ProConAR: A Tool Support for Model-based AR Product Configuration*

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. Mobile shopping apps have been using Augmented Reality (AR) in the last years to place their products in the environment of the customer. While this is possible with atomic 3D objects, there is is still a lack in the runtime configuration of 3D object compositions based on user needs and environmental constraints. For this, we previously developed an approach for model-based AR-assisted product configuration based on the concept of Dynamic Software Product Lines. In this demonstration paper, we present the corresponding tool support ProConAR in the form of a Product Modeler and a Product Configurator. While the Product Modeler is an Angular web app that splits products (e.g. table) up into atomic parts (e.g. tabletop, table legs, funnier) and saves it within a configuration model, the Product Configurator is an Android client that uses the configuration model to place different product configurations within the environment of the customer. We show technical details of our ready to use tool-chain ProConAR by describing its implementation and usage as well as pointing out future research directions.

Keywords: Product Configuration · Augmented Reality · Model-based · Tool Support

1 Introduction

In the last years, Apple and Google have pushed the topic of Augmented Reality (AR) in mobile apps within their mobile ecosystems. One area of focus is mobile shopping [3], where AR can support the decision process of the customer with additional product information, direct placement of products in the environment, and a greater product choice [4]. Examples for these mobile apps are the placing of an atomic product in the environment as in IKEA Place app[3], the configuration of products like windows in the VEKA Configurator AR app[4],

* 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 at Apple's AppStore: https://apps.apple.com/us/app/ikea-place/id1279244498

[4] VEKA Configurator AR at Apple's AppStore: https://apps.apple.com/us/app/vD0B5kD0B0-configurator-ar/id1450934980

Preprint, cite this paper as:
Gottschalk S., Yigitbas E., Schmidt E., Engels G. (2020) ProConAR: A Tool Support for Model-Based AR Product Configuration. In: Bernhaupt R., Ardito C., Sauer S. (eds) Human-Centered Software Engineering. HCSE 2020. Lecture Notes in Computer Science, vol 12481. Springer, Cham

The final authentificated version is available online at:
https://doi.org/10.1007/978-3-030-64266-2_14

or the placement of multiple products in an environment as recently announced by Amazon[5]. While these approaches focus on simple placements, they neglect the runtime configuration of 3D object compositions based on user needs and environmental constraints.

To improve this configuration process, we previously presented a model-based AR-assisted product configuration [6] that uses the concept of Dynamic Software Product Lines (DSPLs) [2]. The approach consists of a *Design Stage* and a *Runtime Stage*. At the *Design Stage*, we use the concepts of feature models and reuseable assets from Software Product Lines (SPLs) [1] to split the product representation and its possible configurations from the corresponding 3D objects. Out of both, we export a configuration model that is used at runtime. At the *Runtime Stage*, we import the configuration model and use the MAPE-K architecture [7] to monitor the user needs in the form of user inputs, the actual product configuration, and the environment. Based on the measured inputs, we configure possible product configurations within the environment.

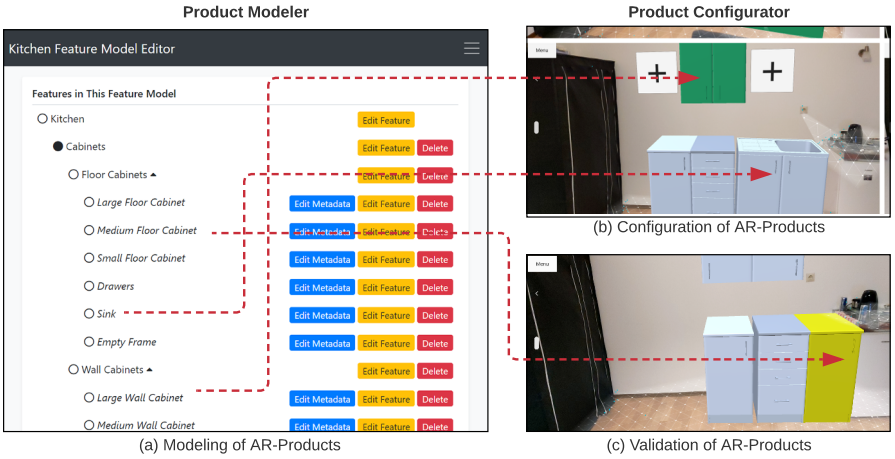


Fig. 1. ProConAR consists of a *Product Modeler* used by the *Developer* and a *Product Configurator* used by the *User*

In contrast to our research paper [6], in this demonstration paper we focus on the technical implementation of the corresponding tool ProConAR and point our future research directions. ProConAR, as shown in Fig. 1, consists of a *Product Modeler* and a *Product Configurator*. The *Product Modeler*, which is used at the *Design Stage*, is an Angular web app based on an existing feature model editor in [5]. It can be used to create feature models of possible product configurations (see Fig. 1 (a) for a developed kitchen model). Moreover, for

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

each feature, a 3D object together with additional attributes can be saved. The created configuration model can be exported for the *Product Configurator*. The *Product Configurator*, which is used at the *Runtime Stage*, is an Android client based on the Unity Engine[6] . It imports the configuration model and uses the information to configure products within the environment (see Fig. 1 (b) for kitchen configuration in the environment). Moreover, it adapts the product configuration to the user needs, the product requirements, and the environmental constraints (see Fig. 1 (c) for validation of the product in the environment).

