Computer Science and Information Systems 14(2):517–536 DOI: 10.2298/CSIS160930015W

Visual Requirements Modeling for Cross-Device Systems

Dennis Wolters[1], Christian Gerth[2], and Gregor Engels[1]

[1] Department of Computer Science,
Paderborn University, Paderborn, Germany
{dennis.wolters, engels}@uni-paderborn.de
[2] Faculty of Business Management and Social Sciences,
Osnabrück University of Applied Sciences, Osnabrück, Germany
c.gerth@hs-osnabrueck.de

Abstract. Modern information systems have to support a variety of different device types like desktop computers, smartphones, or tablets. Furthermore, it is important to enable users to use device types that fit their needs or are suited for the tasks at hand, e.g., allowing them to use multiple devices in parallel or sequentially by switching from one device to another. Such cross-device interactions must be taken into account already during requirements analysis to ensure that they are properly addressed in later phases of development. Unfortunately, current requirements modeling techniques do not provide adequate techniques to model cross-device systems. In this paper, we present an extended form of use case diagrams able to model such systems. Using our approach it is possible to specify which device types can be used when performing a certain use case and what kinds of cross-device interactions are supported. Based on this, we show how this information can be refined by integrating extended use case diagrams with our existing approach to model cross-device interactions in process diagrams. Thereby, we explain how requirements can be modeled visually in a model-based development process for cross-device systems.

1. Introduction

Over the last few years, the number of available computing devices as well as their diversity increased rapidly. In addition to computers[3], people often own smartphones and tablets as well. With new device types like smart TVs, smart watches, or other wearables, the variety of devices will increase even more. To enable users to use multiple devices and to allow them to choose devices according to their needs, information systems are often no longer developed for a single device type like computers, instead they target multiple device types. Due to the different properties of those device types, the functionality offered on the different device types can differ. For instance, travel routes can be planned on desktop computers and smartphones, but navigation on a specific route is only possible on smartphones, because they are location-aware mobile devices.

[3] We use the term computer as a synonym for desktop computers and laptops.

Due to the availability of multiple devices, users start using information systems in a cross-device manner [3,20], i.e., by performing tasks on different devices, use them in parallel or switch from one device to another. If a software system does not support such cross-device interactions, users have to coordinate these interactions themselves, which is usually a cumbersome task because application states have to be recreated or synchronized somehow [3], e.g., by manually copying data or using a synchronization tool. For seamless interactions as described by Satyanarayanan [21], cross-device interactions must be considered at design time in such a way that the realized system coordinates the interaction and the user has little to none coordination overhead. Hence, cross-device usage of a system needs to be considered in early phases of development, e.g., in the requirements analysis.

During requirements analysis, it is typical to use UML use case diagrams [16] to depict use cases of the system to be built. Use case diagrams visualize the relation between use cases, actors, and other systems. This is sufficient to visualize the requirements of systems that allow just a single device type to access the system or if the device usage information is being neglected. Yet, in the scope of cross-device systems, where multiple different types of devices shall be used to interact with a system, the information about supported device types and interactions spanning across multiple devices has to be taken into account as well. However, use case diagrams only allow to model device usage and cross-device interactions in a very limited fashion and only by using workarounds such as mixing this information with other aspects of the modeled system, e.g., replicate a use case for every device type on which it is provided. These workarounds lead to redundancies, do not allow a proper separation of concerns, and are not expressive enough to clearly specify cross-device systems. In this paper, we present extended use case diagrams which allow modeling such systems. Our extension allows refining associations between actors and use cases so that it is specifiable which device types can be used and what kinds of cross-device interactions are supported.

In addition to use case diagrams, process diagrams are used during requirements analysis to show an integrated view of execution scenarios defined for each use case, e.g., what tasks have to be done to reach the goal of a use case. Popular process modeling languages like the Business Process Model and Notation (BPMN) allow specifying who executes a task by using pools and lanes. This can be used to represent device usage up to a certain extent, but it is not expressive enough to fully specify cross-device interactions and has various drawbacks like redundancies and unnecessary complex control flows. As a consequence, we proposed an extension of BPMN in [2]. In this paper, we show how to use our BPMN extension to refine the definitions made with extended use case diagrams. In combination these two approaches allow the visual modeling of requirements for cross-device systems.

This paper is an extended version of [28]. In addition to the parts presented in the original version, this paper includes a comprehensive overview of the terminology used in the area of cross-device systems. Based on this, we provide more elaborate explanations on extended use case diagrams and extend the approach itself to better distinguish between different forms of cross-device interactions. Additionally, we describe an iterative process to externalize the information about relevant device types in terms of an ontology, which is more expressive than taxonomy-

based externalization presented in [28]. Furthermore, we show how extended use case diagrams can be refined by using our extended process diagrams, which we introduced in [2].

The remainder of this paper is structured as follows: Section 2 describes the terminology we use in this paper. Section 3 discusses related work. Section 4 introduces the running example, shows limitations of standard UML use case diagrams, and derives requirements towards an extension, which enables the modeling of cross-device systems with use case diagrams. Section 5 presents our extension of use case diagrams. Section 6 explains how to externalize the information about device types into an ontology. Section 7 shows how extended use case diagrams can be refined by using our extended process diagrams. Finally, Section 8 concludes the paper and gives an outlook on future work.

2. Terminology

As of today, there is no common understanding of the terms device and cross-device interaction. In this section, we explain how we interpret these and related terms for this paper.

