Software and Systems Modeling (2020) 19:1057–1081
https://doi.org/10.1007/s10270-020-00777-7

SPECIAL SECTION PAPER

Integrated model-driven development of self-adaptive user interfaces

Enes Yigitbas[1] · Ivan Jovanovikj[1] · Kai Biermeier[1] · Stefan Sauer[1] · Gregor Engels[1]

Received: 8 February 2019 / Revised: 28 October 2019 / Accepted: 6 December 2019 / Published online: 27 January 2020

Abstract
Modern user interfaces (UIs) are increasingly expected to be *plastic*, in the sense that they retain a constant level of usability, even when subjected to context changes at runtime. Self-adaptive user interfaces (SAUIs) have been promoted as a solution for context variability due to their ability to automatically adapt to the context-of-use at runtime. The development of SAUIs is a challenging and complex task as additional aspects like context management and UI adaptation have to be covered. In classical model-driven UI development approaches, these aspects are not fully integrated and hence introduce additional complexity as they represent crosscutting concerns. In this paper, we present an integrated model-driven development approach where a classical model-driven development of UIs is coupled with a model-driven development of context-of-use and UI adaptation rules. We base our approach on the core UI modeling language IFML and introduce new modeling languages for context-of-use (*ContextML*) and UI adaptation rules (*AdaptML*). The generated UI code, based on the IFML model, is coupled with the context and adaptation services, generated from the *ContextML* and *AdaptML* model, respectively. The integration of the generated artifacts, namely UI code, context, and adaptation services in an overall rule-based execution environment, enables runtime UI adaptation. The benefit of our approach is demonstrated by two case studies, showing the development of SAUIs for different application scenarios and a usability study which has been conducted to analyze end-user satisfaction of SAUIs.

Keywords Model-driven UI development · UI adaptation · Self-adaptive UIs · Context-aware applications

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 [8]. Today's user interfaces of interactive systems become increasingly complex as many heterogeneous contexts-of-use have to be supported. Hence, 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 as 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 like UsiXML [22], MARIA [29], and IFML [6] support the abstract modeling of user interfaces and their transformation to final user interfaces. However, in the aforementioned classical MDUID approaches, the modeling of context management and UI adaptation aspects introduce additional complexity as they characterize crosscutting concerns. This results in a tightly interwoven model landscape that is hard to understand and maintain. Therefore, an integrated model-driven development approach is needed where a classical model-driven development of UIs is coupled with a separate model-driven development of context-of-use and UI adaptation rules. Hence, in order to support the development of self-adaptive UIs in a systematic way, the following challenges have to be addressed to integrate context management and adaptation aspects into MDUID:

Context Management Challenges:

– *C1: Specification of contextual parameters*: A modeling language is required for specifying different contexts-of-

Communicated by A. Pierantonio, A. Anjorin, S. Trujillo, and H. Espinoza.

✉ Enes Yigitbas
enes@mail.upb.de

[1] Paderborn University, Fürstenallee 11, 33102 Paderborn, Germany

use. Such a modeling language should enable modeling of different contextual situations that can occur during usage of the UI. With the help of this language, developers should be able to separately specify needed context sensor services to monitor various contextual parameters.
- *C2: Generation of context services*: A transformation method for automatic generation of heterogeneous context services is needed. Based on the specified context-of-use model, code for the required context services needs to be generated for monitoring context information and triggering the adaptation at runtime.
- *C3: Execution of context services and runtime monitoring*: An execution environment is required for executing the generated context services. For supporting runtime monitoring of dynamic context changes, the generated context services should observe the context sensors and provide context information data within the overall UI execution environment.

UI Adaptation Challenges:

- *C4: Specification of UI adaptation rules*: A language conform to the core UI modeling language IFML [24], 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 for different contexts-of-use which can adapt the UI at runtime.
- *C5: Generation of UI adaptation services*: A transformation method for automatic generation of different adaptation services is needed. Based on the specified abstract UI adaptation rules, the code for the executable adaptation services needs to be generated for supporting UI adaptation capabilities at runtime.
- *C6: Execution of UI adaptation at runtime*: An integrated execution environment is required for executing the generated context and adaptation services. For supporting runtime UI adaptation enabling automatic reaction to dynamic context-of-use changes, the generated adaptation services need to be coupled with generated code for the UI and context services as well as integrated in an overall UI execution environment.

To address the above-described challenges, we present an integrated model-driven development approach for self-adaptive UIs. Our approach covers the modeling, transformation, and execution of self-adaptive UIs with a special focus on the concerns context management and adaptation. In particular, our integrated model-driven development approach covers the following contributions: Firstly, we introduce two domain-specific languages, called *ContextML* and *AdaptML* which support the specification of context-of-use parameters (e.g., brightness level, movement, or user's mood) and abstract UI adaptation rules that cover various adaptation dimensions (e.g., layout, navigation, or task-feature set adaptation), respectively. Additionally, we describe our approach which supports the generation of context and UI adaptation services by transforming the context model and abstract UI adaptation rules into an executable representation of the target UI execution environment. Finally, we present the integration of a rule engine in our UI execution environment for monitoring various context-of-use parameters and 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 Sect. 3, we present the integrated modeling approach for self-adaptive UIs. Section 4 deals with the implementation of our integrated transformation and execution environment. Section 5 shows the benefit and usefulness of our approach based on two case studies showing the development of self-adaptive UIs and a usability evaluation analyzing end-user satisfaction of SAUIs. Related work is presented in Sect. 6, and finally, Sect. 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 Fig. 1 and consists of three development paths.

The first development path (left side of Fig. 1) addresses the model-driven development of UIs. It 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 [28], and we already presented the realization and application of an MDUID approach for different target platforms, e.g., smartphone, desktop, and self-service systems [37,38] based on the OMG standard IFML. The first development path supports efficient development of heterogeneous UIs for different target platforms. However, on its own, it is not enough to support UI adaptation capabilities.

Therefore, we extended our existing MDUID solution architecture with parallel development paths which support model-driven context management and development of UI adaptations. In this way, the model-driven UI development path is complemented by analog development paths responsible for the context management and UI adaptation concerns. As these complementary paths are also based on the paradigm of model-driven development, the solution preserves various

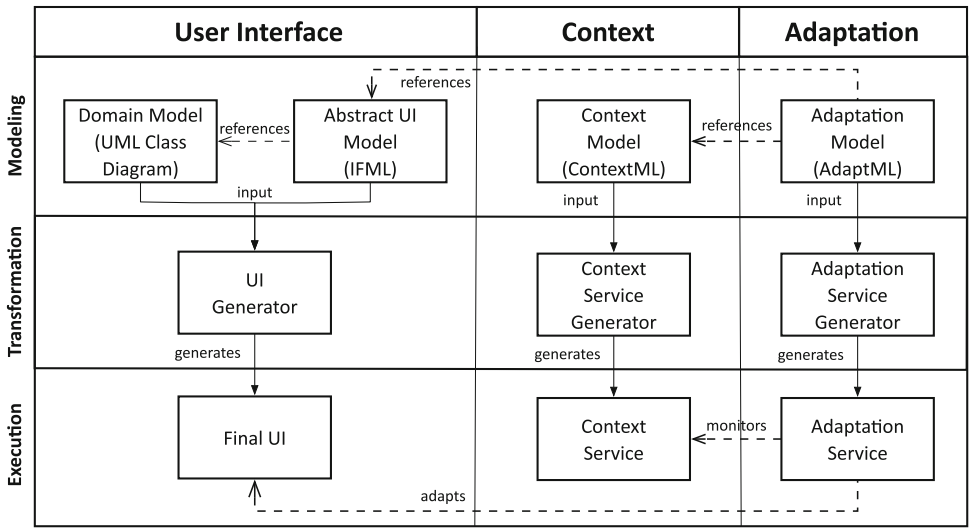


Fig. 1 Model-driven architecture for self-adaptive UIs

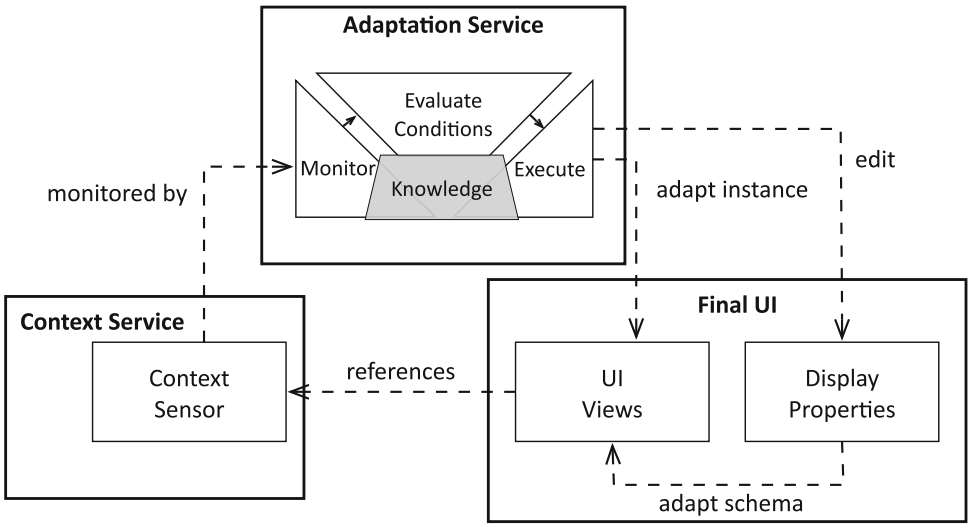


Fig. 2 Runtime perspective: architectural overview for self-adaptive UIs

advantages of model-driven software development like separation of concerns, extensibility, or maintainability.

The second development path (in the middle of Fig. 1) is responsible for characterizing the dynamically changing context-of-use parameters. A *Context Model* supports the abstract specification of heterogeneous context-of-use situations. Based on the *Context Model*, the *Context Service Generator* enables the generation of various *Context Services* which monitor context information like accelerometer, GPS, brightness, or noise level.

The model-driven UI adaptation development path is depicted on the right side of Fig. 1. In general, it supports the specification of an *Adaptation Model* in the means of abstract UI adaptation rules in alignment to the standardized abstract UI modeling language IFML. The UI adaptation rules, specified in the *Adaptation Model*, reference the *Context Model* to define the context constraints for triggering adaptation rules. The UI adaptation rules of an *Adaptation Model* also have a reference to the *Abstract UI Model* to define which UI elements are scope of an UI adaptation change. The specified *Adaptation Model* serves then as an input for the *Adaptation Service Generator* which transforms it to an *Adaptation Service*. The *Adaptation Service* is responsible for monitoring the context information provided by the *Context Service* and adapting the generated *Final UI* at runtime.

For illustrating the interplay between the generated *Final UI*, the *Context Service*, 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 aspect of UI adaptation. Figure 2 shows a detailed overview of the runtime perspective for UI adaptation containing the main components for realizing self-adaptive UIs that are able to automatically react to changes in their context-of-use. Our solution architecture for realizing such self-adaptive UIs is based on IBM's MAPE-K [18] architecture which is common in the field of self-adaptive software systems. In the follow-

ing, the specific components of our solution architecture will be described.

An important component for UI adaptation at runtime is the *Context Service*. The *Context Service* provides context information to the *Adaptation Service* based on the *Context Sensors* which are specified in the *Context Model*. The provided context information is monitored by the *Adaptation Service*. Unlike the MAPE-K loop with its analyze 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. The rules that satisfy the conditions are executed in the *Final UI*. The *Final UI* consists of two subcomponents: The *UI 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. The executed adaptation operations can modify the UI directly or edit the *Display Properties*. In general, 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 mainly serves to log and store context information data and previously triggered UI adaptations.

3 Integrated modeling environment

In this section, we present our integrated modeling environment for self-adaptive UIs. For this purpose, first, we introduce our integrated modeling environment itself in Sect. 3.1. After that, we present details about the language design and definition of our context modeling language *ContextML* in Sect. 3.2 and UI adaptation modeling language *AdaptML* in Sect. 3.3. Finally, in Sect. 3.4, we shortly present the implemented tooling for our integrated modeling environment.

3.1 Integrated modeling environment

Modeling self-adaptive UIs is a challenging task which should be supported in a systematic way, so that essential concerns like context management and UI adaptation are appropriately treated. To address this issue and reduce the complexity in development of self-adaptive UIs, we developed *ContextML* and *AdaptML* to support the context modeling and UI adaptation concerns through dedicated domain-specific languages.

Special attention is required for context management due to the complexity of capturing context information through various sensors from heterogeneous sources and monitoring dynamically changing context-of-use parameters. Therefore, in our previous work [34], we introduced the context modeling language *ContextML*. It allows to define a set of context properties and the needed context provider interfaces to capture the relevant context information. These interfaces are later on referenced as *Context Providers*. For example, the environmental light condition of a context-of-use can be captured by using a provided API for the ambient light. Also, the data types of such context properties as well as their behavior in terms of data updating have to be defined and are covered by *ContextML*.

Beside *ContextML*, we developed another domain-specific language, *AdaptML*, introduced in [39], which supports the modeling of UI adaptations through UI adaptation rules. *AdaptML* is designed as a complementary modeling language to OMG's core UI modeling language IFML and allows domain experts, for example, Web designers, to model adaptation concerns by specifying the conditions and actions for UI adaptations. To support various adaptation techniques for devising self-adaptive model-driven UIs, *AdaptML* enables specification of different categories of UI adaptation rules. Based on [26], the following main categories of UI adaptation types are supported: task-feature-set, navigation, layout, and modality adaptation. The 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. Layout adaptation deals with adaptation rules that support layout optimization like changing font size, colors, or positioning. Finally, modality adaptation characterizes changes in the interaction mode where UI modality is changed, for example, from graphical to vocal UI.