The rest of the paper is structured as follows: Sect. 2 provides the solution architecture of ProConAR in the form of a component diagram. Sect. 3 shows technical details in the modeling of the product, the transfer of the assets from the *Product Modeler* to the *Product Configurator*, and the configuration of the product. Finally, in Sect. 4 we conclude our paper and point out future research directions.

2 Solution Architecture

In this section, we provide a solution architecture for ProConAR based on a component-based architecture as shown in Fig. 2. The solution 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 as presented in [6]. Moreover, the assets for the configuration parts are made available with the *Asset Model Exporter* which is an external graphic modeling tool. Both, the *Feature Model* and *Assets*, are connected so that each feature can consist of an asset with a specific type (e.g. product, part of product, texture) and positioning (e.g. left slot, right slot, upper slot) to other features. Next, both are serialized through the *Asset Binder* to create a single *Asset Feature Bundle* which can be transferred to the *Product 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. Here, we use the *Feature Model* to store all possible configurations, the *Configuration Model* to store the actual product configuration, the *User Model* to store the user needs and the *Environment Model* to store the environmental constraints. 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 based on the MAPE-K architecture and explained in [6], receives 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

[6] Unity Engine: `http://www.unity.com`

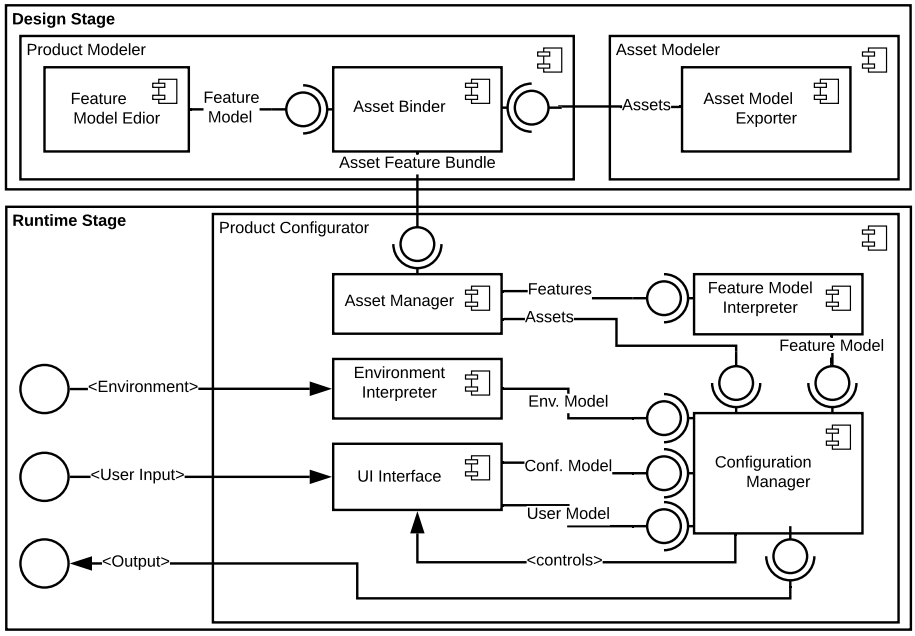


Fig. 2. Component overview of the developed *Product Modeler* and *Product Configurator* together with the external *Asset Modeler*

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*.

3 Technical Implementation

In this section, we present the technical implementation of the *Product Modeler*[7] and the *Product Configurator*[8]. Both tools are built on top of well-accepted development techniques to allow the reusing for other researchers. The *Product Modeler* is built on an Angular web app and can run directly in the web browser. Moreover, we use an existing feature modeler [5] to cover all possible feature restrictions and dependencies. The *Product Configurator* is built on ARCore, the AR SDK of Android. Moreover, we use the Unity SDK that provides solutions for most interactions with the environment like detecting obstacles or modifying 3D mesh models. For the transfer between both tools, we use the assets bundling

