Model-Based Product Configuration in Augmented Reality Applications

Sebastian Gottschalk, Enes Yigitbas, Eugen Schmidt, Gregor Engels

▶ **To cite this version:**

Sebastian Gottschalk, Enes Yigitbas, Eugen Schmidt, Gregor Engels. Model-Based Product Configuration in Augmented Reality Applications. 8th International Conference on Human-Centred Software Engineering (HCSE), Nov 2020, Eindhoven, Netherlands. pp.84-104, 10.1007/978-3-030-64266-2_5 . hal-03250496

HAL Id: hal-03250496
https://inria.hal.science/hal-03250496

Submitted on 4 Jun 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Model-based Product Configuration in Augmented Reality Applications*

Sebastian Gottschalk[1], Enes Yigitbas[1], Eugen Schmidt[2], and Gregor Engels[1]

[1] Software Innovation Lab, Paderborn University, Germany
{sebastian.gottschalk,enes.yigitbas,gregor.engels}@uni-paderborn.de
[2] Paderborn University, Paderborn, Germany
eschmidt@mail.uni-paderborn.de

Abstract. Augmented Reality (AR) has recently found high attention in mobile shopping apps such as in domains like furniture or decoration. Here, the developers of the apps focus on the positioning of atomic 3D objects in the physical environment. With this focus, they neglect the configuration of multi-faceted 3D object composition according to the user needs and environmental constraints. To tackle these challenges, we present a model-based approach to support AR-assisted product configuration based on the concept of Dynamic Software Product Lines. Our approach splits products (e.g. table) into parts (eg. tabletop, table legs, funnier) with their 3D objects and additional information (e.g. name, price). The possible products, which can be configured out of these parts, are stored in a feature model. At runtime, this feature model can be used to configure 3D object compositions out of the product parts and adapt to user needs and environmental constraints. The benefits of this approach are demonstrated by a case study of configuring modular kitchens with the help of a prototypical mobile-based implementation.

Keywords: Product Configuration · Augmented Reality · Runtime Adaptation · Dynamic Software Product Lines

1 Introduction

Mobile shopping has become a big trend in the last years, as it allows purchasing from anywhere and at any time [22, 27]. Because of the growing performance of mobile devices, Augmented Reality (AR) is becoming one key focus of these shopping apps [9]. AR allows users to virtually try-out physical objects in their environment before purchasing them. Examples of these apps are IKEA Place[3] to determine the appearance of furniture in certain rooms, the configuration of products like windows in the VEKA Configurator AR App[4], or the placement of

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Center "On-The-Fly Computing" (CRC 901, Project Number: 160364472SFB901)

[3] IKEA Place App: https://apps.apple.com/us/app/ikea-place/id1279244498

[4] VEKA Configurator AR App: https://apps.apple.com/us/app/vμ-configurator-ar/id1450934980

multiple products in an environment as recently announced by Amazon[5]. With these apps, the users have the possibility to get more information about the products, be more certain about buying what they want, and have a greater product choice and variety [12]. For this, most of the existing solutions focus on the positioning of atomic 3D objects in the physical environment. With this focus, they neglect the configuration of multi-faceted 3D object composition. This 3D object composition, in turn, can support a runtime adaptation of parts of the product according to the product requirements (e.g. exclusion of different product parts to each other), the user needs (e.g. maximum price), and the environmental constraints (e.g. detection of obstacles).

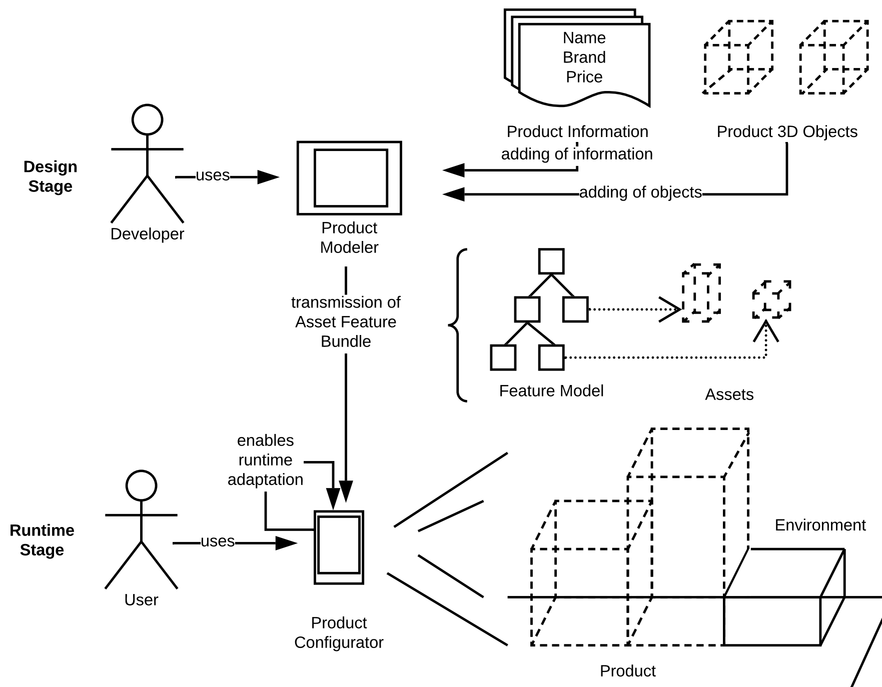

Fig. 1. Overview of the approach, where the *Developer* can model products at the *Design Stage* which can be configured by the *User* at the *Runtime Stage*

To overcome these limitations, we present a model-based AR-assisted product configuration that uses the concept of Dynamic Software Product Lines (DSPLs) [8] to provide a runtime adaptation of product parts to the product requirements, user needs, and environmental constraints. An overview of the approach, which consists of a *Design Stage* and a *Runtime Stage*, can be seen in Fig. 1. At the

[5] Announcement of Amazon: `https://techcrunch.com/2020/08/25/amazon-rolls-out-a-new-ar-shopping-feature-for-viewing-multiple-items-at-once/`

Design Stage, we use the concept of reusable assets [3] to split the definition of products and their possible configurations from their actual representation in 3D objects. For this, the *Developer* uses the *Product Modeler* to model valid product configurations. These configurations are based on the *Product Information* (e.g. name, brand, price) which are used to create a *Feature Model* where each feature (e.g. tabletop, table leg) can be linked to a specific *Product 3D Object* (e.g. tabletop model, table leg model). This so-called *Asset Feature Bundle* is then transmitted to the *Product Configurator* at the *Runtime Stage*.