Figure 3 shows a simplified modeling example for self-adaptive UIs, where UI, context, and adaptation concerns are specified based on *IFML* [24], *ContextML*, and *AdaptML* in an integrated manner. On the top of this figure, small excerpts of the domain model in the form of a UML [25] class diagram and core UI models are depicted. There is an abstract UI model for a simplified library application based on IFML which shows the representation of three UI view containers *LoginView*, *BooksView*, and *BookDetailsView* which are connected by the navigation edges *submit* and *showDetails*. To enable the specification of data bindings in IFML, the corresponding classes from the domain model are referenced, and in our case, it is the class *Book*. For specifying the different context-of-use parameters, in the bottom left corner of Fig. 3, an exemplary context model based on *ContextML* is depicted. The shown *Context Model* contains different con-

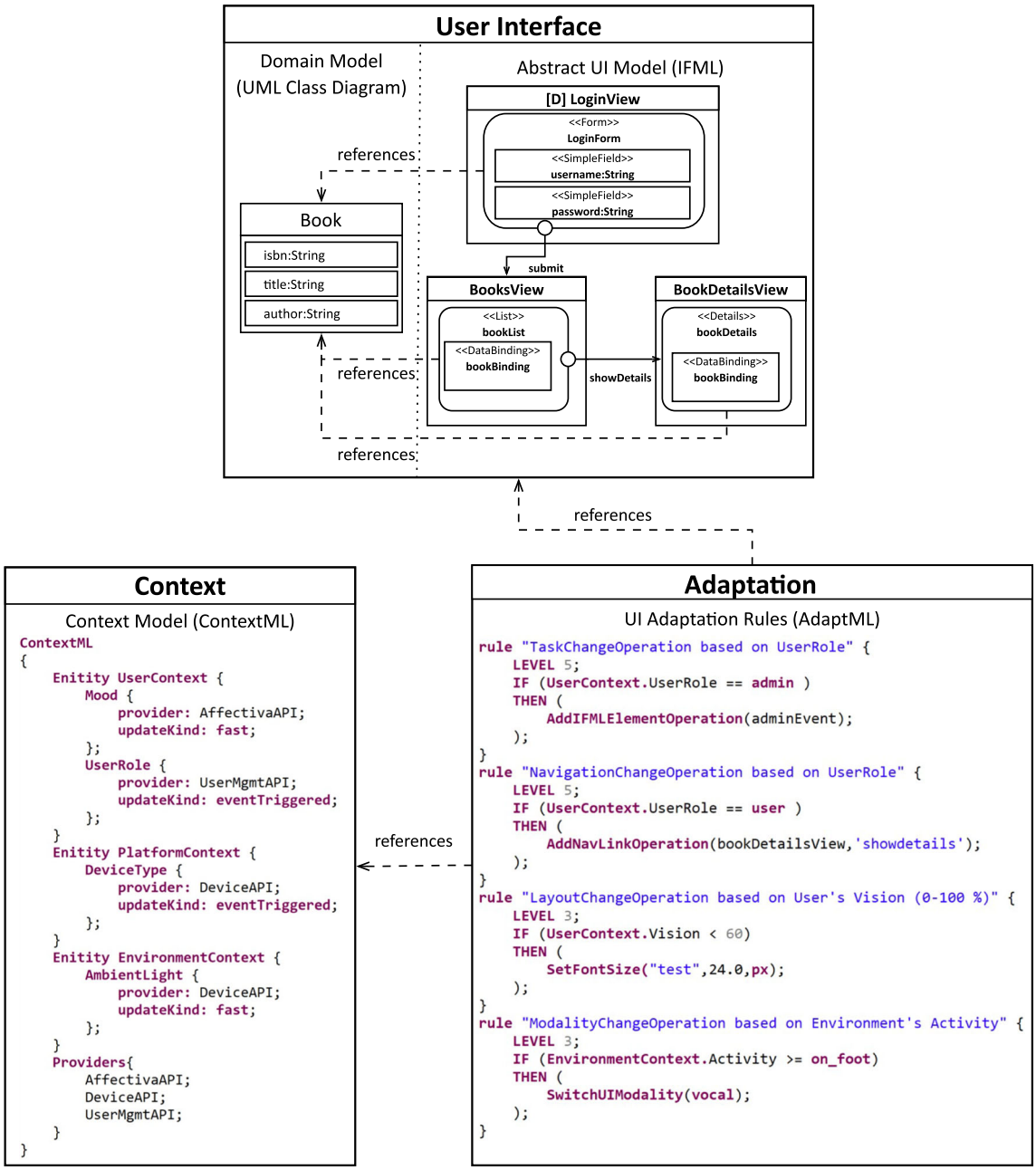


Fig. 3 Example: Integrated modeling of self-adaptive UIs

text *Entities* to characterize various context properties like *UserContext*, *PlatformContext*, or *EnvironmentContext*. A specific context *Entity* is described in detail by its context property (e.g., mood of a user or used device type) which is defined though a context provider and update type. In the example context model, for instance, the mood of the user is specified as a relevant context property which is gathered through the *AffectivaAPI* using the smartphone's camera. To support the separate specification of UI adaptation rules in addition to the IFML and context model, *AdaptML* allows to specify and bind different adaptation rules to the IFML model. The UI adaptation model, depicted on the bottom right side of Fig. 3, contains exemplary UI adaptation rules. The first adaptation rule specifies a *TaskChangeOperation* based on the user role. In this case, it is checked whether the user has the role *admin*. If this is the case, then an additional model element can be shown by using the *AddIFMLElementOperation*. More details about the available adaptation operations are provided in Sect. 3.3. The second adaptation rule is called *NavigationChangeOperation based on UserRole* and defines that the specific view *BookDetailsView* can be only reached, if a specific user context is satisfied. For defining this adap-

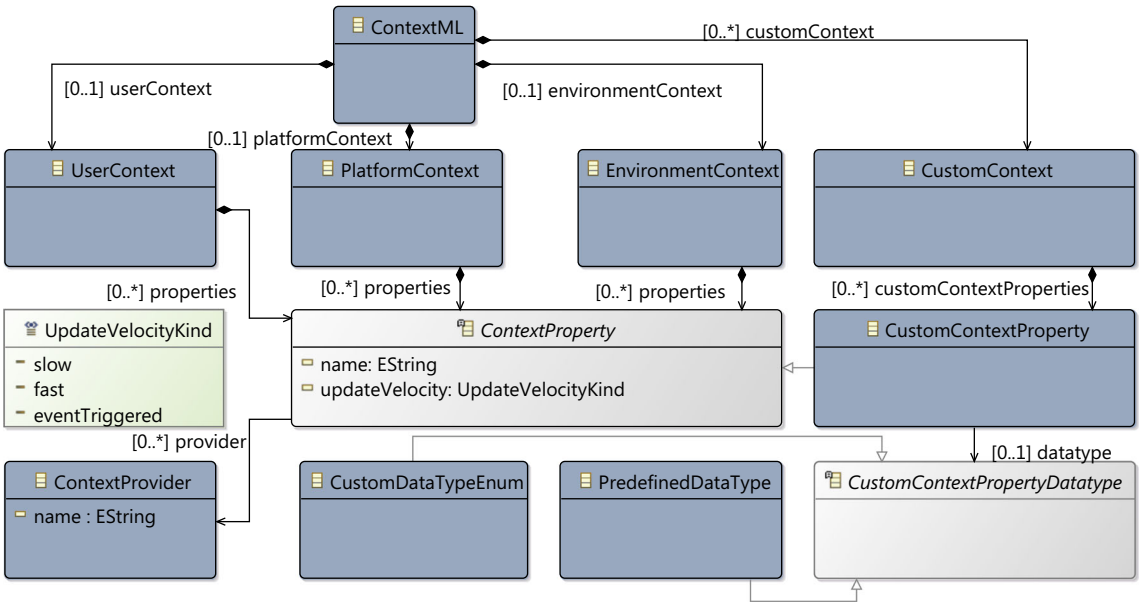


Fig. 4 *ContextML*: Context metamodel

tation behavior, *AdaptML* rules are referencing the context model where relevant contextual parameters are described and the IFML model to reference the specific UI model elements that has to be changed. In the case of our example, the user role *user* has to be satisfied, so that the *BookDetailsView* can be reached. In a similar way, various other UI adaptation rules can be specified. Analogously, the third rule is defining a *LayoutChangeOperation* to increase the font size if the user's vision is under the specified threshold value. Finally, the last adaptation rule specifies a *ModalityChangeOperation* which switches the UI modality from graphical to vocal if a movement is detected.

3.2 Context modeling with *ContextML*

For addressing the challenge *C1: Specification of contextual parameters*, we introduce and elaborate on the context modeling language *ContextML* that is designed conform to the context metamodel depicted in Fig. 4. The purpose of the context modeling language *ContextML* is to provide support for specifying various context properties and the needed context provider interfaces to capture the relevant context information needed for UI adaptation.

The root element and central class of the metamodel is the class *ContextML* that connects all parts of the metamodel. The root class *ContextML* is further specialized by specific context entities, such as *UserContext*, *PlatformContext*, *EnvironmentContext*, and *CustomContext*. Each context entity *UserContext*, *PlatformContext*, and *EnvironmentContext* has a *ContextProperty* (abstract class) which covers *name* and *updateVelocity* of a contextual parameter. Moreover, the reference to the desired *ContextProvider* is saved, which is the source of context information and is provided through a context sensor. The data type of a *ContextProperty* provided through a *ContextProvider* can be a standard type like *Integer*, *String*, or *Boolean*, but also a user-defined type is supported. The *updateVelocity* describes the way, how a single data set shall be updated. This can be "*slow*," "*fast*," or "*eventTriggered*," which is whenever a context information change occurs.

The *ContextML* metamodel is designed for a broad scope of context modeling aspects and allows adding further context entities via *CustomContext*. Each *CustomContext* can have a *CustomContextProperty* inherited by *ContextProperty*. Moreover, each *CustomContextProperty* has a *CustomContextPropertyDatatype* which can be a *CustomDataTypeEnum* or a *PredefinedDataType*.

In order to support a broad spectrum of contextual parameters and ease the work of the developers in specifying various context-of-use situations, *ContextML* comes up with a fine-grained modeling support to cover the context triplet *UserContext*, *PlatformContext*, and *EnvironmentContext*. In the following, those context categories are described in more detail.

Figure 5 depicts a refinement of *UserContext* as part of the *ContextML* metamodel. It covers a several context information related to the actual user, and each user has a unique *UserID*. Typical context properties about the user are age, gender, and language. The context property age can be specified as *AgeUserDefined* when it is gathered through a user prompt or *AgeCalculated* when it is detected through a camera and based on a recognition service such as AffectivaAPI. Beside that, the *Mood* of the user (happy, angry, surprised, etc.) and if she/he wears *Glasses* and has a *Vision* problem

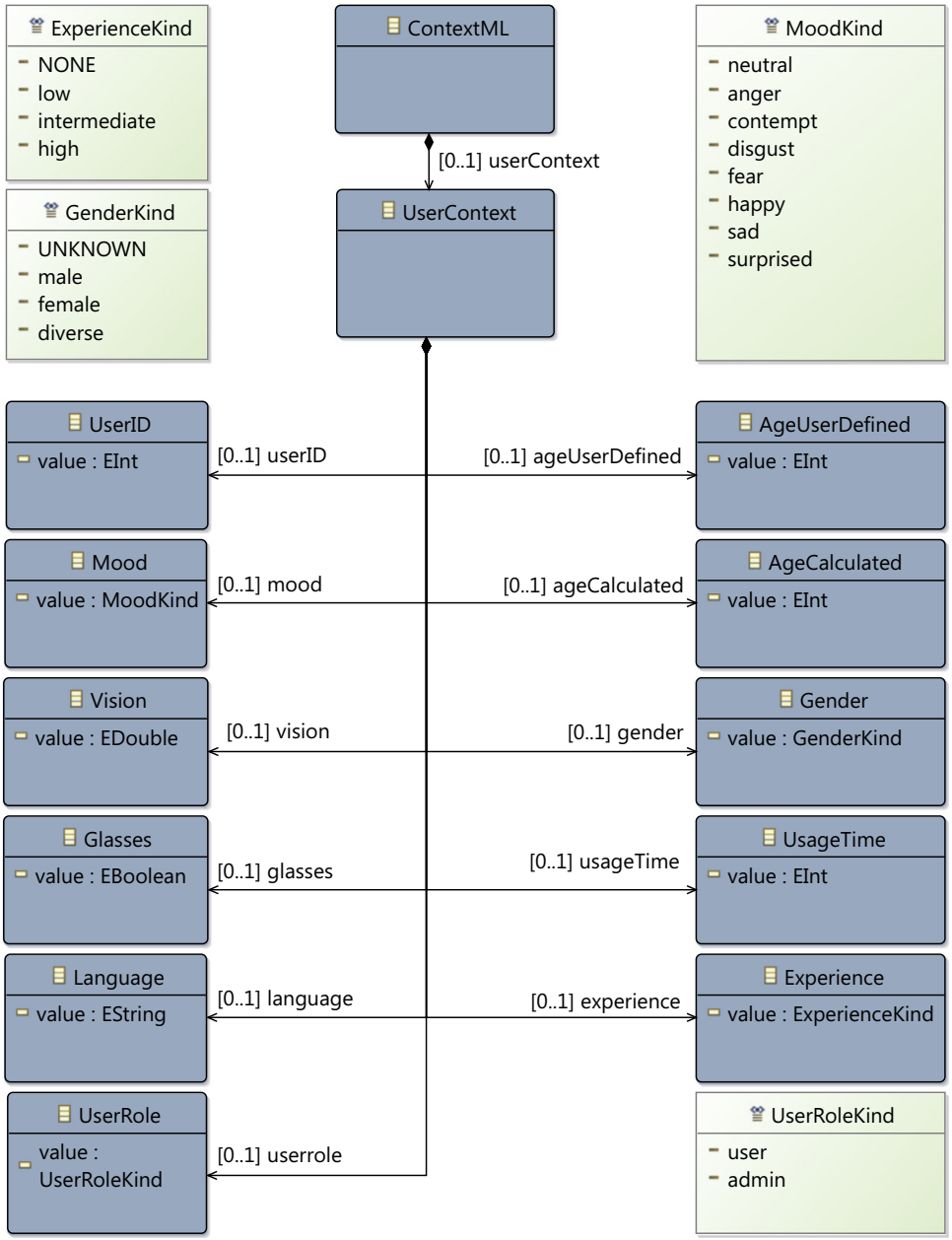


Fig. 5 *ContextML* metamodel for *UserContext*

can be also detected through a camera and face detection API. Furthermore, when a user model is specified, we can ask the users for context information about their *Experience* level or *UserRole* for interacting with the system. Lastly, we can specify and track the *UsageTime* of each user.