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

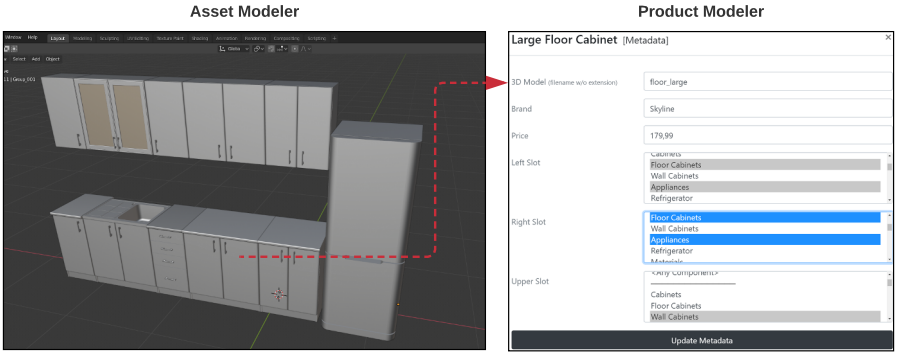
[8] Source Code of the Product Configurator: `https://github.com/sebastiangtts/ar-product-configurator`

mechanism of Unity by using an *Importer Script* for the feature model, stored as JSON file, and the assets in the form of 3D meshes, stored as FBX files, and textures, stored as PNG files.

In the following, we are focusing on the steps of product modeling, the transfer of product configurations, and the product configuration. While this demonstration paper just roughly goes through the whole process of ProConAR, the detailed description of the installation and usage is provided within the repositories.

3.1 Step 1: Product Modeling

In the first step of our process, we have to define the products which we want to configure in the environment. For that we need to design the parts of our product in the *Asset Manager* and model the possible product configurations within the *Product Modeler* as shown in Fig. 3.



(a) Export of Product Configurations (b) Import of Product Configurations

Fig. 3. Creating of products with the *Asset Modeler* and modeling them with the *Product Modeler*

At first, we use an *Asset Manager* like Blender[9] or AutoDesk[10] to separate the different product parts from each other and create a single model for each part (e.g. fridge of a kitchen). Moreover, we separate the 3D mesh as FBX from the texture as PNG to allow a reusing of the texture for different meshes. After that, we store both within our Unity product in the *Assets/Inputs* folder so that we can reference them with the corresponding *Importer Script*.

Next, we have to model the possible product configurations within our *Product Configurator*. For that, we create a new feature model that adds multiple features that can have the type of wrapper, product-part, or textures. While wrapper features are just used for structuring the model itself, the product-part

[9] Homepage of Blender: `https://www.blender.org`
[10] Homepage of AutoDesk: `https://www.autodesk.com`

and texture features contain corresponding 3D mesh files or images. These files are referenced in the assets folder of the importer by choosing the same naming. For each feature of type product-part, we can set a brand, a price, and slots for left, right, and upper positioning. After adding all features to the feature model, we can export the model as a JSON file and store it within the *Assets/Inputs* folder so that it is useable within the *Importer Script*.

3.2 Step 2: Transfer of the Product Configuration

In the second step, we need to transfer the product configuration from the *Product Modeler* to the *Product Configurator*. The transfer process is shown in Fig. 4. In the last step, we have already shown the design and modeling of the product and its transfer to the *Importer Script*. This script is written for the Unity Game Engine and imports all files from the *Assets/Inputs* folder. After validating the files (e.g. looking if all references of the feature model are inside the folder), it bundles the JSON with the 3D meshes and textures by using the asset bundling[11] which is a build-in technique of Unity to load non-code assets at runtime into an application. After the bundling process, the DB file is stored in the *Assets/Bundles*. This bundle can be uploaded to a web server, for which we used GitHub, and downloaded by the *Product Configurator*. The *Product Configurator* downloads the asset bundle and interprets it which is described in the next step.