At the *Runtime Stage*, we use the MAPE-K architecture [21] to monitor the user needs in the form of user inputs, the actual product configuration, and the environment. The gathered information is validated and used to adapt to the product configuration in the physical environment. For this, the *User* uses the *Product Configurator* by defining his own needs (e.g. favorite brand, maximum price). After that, she starts the configuring process by selecting features (e.g. table) and placing them in the environment. During the placement, the need of the *User* (e.g. price too high), the configuration of the *Product* (e.g. missing fridge in the kitchen), and the constraints of the *Environment* (e.g. collisions with other objects) are checked dynamically in the background.

With this paper, we present a novel model-based approach, which provides a twofold contribution to the research of AR-assisted product configurations: First, the separation of product modeling from 3D object implementation ensures a fast and flexible extension the configurator for new products. Second, the runtime configuration provides validation of product requirements and adaptation to user needs and environmental constraints. We demonstrate both benefits with a case study of a kitchen configurator.

The rest of the paper is structured as follows: Sect. 2 is providing the background of DSPLs which is used in our approach. Sect. 3 defines the requirements of our solution for the three models of the product, the customer, and the environment. Based on these requirements, Sect. 4 explains a solution concept that architecture is presented in Sect. 5. Sect. 6 shows the benefits of our approach based on the case study of a kitchen configurator. Sect. 7 presents the related work of our approach. Finally, we conclude our paper in Sect. 8.

2 Background

In this section, we explain the background on which we built our approach. We divide this background into Software Product Lines (SPLs) and Dynamic Software Product Lines (DSPLs).

2.1 Software Product Lines

Software Product Lines (SPLs) can be defined as "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [10]. Here, a feature refers to a

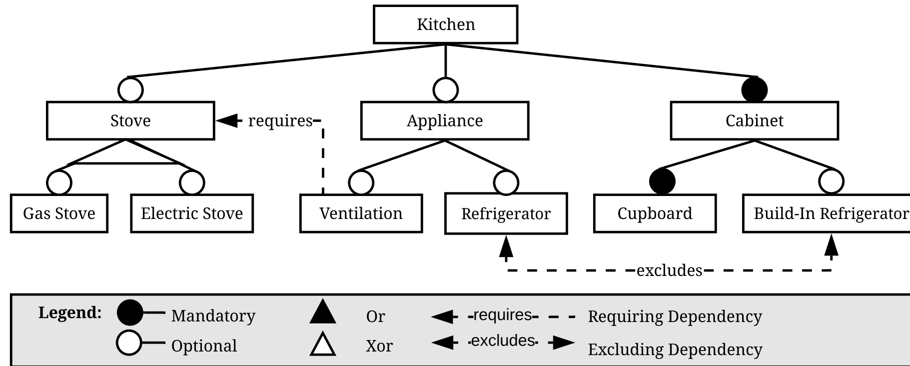

Fig. 2. Structure of feature models illustrated with a subset of a kitchen model which is used in the case study

prominent characteristic of a product which can be, depending on the stage of development, a requirement, a component in an architecture, or a piece of code [6]. To use these SPLs, a structure and an engineering process are needed.

The structure of SPLs can be represented using hierarchical feature modeling which is shown in Fig. 2. Features can be mandatory (i.e. *Cabinet*) or optional (i.e. *Ventilation*) for the model instances. Moreover, there can be Or (at least one sub-feature is selected), and Xor (exactly one sub-feature is selected) relationships between a parent and a child feature. To refine the model instance, cross-tree constraints for requiring (i.e. *Stove* requires *Ventilation*) and excluding (i.e. *Refrigerator* and *Build-In Refrigerator*) dependencies can be made. A big issue in SPL development is to find the right granularity for the features [20]. While the classical feature model does not allow modelings like the usage of multiple feature instances and additional attributes, some approaches extend the modeling approach for these cases [11]. For this, they add these additional modelings to the meta-model of the feature model. In our approach, we use feature models to structure the possible product configuration. For this, we need to extend the meta-model of feature models with meta-data in the form of attributes, positioning, and links to 3D objects.

The engineering process for SPLs is shown in Fig. 3. The process can be divided into the *Domain Engineering*, which consists of the analysis of the domain and the development of reusable artifacts, and the *Application Engineering*, which uses these artifacts for the development of a specific software product for a user group. Moreover, the *Problem Space* describes the user perspective on the requirements of the software product, while the *Solution Space* covers the developer's perspective on the design and implementation of the software product. Based on this classification the process consists of the following four steps [3]:

– **A. Domain Analysis** identifies the domain scope of the different products which can be developed with the SPL. From this analysis the reusable artifacts are identified and modeled as a feature model.

– **B. Domain Implementation** develops the different reusable artifacts (e.g. source code, test scripts) for further usage in the product derivation.
– **C. Requirement Analysis** extracts the requirements of a single user for the product. This requirements can lead to a feature selection of the SPL or the adding of new requirements to the domain analysis.
– **D. Product Derivation** is the matching of user requirements and reusable artifacts to build a product.

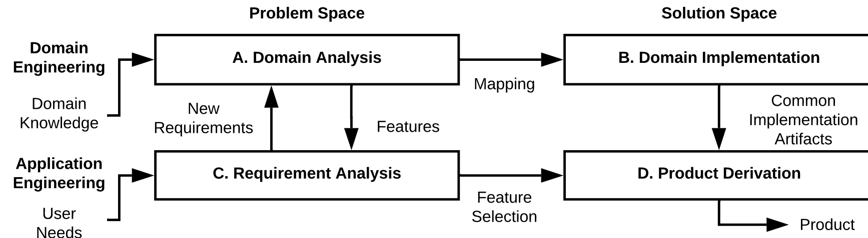

Fig. 3. Engineering process of SPLs which splits the generic *Domain Engineering* from the specific *Application Engineering*

We use the division of *Domain Engineering* and *Application Engineering* to configure different possible products from the same model. Moreover, we use the division of problem and solution space to separate the information about the product from their 3D objects.

2.2 Dynamic Software Product Lines (DSPLs)

Dynamic Software Product Lines (DSPLs) extend "the concept of conventional SPLs by enabling software-variant generation at runtime. The reason for this extension is that typical SPL approaches do not focus on dynamic aspects: the adaptation of their products is known and occurs during product line development"[7]. For this, adaptive systems can be used [19].