In a similar way, *PlatformContext* (see Fig. 6) entity covers aspects according to the execution platform that should be considered when maintaining usability of the UI. As with *UserContext*, each *PlatformContext* has a unique *PlatformID* so that the platform model is mapped to the correct execution platform. Common context properties regarding execution platform are *OSName* and *OSVersion* denoting the name of the running operating system and its version number, respectively. In addition to that, the context property *DeviceType* characterizes the device type of the execution platform, such as desktop, mobile, or tablet. Also, the specific *DeviceName* and *TimeZone* can be specified and observed as an additional context information. A pair of context properties that is essential for the UI is the screen dimension. In our metamodel, it is represented by *ScreenHeight* and *ScreenWidth* for the respective dimension height and width. These context properties are often used to scale the UI to the respective device. The most important scaling property is that for font, because UIs mostly contain text elements. As text elements play a major role for UIs, a *FontScale* is introduced. It carries the default font scale for the respective device to take into consideration when

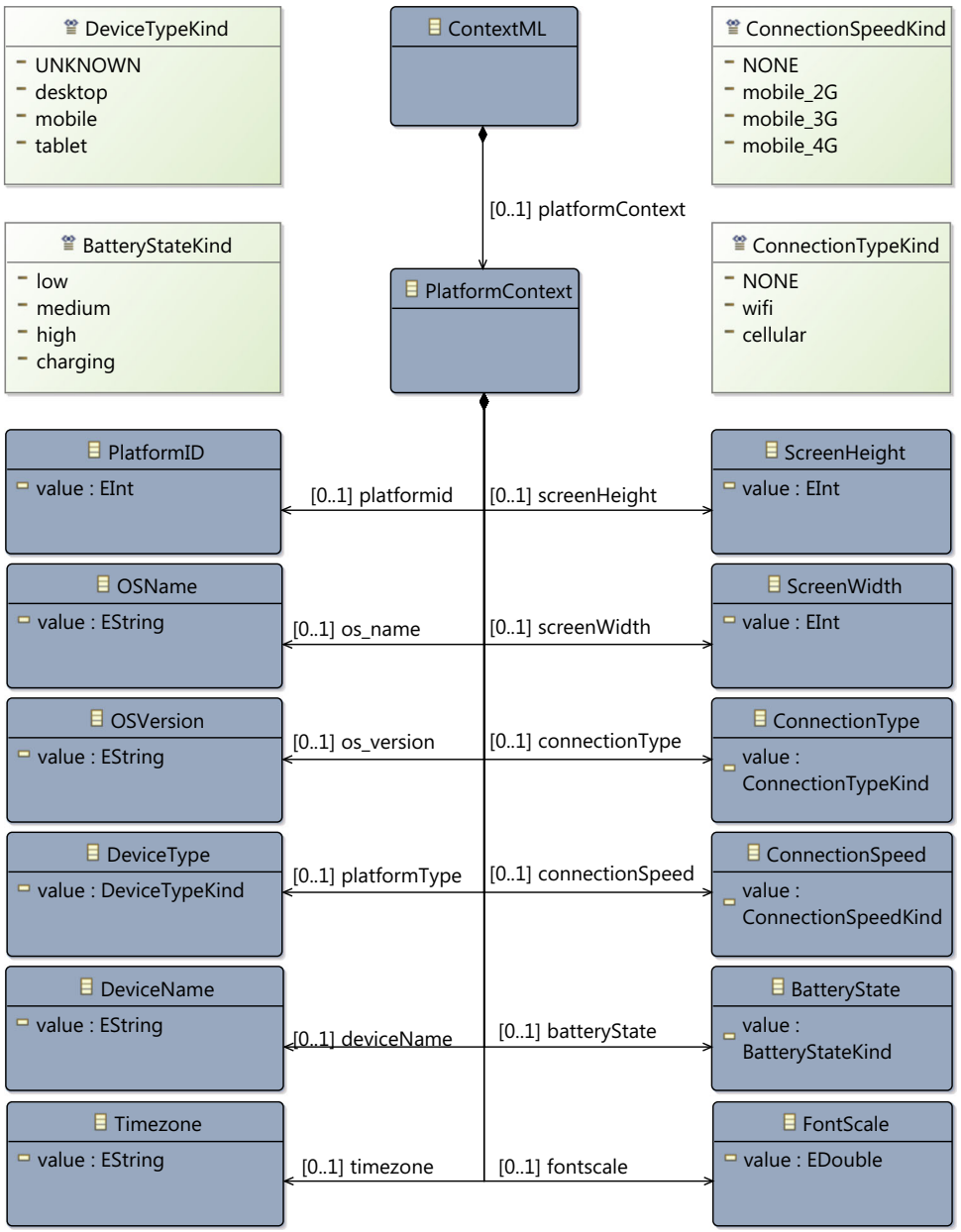


Fig. 6 *ContextML* metamodel for *PlatformContext*

modifying its value. Furthermore, dynamic platform context properties such as *ConnectionType* (Wi-Fi, cellular, etc.) and *ConnectionSpeed* (mobile_2G, mobile_3G, etc.) can be used to track the quality and speed of the internet connection. Finally, the dynamic platform context property *BatteryState* is an important context information that can influence functionality and usability of the UI.

The *EnvironmentContext* part of the *ContextML* metamodel, depicted in Fig. 7, is especially important when considering the mobile scenario. In the mobile scenario, the context parameters regarding the interaction environment can dynamically change. Several dynamic context changes and various combinations of environmental context properties are conceivable. In *ContextML*, the most relevant environmental properties are considered as *Date*, *Time*, *AmbientLight*, *NoiseLevel*, *Weather*, and *Activity*. While *Date* and *Time* constitute the temporal dimensions, *Light*, *NoiseLevel*, *Weather*, and *Activity* constitute the space dimension. As part of the space-related context properties, *Activity* is defined by an enumeration over different states of movement such as *standstill*, *on_foot*, and *on_bicycle*. Similarly, context information about the *Weather* is defined through an enumeration over different states (clear, sunny, rainy, etc.).

An excerpt of an example context model based on this metamodel is depicted in Fig. 8. It shows a set of possible context entities. For illustrating the context modeling language, exemplary entities which contain some exemplary context properties are shown. This model snippet is also used

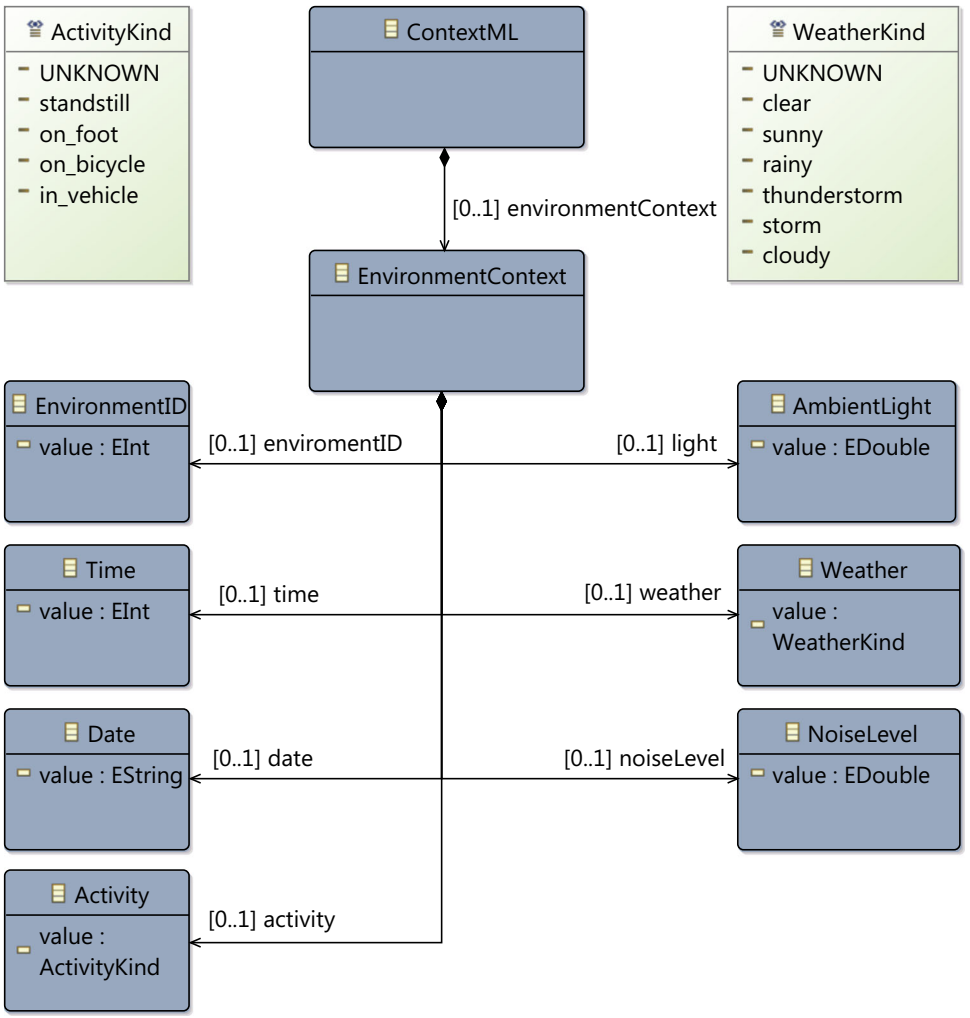


Fig. 7 *ContextML* metamodel for *EnvironmentContext*

to demonstrate the generation process in the next section. Based on the metamodel, we also created a concrete syntax for *ContextML* using *Xtext*.[1] Based on *Xtext*, we created an Eclipse plugin for context modeling which allows an easy modeling of the context, due to error highlighting and code completion. This way, the required programming knowledge and error potential is reduced. An example is displayed on the right side of Fig. 8.

3.3 Adaptation modeling with *AdaptML*

In order to support the UI adaptation modeling and address challenge *C4: Specification of UI adaptation rules*, we introduce *AdaptML*. The main purpose of *AdaptML* is to provide a dedicated modeling perspective which allows the separate specification of UI adaptation rules complementary to the UI and context model. *AdaptML* aims to support the specification of various UI adaptations by covering different adaptation techniques and reduce complexity in designing and maintaining adaptation rules.

An overview of the general structure of the *AdaptML* language is shown in Fig. 9. The root element of the metamodel is the *AdaptML* class. For the purpose of integration, the *AdaptML* class has a reference to the *ContextML* class to evaluate the context conditions and a reference to the *IFMLModel* class to define UI adaptations for specific UI elements. *AdaptML* consists of *AdaptationRule* elements which have a rule *name* and a priority *level* as attributes. The priority *level* is used as an indicator for priority to decide in which order rules are executed if more than one satisfies all conditions. A rule with higher priority level is executed before rules with lower level. Each *AdaptationRule* consists of one or more *Premise* and *AdaptationOperation* elements. The *Premise* class characterizes the condition part of a UI adaptation and consists of an abstract class *Condition* which is the base for describing simple and complex conditions. Simple conditions can be specified based on the *PrimeCondition* class which defines a concrete simple constraint on one *ContextProperty*. Therefore, *PrimeCondition* has the attributes

[1] https://eclipse.org/Xtext/.

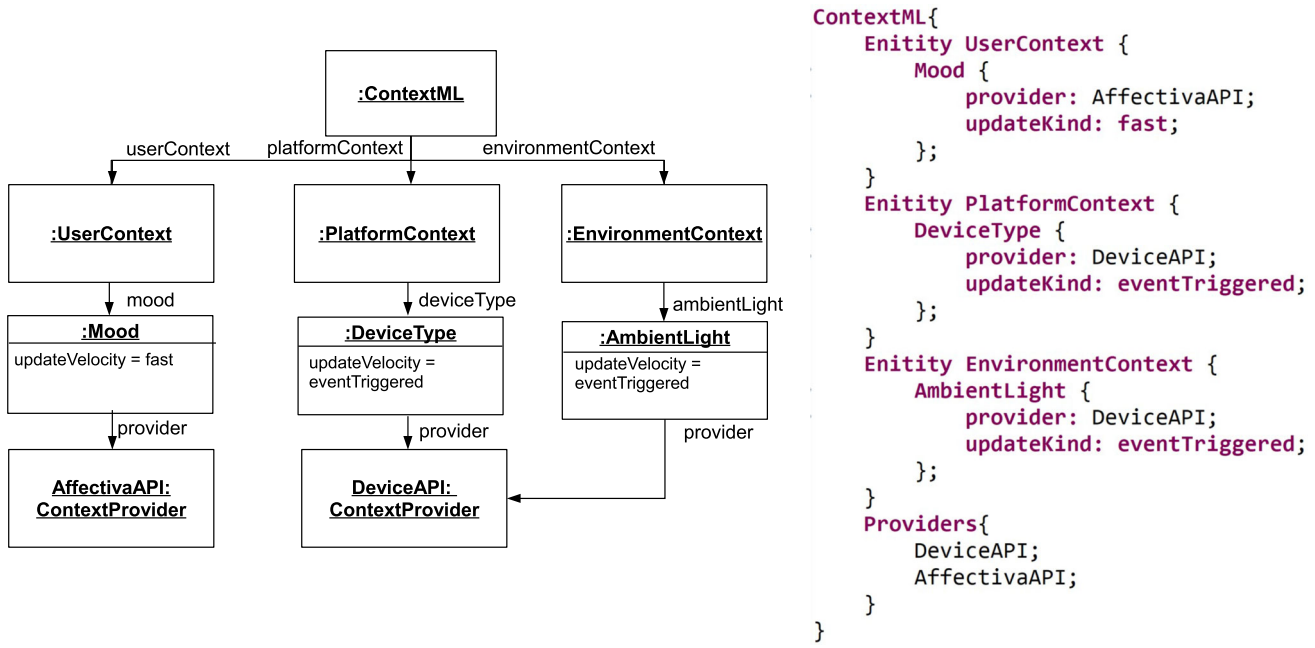


Fig. 8 Excerpt of a context model in *ContextML*, graphical (left) and textual (right) concrete syntax