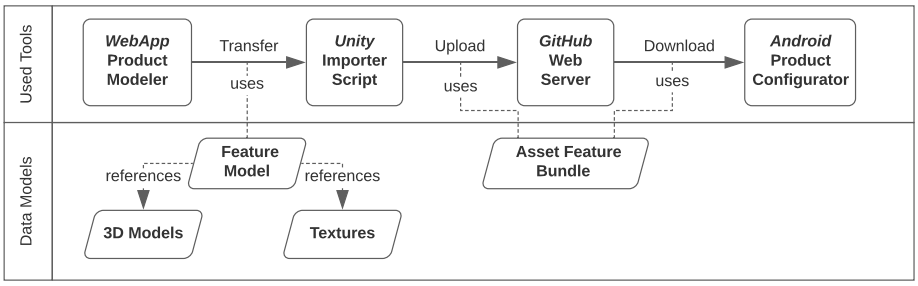


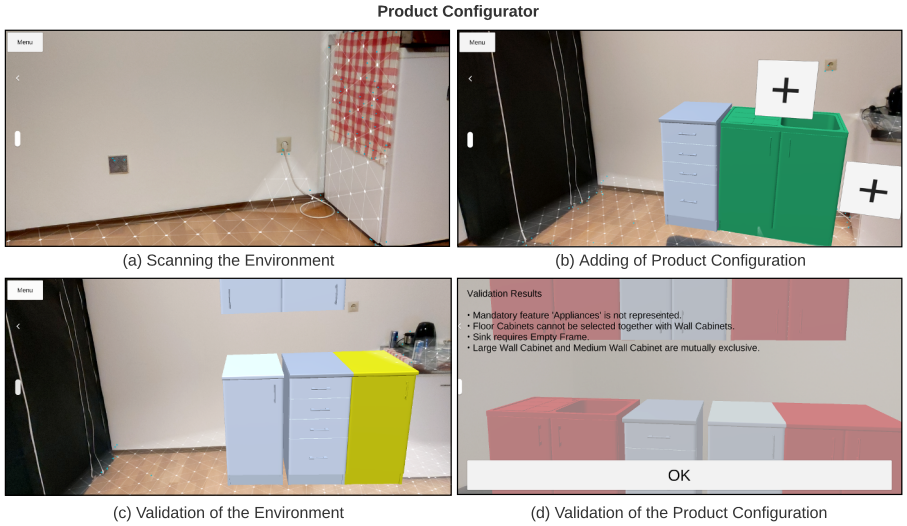
Fig. 4. Transfer of the product configuration from the *Product Modeler* to the *Product Configurator* by using the *Feature Asset Bundle*

3.3 Step 3: Configuration of Products

In the last step, we need to configure the product out of the asset file as shown in Fig. 5. In the beginning, the app loads the file from the webserver and splits into

[11] Asset Bundling of Unity: `https://docs.unity3d.com/Manual/AssetBundlesIntro.html`

the JSON feature model and the assets in the form of 3D meshes and textures. After that, the app deserializes the JSON with an external library[12] and builds an internal structure of the model to validate violations at runtime.



(a) Scanning the Environment (b) Adding of Product Configuration

(c) Validation of the Environment (d) Validation of the Product Configuration

Fig. 5. Configuration of the product by scanning the environment, adding product parts and validating the product configuration

After that, the customer can scan the environment to detect horizontal surfaces on which the product parts will appear standing and vertical surfaces that will provide spatial boundaries of the environment (see Fig. 5 (a)). After that, the customer can tap on the environment and select the first product part he wants to add to the environment (see Fig. 5 (b)). From this point, the customer clicks on the plus buttons next to the placed product parts to extend the configuration and is informed with visual hints if an error (e.g. two stoves in one kitchen) duo to the placement has occurred (see Fig. 5 (c)). Moreover, the configurator validates at runtime if constraints of the underlying feature model or the user needs are violated. For that, we have developed our own feature model interpreter script. At the end of the configuration (see Fig. 5 (d)), the configurator check also violations which can not be displayed during the runtime.

4 Conclusion and Outlook

Mobile shopping apps have been used Augmented Reality (AR) in the last years to place their products in the environment of the customer. While this is possible

[12] Unity-Library for JSON Interpretations: `https://assetstore.unity.com/packages/\-tools/input-management/json-net-for-unity-11347`

with atomic 3D objects, there is is still a lack in the runtime configuration of 3D object compositions based on user needs and environmental constraints. For this, we previously developed an approach for model-based AR-assisted product configuration based on the concept of Dynamic Software Product Lines. Based on this approach, we have shown the tool-support ProConAR with a Product Modeler and a Product Configurator. For both, we have shown a solution architecture together with the most important parts of the technical implementation.

4.1 Research Directions

We want to extend our approach and our tool support so that the underlying concepts can be used domain-independent across VR and AR apps. For that, we improve the modeling of the user and the environment by separating them into distinct models as represented in our research paper. Based on that, we want to transfer our model-based approach into a model-driven framework that can generate code for different platforms. These platforms can be all kinds of mixed reality and allow a fluid usage of product configurations between the platforms.

Furthermore, we want to extend the framework in various ways: First, we plan to configure parts of products in each other (e.g. changing the grips of a kitchen). This should support the modeling of product configuration with finer granularity and extend the possible use cases. Second, we want to add an intelligent obstacle detection based on machine learning (e.g. detecting sockets, water connections). This should improve the constraints space of the possible product configurations with the environment. Third, we add support for collaborative product configuration (e.g. changings of the product configuration by the customer and the kitchen salesman). This will allow a multi-user experience in the configuration of products by taking also a product validation based on different user feedback into account.

References

1. Apel, S., Batory, D., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
2. 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)
3. 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)
4. Dacko, S.G.: Enabling smart retail settings via mobile augmented reality shopping apps. Technological Forecasting and Social Change **124**, 243–256 (2017)
5. 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)
6. Gottschalk, S., Yigitbas, E., Schmidt, E., Engels, G.: Model-based Product Configuration in Augmented Reality Applications. In: Human-Centered Software Engineering. Springer (2020)
7. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer **36**(1), 41–50 (2003)