The term *device* is very generic and several definitions exist. For instance, the architecture modeling language ArchiMate [26] describes a device as a "physical IT resource upon which system software and artifacts may be stored or deployed for execution". Unfortunately, ArchiMate's definition does not take the user or interaction with the device into account, which is important for our approach, since we are concerned with systems allowing interactions across multiple devices. Therefore, we use the definition from the Model-based User Interface Glossary [1], which defines a device as *"an apparatus [...] which appears to a user as a functional unit through which to perform an interaction process. A device can include computational abilities, act as a stand-alone interactive system, or be part of a network."* Since we focus on software engineering, we require the computational abilities. Thus, when we speak of a device in this paper, we actually mean *computing device*. Based on this definition, a concrete smartphone like a Samsung Galaxy S7 would be a device while a server in a rack would not be considered as a device, since it does not support user interactions. Furthermore, the functional unit aspect is important. If a printer provides services on its own, e.g., it supports printing from a USB drive, it is seen as a device. Otherwise, the printer is not a functional unit, just periphery [1] of another device like a computer.

The term *device type* is used to *describe devices that have common properties*, e.g., input capabilities, mobility, or peripherals. This closely relates to the definition of the term platform in [1]. However, we choose to distinguish between the device type as a concept to interact with the system, and a platform as a concept to realize the system. Thus, for the term *platform*, we adopt the definition of [15] which describes it as *a set of resources for implementing or supporting a software system*, e.g., web technologies, operating systems, or other technologies like .NET or Java. This distinction is important for our approach as we consider device types before we decide which concrete platforms shall be used.

The term *cross-device interaction* is often used, but a precise definition is missing. Scharf et al. try to provide a definition in [22], but it does not include the possibility to migrate a running application to another device, which is often part of cross-device applications [14,18]. Therefore, we define cross-device interaction as the *interaction of at least one human with more than one device in the scope of the same task or goal. This interaction can be in parallel or sequentially.* A goal is achieved by performing at least one task and where a task is defined according to [1] as an activity to achieve a goal. This definition is broader than the one provided in [22], but it leaves room for various forms of cross-device interactions, including switching from one device to another while keeping the current state.

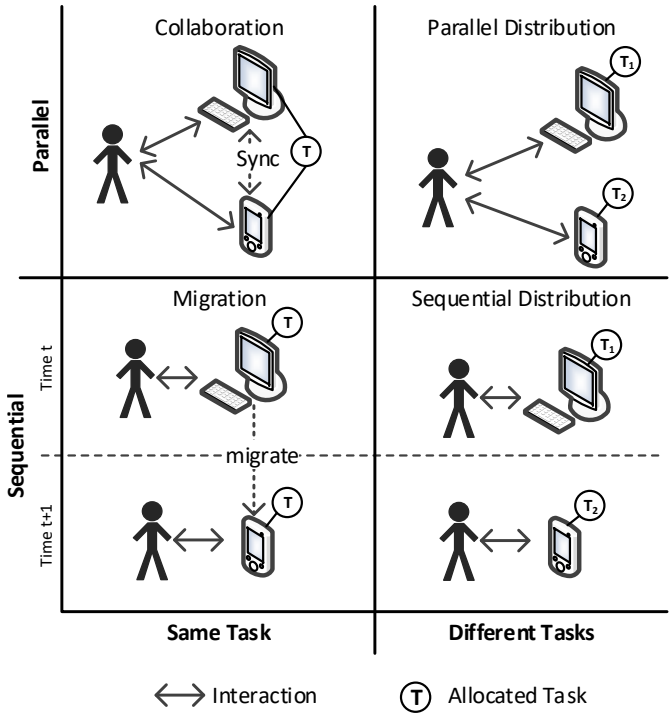

Fig. 1. Visualization of cross-device interactions

Figure 1 classifies different types of cross-device interactions: *Collaboration* describes the case where multiple devices work on the same task, e.g., editing a document on multiple devices at once. In contrast, if different tasks related to the same goal are assigned to multiple devices, we speak of *distribution*. In a *parallel distribution*, multiple devices are used simultaneously, e.g., when reviewing a paper, one device might permanently show the references, another one might be used to read the paper, while a third device is used to write the review. If different tasks related to the same goal are performed at different points in time, it is a *sequential distribution*, e.g., defining a travel route upfront on a desktop computer, while later

on doing the actual navigation on a smartphone. If a user switches from one device to another, but stays in the context of the same task, we speak of *migration*, e.g., a user starts writing an e-mail on his desktop computer and continues writing when switching to his smartphone. These four types of cross-device interactions can be refined, e.g, Santosa and Wigdor [20] describe different patterns of distribution like producer-consumer or performer-informer. If a systems supports cross-device interactions, we call it a *cross-device system*. Such systems can consist of one or more applications being deployed on multiple devices.

3. Related Work

The development of cross-device systems is supported on various levels: On the implementation level, approaches like Panelrama [30] for web applications or Conductor [8] for Android applications can be used to realize applications supporting cross-device interactions. In [13], test and debug tools for cross-device systems are provided. Instead of considering cross-device interactions at design time, approaches like [5] and [9] allow using existing web applications in a cross-device manner. However, these approaches do not work for every web application and they are limited to certain technologies. All of these approaches consider cross-device usage at later stages of development, some even after implementation, and they focus on concrete platforms, mostly web technologies. The design decisions leading to the usage of one of these approaches are based on the requirements which a cross-device system must address. Consequently, cross-device aspects of a system need to be documented during requirements analysis.

As we show in the upcoming Section 4, standard UML use case diagrams are not suited to model cross-device systems. However, several extensions of use case diagrams have been proposed: Koch et al. add stereotypes to capture concerns specific to web applications [10]. Sindre et al. introduce misuse case diagrams allowing to model safety [24] and security aspects [25]. Von der Maßen and Lichter [12] focus on modeling variability between use cases by defining alternatives for including other use cases or by specifying that the inclusion of another use case is optional. The most relevant approach for modeling cross-device systems is presented by Gopalakrishnan et al. [6]. They explore different techniques to model device usage in use case diagrams, e.g., using notes or different colors/shapes. Unfortunately, all of their proposed techniques have only very limited support to express cross-device interactions. For instance, it is possible to specify relevant device types for a use case, but not by which actor a certain device type is used or if these device types can be used simultaneously or only mutual exclusive. In [7], Gopalakrishnan et al. extend their approach to also cover the environment in which the system is used. However, they employ the same techniques for modeling the environment and the usable devices, which hinders an easy differentiation of these two aspects.