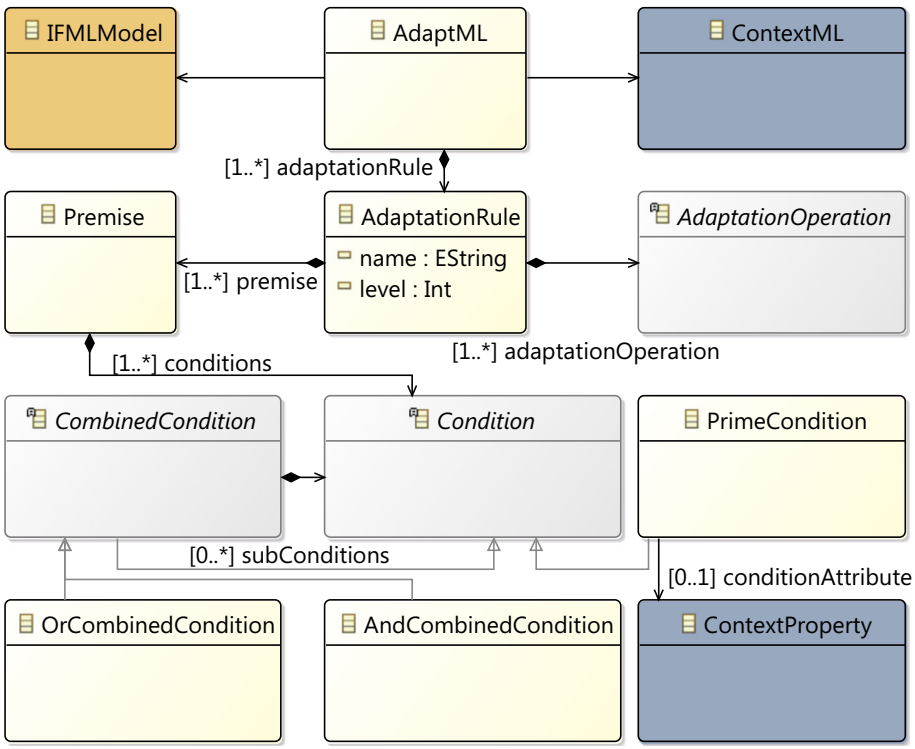


Fig. 9 *AdaptML*: Adaptation metamodel overview

operator and *value* which are needed to define a condition of a UI *AdaptationRule* expressed as a logical expression. More complex conditions can be specified through the abstract class *CombinedCondition* which allows a combination of conditional expressions concatenated by OR-operators and AND-operators. For this purpose, the subclasses *OrCombinedCondition* and *AndCombinedCondition* are defined in the *AdaptML* metamodel.

Beside the above-described conditional expressions, an *AdaptationRule* enables to specify different UI adaptation techniques that are executed if the associated conditions are satisfied at runtime. For this purpose, each *Adaptation-*

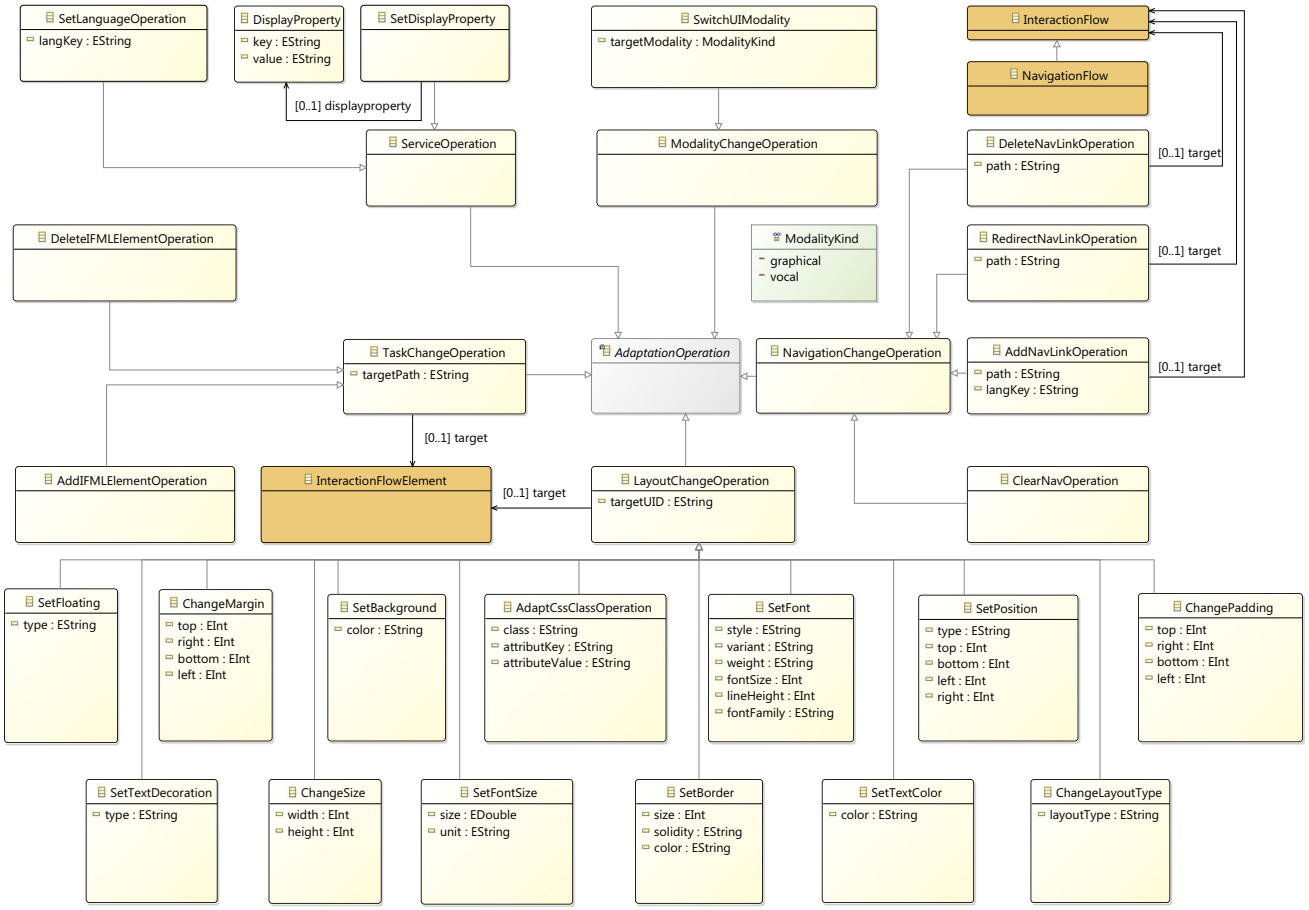


Fig. 10 *AdaptML*: Adaptation operation overview

Rule consists of one or many *AdaptationOperation* elements. As shown in Fig. 10, *AdaptML* supports different types of adaptation techniques: *TaskChangeOperation*, *NavigationChangeOperation*, *LayoutChangeOperation*, *ModalityChangeOperation*, and *ServiceOperation*.

Also, a combination of multiple adaptation techniques is possible. This is implicitly modeled by the composition relation between *AdaptationRule* and *AdaptationOperation* in *AdaptML*.

TaskChangeOperation enables the specification of UI adaptations which allow to decrease and increase the task-feature-set to provide a more minimalistic or detailed UI view upon the current context-of-use. For hiding and showing specific UI elements, *TaskChangeOperation* supports the *AddIFMLElementOperation* and *DeleteIFMLElementOperation* through a *targetPath* which enables to apply these operations on specific UI elements of the specified IFML model. For this purpose, *TaskChangeOperation* has a target reference to the IFML class *InteractionFlowElement*.

Similarly, *NavigationChangeOperation* enables to specify UI adaptations where the navigation flow of a UI can be changed based on the context-of-use. For this purpose, *NavigationChangeOperation* has the subclasses *AddNavLinkOperation*, *DeleteNavLinkOperation*, *RedirectNavLinkOperation*, and *ClearNavOperation*. The first three classes or navigation change operations support the addition, deletion, and redirection of a specific navigation edge, denoted as *NavigationFlow* in the IFML model, which is referenced through a *targetPath*. Lastly, the navigation change operation *ClearNavOperation* can be used to remove all links that are currently stored in the navigation component.

As a further UI adaptation technique, *AdaptML* supports the specification of layout changes which is characterized through the *LayoutChangeOperation* class. Although IFML in general is an abstract UI modeling language which is not directly focusing on platform-specific details like layout, we decided to incorporate layout change operations in *AdaptML* as we see a higher potential and flexibility for UI adaptations through this possibility. The *LayoutChangeOperation(s)* are mainly inspired by commonly used Cascading Style Sheet (CSS)[2] properties. The main idea is to ease the developers work in specifying layout change operations by

[2] https://www.w3.org/Style/CSS/Overview.en.html.

reusing a common standard and integrating it into our modeling approach as it is also the case with IFML. Similar to the *targetPath* attribute of the *TaskChangeOperation(s)*, the *LayoutChangeOperation* class contains the *targetUID* attribute identifying a UI element uniquely. This way, different *LayoutChangeOperation(s)* can be applied to specific UI elements which are contained in the IFML class *InteractionFlowElement*. Typical layout change operations commonly known from CSS are, for example, *SetFontSize*, *SetPosition*, or *SetTextColor*. Furthermore, we added the *ChangeLayoutType* operation to support a switch between different layout types like grid or linear layout. Finally, we also added a generic *AdaptCSSClassOperation*. With this operation, it is possible to set fine granular style properties for a specific class of UI elements. The *class* attribute in *AdaptCSSClassOperation* is equal to a CSS class, while the *attributeKey* is intended to be a CSS property and the *attributeValue* a valid value to that key. The *AdaptCSSClassOperation* is not intended to be used in general, but represents an alternative solution in cases where the specific *LayoutChangeOperation* is not implemented in *AdaptML*.

In addition to *LayoutChangeOperation*, *AdaptML* also supports the specification of basic modality changes for the interaction with the UI. For this purpose, the *ModalityChangeOperation* class is provided which enables to switch the interaction modality type via the *SwitchUIModality* class between graphical and vocal user interface.

As a last type of adaptation technique, *AdaptML* supports a *ServiceOperation* in the target language of the UI. In our approach, reusable predefined Angular[3] services were provided to support UI adaptations for the Web platform. The definition of these services enables to use them later on in the rule specification. A *ServiceOperation* is defined by its name and relative location to the *Services* folder of the Angular implementation. A *ServiceOperation* can contain interfaces to functions. *ServiceOperation(s)* are helpful to specify UI changes that affect bigger parts or a group of UI elements. For this purpose, *ServiceOperation* has a subclass *SetDisplayProperty* which takes a *DisplayProperty* object as input and enables a group of changes in the UI. As an example, *AdaptML* comes up with a predefined *SetLanguageOperation* which supports internationalization of the UI language. Therefore, each text on the UI is represented by a language key which is automatically set to the detected language on the used device.

A set of exemplary UI adaptation rules based on *AdaptML* are shown in Fig. 11.

The first UI adaptation rule specifies a *TaskChangeOperation* based on user's mood. If a "sad" mood is detected through the face detection API, a "helpEvent" is added through the *AddIFMLElementOperation* to provide, for instance, a textual help for the user. In the second adaptation rule, a *LayoutChangeOperation* based on the environmental light condition is specified. When the light condition is under a certain threshold value, the contrast of the UI is increased through the *adaptCSSClass* operation. Similarly, the third adaptation rule specifies a *LayoutChangeOperation* based on user's experience, while the fourth rule encodes a *ModalityChangeOperation* to trigger a switch into the vocal UI if a movement is detected. Finally, a language adaptation is shown in the last adaptation rule where the UI language is switched based on the user's currently used language using the *ServiceOperation*.

```
rule "TaskChangeOperation based on User's Mood"{
    LEVEL 1;
    IF(UserContext.Mood == sad) THEN
    (
        AddIFMLElementOperation(helpEvent);
    );
}
rule "LayoutChangeOperation based on Environment's Light (0-100 %)"{
    LEVEL 2;
    IF(EnvironmentContext.AmbientLight < 100) THEN
    (
        AdaptCssClassOperation("container","filter","contrast(0.5)");
    );
}
rule "LayoutChangeOperation based on User's Experience"{
    LEVEL 1;
    IF(UserContext.Experience <= low ) THEN
    (
        ChangeLayoutType("sidebar-navbar", grid);
    );
}
rule "ModalityChangeOperation based Environment's Activity"{
    LEVEL 3;
    IF(EnvironmentContext.Activity >= on_foot ) THEN
    (
        SwitchUIModality(vocal);
    );
}
rule "ServiceOperation based on User's Language"{
    LEVEL 1;
    IF(UserContext.Language === de ) THEN(
        SetLanguageOperation("dede");
    );
}
```

Fig. 11 Exemplary UI adaptation rules based on *AdaptML*

In summary, *AdaptML* supports integrated modeling and maintenance of UI adaptation rules complementary to the core UI and context model. For this purpose, it enables the specification of different adaptation techniques which can be combined to reach complex means of UI adaptations.

3.4 Integrated modeling workbench

For supporting the modeling task of self-adaptive UIs through an adequate tooling, we have developed an integrated modeling environment in terms of an Eclipse plugin. Figure 12 shows a screenshot from our modeling workbench which provides three separated modeling views for the essential concerns UI modeling, context modeling, and adaptation modeling. The UI modeling view (left side of Fig. 12) is based on the open source IFML editor Eclipse plugin[4] which is integrated into our modeling workbench to support the

[3] https://angular.io.