One of the most prominent reference architectures for autonomic computing is the MAPE-K loop [21]. MAPE-K, see Fig. 4 for the example of a socket detector consists of a *Managed Element* and an *Adaptation Manager*. The *Managed Element* provides sensors to get corresponding data (e.g. camera screens) and effectors to change its state (e.g. display view). The *Adaptation Manager* monitors the sensors (e.g. screen the environment) and analyzes the gathered data (e.g. detect sockets). Based on that, an adaptation is planned (e.g. plan an overlay for the sockets) and executed (e.g. add the overlay). Moreover, the knowledge base stores knowledge which can support the different steps (e.g. socket library for image recognition). Depending on the use case, the adaptation can be divided into different dimensions of goal (type, evolution), cause (type), and mechanism (autonomy, type) [4].

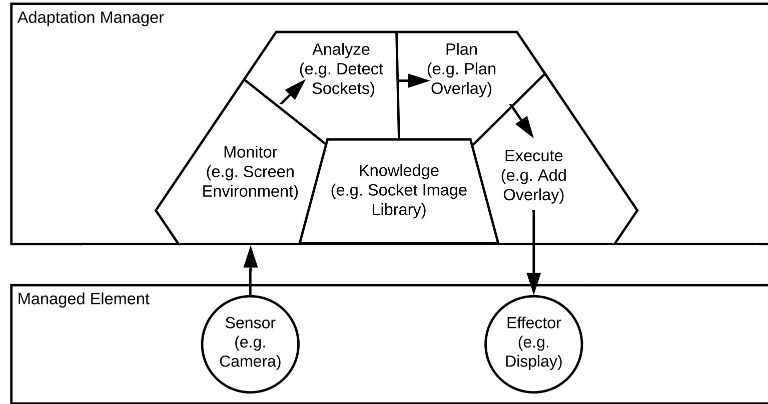

Fig. 4. Process of MAPE-K illustrated with the example of detecting and marking sockets in the environment

We use the MAPE-K loop to continuously monitor the user needs and environmental constraints for the product configuration. Moreover, the goal of the configurator will be of type self-configuring with static evolution. Moreover, the cause types are the user, product, and environment together of the manual component mechanism [4].

3 Model Requirements

In this section, we describe the requirements on the models for model-based AR product configuration. Based on the context in the *Runtime Stage*, we divide the requirements and the resulting models into the primary sources of *Product*, *User*, and *Environment*.

- **Product:** Within the product model, also referred to as the feature model in the next sections, we store all information that is needed to configure the products out of their parts. For this, we use the modeling structure of feature models [10] to model each product part as a single feature. Therefore, each valid product configuration of the feature model is also a valid AR product configuration. We extend this model with product information and asset management. In the product information, we store additional attributes (e.g. name, price, brand) for each product part. In asset management, we link the product parts to specific assets (e.g. 3D object, texture) and give additional information about their usage (e.g. single product, part of product, texture of product).
- **User:** Within the user model, we store all information that is needed to validate the user needs during the runtime configuration. Depending on the actual use case of the product configuration this could the maximum price which the user wants to pay for the configured product or the handedness to choose the side of the hinges.

– **Environment:** Within the environment model, we store all information that is needed to place the products in the physical environment during the runtime. We divide this information into the types of placement and obstacle management. In the placement management, we store a mesh structure of the environment that can be used to detect free places for the products and validates collisions with other objects. In the obstacle management, we store the positions of obstacles which can be needed by parts of the product (e.g. sockets, water supply).

Moreover, there can be dependencies between the different models. Because the product configuration is our main artifact, we consider just the combinations of *Product-User* and *Product-Environment*.

– **Product-User:** Between the product and the user, we specify dependencies that can exist between both models. Mostly these connections are between some product information and a specific user need. and For example, the product can have the dependency of a minimum height for a user or the user can prefer a specific brand of the product part.
– **Product-Environment:** Between the product and the environment, we specify dependencies that can exist between both entities. Mostly these connections are between some product information and a specific constraint in the environment. For example, the product can have the dependency of a water supply or the environment sets a maximum height for products because of a window as an obstacle.

4 Solution Concept

In this section, we present a solution concept for model-based AR product configuration. For this, we first show how the model requirements for the product, user, and environment can be modeled. Based on that we apply the engineering process of SPLs to AR product configuration. In the end, we show how the runtime adaptation to the models can be done.

4.1 Modeling of Product, User and Environment Requirements

To create a model-based AR product configuration, we have to first create models for all parts which our approach should effect. Based on the model requirements in the last section, we create models for the user (*UserModel*), the environment (*EnvironmentModel*), and the product (*FeatureModel*). The model of the user consists of simple attributes of the user (e.g. gender, maximum budget) which are characterized as important by the developer. Moreover, the modeled environment consists of a mesh structure of the physical environment together with positions of obstacles (e.g. windows, sockets) that need to be defined by the developer. Last, the model of the product consists of configuration parts that can be resolved to complete products. Because of the special interest of the product model in AR product configuration, we focus on this model within this section.

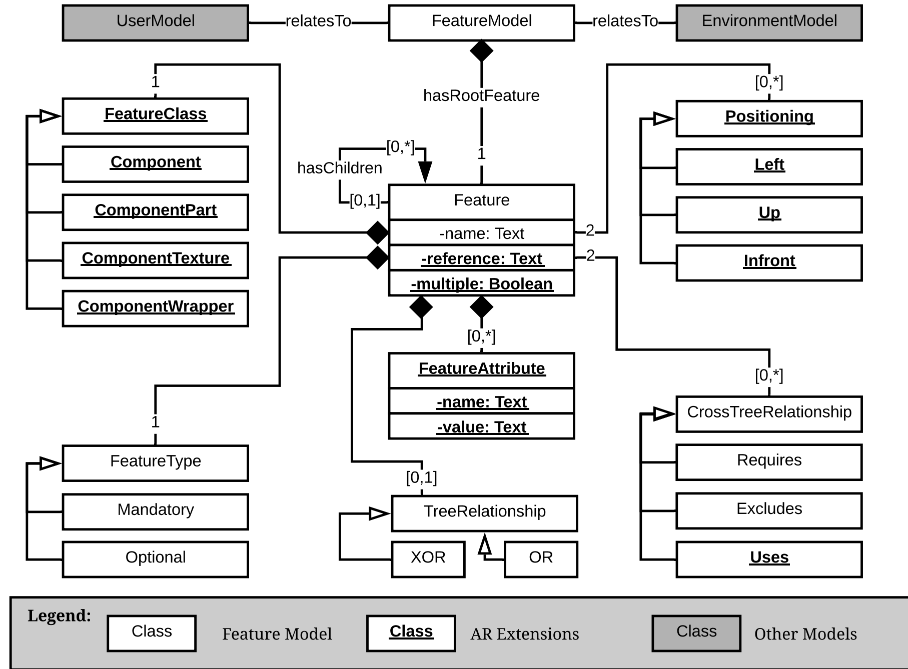

Fig. 5. Meta-Model of the *FeatureModel* with extensions for modeling the AR products and dependencies to the *UserModel* and the *EnvironmentModel*