The Systems Modeling Process (SYSMOD) [27] adds the notion of a user system to use case diagrams, which enables the definition of device types used by actors to interact with a system. The relation between a human actor and a user system is modeled with information flows. This, however, does not enable the specification of different alternatives with respect to the used user system. The PLUSS

approach [4] enables modeling variability in textual use case descriptions instead of diagrams and this could be used to link use cases to certain device types. However, the relation between devices and human actors cannot be expressed; neither can cross-device interactions be specified.

Prehofer et al. [19] extend the task model language ConcurTaskTrees [17] so that it allows capturing aspects of cross-device systems. Task models, however, are not a substitute for use case diagrams. Instead, they can be used to refine single use cases with respect to interaction with end users.

UML deployment diagrams [16] allow specifying the deployment of system components and outlining communication paths between them. This can also include communication with human actors. While this allows to model device usage up to a certain extent, we cannot use this to define cross-device interactions, e.g., there is no possibility to define migration. Furthermore, deployment diagrams require that the different system components are already known, which is usually not the case during requirements analysis.

Adapt Cases [11] is an approach based on use case diagrams, which enables the specification of adaptive systems. Migrating from one device to another or distributing parts of a system can be described as an adaption of the deployment. Even though Adapt Cases can be used to express this, the description of the adaption logic is on a lower level of abstraction and not suited for the initial requirements analysis of cross-device systems. Nevertheless, Adapt Cases can be used as a refinement of the approach, which we present in Section 5, e.g., to describe automated migration or distribution rules.

4. Running Example and Requirements

In this section, we explain our running example for this paper. We use this example to explain the drawbacks of standard UML use case diagrams with regard to modeling device usage and specifying the support for cross-device interactions. Based on these drawbacks, we define requirements towards an approach, which enables the modeling of cross-device systems with use case diagrams.

In our running example, a railway company wants to provide a ticket system, which enables users to buy tickets at Ticket Vending Machines (TVMs), on computers, and with smartphones. Due to a mobile first strategy, everything that can be done with a computer shall be doable with a smartphone, too. Thus, a smartphone is seen as a special kind of computer, but in addition, smartphones are mobile devices that shall provide location-specific services, e.g., like providing live trip information when traveling by train. Consequently, in this scenario, there are use cases that are supported on smartphones but not on computers.

Since standard UML use case diagrams do not allow modeling the device types being used to perform a use case, we need to use workarounds to model this scenario, such as using concepts intended for specifying other information. For instance, by representing each device type as a separate system like in Figure 2.a. Alternatively, if we want to express that it is a coherent ticket system, we can model it like in Figure 2.b by stating in each use case on which device type it is performed. However, both approaches require multiple use cases to describe that a user shall be

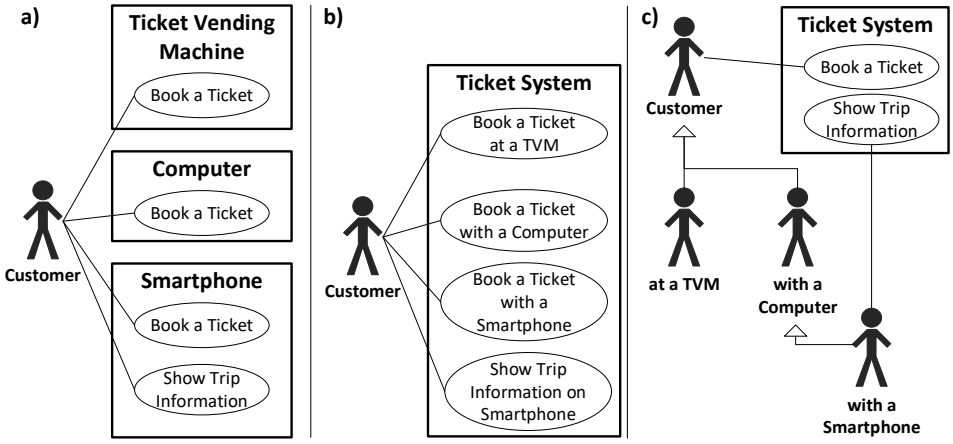

Fig. 2. Different workarounds to partially model the running example with standard UML use case diagrams: a) Depict every device type as a separate system, b) encode device type information within use cases, c) combine device type information with actor roles.

able to buy a ticket on different devices. In Figure 2.c, we define different customers: one at a TVM, one with a computer, and one with a smartphone. Thereby, we have no redundancies in the use case definitions. Furthermore, by defining a customer with a smartphone as a subtype of a customer with a computer, we express that everything that can be done with a computer can be done with a smartphone as well. Even though the last variant seems promising, once we actually want to differentiate between different kinds of customers based on other criteria than device types, we again have redundancies, but this time in the actor definitions.

Please note, for the use case diagrams depicted in this paper, we assume a default multiplicity of 1 on the actor's end of an association and 0..1 on the use case's end. Hence, by default, an actor does not have to perform the use case but if a use case is performed, the associated actors are required.

All variants depicted in Figure 2 have the same problem, they mix information about which device type is being used with other concepts: systems, use cases, or actors. Consequently, we have redundancies and no proper separation of concerns. Hence, specifying which device type a user uses to interact with a system cannot be properly modeled with standard use case diagrams.

In addition to providing the possibility to buy tickets with different devices, the railway company wants to allow the following scenario: A customer starts the booking process on his computer at home by selecting a train connection and choosing the desired ticket options. While on his way to the train station, the customer pays the ticket with his smartphone. When arriving at the train station, the customer switches to a TVM to print the ticket. This kind of cross-device interaction cannot be modeled with any of the three variants displayed in Figure 2, since there is no possibility to express that the customer is able to migrate from one device to another. In (a) and (b), migration possibilities could be specified

using «extend» associations between the use cases. However, modeling transitions between different devices with «extend» associations is not intuitive because the use cases are not extended, just the device type is changed.