[4] https://ifml.github.io.

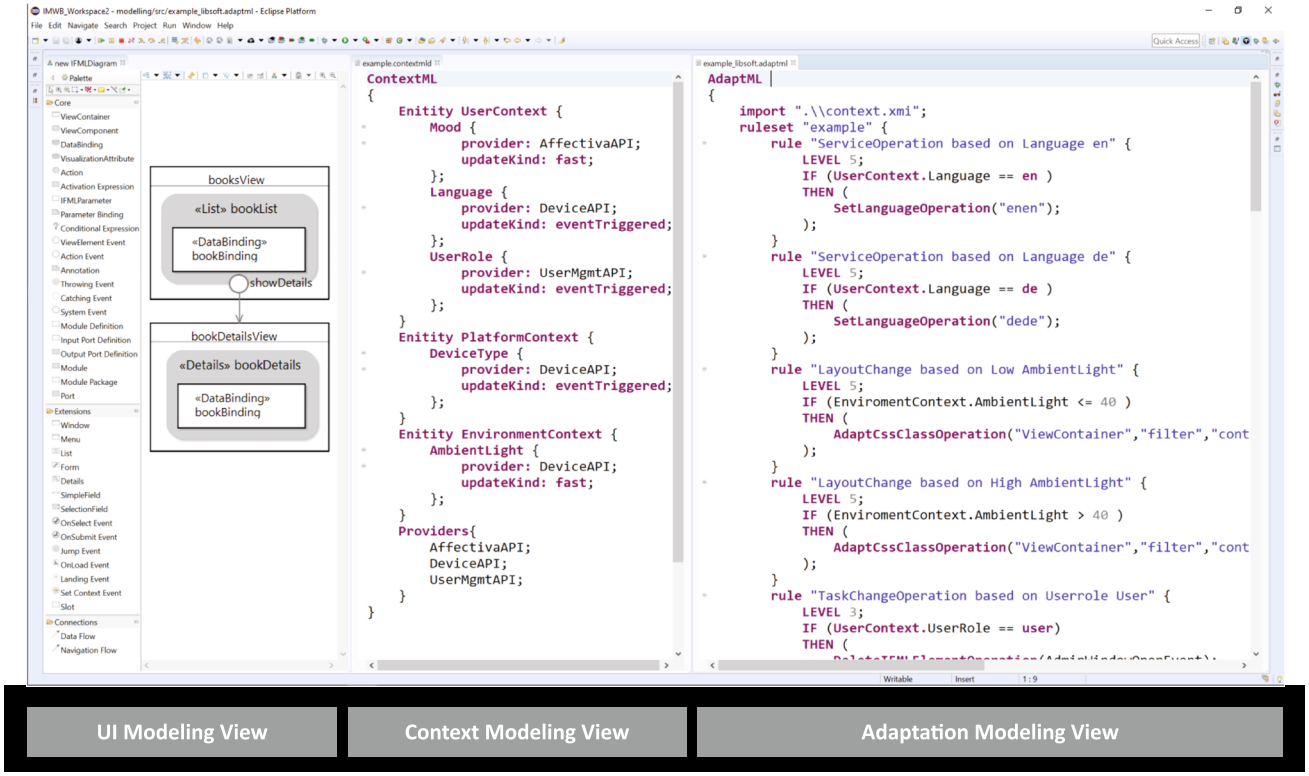


Fig. 12 Integrated modeling workbench for self-adaptive UIs

specification of core UI aspects as described in the previous Sect. 3.1. The context modeling view based on *ContextML* is depicted in the middle of the screenshot and allows the specification of various context-of-use situations as presented in Sect. 3.2. Finally, the adaptation modeling view based on *AdaptML* is shown in the right side of the screenshot and supports the specification of UI adaptation rules as described in Sect. 3.3.

4 Integrated transformation and execution environment

In order to support the utilization of our modeling and development approach for devising self-adaptive UIs for different target platforms like mobile or desktop, we implemented a code generator for self-adaptive UIs (*SAUI-Generator*). Figure 13 shows the overall architecture of the *SAUI-Generator*.

It consists of a main generator, *Generator Core*, and three subgenerators *UI Generator*, *Context Service Generator*, and *Adaptation Service Generator*. The *Generator Core* gets as input the *IFML* and *Domain Model* as well as the *Context* and *Adaptation Model* which were specified based on the presented integrated modeling workbench. The models are then delegated to the corresponding subgenerators. Based on the *IFML* and *Domain Model*, the subgenerator

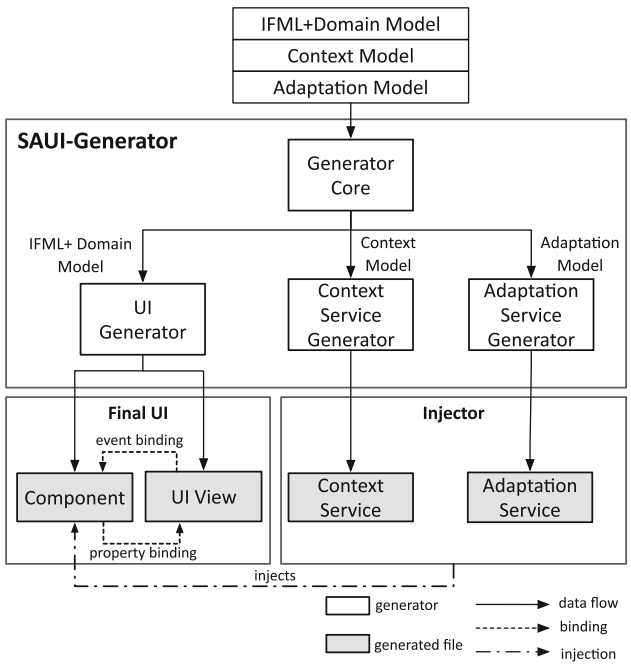


Fig. 13 Architecture of the self-adaptive user interface generator

UI Generator automatically creates the *Final UI* as Angular *Components* and *UI Views*. Furthermore, based on the specified *Context* and *Adaptation Model*, the *Context* and

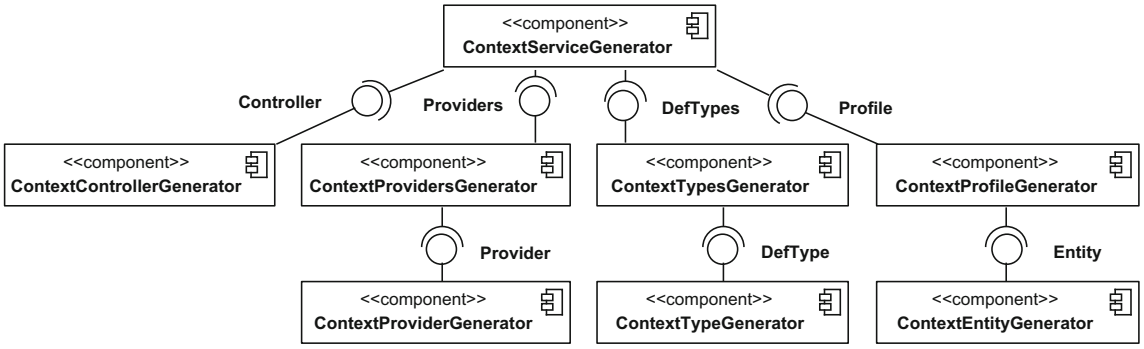


Fig. 14 Component diagram of *Context Service Generator*

Adaptation Services are generated, respectively. The generated services are injected into the *Component* element of the *Final UI* by using the Angular *Injector*[5] technique. The main idea of the generation process relies on the idea of model-to-text (M2T) transformations. As the generation of the *Final UIs* is a common task in existing model-driven UI development approaches (see, for example, [37]), in the following, we focus on and describe the implementation of the generation process responsible for automatically creating the *Context* and *Adaptation Services*. The interplay of those generated services within the execution environment is essential to support UI adaptation at runtime.

4.1 Context service generation and context monitoring at runtime

To address the challenges C2 and C3 from the Introduction section, we describe the process of generating *Context Services* represented as Angular code in the form of Typescript. The *Context ServiceGenerator* gets as input the previously mentioned *Context Model* which is based on *ContextML*. The structure of the *Context Service Generator* is shown in Fig. 14. It has a main generator that splits the generation into four kinds of files that will be generated: *ContextControllerGenerator*, *ContextProvidersGenerator*, *ContextTypesGenerator*, and *ContextProfileGenerator*. Our *Context Service Generator* is a template-based code generator that is implemented with *Xtend*.[6]

First, the *ContextServiceGenerator* invokes the *ContextControllerGenerator* which generates the main Angular service that connects and controls all the other parts. The generated *Context Controller* contains subscriptions to context properties, which push changed data automatically to the subscriber based on the *RxJS observer pattern*.[7] Furthermore, it contains timers for the properties which are not updated in an event-based manner.

The *ContextProvidersGenerator* invokes the *ContextProviderGenerator* for each provider that is listed in the *Context Model*. This creates a folder with all provider files. Each file contains standard imports and used *DefTypes*. The business logic code for controlling and managing of sensor sources like APIs or libraries has to be inserted manually. This is due to the very individual structure of numerous interfaces. Those can be fairly easy to use, like standard HTML5 APIs,[8] but can be individual and more complex as well, like the Affectiva SDK for emotion recognition.

The *ContextTypesGenerator* invokes similar to the *ContextProvidersGenerator* the *ContextTypeGenerator* for each user-defined *DefType*. This creates a folder with type files that are imported by the providers. Each file contains the *Enums* defined in the *Context Model*.

The last generator component is the *ContextProfileGenerator* that creates a central context data profile file and invokes the *ContextEntityGenerator* for each declared entity in the *Context Model*. This creates a file for each entity which contains all the defined properties and the corresponding *getter* and *setter* methods. The generated context service files are injected into the Angular UI framework as modular components.

An example for the generation of a *Context Provider* is depicted in Fig. 23 (see "Appendix"). The code excerpt depicted in Fig. 23 represents on the left side the Xtend template for generating a specific context provider for capturing the ambient light level through a sensor library. On the right side of Fig. 23, the generated code for the *AmbientLight* provider is illustrated. The code of the generated context provider is responsible for monitoring the environmental lighting condition at runtime by using the *AmbientLightAPI*.

At runtime, the generated *Context Service* works as a background service that can be used by any Web application based

[5] https://angular.io/api/core/Injector.

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

[7] https://github.com/Reactive-Extensions/RxJS.

[8] https://www.w3.org/2009/dap/.

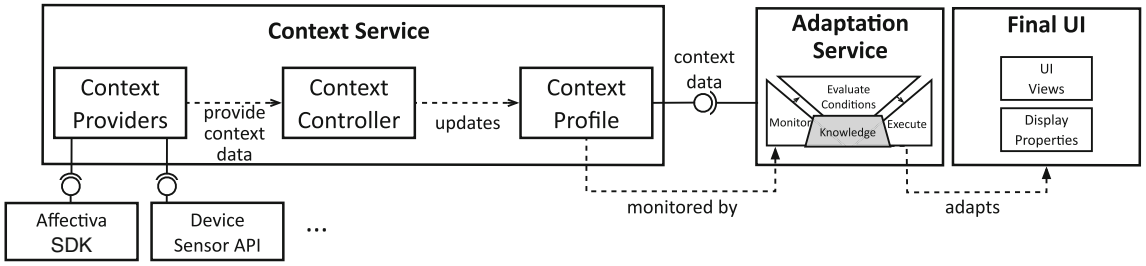


Fig. 15 Overview on *Context Service* at runtime

on the *Angular* framework. A runtime system overview is depicted in Fig. 15.

Depending on the defined update types of the context properties, the *Context Providers* either access the information event-based or triggered by the timer of the *Context Controller*. Through the subjects of the observer pattern, new data are directly pushed to the subscriptions of the controller. At the same time, the corresponding property is updated in the *Context Profile*. Based on the provided context information data through the *Context Service*, an *Adaptation Service* is able to dynamically monitor context information and adapt the *Final UI*.

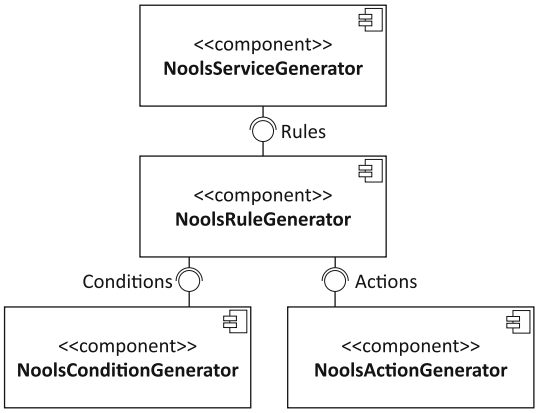


Fig. 16 Structure of the adaptation service generator

4.2 Adaptation service generation and runtime UI adaptation

For addressing the challenges C5 and C6 from the Introduction section, we describe the generation process of an *Adaptation Service* and how it is utilized at runtime to support UI adaptation. The goal of the *Adaptation Service Generator* is the automated creation of an Angular service that allows UI adaptation at runtime. The adaptations to the UI are expressed, as previously introduced, in a rule-based form using *AdaptML*. Based on this input file, the *Adaptation Service Generator* generates an Angular service containing the JavaScript rule engine *Nools*.[9] *Nools* is an efficient RETE-based rule engine written in JavaScript and provides an API for specifying facts and rules. The *Adaptation Service Generator*, which is synonymously called as *NoolsServiceGenerator* in this work due to the name of the used rule engine, is implemented with Xtend and receives the UI adaptation rules as input. Structurally, as shown in Fig. 16, it consists of the components *NoolsServiceGenerator*, *NoolsRuleGenerator*, *NoolsConditionGenerator*, and *NoolsActionGenerator*. These components are responsible for creating an injectable Angular service for monitoring the context model and executing adaptation operations.