The model of the product, see Fig. 5, is based on the concept of feature models [10]. For this, we implement all classes of feature models and extend the model with an AR specific extension. Each feature consists of a reference to a 3D object and an attribute if multiple instances (e.g. multiple cabins in one kitchen) can be created. Moreover, we added the classes of *FeatureClass*, *FeatureAttribute*, and *Positioning* together with extending *CrossTreeRelationship*. The *FeatureClass*, which is interpreted within the product configurator. Here, a *Component* relates to a 3D object which can be placed within the environment (e.g. table). A *ComponentPart* is a 3D object which can not be placed in the environment alone but as a part of another *Component* (e.g. tabletop). A *ComponentTexture* is a texture that can be applied to an existing *Component* (e.g. wood). A *ComponentWrapper* is used to structure the features within the product modeler and is not interpreted within the product configuration (e.g. electrical appliances). The *FeatureAttribute* is used to add product information (e.g. name, price) to each feature. The *Positioning* restricts the placed of features to each other. The new class of *Uses* within *CrossTreeDependencies* can be used to link *Components* to *ComponentTextures*.

Moreover, the model of the product has relationships with the models for the user and the environment. Here, the *FeatureAttributes* are linked to the

requirements of the user (e.g. specific brand or color). Moreover, the *Feature[s]* are linked to obstacles that are needed to use the feature (e.g. sockets or water supply).

4.2 Engineering Process of Model-based Product Configuration

The engineering process for AR product configuration can be seen in Fig. 6. The process can be divided into the *Design Stage*, which analyses the different product configuration parts and links them to the 3D objects, and the *Runtime Stage*, which analyses the needs of the user and environmental constraints to derive valid product configurations. Moreover, the *Problem Space* describes the user's perspective on the requirements of the product, while the *Solution Space* covers the developer's perspective on the implementation in the physical environment. Based on this classification, the process consists of the following four steps of *Product Analysis*, *Product Implementation*, *Configuration Analysis*, and *Configuration Derivation.*

- **A. Product Analysis** identifies the domain scope of the different products and develops a feature model out of them. In *A.1. Product Scoping* the possible products are identified, additional product information is collected and the different configuration parts are split up into reusable parts. In *A.2. Product Scoping* the possible product configurations are modeled as features of the feature model. Therefore, for each features the multiplicity, the feature class, the product information in the form of feature attributes and the possible positions are set (i.e. see section 4.1).
- **B. Product Implementation** develops the reusable assets which are used in the configuration derivation. In *B.1. Assets Modeling* the 3D objects and textures are created and split up into the parts of possible product configurations of the feature model. In the *B.1. Assets Binding* the created assets are linked to the feature model. So that the assets are inserted as a reference in the feature model (i.e. see section 4.1).
- **C. Configuration Analysis** extracts the requirements of a single user based on user needs and possible product configurations. In *C.1. User Input Validation* the user input is validated against the current product configuration to detect violations that need to be resolved. In *C.2. Product Selection Validation* the product configuration is validated according to the underlying feature model.
- **D. Configuration Derivation** is the matching of selected features and assets to place the product configuration in the environment. In the *D.1. Environment Validation* the configuration is validated against possible obstacles in the environment. Moreover, the *Feature Selection Change* of the *C. Configuration Analysis* can be changed. In the *D.2. Product Configuration Derivation*, the current *Product Configuration* is selected and can be exported for further usage (e.g. save for later, buy the product).

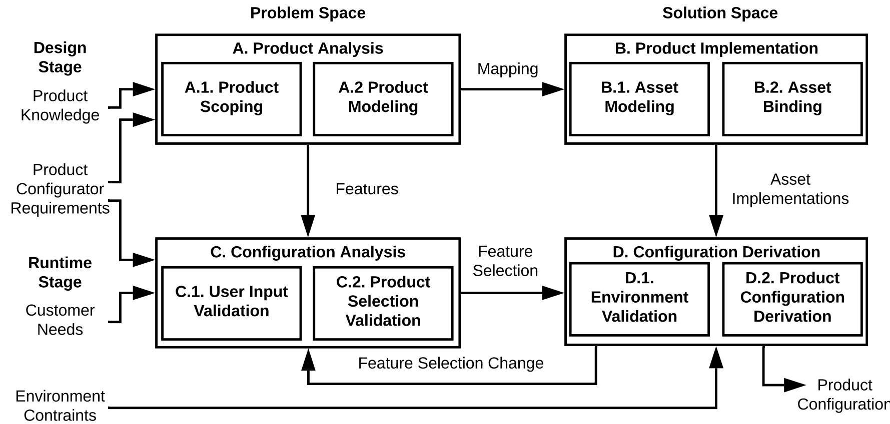

Fig. 6. Engineering process for an AR product configurator with a *Design Stage* to model the products and a *Runtime Stage* to configure the products

4.3 Runtime Adaptation according to the Modeled Requirements

The runtime adaptation for AR product configuration can be seen in Fig. 7. The concept can be divided into the *Mobile Application*, which provides the sensors and effectors for the adaptation and the *Configuration Manager*, which structures the adaptation process.

The *Mobile Application* provides sensors to provide information for the monitoring and effectors to change the display according to the execution. In the *Context Service* the inputs of the *Camera* and *User Input Management* are detected to update the *Configuration Model*, the *User Model*, and the *Environment Model*. These models can be monitored by the *Context Monitoring*. The effectors of the *Display Service* are triggered by the *Display Adaptation*. They are used to change the screen of the mobile application by *Display [the] AR Configuration* and *Display [the] Validation Issues*.