Summarizing, we derive the following requirements for an approach to specify cross-device interactions in extended use case diagrams:

R1 Explicit device type modeling: Device types being relevant for the system must be modeled explicitly so that it is clear which device types are in the scope of the system.

R2 Definition of device usage: A use case might involve multiple human actors which use certain devices to interact with the system. It must be specifiable which device types have to be used and by whom. This information shall be treated separately and not be mixed with other concerns.

R3 Variability in the device usage: Human actors may have the choice between different device types to perform a use case. Such variability in the device usage must be expressible.

R4 Specification of cross-device interactions: Cross-device interactions must be specifiable, e.g., distributing a use case across different devices, the ability for devices to collaborate on the same task, or allowing migration from one device to another.

5. Modeling Cross-Device Systems with Extended Use Case Diagrams

In this section, we present extended use case diagrams which allow modeling cross-device systems. Throughout this section, we mainly focus on our running example to explain the different concepts of our approach, but we introduce further examples when necessary. The following four subsections address the Requirements R1 to R4, whereas the last subsection summarizes our approach.

5.1. Representing Device Types in Use Case Diagrams

As we discuss in Section 4, it has various drawbacks when device types are mixed with other concerns of use case diagrams, e.g., actor, use case, or system definitions. As a response to these drawbacks, we formulated Requirement R1, which states the need for the explicit modeling of device types. To address this requirement, we extend use case diagrams by adding a new model element to specify device types. In our extension, a device type is represented by a UML class named after the respective device type and tagged with the stereotype «device type». For instance, the extended use case diagram depicted in Figure 5 contains three device type classes: `TVM`, `Computer`, and `Smartphone`.

The next subsection explains how this new model element can be used to define the device types relevant for a certain use case. Furthermore, we allow to define inheritance relations between device types, which is explained in Subsection 5.3. In addition to the representation of device types within use case diagrams, we propose a separate model in Section 6 to externalize and refine information about the relevant device types.

5.2. Refining the Association Between Actors and Use Cases

In the following, we use our new model element, the device type class, to introduce device usage as a separate concern in extended use case diagrams. Thereby, we address Requirement R2 which expresses the need to model the device usage.

Within extended use case diagrams, we refine associations between actors and use cases to express which device types actors can use to perform certain use cases. Such a refinement is shown in Figure 3 where a device type class is modeled as an intermediate entity between an actor and a use case. An association between an actor and a device type specifies that the actor interacts with a device of the corresponding type, while an association between a device type and a use case defines which device type is needed to perform a use case. To get from direct associations between actors and use cases to refined associations describing the device usage, we start by using a generic device type called "Device" as an intermediate entity. This can be refined later on by using more specialized device types.

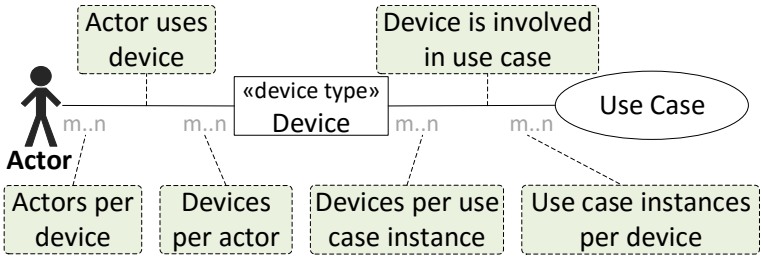

Fig. 3. Refining the association between actors and use cases to model device usage

Using multiplicities, we are able to define the relation between actors and device types as well as between device types and use cases more precisely. For instance, in Figure 4.a, we model that multiple users work on a single smartboard to edit a document. While in Figure 4.b, we specify that multiple tablets can be used but only one user is expected to work on a single tablet. We explain in next subsection how both variants can be combined to express that the users can do both, collaborate on a smartboard or contribute using their own tablet.

For standard UML use case diagrams, we define in Section 4 that the default multiplicity is 1 on the actor's end and 0..1 on the use case's end. For a refined association, we define it in a similar manner: The default multiplicity on an association between a human actor and a device type is 1 on the human actor's end and 0..1 on the other end. Thus, not every human actor has to use all devices of all types but when a device of a certain type is used, the associated actors need to be present. Similarly, the default multiplicity on associations between a use case and a device type is 1 on the device type's end and 0..1 on the end of the use case. Thereby, the default multiplicities define that not every device of a certain type necessarily has to perform all use cases to which the device type is connected but if a use case is performed, the devices of the respective types are required. In Figure 4, the default multiplicities are colored grey.

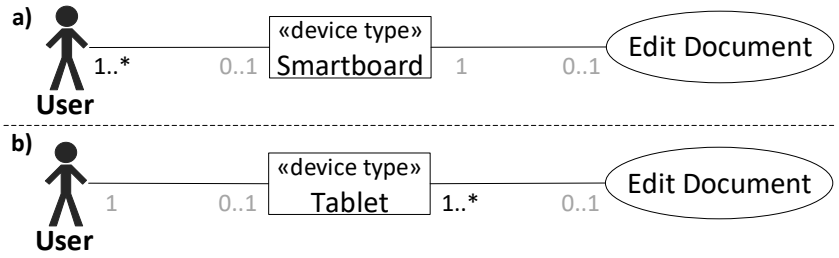

Fig. 4. Example for editing a document with multiple users: (a) All users collaborate on a smartboard and there is only one smartboard per use case instance, (b) multiple tablets are used to collaborate and there is only a single user per tablet.