The base structure of the Angular service, generated by the *NoolsServiceGenerator*, consists of the required Angular 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 based on *AdaptML*. 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 of the *Adaptation Service* 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 *AdaptML* rule specification. 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 (see Fig. 10), 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. Additionally, the *NoolsActionGenerator* gets as input parameter the mapping of services and functions defined in the abstract UI adaptation rule specification. However, there is a defined set of actions. If the action element is unknown, no code is created. This means, if there are new possible actions added to

[9] http://noolsjs.com/.

the schema definition, they also need to be implemented in the *NoolsActionGenerator*.

An example for the generation of a *Nools Adaptation Service* is depicted in Fig. 24 (see "Appendix"). The code excerpt depicted in Fig. 24 represents on the left side the Xtend template for generating a *Nools Adaptation Service*. On the right side of Fig. 24, the generated code for the *Nools Adaptation Service* is illustrated, which is at runtime responsible for executing the UI adaptations.

Considering the runtime perspective, we have the components *Adaptation Service*, *Final UI*, and *Context Service*. The *Final UI* is generated by the subgenerator *UI Generator*. Its Angular *UI View* consists of an HTML template, which is used to render the UI in the browser and an Angular *Component*, which is implemented in TypeScript and manages the *UI View*. Likewise, the *Adaptation Service* is generated as Angular 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 Service*. Similar to the context service generation approach, the *Adaptation Service* generation approach uses code injection to integrate the generated *Nools Adaptation Services* into the *Nools* rule engine. The *Nools* rule engine again was integrated into our Angular UI framework architecture to realize UI adaptation capabilities for the *Final 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 of the *Final UI* on instance level, JQuery[10] is used to directly manipulate the DOM tree of the *UI View*. Changes only affect the current *UI View* element and do not persist in other *UI views*. When changing the schema for a group of *UI View* elements in the *Display Properties*, the adaptation affects the properties of all *UI View* elements of this type. This also includes instances of this *UI View* element type on subsequently visited *UI Views*. This is done by binding the layout class of the *UI View* elements of this type, represented by CSS classes, to the properties stored within the *Display Properties*.

5 Evaluation

In this section, we demonstrate potentials and limits of our integrated model-driven engineering approach for self-adaptive user interfaces on the basis of two case studies and a usability evaluation study. The first case study, presented in 5.1, deals with a cross-device library Web application for which we devised self-adaptive UIs. The second case study deals with an e-mail client application with UI adaptation capabilities and is described in Sect. 5.2. Moreover, based on the last application scenario, we have conducted a usability experiment to evaluate end-user satisfaction of UI adaptation features. Main results of this usability evaluation are presented in Sect. 5.3. Finally, a discussion of the main potentials and limits of our engineering approach as well regarding the usability evaluation is presented in 5.4.

5.1 Library application with self-adaptive UI

The case study setting is based on an example scenario which is derived from the university library management domain (see Fig. 17). 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/he can begin an interaction using one channel (search and reserve a book with her laptop at home), modify the book reservation on her way using a mobile channel, and finalize the book lending process at the university library via self-checkout terminal or at the staff desk. In the above-described example scenario, each channel has its own special context-of-use and eventually, the contextual parameters regarding user, platform, and environment can dynamically change. Figure 18 shows a concrete context-of-use (*CoU*) change from *CoU2* to *CoU4* (compare Fig. 17). The depicted context-of-use object model excerpts in Fig. 18 illustrate how different contextual parameters regarding user, platform, and environment change. Already a small set of contextual parameters can highly influence the usability of the UI as lots of context situations can occur if the context-of-use parameters dynamically 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.

For utilizing our integrated model-driven development approach in the case study setting, an *IFML Model*, a *Domain Model*, a *Context Model*, and an *Adaptation Model* with a set of UI adaptation rules were created as described in Sect. 3. Using our *SAUI-Generator*, the specified models were transformed to final user interfaces including the generated code for context and adaptation services.

Exemplary screenshots of the resulting self-adaptive UI are depicted in Fig. 19. According to the monitored context information for *CoU2*, the layout for the UI is optimized for

[10] https://jquery.com.

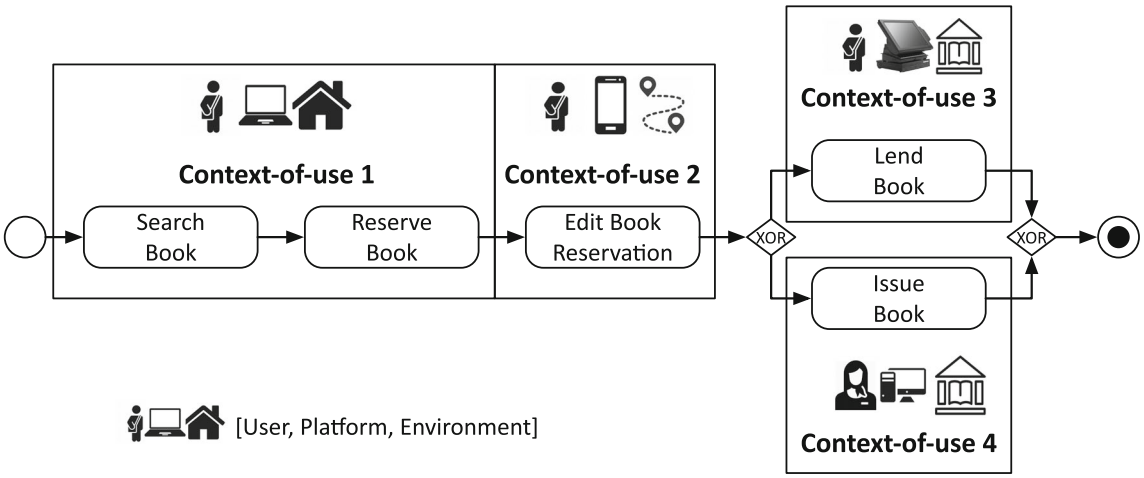


Fig. 17 Example scenario: UIs in dynamically changing context-of-use situations

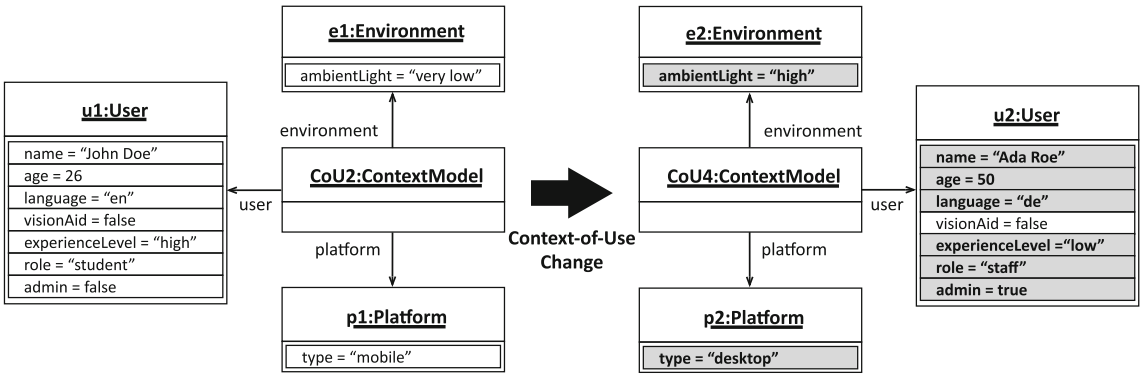


Fig. 18 Library application: context-of-use object model excerpts

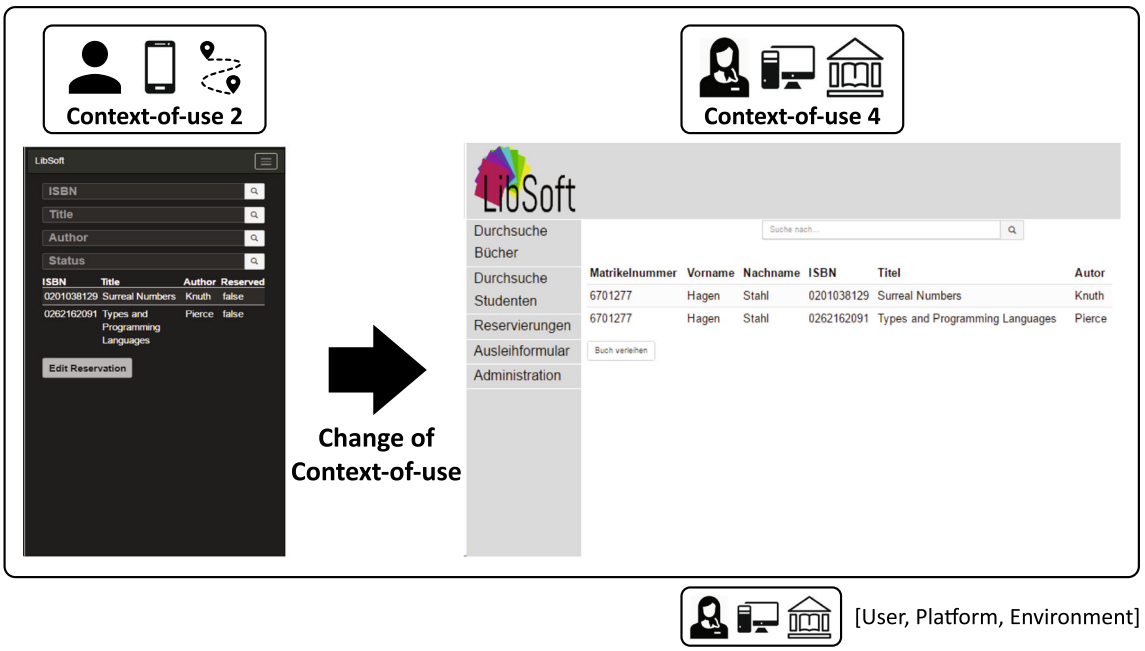


Fig. 19 Library application: UI adaptations according to different contexts-of-use

a mobile device used in a darker environment, because the user John is editing his book reservation while travelling to the library, and it is already quite dark outside (see left side of Fig. 19). 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 John. As John is recognized as an experienced user with the application (based on his usage time), he 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, Ada Roe, uses her desktop computer to issue the book to John. Additionally, to the functionalities and functions available to staff members, Ada is provided with a link to the administration interface, because she 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 she just started using LibSoft and is, therefore, not yet experienced with the application. As the location is a well-lit library, the default brightness level is shown on the screen of the desktop computer.

The case study demonstrates the benefit of our approach for supporting the development of self-adaptive UIs. By using our integrated modeling workbench and the corresponding *SAUI-Generator*, we were able to model and generate self-adaptive UIs. To sum up, our solution approach addresses the introduced challenges C1–C6 as it provides a systematic and integrated way for developing self-adaptive UIs.

5.2 E-mail application with self-adaptive UI

E-mail applications are one of the most recurrently used applications on mobile devices. People read and write mails while commuting to work, before going to sleep, walking or watching TV, or doing different other activities. As various dynamically changing context-of-use situations are faced when using such an e-mail application, we decided to develop an e-mail application with UI adaptation capabilities. Therefore, we used again our integrated modeling environment for specifying the UI, context, and adaptation concerns. The specified input models were given as input to our *SAUI-Generator* to generate the views, context, and adaptation services for the self-adaptive UI of the e-mail application.

Figure 20 depicts an exemplary sequence of context changes and how the UI of the e-mail application adapts to the changed context in each case.

Each state is a pair of the self-adaptive UI, depicted as a screenshot of the e-mail application, and the current context as experienced by the user. For our simplified example, the

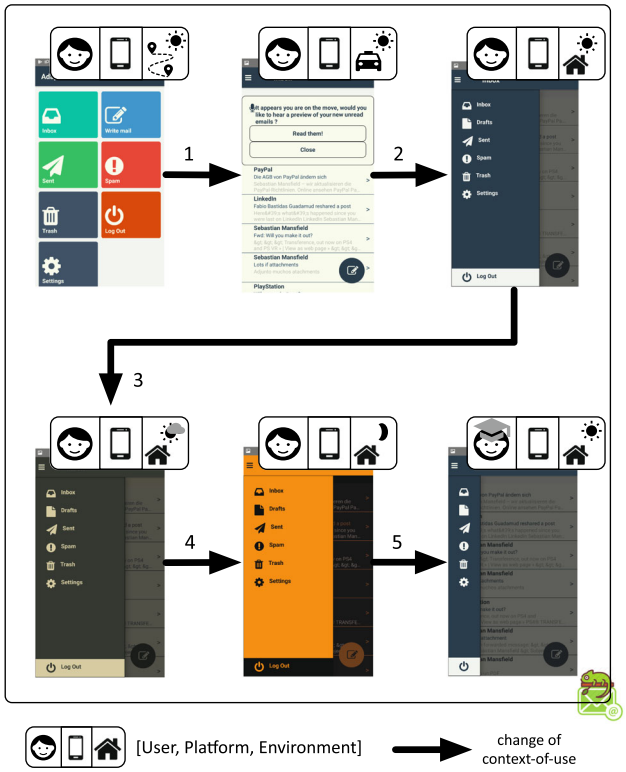


Fig. 20 E-mail application: UI adaptations according to different context changes

context is reduced to three components: (i) if the user is on the move, in a moving vehicle, or immobile (and probably at home), (ii) if the brightness level (ambientLight) is high (sunny), low (cloudy), or very low (nighttime), and finally, (iii) if the user is a novice or experienced user, based on a threshold value of usage time.

The first state (left upper corner in Fig. 20) represents a novice user on the move and experiencing high brightness levels (ambient light). The corresponding self-adaptive UI recognizes the context properties *movement* and *ambientLight* and uses a grid layout to simplify haptic interaction. Figure 21 shows exactly this change *CoU1* to *CoU2* (compare Fig. 20) using an object diagram. The depicted context-of-use object model excerpt in Fig. 21 illustrates how different contextual parameters regarding user, platform, and environment change.