The *Configuration Manager* provides the adaptation logic to change the product configuration according to the requirements of the product, the user, and the environment. For this, the *Monitor* measures changes in the models of the configuration, the user, and the environment. While the user and configuration requirements can be directly used in the models, the environments need to be interpreted to create a mesh structure and detect the obstacles which should be considered. In *Evaluate Conditions* these changes are validated if the user needs, valid product configurations, or environmental constraints are violated. While the user and the configuration can be validated with the *FeatureModel*, the for the environment the positions in the configuration need to be compared to the structure in the *EnvironmentModel*. If all conditions are fulfilled, the *Execute* process the instructions to change the screen of the application are given. During the full configuration process, *Knowledge* is stored which can be used in the different steps. This *Knowledge* consists of the *FeatureModel*, which holds all

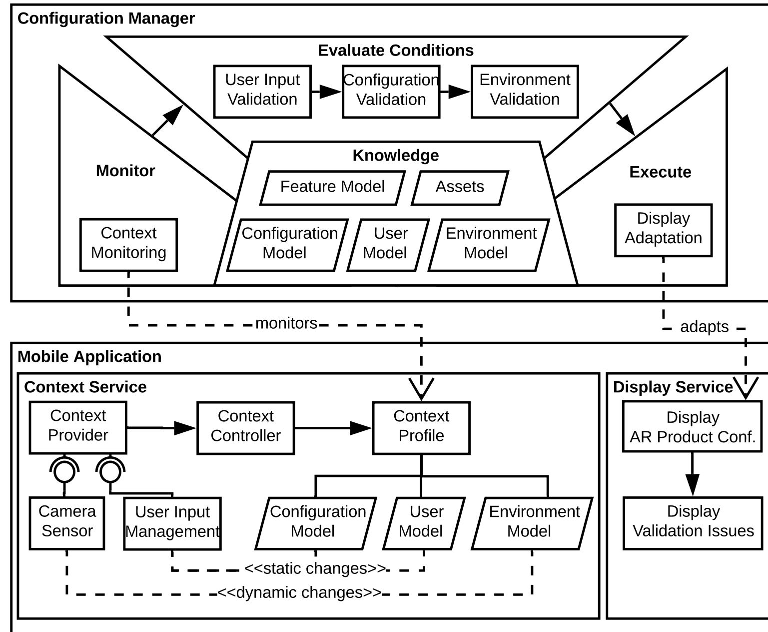

Fig. 7. Runtime adaptation based on the MAPE-K with a *Configuration Manager* (Adaptation Manager) and a *Mobile Application* (Managed Element)

possible product configurations, and the *Assets*, which display these configurations in the form of 3D objects and textures. Moreover, the current state of the *ConfigurationModel*, the *UserModel* and the *EnvironmentModel* are stored from the *Monitor* to use them in the other steps.

5 Solution Architecture

In this section, we provide a solution architecture for model-based AR product configuration. Based on our definition of the engineering process in section 4.2, we provide a component-based architecture for the implementation. This architecture is divided into the *Design Stage*, which consists of the *Product Modeler* and the *Asset Modeler*, and the *Runtime Stage*, which consists of the *Product Configurator*.

In the *Design Stage*, the configuration parts of the products are modeled through a *Feature Model Editor* together with the attributes for additional product information. Moreover, the assets for the configuration parts are made available with the *Asset Model Exporter*. Both, the *Feature Model* and *Assets*, are connected so that each feature can consist of an asset with the type (e.g. product, part of product, texture). Next, both are serialized through the *Asset Binder* to create a single *Asset Feature Bundle* which can be transmitted to the *Product*

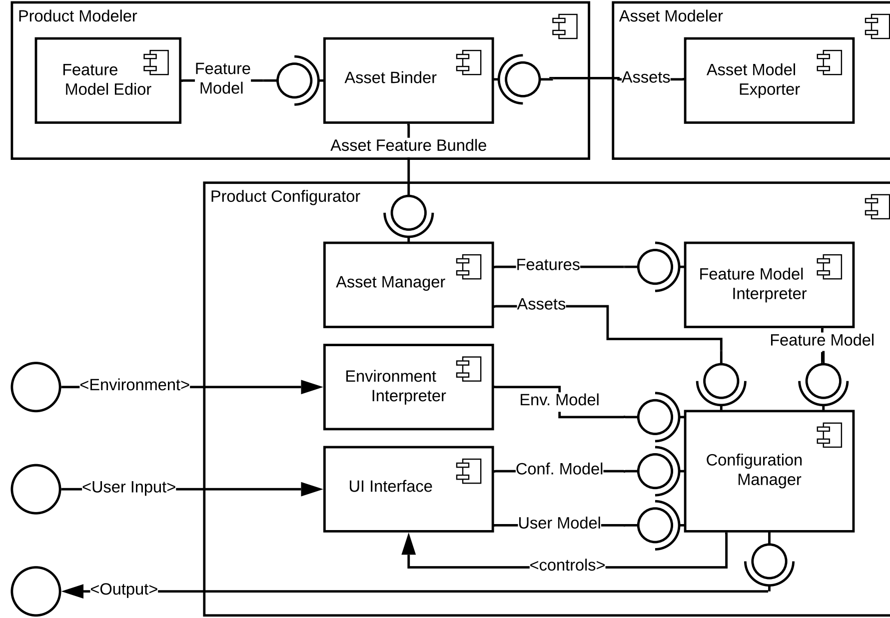

Fig. 8. Component overview of the developed *Product Modeler* and *Product Configurator* together with the external *Asset Modeler*

Configurator. By choosing the *Asset Feature Bundle* as a loose coupling between both stages, the tools of each stage can be exchanged independently.

In the *Runtime Stage*, the *Asset Manager* is used to deserialize the *Asset Feature Bundle* into components. While the *Assets* can be directly used in the *Configurator Manager*, the *Features* need to be analyzed by the *Feature Modeler Interpreter* to derive the *Feature Model*. Moreover, the *Configuration Manager*, whose activities are explained in section 4.3, received the requirements of the user, the configuration, and the environment during the runtime. While the *User Input* of the *UI Interface* can be directly restricted to use the *Configuration Model* and *User Model* , the *Environment* needs to be analyzed by the Environment Interpreter to derive the *Environment Model*. This is done by analyzing the images of the camera. With all these requirements the *Configuration Manager* can control the *UI Interface* and provide the configuration to the *Output*.

6 Case Study

In this section, we show how the principles of the solution concept can be applied in a concrete usage scenario. For this, we present a use case in which a manufacturer of modular kitchens uses our approach to provide to their customers the ability to order customized kitchens. First, we show our implementation of the

Product Modeler and the *Product Configurator*. Second, we discuss the current limitations of the implementation.

6.1 Instantiation

To show the benefits of our approach, we implemented a prototype of the *Product Modeler* and the *Product Configurator*. While the *Product Modeler* is based on a web-based implementation of a feature modeler in [17], the *Product Configurator* is based on the Unity framework[6]. Although this section gives an overview of the implementation, our demonstration paper shows more technical details [18].