5.3. Modeling Device Variability

Requirement R3 states the need to express that a variety of different device types can be used to perform a use case. To specify this variability, we offer two possibilities: (i) Either use inheritance relations to define a common ancestor of the device types that shall support a use case, or (ii) by explicitly listing all device types, which support a use case. In Figure 5, both options are used. Since the device type `Smartphone` is defined as a subtype of `Computer` (see ❶), it inherits all supported use cases, e.g., in this case booking a ticket. This, however, does not imply that inherited use cases are done in exactly the same manner. For instance, a computer shall enable a user to book a ticket, and therefore, a smartphone shall do the same, but the way how this goal is reached can vary based on the different device properties, e.g., a smartphone might allow to payment by carrier, which is not offered on computers. In contrast to the device type `Smartphone`, the device type `TVM` is explicitly associated with the ticket booking use case since it is not defined as a special type of `Computer` (see ❷).

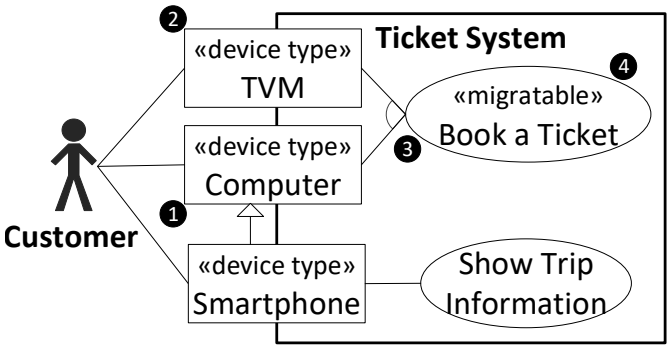

Fig. 5. Running example modeled using extended use case diagrams

To refine the variability in the device usage, we enrich the concrete syntax of UML use case diagrams with OR and XOR operators known from feature diagrams [23]. These operators are a new concrete syntax for certain UML constraints (cf. with XOR in [16]). We find it very helpful to use this syntax since it is widely known, especially when it comes to the topic of variability, and in contrast to UML constraints like XOR, the feature diagram operators imply a reading direction for the constraint. An example for an exclusive choice operator is shown in Figure 5 (see ❸). It describes that the booking of a ticket can either be done on a device of type `Computer` or `TVM` but not on both.

By listing multiple device types, we are also able to integrate the diagrams shown in Figure 4. This integrated version is depicted in Figure 6. There, an inclusive choice operator is used (see ①) to define that a smartboard, or tablets, or both can be used to edit a document.

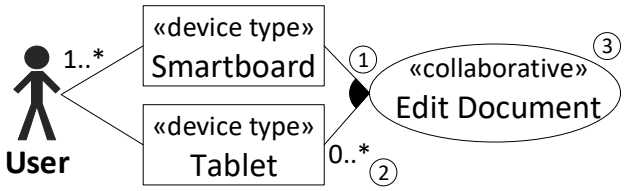

Fig. 6. Integrated version of the diagrams depicted in Figure 4

When an OR or XOR operator is used the lower bound of all multiplicities on the opposite end must be 0. Otherwise, the multiplicities contradict the operator. To avoid this in our document editing example, the multiplicity 1..∗ used in Figure 4.b changes to 0..∗ in the integrated version (see ②), because of the usage of the OR operator (see ①). If no explicit multiplicities on the opposite ends are defined, a default multiplicity of 0..1 is implied. For instance, if we would not imply a lower bound of 0 on the opposite side of the XOR operator in Figure 5, the multiplicities would specify that both a computer and a TVM are required, which contradicts the XOR operator (see ❸) that specifies that only one of these device types shall be used.

5.4. Expressing Cross-Device Interactions

Requirement R4 demands that cross-device interactions like migration, distribution, and collaboration (see Figure 1) must be specifiable. For specifying migration, we introduce the stereotype «migratable», which can be applied on use cases to define that the device being used can be substituted at runtime by another device while keeping the current state. This stereotype is applied on the use case "Book a Ticket" (see ❹) to describe that we can switch between computers, smartphones, and TVMs while booking a ticket. Similarly, to define collaboration, the stereotype «collaborative» can be applied to use cases. An example for this can be seen in Figure 6 (see ③). There, we define multiple devices that can be used collaboratively while editing a document.

The possibility to distribute a use case on multiple devices is indicated by listing multiple device types for a use case or using multiplicities with an upper bound greater than 1. In Figure 4, multiple tablets can be involved in the use case "Edit Document" as indicated by the multiplicity $0..*$ (see ②). Moreover, the OR operator (see ①) allows that both device types associated with the use case can be used at the same time. Hence, a smartboard can be used in addition or as an alternative to multiple tablets.

5.5. Summary

In summary, our extended use case diagrams include a new model element for representing device types. This is used to refine associations between use cases and actors. Variability in the device usage, e.g., possibilities to use multiple devices in parallel, are specified by defining inheritance relations between device types and by associating actors and use cases with multiple device types. To constrain variability, we add the OR and XOR operator known from feature diagrams to the UML. Additionally, we use stereotypes to mark use cases that can be done collaboratively or allow the migration to other devices.

The stereotypes introduced by our approach and the additions to the concrete syntax of the UML are summarized in Figure 7. The UML metamodel already allows to associate actors and use cases with classes (in our case representing device types). Hence, aside from our stereotypes (see Figure 7.a), which are defined as a UML profile, no additional changes to the UML metamodel are necessary. The usage of OR/XOR operators is optional, since they could be defined, in a less readable form, using UML constraints (see Figure 7.b). Please note that the OR constraint is not officially defined in the UML specification [16], but, except for a small example, there is no proper definition of the XOR constraint either.

Our approach provides a clear separation of concerns by modeling device usage as an additional dimension in extended use case diagrams. Thereby, we avoid mixing it with other concerns like in Figure 2, and allow changing the level of abstraction or the viewpoint whenever needed in the respective project. For instance, cross-device interactions may not be of interest in any case or for all stakeholders. In such cases, we can automatically abstract from device usage by replacing an association from an actor to a device type and from a device type to a use case with a direct association between the actor and the use case. Similarly, it is also possible to change the viewpoint by focusing on use cases offered by certain device types, or alternatively, by focusing on the device usage of certain use cases.