In response to the context change (depicted in Fig. 20 as labeled arrows—in this case with Label 1) leading to a state where the user is now in a moving vehicle, the UI switches its modality to audio-based interaction, offering to read new e-mails aloud and enabling control of the application via audio commands. When the user is immobile for some time (and can be assumed to be seated in a building—see Label 2), the UI responds by reverting to standard haptic-based modality and additionally uses a list of icons instead of a grid for more efficient screen space usage. The next two context changes

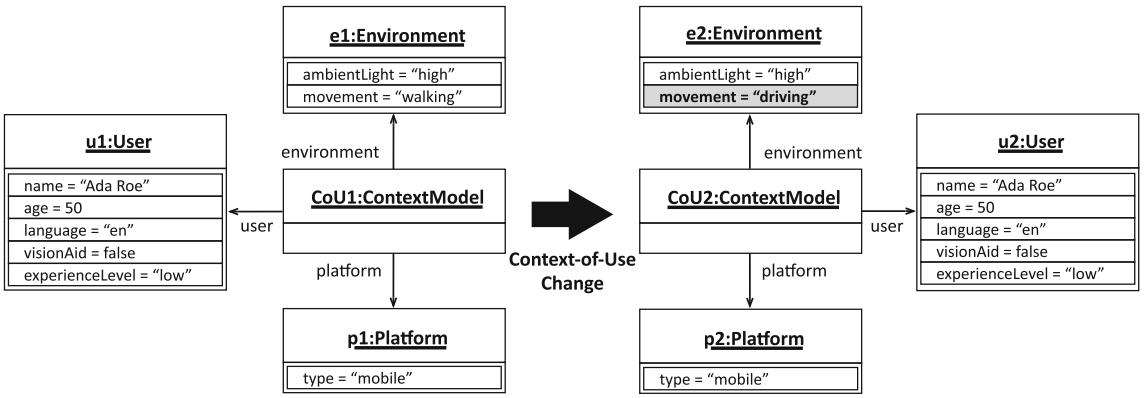


Fig. 21 E-mail application: context-of-use object model excerpts

(Label 3 and 4) represent changes in brightness level to low brightness and nighttime. The UI responds to low brightness levels by dimming the screen and using sepia tones instead of white/black and to nighttime by inverting the color scheme. The final context change (Label 5) is triggered when the user passes a certain usage-time threshold. The UI assumes that the user must now be accustomed enough to the icons and saves screen space by removing the explanatory labels for each icon.

As the second case study illustrates, the integrated modeling environment and the implemented *SAUI-Generator* allow the modeling and transformation of self-adaptive UIs. The identified context changes and the UI adaptations in action demonstrate that the generated self-adaptive UIs are able to continuously monitor their context-of-use parameters and automatically adapt the UI at runtime.

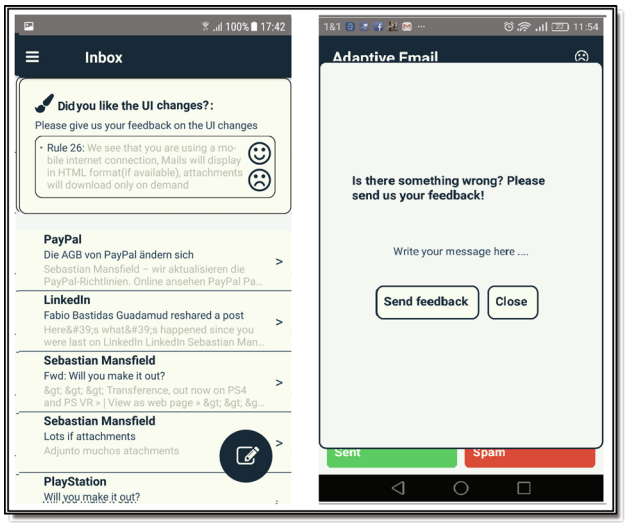


Fig. 22 Feedback prompts

5.3 Usability evaluation

With regard to self-adaptive user interfaces, usability evaluation of specific UI adaptations is still a challenging task, especially in the ubiquitous domain of mobile UI platforms, where dynamically changing context-of-use situations are usual. In the past, classical usability evaluation methods like usability tests, interviews, or cognitive walk-throughs were applied to evaluate the usability of self-adaptive UIs [33]. However, these methods are not sufficient for a proper evaluation of dynamically changing UI adaptation features at runtime. The reason is that these methods mostly focus on *a posteriori* analysis techniques. However, the acceptance of each UI adaptation feature should be evaluated at the very moment and in context-of-use when the adaptation is triggered at runtime. To address this issue, in our previous work [36], we introduced an on-the-fly (OTF) usability evaluation approach that integrates UI adaptation features and a user feedback mechanism into a mobile app. The developed solution enables us to continuously track various context information data and collect user feedback, e.g., whether the users like or dislike the triggered UI adaptations. Figure 22 illustrates how we integrated a feedback prompt into the adaptive e-mail app from the last described application scenario.

The left screenshot in Fig. 22 shows how the feedback prompt was placed in the mail app. On the top of the screen, the feedback prompt is shown whenever context changes were detected that lead to UI adaptations. The triggered UI adaptations are explained in the feedback prompt, and the user is able to provide feedback by clicking the positive or negative smiley indicating whether the user liked the UI adaptation or not. In some cases, for example, when the user is in a sad mood and the app detects this via camera, a feedback prompt in the form of a text field appears (see right screenshot in Fig. 22) that allows the users to provide more detailed feedback.

During the usability experiment, the developed app was used for one week by 23 participants. All users were made aware of the fact that their interaction with the app would

be closely monitored (e.g., using facial recognition). During the usability experiment, various data about the users and their usage context, while feedback was given, have been collected to evaluate the usability of the UI adaptation features in detail. In the following, we shortly describe some of the collected data to show the potential of our usability evaluation solution. During the conduction of the experiment, there were 104404 detected context changes from all devices. Of these, only 37465 triggered an adaptation by the rule engine. However, users gave feedback on the adaptation rules in only 663 cases. Every time an adaptation rule received feedback, the previous context additionally to the current context was saved. In total, the users gave positive feedback in 616 cases and negative feedback in 47 cases. With about 93% of the feedback provided by users being positive, this means that most of the user interface adaptations were liked by the users. To gain a deeper insight into the usability experiment and a fine-grained data-driven analysis of the usability evaluation, the interested reader may refer to our work [35].

5.4 Potentials and limits

Considering the presented case studies above, we observed that our introduced domain-specific languages *ContextML* and *AdaptML* are a suitable complement to OMG's UI modeling language IFML to specify context management and UI adaptation concerns. Particularly, the separation of different modeling views for UI, context, and adaptation eases the modeling of self-adaptive UIs and also supports the maintenance of evolving context and adaptation models. As the presented case studies are showing, our integrated development approach allows the generation of self-adaptive UIs that can have quite complex UI adaptation features. In this regard, we have to point out that our approach is not supporting the analysis and resolution of conflicting UI adaptation rules. Although we have introduced priority levels to determine the execution order of UI adaptation rules, still it is a complex and error-prone task to manually specify a sound set of UI adaptations. Furthermore, our approach is focusing on the generation of view, context, and adaptation aspects, while the generation of application logic is out of scope.

Regarding the usability evaluation results, we can sum up that most of the triggered UI adaptation features were positively rated. Although this is a positive indicator for the resulting self-adaptive UIs, it should be noticed that users can ignore feedback questions in our usability experiment setup and that the absence of explicit user feedback should not be interpreted as a positive result for end-user satisfaction in general. While collecting context information, also issues regarding data privacy can arise and should be considered. On the one hand, a fine-grained way of collecting explicit instant user feedback can annoy the users and result in an intrusive evaluation method [35]. On the other hand, the collection of instant user feedback can be used to further optimize UI adaptations through machine learning techniques. This way, log data (context information, previous adaptations) and instant user feedback can be combined and analyzed to learn the most suitable adaptations for future context situations.

6 Related work

Recent research provides various approaches that support the model-based and model-driven development of UIs, their context management, and adaptations. In the following, relevant approaches are described and compared to our integrated model-driven engineering approach for self-adaptive UIs.

6.1 Model-driven UI development

Model-based and model-driven development approaches 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) [8] 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 [22], MARIA [29], and IFML [6] 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 context management as well as UI adaptation aspects in the development process by providing a context and adaptation model which enable the generation of context and adaptation services for supporting runtime UI adaptation.

6.2 Context management

Various approaches in the area of context-aware computing were presented in the past years to deal with the topic of context management. An important architecture for building context-aware applications was already presented by Dey et al. [10]. They developed a context toolkit that enables rapid prototyping of context-aware applications. The architecture of their context toolkit consists of sensors to collect context information, widgets to encapsulate the contextual information and provide methods to access the information, as well as interpreters to transform the context information into high-level formats that are easier to handle. Beside this approach, various other frameworks like WildCAT [12] or JCAF [4] were introduced to support the development of context-aware applications. Both, WildCAT and JCAF are

frameworks based on the programming language Java and they support context management by allowing the definition of a dynamic data model to represent the execution context for several application domains. In addition, they offer a programming interface to discover, interpret, and monitor the events occurring in an execution context and record every change occurring in the context model. A systematic and methodological approach for developing context-aware systems is presented in [19]. In this paper, the authors present a model-based approach that addresses the development of context-aware applications from both the theoretical and practical perspectives.

While the before mentioned approaches consider context management for a broad spectrum of applications areas, there are also specific approaches that deal with context management and context modeling specifically for supporting the adaptation of user interfaces of interactive systems. One holistic approach in this direction is the conceptual framework named TriPlet [23]. TriPlet contains a context-aware metamodel (CAM) that defines concepts required to implement and run a context-aware user interface. Context modeling and context awareness-related approaches were also applied in the mobile context. In [15], for example, the authors present a sensor-driven software framework for rapid prototyping of mobile applications. Closely related to our work is also the approach [30], where the authors deal with the topic of context-aware self-adaptation. They present a model-driven engineering approach to generate context-aware self-adaptation mechanisms. Our approach relies on and extends existing model-driven context management approaches by supporting the automatic generation of context services to monitor and detect context information changes.

6.3 User interface 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 [32], 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 architectures for self-adaptive systems can be also applied to self-adaptive UIs. The MAPE-K loop, which was used in our approach, was created by IBM as a reference model for autonomic computing [18]. 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 [14] and the Three-Layer Architecture [20]. Beside these general architectures for self-adaptive systems, there are also specific reference architectures for adaptive UIs like CAMELEON-RT [5], CEDAR [1], or FAME [11]. Furthermore, different approaches like Supple [16], MASP [13], MyUI [27], or RBUIS [3] present methods, techniques, and tools for supporting the development of adaptive UIs. However, these approaches do not focus on the generation of context and UI adaptation services in an integrated manner.

On the intersection of MDUID and UI adaptation, several transformation-based approaches like [21] or [31] 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 [17] which uses machine learning or [7] where a genetic algorithm is used to calculate a well-suited UI adaptation. Compared to these approaches, our integrated model-driven development approach for self-adaptive UIs provides dedicated modeling languages for context and adaptation modeling and supports the generation of context and adaptation services enabling runtime UI adaptation.

7 Conclusion and outlook

In this paper, we present an integrated model-driven development approach for self-adaptive UIs based on a classical model-driven development of UIs which is enhanced and coupled with a complementary model-driven development of context-of-use and UI adaptation rules. Based on OMG's core UI modeling language IFML, we propose new modeling languages for context management and UI adaptation, the languages *ContextML* and *AdaptML*, respectively. We present how generated UI code is coupled with context and adaptation services generated from *ContextML* and *AdaptML* models and integrated in an overall UI execution environment. 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 two case studies, showing the development of self-adaptive UIs for a university library and an e-mail application. Furthermore, we report on a usability evaluation study which has been conducted to analyze the end-user satisfaction of self-adaptive UIs. Main results of the usability evaluation show that the generated self-adaptive UIs based on our integrated model-driven development approach are mostly accepted by the end-users.

In ongoing research, we investigate the application of quality assurance techniques to our presented model-driven UI adaptation approach, which enable the provisioning of hard guarantees concerning self-adaptive characteristics such as adaptation rule set stability and deadlock freedom. Furthermore, we plan to enhance our proposed UI self-adaptation loop through the implementation 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. Further research will cover the development and application of self-adaptive UIs for various other domains.

Acknowledgements Open Access funding provided by Projekt DEAL.

Code examples

See Figs. 23 and 24.