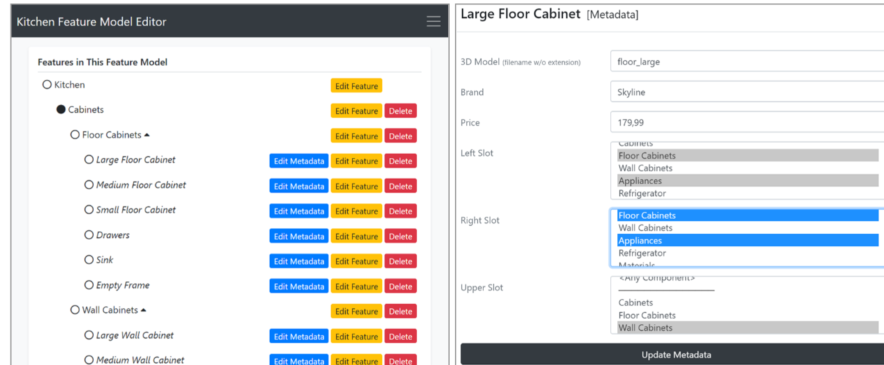

Fig. 9. *Product Modeler*: Creation of product parts and adding of meta-data inlcuding the 3D object for later AR representation

The *Product Modeler*[7], see Fig. 9 for screenshots, can be used to create the *FeatureModel* of the product that should be configurated. For this, all dependencies and constraints of feature models [3] can be used within the configuration by using the feature modeling tool in [17]. We extend this tool by adding meta-data for each feature (i.e. product part). In this meta-data, each feature can be linked to a 3D object, which is placed in a specific folder of Unity. Moreover, we add a price and a brand as feature attributes, which can be chosen to configure the product according to the customer needs. These feature attributes are also specified in our meta-model in Fig. 5. To place the product parts (e.g. cabin, stove) to each other, we allow the selection of valid other parts of the left, right, and upper slot of each product part. This, in turn, allows the flexible configuration of the kitchen in front of a single wall. After the full feature model is created, we need to bind the features to the corresponding 3D objects. For this, we export

[6] Unity Framework: `https://unity.com/`
[7] Source Code of the Product Modeler: `https://github.com/sebastiangtts/feature-modeler`

the whole model as a JSON file. After this, we use a developed importer script in Unity, which serialized the features and 3D objects in a single file. This file can be now uploaded to a web server. From this web server, the product configurator can download the *Feature Asset Bundle* and use it in the configuration process. With the *Project Modeler* it is possible to flexibly change the possible product configurations. These changes are automatically applied to all mobile clients in the form of *Product Configurators*.

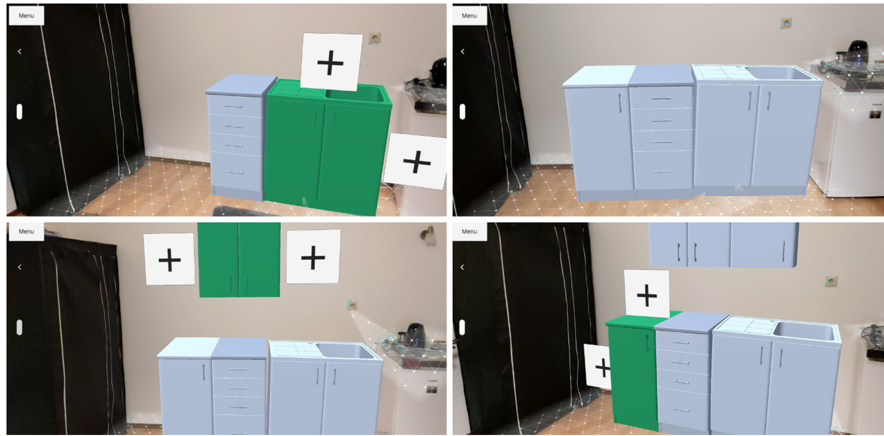

Fig. 10. *Product Configurator*: Configuring the product by selecting an existing product part and clicking on the "+"-button

The *Product Configurator*[8], see Fig. 10 and 11 for screenshots, can be used to configure a product out of the feature model. For this, the user opens the app, which downloads the *Feature Assets Bundle* from the webserver. After that, the user scans the environment to detect where the different parts of the kitchen can be placed. This information is stored inside the *EnvironmentModel* of the approach. Initially, she clicks on the screen to place the first part of the kitchen (i.e. see mesh structure on the floor). Iteratively, she now can click on an existing part of the kitchen, which makes "+"-buttons appear to extend the configuration. By clicking on one of these buttons, a list of all valid product parts is shown for selection.

During the full configuration process, the app checks if product requirements, user needs, or environmental constraints are violated. As an example, the first screen of Fig. 11 shows a cabin which can not be placed because of an object collision in the environment. After the full configuration has been done, the user can go to the check out process. Here, conformance errors, which can not be

[8] Source Code of the Product Configurator: `https://github.com/sebastiangtts/ar-product-configurator`

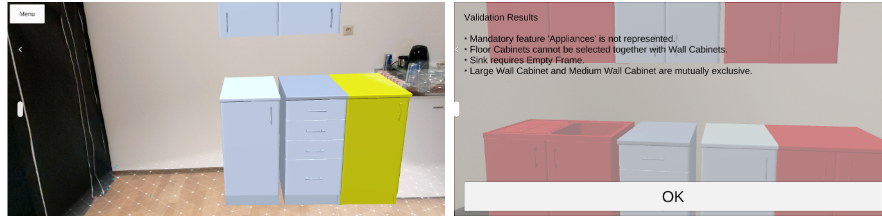

Fig. 11. *Product Configurator*: Validation of the product configuration during the selection process and after the configuration is finished

modeled graphically during the configuration, are validated. As an example, the second screen of Fig. 11 shows the error message that the sink requires an empty frame. Moreover, also requirements of the *UserModel* like the maximum price are applied here. With the *Product Configurator* it is possible to detect violated requirements of the product, the user, and the environment at runtime. Based on that, the violation can be resolved or conformance errors can be shown.

6.2 Discussion

With the implementation of the modular kitchen, we show the two main benefits of our approach. On the one hand, the separation of the modeling of products from their actual configuration, and, on the other hand, the adaptation of the configuration to the requirements of the product, the user and the environment. Nevertheless, the current implementation is just a prototype and has limitations according to the *Model Management* and *Platform Compatibility*.

For the *Model Management*, we develop a feature model where the configuration parts can be enriched with meta-data like product attributes and 3D objects. In the current prototype, we have not considered the user and the environment as explicit models. While for the user we fixed the data of the user at the beginning of the implementation, the *EnvironmentModel* is based on an internal model of AR Core [9]. In the future, we want to improve the prototype so that all requirements of the product, the user, and the environment are based on distinct models and not internal models like used by AR Core. Moreover, it should be possible that changes in the models (e.g. adding new feature attributes) are also covered directly in the prototype and not need to be fixed at the beginning of the implementation.