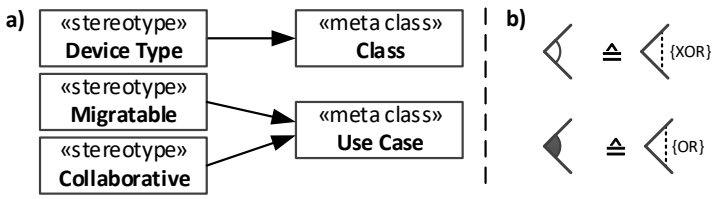

Fig. 7. a) Stereotypes added by our approach; b) Additions to the concrete syntax

6. Externalize the Device Type Information

In our extended use case diagrams, we can model relevant device types by using our new model element, the device type class (see previous section). To reuse the information about relevant device types in multiple diagrams, we propose the externalization of this information in form of a *device type ontology*. This ontology explicitly models interrelations and properties of device types relevant for the system under development. Thereby, we allow to increase the level of detail of these device type descriptions, enable reuse in other diagrams, and support the identification of device types based on their properties. In the following, we first describe the process of creating a device type ontology. Then we detail the different steps of this process, and finally, we explain how the ontology can be formalized using the Web Ontology Language (OWL).

6.1. Creating a Device Type Ontology

To create a device type ontology, we propose four steps (see Figure 8): In Step 1, we identify an initial set of device types. Subsequently, in Step 2, inheritance relations between these device types are specified. Both of these steps can be done within extended use case diagrams (see Section 5) and can be externalized afterwards. In Step 3, we define specify relevant device properties, and in Step 4, we assign them to the device types. This process is iterative to allow constant refinement and addition of new device types and properties.

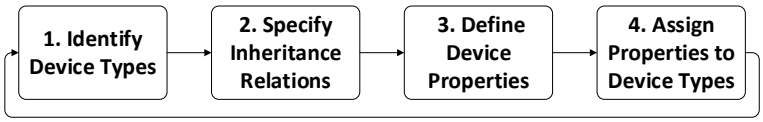

Fig. 8. Steps towards defining a device type ontology

The following subsection focuses on externalizing the information previously contained in extended use case diagrams, Subsection 6.3 explains Steps 3 and 4 in detail, and Subsection 6.4 discusses the formalization of the created ontology.

6.2. Specifying a Device Type Hierarchy

In Section 5, we explain how device types can be modeled within extended use case diagrams. Aside from introducing a new model element for device types, we also show how inheritance relations can be defined between device types. In this section, we externalize the specified device types and their inheritance relations into a device type ontology. This is particularly useful for cross-device systems supporting many different device types and when this device type information is needed beyond extended use case diagrams, i.e., the device usage information can be refined by using other diagrams as shown in Section 7.

The device types specified within extended use case diagrams are externalized to the ontology by specifying them as a subtype of a generic device type called `Device`, either directly or indirectly by inheriting from another device type. The latter allows the preservation of inheritance relations used in extended use case diagrams. The benefit of having a generic root device type is that it can be used if any device type is applicable, e.g., like when starting to refine the association between an actor and a use case (see Subsection 5.2). Figure 9 shows the externalized device type hierarchy of our running example. While `Computer` and `TVM` are directly defined as subtypes of the generic device type `Device`, we preserve the inheritance relation defined in Figure 5 by specifying `Smartphone` as a subtype of `Computer`.

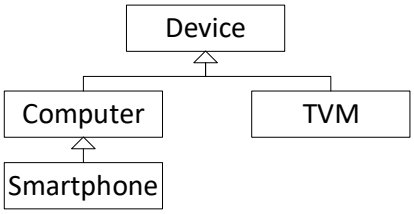

Fig. 9. Externalization of the device types specified for the running example

In each iteration, we can refine this hierarchy, e.g., by adding a new intermediate device type to highlight common aspects of existing device types. For instance, instead of directly listing `Smartphone` as a subtype of `Computer`, we could define an intermediate device type `Mobile Computer`. This can be very handy if we plan to support further mobile devices, like tablets, at a later point.

6.3. Modeling Device Properties

Once we have defined an initial set of device types, we start to identify relevant properties of these device types (Step 3 of Figure 8). These properties help to distinguish different device types and can be used to identify device types based on their properties, which is very helpful if only a certain device property is required instead of a concrete device type. Examples for this can be found in Section 7.

We use feature diagrams [23] to specify relevant device properties. A feature diagram is a tree structure where every node is a feature and the children of a node represent a refinement of their parent. By using operators like AND, OR, XOR, MANDATORY, or OPTIONAL it can be specified which features are mandatory, optional, or exclude each other. In our case, every feature represents a device property, e.g., owner of the device or supported payment methods. By using feature diagrams, we are able to demand certain properties for each device type or define them as mutual exclusive.

Relevant device properties can be derived by inspecting use cases defined for the system to be developed. For instance, the upper half of Figure 10 shows a feature diagram describing the relevant device properties for our running example. The top of such a feature diagram is labeled “Device” as it describes all possible configurations of device properties. Our example contains two optional properties, one for declaring whether the device is mobile or not, and another to indicate printing support. The former is important for showing live trip information, while

