation services for supporting runtime UI adaptation.

In recent research, adaptive or self-adaptive UIs have been promoted as a solution for context variability due to their ability to automatically adapt to the context-of-use at runtime [2]. A key goal behind self-adaptive UIs is plasticity denoting a UI's ability to preserve its usability despite dynamically changing context-of-use parameters [9]. In practice, especially in the context of web design, the paradigm of Responsive Web Design (RWB) is widely used to adapt the layout of a web page in response to the characteristics of the used device. While RWB adaptation rules are mainly focusing on the contextual parameter *Platform*, considering device characteristics like screen size or resolution, our approach also focuses on the contextual parameters *User* and *Environment* allowing the specification of advanced adaptation rules and automatic adaptation to complex context-of-use situations.

In [10] the authors present a hierarchy of adaptability properties for software systems, referred to as self-* properties. Based on this work, the authors present in [2] how some of these properties are applicable to the domain of self-adaptive UIs. Similar to the idea that self-* properties of self-adaptive software systems can be applied to self-adaptive UIs, it is possible that general reference architec-

tures for self-adaptive systems can be also applied to self-adaptive UIs. We will give a brief overview of these architectures. The MAPE-K loop, which was used in our approach, was created by IBM as a reference model for autonomic computing [11]. MAPE-K considers software systems as a set of managed resources that is adapted by an adaptation manager which consists of the components Monitor, Analyze, Plan, Execute, and Knowledge. Similar reference architectures for self-adaptive systems are Rainbow [12] and the Three Layer Architecture [13]. Beside these general architectures for self-adaptive systems, there are also specific reference architectures for adaptive UIs like CAMELEON-RT [14], CEDAR [15] or FAME [16]. Furthermore, different approaches like Supple [18], MASP [19], MyUI [20] or RBUIS [21] present methods, techniques and tools for supporting the development of adaptive UIs. However, these approaches do not focus on the generation of UI adaptation logic in the means of adaptation services.

On the intersection of MDUID and UI adaptation, several transformation-based approaches like [22] or [23] were proposed that make use of adaptation rules based on a context model to adapt UIs. There are also other approaches using different techniques to adapt UIs, like [24] which uses machine learning or [17] where a genetic algorithm is used to calculate a well suited UI adaptation. Compared to these approaches, our model-driven approach for developing self-adaptive UIs, provides a dedicated rule-based UI adaptation language and supports the generation of adaptation services allowing runtime UI adaptation.

7 Conclusion and Outlook

In this paper, we present an integrated model-driven development approach for self-adaptive UIs where a classical model-driven development of UIs is enhanced and coupled with a separate model-driven development of UI adaptation rules and context-of-use. Based on OMG's core UI modeling language IFML, we propose a new modeling language for UI adaptation rules, the language AdaptUI. We present how generated UI code is coupled with adaptation services generated from AdaptUI adaptation rules and integrated in an overall UI framework. This allows runtime UI adaptation realized by an automatic reaction to dynamically changing context-of-use parameters like user profile, platform, and usage environment. We demonstrate the benefit of our approach by a case study, showing the development of self-adaptive UIs for a university library application, utilizing the Angular 2 framework.

In ongoing research, we investigate the acceptance and user-friendliness of self-adaptive UIs by conducting usability studies with potential end-users. In addition to that, we analyze how additional context information properties can be automatically monitored and generated by context sensors to be used for further UI adaptation. Further research will also cover the application of quality assurance techniques to our presented model-driven UI adaptation approach, which enable the provisioning of hard guarantees concerning self-adaptivity characteristics such as adaptation rule set stability and deadlock freedom. Furthermore, we plan to enhance our proposed UI self-adaptation loop through the imple-

mentation of a knowledge component. In this context, it is conceivable to apply learning algorithms based on the user's assessment of executed adaptation operations to further improve UI adaptations.

Acknowledgement

This work is based on "KoMoS", a project of the "it's OWL" Leading-Edge Cluster, partially funded by the German Federal Ministry of Education and Research (BMBF).