For the *Platform Compatibility*, we develop the prototype based on the technologies of Android and Unity. While the adaptation logic is written for Angular's AR Core, the *Asset Binding* is built on a script in Unity. Here, the *Asset Feature Bundle* has also the limitation that the whole information is transmitted at the starting of the application which limits the scalability of the approach.

[9] AR Core: `https://developers.google.com/ar`

Therefore, the current prototype has limitations in using other technology platforms. In the future, we want to modify the prototype so that different technology platforms can be used and interexchange. Moreover, we want to modify the *Asset Feature Bundle* so that assets are only transferred when they are needed at the runtime.

7 Related Work

In this section, we provide an overview of the related work of our approach. For this, we analyze relevant approaches considering the topics of Product Configuration, Augmented Reality, and Runtime Adaptation that are described and compared to our solution approach.

7.1 Product Configuration

Product configuration is a rich topic with a substantial amount of research behind it which is applied in various domains such as industrial applications, management, marketing, and computer science. In the following, we briefly focus on and describe product configuration approaches that rely on a model-based solution approach.

For model-based approaches, we use feature models to model the product features and business models of mobile applications [16]. We call our concept a Business Model Decision Line (BMDL). In this paper, we use our BMDL Feature Modeler [17] as the basis of our AR Product Modeler. Furthermore, Bashari et al. present a reference framework for dynamic software product line engineering which also serves as an inspiration for extending our feature modeler towards a DSPL approach that supports dynamic product configuration in AR [4]. For model-based approaches, one of the first things a product configuration system needs to do is validate the feature model itself. A lot of research has been conducted in the area of feature model validation [5], partially because feature models can be prone to errors (e.g. adding a new constraint might accidentally create a constraint deadlock or remove some valid products from the product line). One of the more advanced solutions in this area was proposed by Trinidad et al. who have developed a 3-step framework for automated feature model validation [26]. Their approach can detect some situations where there is a discrepancy between what the feature model describes and the actual product line it is supposed to model (e.g. feature models that produce no valid products or features that cannot appear in any valid product).

Another related area of research is optimization of product configurations (i.e. using algorithms to scan through all valid product configurations to find one that is optimal concerning some criteria). Depending on the criteria used, optimization can bring benefits to both the manufacturer and the customer. For example, an algorithm can be used by the manufacturer to find configurations that are easiest to produce or would be fastest to deliver, while the customer can use an optimization system to find a configuration with the lowest price. For

example, Yeh et al. focus specifically on using advanced algorithms to optimize the price of configurable products [28]. However, using such advanced techniques is only needed in cases where the feature model is complex enough to induce a very large number of possible configurations that can only be analyzed by a highly efficient algorithm.

7.2 Augmented Reality

With the shift from mass manufacturing to mass customization, the interest in product configuration increases as more and more companies offer their customers the ability to tailor a product to their needs before manufacturing even begins. This customization step often relies on computer-based representations of the product, its variants, and sometimes the used environment. For this reason, technologies that offer computer-generated imagery, such as VR and especially AR, are being researched in this context. In the following, we discuss some of the relevant AR-based approaches for product configuration.

An early study by Gehring et al. explored the usage of a mobile AR configurator app as a means of on-the-fly customization at the point of sale [15]. The app allowed the user to customize the color of a soap dispenser on their phone, either by picking a color directly or by calculating the most fitting color from the environment. Thus, the app exhibited a form of dynamic adaptivity to both user-based and environmental constraints. However, the focus of the study was not on using adaptive techniques for product configuration. Rather, the app was envisioned to be used in concert with a smart factory.

Moreover, the usage of AR in product customization is a related topic of this work. As a result, an additional related area of study in AR-based product configuration has emerged: virtual try-on. This discipline aims to use AR's capabilities to allow the customer to not only customize a wearable product but also to immediately try their design on themselves. This research field is in certain aspects similar to the idea of mobile product configuration, as both aim to enable the end-user to test and preview a product with the help of computer-generated imagery. Here, Eisert et al. developed an AR-based try-on system that they called Virtual Mirror [13]. This system was essentially an AR-based mirror that recorded the customer's feet and displayed the video output in real-time, horizontally flipped. The customer could then use a computer to select a shoe model and make various cosmetic changes such as color, materials, and embroidery.

Overall, approaches that focus purely on the try-on aspect, as exemplified by Eisert et al., can be considered as fulfilling the user–product requirement type due to their often minimal configuration capabilities [13]. Furthermore, they can be said to display a degree of adaptivity to the user, because they adapt their video output to the user's physical properties and movement. However, despite being AR-based, virtual try-on systems do not take their environment into account, as their AR capabilities are aimed at augmenting the image of the user, rather than the environment.

7.3 Runtime Adaptation

For enabling runtime adaptation in software systems, the research area of self-adaptive software systems [24] has emerged which provides extensive approaches to support the adaptation of software systems to changing context situations. As AR-based product configuration approaches have to monitor and adapt a viable product configuration regarding various aspects such as user and environmental requirements, we discuss relevant approaches based on runtime adaptation.

Here, especially 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 [1]. RBUIS [2] and Adapt-UI [31] are two representative approaches that present methods, techniques, and tools for supporting the development of adaptive UIs. With regard to adaptive UIs, most existing approaches make use of a context model consisting of a user, platform, and environment model as described by us in [30]. Besides these approaches, there are also approaches that apply the idea of runtime adaptation for AR applications. While in [29], we apply the idea of context-awareness for VR applications, in [23] we propose a development framework for context-aware AR applications. Mostly, the before mentioned approaches are based on a runtime monitoring and adaptation loop which is characterized through the MAPE-K loop [21] which was also applied in this work. In contrast to the described approaches, our proposed solution in this paper combines the advantages of DSPLs with runtime adaptation to support model-based product configuration.

Moreover, some approaches use the concept of DSPLs to model the UI and the context of use (user, platform, environment) as feature models. Here, Gabillon et al. create a single feature model for the UI features and the contexts of use [14]. Moreover, they add logic rules to the UI features so that they can adapt to changes in the context and apply the concept of a case study of a desktop application. Sboui et al. model both as separate feature models and create a configuration model to link both feature models together [25]. They apply their approach to a case study of a mobile application. In contrast to the described approaches, our proposed solution in this paper combines the advantages of DSPLs with runtime adaptation to support model-based product configuration of AR products.