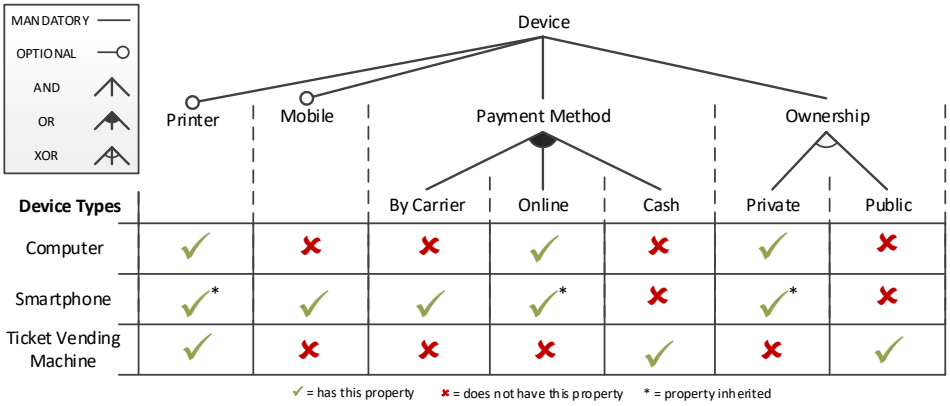

Fig. 10. Specification of device properties in terms of a feature diagrams (upper part) and assigning device properties to device types (lower part)

the latter is needed to print a ticket. Further, there is a mandatory device property "Payment method", which every device type needs to have, since all device types shall enable the purchase of a ticket. It is refined to three concrete payment methods. The OR operator specifies that each device must at least support one of these payment methods. The third property specifies whether a device is either owned by a single person or is publicly available. The mutual exclusion of these properties is defined by the XOR operator. Which device properties are modeled depends on the concrete project and on the level of detail. In our running example, the mobility and the owner of a device can influence which type of train ticket is issued, i.e., a digital ticket is not useful on public non-mobile devices, instead those devices should provide a printed ticket.

After defining device properties, we specify for each device type mentioned in the ontology if it has a property or not (Step 4 of Figure 8). Thus, every device type represents a valid configuration of our feature diagram. The lower half of Figure 10 shows the assignment of properties to the device types of our running example. We specify for the type `Computer` that it has printing support, is not mobile, only supports online payment, and that it is a private device. The type `Smartphone` is defined as a private, mobile device, supporting online payment and payment by carrier. Since it is defined as a subtype of `Computer`, it must have the same properties, but it can have properties which a `Computer` does not have like being mobile or supporting payment by carrier. The type `TVM` is specified as a public device only supporting cash payment.

6.4. Formalizing the Ontology

To leverage from the device type ontology, a proper formalization is needed. This enables the automated processing of the ontology, e.g., to query the ontology for device types having certain properties. In the following, we explain the formalization of a device type ontology using the Web Ontology Language (OWL).

Listing 1 shows the OWL specification of the device type ontology depicted by Figures 9 and 10. The basis for formalizing a device type ontology with OWL is given by predefined classes[4] for device types and device properties (see Lines 2 and 3), as well as an OWL object property allowing the assignment of device properties to device types (see Lines 4-6). To formalize device properties, inner nodes of the feature diagram are specified as subclasses of the class `dto:DeviceProperty` (see Lines 9 and 10), whereas the leafs are defined as instances of this class or the respective subclass (see Lines 12 to 18). Each device type of the ontology is explicitly defined as an instance of `dto:DeviceType` (see Lines 20 and 24) or implicitly by defining it as a subclass of another device type (see Line 28). Additionally, we define which device properties a device type has (see Lines 21-23, 25-27, and 29-30). Due to the subclass relation, we only need to define the device properties the type `Smartphone` has in addition to type `Computer`.

Listing 1. Device type ontology for the running example

```
1  #Predefined part of the ontology
2  dto:DeviceType       a                owl:Class.
3  dto:DeviceProperty   a                owl:Class.
4  dto:hasProperty      a                owl:ObjectProperty;
5                       rdfs:domain      dto:DeviceType;
6                       rdfs:range       dto:DeviceProperty.
7
8  #Definitions for the Ticket System Example
9  ex:PaymentMethod     rdfs:subClassOf  dto:DeviceProperty.
10 ex:Ownership         rdfs:subClassOf  dto:DeviceProperty.
11
12 ex:Printer           a                dto:DeviceProperty.
13 ex:Mobile            a                dto:DeviceProperty.
14 ex:CarrierPayment    a                ex:PaymentMethod.
15 ex:OnlinePayment     a                ex:PaymentMethod.
16 ex:CashPayment       a                ex:PaymentMethod.
17 ex:Private           a                ex:Ownership.
18 ex:Public            a                ex:Ownership.
19
20 ex:TVM               a                dto:DeviceType;
21                      dto:hasProperty  ex:Printer,
22                                       ex:CashPayment,
23                                       ex:Public.
24 ex:Computer          a                dto:DeviceType;
25                      dto:hasProperty  ex:Printer,
26                                       ex:OnlinePayment,
27                                       ex:Private.
28 ex:Smartphone        rdfs:subClassOf  ex:Computer;
29                      dto:hasProperty  ex:Mobile,
30                                       ex:CarrierPayment.
```

[4] Predefined parts have the prefix `dto`

7. Using Process Diagrams to Refine Use Cases

In addition to use case diagrams, there usually exist further specifications for each use case, i.e., a textual description summarizing important information about a use case. This includes a description of different scenarios that describe the execution of a use case. These scenarios can be depicted as an integrated process diagram, which visualizes what has to be done to reach the use case's goal. In [2], we have presented an extension for the Business Process Model and Notation (BPMN) that allows specifying device usage and cross-device interactions within process diagrams. Thereby, we cannot only refine use cases by defining the tasks that have to be done but also which device types are required to perform these tasks and where cross-device interactions are possible. In the following, we use our running example to exemplify this refinement.

Figure 11 shows the extended process diagram for the use case "Book a Ticket" from our running example using extended process diagrams. The process describes that we first have to select a train connection and choose the ticket options. Subsequently, we can pay the ticket using either cash, carrier, or online payment. Finally, we receive either a mobile or a printed ticket. Our extension of process diagrams allows to define the device requirements for the tasks of a process. These requirements describe the type or properties a device needs to have to execute the respective task. For instance, the task "Pay Cash" requires a device of type `TVM`, or to receive a printed ticket, we need a device having printing support. In addition, we can specify that certain tasks allow to be migrated to another device, either by transferring the current state or by restarting the respective task.