Xtend Template for AmbientLightAPI

```
'''
   import { Injectable } from '@angular/core';
   import { Observable } from 'rxjs';
   import { BehaviorSubject } from 'rxjs/Rx';
   «FOR type: typeList»
      import { «type» } from '../types/«type»';
   «ENDFOR»

   @Injectable()
   export class «providerName.toFirstUpper»Service {
     «FOR prop: propertyList»
       «var propName = prop.getNamedItem("name")»
       «var propType = prop.getNamedItem("type")»
       private «propName»: «propType»;
       private _«propName»Subject: BehaviorSubject<«propType»> ...
       public «propName»Subject: Observable<«propType»> = ...
     «ENDFOR»

     constructor(){
     }

     «FOR prop: propertyList»
       «var propName = prop.getNamedItem("name")»
       get«propName.toFirstUpper»(){
         this._«propName»Subject.next(this.«propName»);
       }
     «ENDFOR»
   }
'''
```

Generatred AmbientLightAPI

```
import { Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { BehaviorSubject } from 'rxjs/Rx';
import { Level } from '../types/Level';

@Injectable()
export class AmbientLightAPI {

        private ambientLight: Level;
        private _ambientLightSubject: BehaviorSubject<Level> ...
        public ambientLightSubject: Observable<Level> ...

        constructor(){
            window.addEventListener('devicelight', event => {
                var html = document.getElementdsByTagName('html')[];
                if (event.value > 300) {
                    this.ambientLight = 2;
                }else if(event.value > 100){
                    this.ambientLight = 1;
                }else{
                    this.ambientLight = 0;
                }
                this.getAmbientLight();
            });
        }

        getAmbientLight(){
            this._ambientLightSubject.next(this.ambientLight);
        }
}
```

Fig. 23 Xtend template excerpt for ContextProviderGenerator and its generated code

Xtend Template for the NoolsService

```
''
  import {Injectable, DynamicComponentLoader, Injector} from '@angular/core';
  import { Router } from '@angular/router';

  import { Profile } from '../helper/profile';
  import { DisplayProperties } from '../helper/displayProperties';

  import { ResourceService } from './resource.service';
  import { «ServiceCollection.sharedInstance.displayProperties.name.toFirstUpper» } from
      '..«ServiceCollection.sharedInstance.displayProperties.location»';
  import { LoggerService } from './logger.service';

  declare var nools: any;
  declare var $: any;

  @Injectable()
  export class NoolsService {

      private flow;

      constructor(
          private dcl: DynamicComponentLoader,
          private injector: Injector,
          private _Router: Router,
          private _logger: LoggerService,
          private _ResourceService: ResourceService,
          private _«ServiceCollection.sharedInstance.displayProperties.name.toFirstUpper»:
                  «ServiceCollection.sharedInstance.displayProperties.name.toFirstUpper»){
          this.flow = nools.flow("«flow.attributes.getNamedItem("name").nodeValue»", function(flow){
              «new NoolsRuleGenerator().generateCode(flow.childNodes, serviceMap, functionMap)»
          });
      }

      public getSession(){
          return this.flow.getSession();
      }
  }
''
```

Generatred NoolsService

```
import {Injectable, DynamicComponentLoader, Injector} from '@angular/core';
import { Router } from '@angular/router';

import { Profile } from '../helper/profile';
import { DisplayProperties } from '../helper/displayProperties';

import { ResourceService } from './resource.service';
import { DisplayPropertiesService } from '../services/displayProperties.service';
import { LoggerService } from './logger.service';

declare var nools: any;
declare var $: any;

@Injectable()
export class NoolsService {

    private flow;

    constructor(
        private dcl: DynamicComponentLoader,
        private injector: Injector,
        private _Router: Router,
        private _logger: LoggerService,
        private _ResourceService: ResourceService,
        private _DisplayPropertiesService: DisplayPropertiesService){
        this.flow = nools.flow("Profile Evaluation", function(flow){
            flow.rule("Low Vision", {salience:9},[Profile,"m",
                        "(m.getUser().hasWeakVision() == 'true')"], function(facts){
                              $('.textPrimary').css('font-size','24px');
                              $('.textSecondary').css('font-size','20px');
            });
            |...
        });
    }
    public getSession(){
        return this.flow.getSession();
    }
}
```

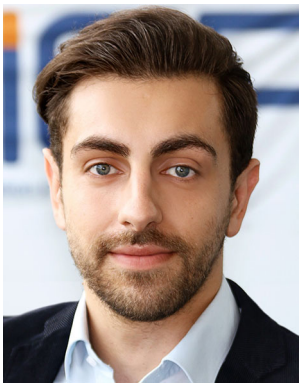
Fig. 24 Xtend template excerpt for NoolsServiceGenerator and its generated code

References

1. Akiki, P.A., Bandara, A.K., Yu, U.: Using interpreted runtime models for devising adaptive user interfaces of enterprise applications. In: ICEIS 2012—Proceedings of the 14th International Conference on Enterprise Information Systems, vol. 3, Wroclaw, Poland, 28 June–1 July, 2012, pp. 72–77 (2012)
2. Akiki, P.A., Bandara, A.K., Yu, Y.: Adaptive model-driven user interface development systems. ACM Comput. Surv. **47**(1), 9:1–9:33 (2014)
3. Akiki, P.A., Bandara, A.K., Yijun, Y.: Engineering adaptive model-driven user interfaces. IEEE Trans. Softw. Eng. **42**(12), 1118–1147 (2016)
4. Bardram, J.E.: The java context awareness framework (JCAF)—a service infrastructure and programming framework for context-aware applications. In: Proceedings of the Third International Conference on Pervasive Computing, PERVASIVE'05, pp. 98–115. Springer, Berlin (2005)
5. Balme, L., Demeure, A., Barralon, N., Coutaz, J., Calvary, G.: CAMELEON-RT: a software architecture reference model for distributed, migratable, and plastic user interfaces. In: Ambient Intelligence: Second European Symposium, EUSAI 2004, Eindhoven, The Netherlands, November 8–11, 2004. Proceedings, pp. 291–302 (2004)
6. Brambilla, M., Fraternali, P.: Interaction Flow Modeling Language—Model-Driven UI Engineering of Web and Mobile Apps with IFML. The MK/OMG Press (2014)
7. Blouin, A., Morin, B., Beaudoux, O., Nain, G., Albers, P., Jézéquel, J.M.: 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 System, EICS 2011, Pisa, Italy, June 13–16, 2011, pp. 85–94 (2011)
8. Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J.: A unifying reference framework for multi-target user interfaces. Interact. Comput. **15**(3), 289–308 (2003)
9. Coutaz, J.: User interface plasticity: model driven engineering to the limit! In: Proceedings of the 2nd ACM SIGCHI Symposium on Engineering Interactive Computing System, EICS 2010, Berlin, Germany, June 19–23, 2010, pp. 1–8 (2010)
10. Dey, A.K., Abowd, G.D., Salber, D.: A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Hum. Comput. Interact. **16**(2), 97–166 (2001)
11. Duarte, C., Carriço, L.: A conceptual framework for developing adaptive multimodal applications. In: Proceedings of the 11th International Conference on Intelligent User Interfaces, IUI 2006, Sydney, Australia, January 29–February 1, 2006, pp. 132–139 (2006)
12. David, P.C., Ledoux, T.: WildCAT: a generic framework for context-aware applications. In: The 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing, pp. 1–7. Pub.acm, Grenoble (2005)
13. Feuerstack, S., Blumendorf, M., Albayrak, S.: Bridging the gap between model and design of user interfaces. In: Informatik 2006–Informatik für Menschen, Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI), 2.-6. Oktober 2006 in Dresden, pp. 131–137 (2006)

14. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B.R., Steenkiste, P.: Rainbow: architecture-based self-adaptation with reusable infrastructure. IEEE Comput. **37**(10), 46–54 (2004)
15. Gamecho, B., Gardeazabal, L., Abascal, J.: A sensor-driven framework for rapid prototyping of mobile applications using a context-aware approach. In: Ubiquitous Computing and Ambient Intelligence—10th International Conference, UCAmI 2016, San Bartolomé de Tirajana, Gran Canaria, Spain, November 29–December 2, 2016, Proceedings, Part I, pp. 469–480 (2016)
16. Gajos, K.Z., Weld, D.S., Wobbrock, J.O.: Automatically generating personalized user interfaces with supple. Artif. Intell. **174**(12–13), 910–950 (2010)
17. Hariri, A., Tabary, D., Lepreux, S., Kolski, C.: Context aware business adaptation toward userinterface adaptation. Commun. SIWN **3**, 46–52 (2008)
18. IBM.: An architectural blueprint for autonomic computing. Technical report. IBM (2005)
19. Jaouadi, I., Djemaa, R.B., Ben-Abdallah, H.: A model-driven development approach for context-aware systems. Softw. Syst. Model. **17**(4), 1169–1195 (2018)
20. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, May 23–25, 2007, Minneapolis, MN, USA, pp. 259–268 (2007)
21. López-Jaquero, V., Montero, F., González, P.: T:XML: a tool supporting user interface model transformation. In: Hussmann, H., Meixner, G., Zuehlke, D. (eds.) Model–Driven Development of Advanced User Interfaces. Studies in Computational Intelligence, vol. 340. Springer, Berlin, Heidelberg (2011)
22. Limbourg, Q., Vanderdonckt, J.: 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 (ICWE 2004), Munich, Germany, 28–30 July, 2004, pp. 325–338 (2004)
23. Motti, V.G., Vanderdonckt, J.: A computational framework for context-aware adaptation of user interfaces. In: IEEE 7th International Conference on Research Challenges in Information Science, RCIS 2013, Paris, France, May 29–31, 2013, pp. 1–12 (2013)
24. Object Management Group (OMG).: Interaction Flow Modeling Language (IFML) Specification, Version 1.0. OMG Document Number formal/2015-02-05. https://www.omg.org/spec/IFML/1.0/PDF (2015)
25. Object Management Group (OMG).: Unified Modeling Language (UML) Specification, Version 2.5.1. OMG Document Number formal/2017-12-05. https://www.omg.org/spec/UML/2.5.1/PDF (2017)
26. Paternò, F.: User interface design adaptation. In: Soegaard, M., Dam, R.F. (eds.) The Encyclopedia of Human–Computer Interaction, vol. 39, 2nd edn. Aarhus, Denmark (2013)
27. Peissner, M., Häbe, D., Janssen, D., Sellner, T.: Myui: generating accessible user interfaces from multimodal design patterns. In: ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'12, Copenhagen, Denmark, June 25–28, 2012, pp. 81–90 (2012)
28. Paternò, F., Santoro, C.: A logical framework for multi-device user interfaces. In: ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS'12, Copenhagen, Denmark, June 25–28, 2012, pp. 45–50 (2012)
29. Paternò, F., Santoro, C., Spano, L.D.: MARIA: a universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. ACM Trans. Comput. Hum. Interact. **16**(4), 19:1–19:30 (2009)
30. Ruiz-López, T., Rodríguez-Domínguez, C., Rodríguez-Fórtiz, M.J., Ochoa, S.F., Garrido, J.L.: Context-aware self-adaptations: from requirements specification to code generation. In: Ubiquitous Computing and Ambient Intelligence. Context-Awareness and Context-Driven Interaction—7th International Conference, UCAmI 2013, Carrillo, Costa Rica, December 2–6, 2013, Proceedings, pp. 46–53 (2013)
31. Sottet, J.S., Ganneau, V., Calvary, G., Coutaz, J., Demeure, A., Favre, J.M., Demumieux, R.: Model-driven adaptation for plastic user interfaces. In: Human–Computer Interaction—INTERACT 2007, 11th IFIP TC 13 International Conference, Rio de Janeiro, Brazil, September 10–14, 2007, Proceedings, Part I, pp. 397–410 (2007)
32. Salehie, M., Tahvildari, L.: Self-adaptive software: landscape and research challenges. TAAS **4**(2), 14:1–14:42 (2009)
33. van Velsen, L et al.: User-centered evaluation of adaptive and adaptable systems: a literature review (2008)
34. Yigitbas, E., Grün, S., Sauer, S., Engels, G.: Model-driven context management for self-adaptive user interfaces. In: Ubiquitous Computing and Ambient Intelligence—11th International Conference, UCAmI 2017, Philadelphia, PA, USA, November 7–10, 2017, Proceedings, pp. 624–635 (2017)
35. Yigitbas, E., Hottung, A., Rojas, S.M., Anjorin, A., Sauer, S., Engels, G.: Context- and data-driven satisfaction analysis of user interface adaptations based on instant user feedback. PACMHCI **3**, 19:1–19:20 (2019)
36. Yigitbas, E., Jovanovikj, I., Josifovska, K., Sauer, S., Engels, G.: On-the-fly usability evaluation of mobile adaptive uis through instant user feedback. In: Human–Computer Interaction—INTERACT 2019—17th IFIP TC 13 International Conference, Paphos, Cyprus, September 2–6, 2019, Proceedings, Part IV, pp. 563–567 (2019)
37. Yigitbas, E., Kern, T., Urban, P., Sauer, S.: Multi-device UI development for task-continuous cross-channel web applications. In: Current Trends in Web Engineering—ICWE 2016 International Workshops, DUI, TELERISE, SoWeMine, and Liquid Web, Lugano, Switzerland, June 6–9, 2016, Revised Selected Papers, pp. 114–127 (2016)
38. Yigitbas, E., Sauer, S.: Engineering context-adaptive UIs for task-continuous cross-channel applications. In: Human-Centered and Error-Resilient Systems Development—IFIP WG 13.2/13.5 Joint Working Conference 6th International Conference on Human-Centered Software Engineering, HCSE 2016, and 8th International Conference on Human Error, Safety, and System Development, HESSD 2016 Stockholm, Sweden, August 29–31, 2016, Proceedings, pp. 281–300 (2016)
39. Yigitbas, E., Stahl, H., Sauer, S., Engels, G.: Self-adaptive UIs: integrated model-driven development of UIs and their adaptations. In: Modelling Foundations and Applications—13th European Conference, ECMFA 2017, Held as Part of STAF 2017, Marburg, Germany, July 19–20, 2017, Proceedings, pp. 126–141 (2017)

Enes Yigitbas received his BSc and MSc degrees in computer science from the Paderborn University, in 2009 and 2012, respectively. Currently, he is a researcher in the Database and Information Systems group at Paderborn University. His research interest covers model-driven engineering, human-computer interaction, and self-adaptive software systems.



Ivan Jovanovikj received his BSc degree in computer science in Skopje, North Macedonia, in 2011, and his MSc in computer science from the Paderborn University in 2015. Currently, he is a researcher in the Database and Information Systems group at Paderborn University. His research interest covers model-driven engineering, software testing, software reengineering, and software migration.

Kai Biermeier received his BSc degree in computer science from the Paderborn University in 2019. Currently, he is pursuing his MSc degree in computer science at the Paderborn University and working as a student assistant in the Database and Information Systems group.

Stefan Sauer is Senior Researcher in the Database and Information Systems Group of the Computer Science Department at Paderborn University and Managing Director of the Software Innovation Lab at SICP-Software Innovation Campus Paderborn, a joint research, knowledge, and technology transfer center for software- and data-driven innovation. He is also the Manager of the Center of Competence for Software Engineering in SICP. His main research areas are human-centered software engineering, model-based and model-driven software development, managed software evolution, and situational method engineering.

Gregor Engels holds the chair of Database and Information Systems at Paderborn University since 1997. His research areas include software engineering, focussing on model-based software development, human-centric computing, architectural styles, domain-specific modeling languages, and situational method engineering. Recently, his research work became more inter- and transdisciplinary, covering human aspects in the usage of cyber-physical systems. He is also head of the Software Innovation Campus Paderborn (SICP), a technology and knowledge transfer institute at Paderborn University.