References

1. G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon, and J. Vanderdonckt. 2003. A Unifying Reference Framework for Multi-Target User Interfaces. Interacting with Computers, 15, 289-308.
2. P. A. Akiki, A. K. Bandara, and Y. Yu, Adaptive Model-Driven User Interface Development Systems. 2014. ACM Comput. Surv., vol. 47, no. 1, pp. 64:1-64:33.
3. Q. Limbourg and J. Vanderdonckt. 2004. USIXML: A User Interface Description Language Supporting Multiple Levels of Independence. In Engineering Advanced Web Applications: Proceedings of Workshops in connection with the 4th International Conference on Web Engineering. Rinton Press, 325-338.
4. F. Paternò and C. Santoro, and L. D. Spano. 2009. MARIA: A Universal, Declarative, Multiple Abstraction-Level Language for Service-Oriented Applications in Ubiquitous Environments. ACM Transactions on Computer-Human Interaction 16(4),19:1-19:30.
5. M. Brambilla, and P. Fraternali. 2014. Interaction Flow Modeling Language - Model-Driven UI Engineering of Web and Mobile Apps with IFML. The MK/OMG Press.
6. F. Paternò and C. Santoro. 2012. A logical framework for multi-device user interfaces. In Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems (EICS '12). ACM, New York, NY, USA, 45-50.
7. E. Yigitbas, T. Kern, P. Urban, and S. Sauer. 2016. Multi-Device UI Development for Task-Continuous Cross-Channel Web Applications. In Proceedings of the 1st International Workshop on Liquid Multi-Device Software for the Web at ICWE'16. Springer, LNCS, vol. 9881, pp. 114-127.
8. E. Yigitbas and S. Sauer. 2016. Engineering Context-Adaptive UIs for Task-Continuous Cross-Channel Applications. In Human-Centered and Error-Resilient Systems Development. Springer, LNCS, vol. 9856, pp. 281-300.
9. J. Coutaz. 2010. User Interface Plasticity: Model Driven Engineering to the Limit! In Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems. ACM, 1-8.
10. M. Salehie and L. Tahvildari. 2009. Self-Adaptive Software: Landscape and Research Challenges. ACM Transactions on Autonomous and Adaptive Systems, 4, 1-42.
11. IBM. 2006. An Architectural Blueprint for Autonomic Computing.
12. D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. Computer 37(10), 46-54.

13. J. Kramer and J. Magee. 2007. Self-Managed Systems: An Architectural Challenge. In Proceedings of the Workshop on the Future of Software Engineering. International Conference on Software Engineering. IEEE, 259-268.
14. L. Balme, R. Demeure, N. Barralon, J. Coutaz, G. Calvary, and U. J. Fourier. 2004. Cameleon-RT: A Software Architecture Reference Model for Distributed, Migratable, and Plastic User Interfaces. In Proceedings of the 2nd European Symposium on Ambient Intelligence. Springer, 291-302.
15. P. A. Akiki, A. K. Bandara, and Y. Yu. 2012. Using Interpreted Runtime Models for Devising Adaptive User Interfaces of Enterprise Applications. In Proceedings of the 14th International Conference on Enterprise Information Systems. SciTePress, 72-77.
16. C. Duarte and L. Carric. 2006. A Conceptual Framework for Developing Adaptive Multimodal Applications. In Proceedings of the 11th International Conference on Intelligent User Interfaces. ACM, 132-139.
17. A. Blouin, B. Morin, O. Beaudoux, G. Nain, P. Albers, and J.-M. Jzquel. 2011. Combining aspect-oriented modeling with property-based reasoning to improve user interface adaptation. In Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems (EICS '11). ACM, 85-94.
18. K. Z. Gajos, D. S. Weld, and J. O. Wobbrock. 2010. Automatically Generating Personalized User Interfaces with Supple. Artificial Intelligence 174(12-13), 910-950.
19. S. Feuerstack, M. Blumendorf, and S. Albayrak. 2006. Bridging the Gap between Model and Design of User Interfaces. In R. L. Christian Hochberger, Lecture Notes in Informatics, 131-137.
20. M. Peissner, D. Haebe, D. Janssen, and T. Sellner. 2012. MyUI: Generating Accessible User Interfaces from Multimodal Design Patterns. In Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS'12). ACM, 81-90.
21. P. A. Akiki, A. K. Bandara, and Y. Yu. 2016. Engineering Adaptive Model-Driven User Interfaces. IEEE Trans. Softw. Eng. 42, 12 (December 2016), 1118-1147.
22. V. López-Jaquero, F. Montero, and P. González. 2011. T:XML: A Tool Supporting User Interface Model Transformation. In Model-Driven Development of Advanced User Interfaces, 241-256.
23. J.-S. Sottet, V. Ganneau, G. Calvary, J. Coutaz, A. Demeure, J.-M. Favre, and R. Demumieux. 2007. Model-driven adaptation for plastic user interfaces. In Proceedings of the 11th IFIP TC 13 international conference on Human-computer interaction (INTERACT'07). Springer-Verlag, Berlin, Heidelberg, 397-410.
24. A. Hariri, D. Tabary, S. Lepreux, and C. Kolski. 2008. Context aware business adaptation toward user interface adaptation. In Communications of SIWN. Springer-Verlag, 46-52.