8 Conclusion

Augmented Reality (AR) has recently found high attention in mobile shopping apps such as in domains like furniture or decoration. While these apps focus on the displaying of atomic 3D objects, they neglect 3D object composition and their configuration according to the user and environmental requirements. To solve both issues, we develop a model-based AR-assisted product configuration approach based on the concept of Dynamic Software Product Lines. For this, we split the engineering process into a *Design Stage* and a *Runtime Stage*. While in the *Design Stage* we create a feature model where each product part is modeled as a feature can be linked to an asset, the *Runtime Stage* uses the feature

model and assets to display a product configuration in the physical environment. Moreover, this product configuration can adapt to the requirements of the user and the environment. The benefits of this approach are demonstrated by a case study of configuring modular kitchens with the help of a prototypical mobile-based implementation.

Our future work is twofold and deals with the generalization of the introduced concepts: First, we want to improve the modeling of the user and the environment by separating them into distinct models. Second, we want to create a cross-platform solution for supporting model-based AR/VR product configuration. By combining both, our goal is to develop a model-driven framework for developing AR/VR product configurators.

References

1. 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)
2. Akiki, P.A., Bandara, A.K., Yu, Y.: Engineering adaptive model-driven user interfaces. IEEE Trans. Software Eng. **42**(12), 1118–1147 (2016)
3. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
4. Bashari, M., Bagheri, E., Du, W.: Dynamic Software Product Line Engineering: A Reference Framework. International Journal of Software Engineering and Knowledge Engineering **27**(02), 191–234 (2017)
5. Benavides, D., Segura, S., Cortés, A.R.: Automated analysis of feature models 20 years later: A literature review. Inf. Syst. **35**(6), 615–636 (2010)
6. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated reasoning on feature models. In: Pastor, O., Falcão e Cunha, J. (eds.) Advanced Information Systems Engineering. pp. 491–503. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
7. Bencomo, N., Hallsteinsen, S., de Almeida, E.S.: A View of the Dynamic Software Product Line Landscape. Computer **45**(10), 36–41 (2012)
8. Capilla, R., Bosch, J., Trinidad, P., Ruiz-Cortés, A., Hinchey, M.: An overview of Dynamic Software Product Line architectures and techniques: Observations from research and industry. Journal of Systems and Software **91**, 3–23 (2014)
9. Chatzopoulos, D., Bermejo, C., Huang, Z., Hui, P.: Mobile Augmented Reality Survey: From Where We Are to Where We Go. IEEE **5**, 6917–6950 (2017)
10. Clements, P., Northrop, L.: Software product lines: Practices and patterns. SEI series in software engineering, Addison-Wesley, Boston, 7. print edn. (2009)
11. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice **10**(2), 143–169 (2005)
12. Dacko, S.G.: Enabling smart retail settings via mobile augmented reality shopping apps. Technological Forecasting and Social Change **124**, 243–256 (2017)
13. Eisert, P., Fechteler, P., Rurainsky, J.: 3-d tracking of shoes for virtual mirror applications. In: Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR). IEEE (2008)
14. Gabillon, Y., Biri, N., Otjacques, B.: Designing an adaptive user interface according to software product line engineering. In: ACHI 2015 (2015)

15. Gehring, S., Löchtefeld, M., Schöning, J., Gorecky, D., Stephan, P., Krüger, A., Rohs, M.: Mobile product customization. In: Proceedings of the 28th International Conference on Human Factors in Computing Systems (CHI). pp. 3463–3468. ACM (2010)
16. Gottschalk, S., Rittmeier, F., Engels, G.: Intertwined Development of Business Model and Product Functions for Mobile Applications: A Twin Peak Feature Modeling Approach. In: Software Business, vol. 370, pp. 192–207. Springer (2019)
17. Gottschalk, S., Rittmeier, F., Engels, G.: Hypothesis-driven Adaptation of Business Models based on Product Line Engineering. In: Proceedings of the 22nd Conference on Business Informatics (CBI). IEEE (2020)
18. Gottschalk, S., Yigitbas, E., Schmidt, E., Engels, G.: ProConAR: A Tool Support for Model-based AR Product Configuration. In: Human-Centered Software Engineering. Springer (2020)
19. Hallsteinsen, S., Stav, E., Solberg, A., Floch, J.: Using Product Line Techniques to Build Adaptive Systems. In: Proceedings of the 10th International Software Product Line Conference (SPLC). pp. 141–150. IEEE (2006)
20. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in software product lines. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) Proceedings of the 13th international conference on Software engineering (ICSE). p. 311. ACM (2008)
21. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)
22. Ko, E., Kim, E.Y., Lee, E.K.: Modeling consumer adoption of mobile shopping for fashion products in Korea. Psychology and Marketing **26**(7), 669–687 (2009)
23. Krings, S., Yigitbas, E., Jovanovikj, I., Sauer, S., Engels, G.: Development framework for context-aware augmented reality applications. In: Proceedings of the Symposium on Engineering Interactive Computing Systems (EICS). pp. 9:1–9:6. ACM (2020)
24. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. **4**(2), 14:1–14:42 (2009)
25. Sboui, T., Ayed, M.B., Alimi, A.: A ui-dspl approach for the development of context-adaptable user interfaces. IEEE Access **6**, 7066–7081 (2018)
26. Trinidad, P., Benavides, D., Durán, A., Cortés, A.R., Toro, M.: Automated error analysis for the agilization of feature modeling. J. Syst. Softw. **81**(6), 883–896 (2008)
27. Yang, K.: Determinants of US consumer mobile shopping services adoption: implications for designing mobile shopping services. Journal of Consumer Marketing **27**(3), 262–270 (2010)
28. Yeh, J.Y., Wu, T.H., Chang, J.M.: Parallel genetic algorithms for product configuration management on pc cluster systems. **31**(11/12), 1233 – 1242 (2007)
29. Yigitbas, E., Heindörfer, J., Engels, G.: A context-aware virtual reality first aid training application. In: Alt, F., Bulling, A., Döring, T. (eds.) Proceedings of Mensch und Computer 2019. pp. 885–888. GI / ACM (2019)
30. Yigitbas, E., Jovanovikj, I., Biermeier, K., Sauer, S., Engels, G.: Integrated model-driven development of self-adaptive user interfaces. International Journal on Software and Systems Modeling (SoSyM) (2020)
31. Yigitbas, E., Sauer, S., Engels, G.: Adapt-ui: an IDE supporting model-driven development of self-adaptive uis. In: Proceedings of the Symposium on Engineering Interactive Computing Systems (EICS). pp. 99–104. ACM (2017)