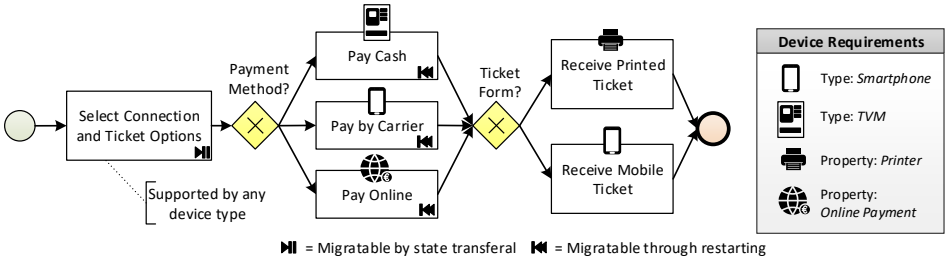

Fig. 11. Extended process diagram for the use case "Book a Ticket"

The device requirements of a task can be specified informally through textual descriptions or formally by using the device type ontology. For instance, the first two requirements in Figure 11 map to types defined in the ontology (see Lines 30 and 33 in Listing 1). Alternatively, we can specify that a device shall have certain properties, e.g., support for printing or online payment. To define device requirements formally, we specify them as SPARQL queries [29] on the device ontology. The query to retrieve all device types supporting online payment is shown in Listing 2. By executing this query on the example ontology, we get the types `Computer` and `Smartphone` as a result.

Listing 2. SPARQL query identifying device types having printing support

```
SELECT ?deviceType
WHERE {
    ?deviceType    a                   dto:DeviceType.
    ?deviceType    dto:hasProperty     ex:OnlinePayment.
}
```

The extended use case diagram in Figure 5 only describes that computers, smartphones, and TVMs can be used to book a ticket. By defining an extended process diagram for this use case, we have refined how the ticket can be booked using the different device types. Devices of all types have in common that they allow selecting train connections and ticket options. However, cash payment shall only be possible on TVMs, online payment is limited to computers and smartphones, whereas payment by carrier is exclusive to smartphones. Tickets can be printed on any device having printing support, and in addition, smartphones support receiving mobile tickets. Furthermore, we defined where the device type can or needs to be changed, e.g., when selecting cash payment on a smartphone, we need to migrate to a TVM. In [2], we also introduce *device change definitions* that can be used to constrain possible device changes, i.e., to specify that it shall not be allowed to migrate from one TVM to another.

8. Conclusion and Future Work

In this paper, we present an extension to use case diagrams enabling the specification of device usage and cross-device interactions. For this purpose, we introduce a new model element to specify device types within use case diagrams. This is used to refine associations between actors and use cases. Thereby, we specify for use cases which device types are required and by whom they are used. In addition, we enable the definition of cross-device interactions by using multiplicities and special stereotypes or by associating actors and use cases with multiple device types. The benefits of our approach are illustrated by using various examples. Moreover, we explain how the information about relevant device types can be externalized into a separate device type ontology. This ontology can be refined independently and serve as basis to reuse the device type information in other models as well. As an example for this, we show how the device usage can be refined for individual use cases by using our extended notation of process diagrams. In combination, our extensions to use case and process diagrams enable the visual modeling of cross-device systems during requirement analysis.

We are working on supporting further steps of a model-based development process for cross-device systems, i.e., modeling the architecture of cross-device systems using component models. Simultaneously, we are developing techniques to integrate services of existing applications into cross-device systems to avoid development from scratch.

References

1. MBUI - Glossary, http://www.w3.org/TR/mbui-glossary/
2. Bokermann, D., Gerth, C., Engels, G.: Use Your Best Device! Enabling Device Changes at Runtime. In: BPM 2014, LNCS, vol. 8659, pp. 357–365. Springer (2014)
3. Dearman, D., Pierce, J.S.: "It's on my other computer!": Computing with Multiple Devices. In: CHI 2008. pp. 767–776. ACM (2008)
4. Eriksson, M., Börstler, J., Borg, K.: The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realizations. In: Software Product Lines 2005, LNCS, vol. 3714, pp. 33–44. Springer (2005)
5. Ghiani, G., Paternò, F., Santoro, C.: Push and Pull of Web User Interfaces in Multi-device Environments. In: AVI 2012. pp. 10–17. ACM (2012)
6. Gopalakrishnan, S., Krogstie, J., Sindre, G.: Extending Use and Misuse Case Diagrams to Capture Multi-channel Information Systems. In: ICIEIS 2011, CCIS, vol. 251, pp. 355–369. Springer (2011)
7. Gopalakrishnan, S., Sindre, G.: Use Case Diagrams for Mobile and Multi-channel Information Systems: Experimental Comparison of Colour and Icon Annotations. In: Enterprise, Business-Process and Information Systems Modeling, LNBIP, vol. 248, pp. 479–493. Springer (2016)
8. Hamilton, P., Wigdor, D.: Conductor: Enabling and Understanding Cross-device Interaction. In: CHI 2014. pp. 2773–2782. ACM (2014)
9. Husmann, M., Nebeling, M., Pongelli, S., Norrie, M.C.: MultiMasher: Providing Architectural Support and Visual Tools for Multi-device Mashups. In: WISE 2014, LNCS, vol. 8787, pp. 199–214. Springer (2014)
10. Koch, N., Knapp, A., Zhang, G., Baumeister, H.: UML-Based Web Engineering. In: Web Engineering: Modelling and Implementing Web Applications, pp. 157–191. Human-Computer Interaction Series, Springer (2008)
11. Luckey, M., Nagel, B., Gerth, C., Engels, G.: Adapt Cases: Extending Use Cases for Adaptive Systems. In: SEAMS 2011. pp. 30–39. ACM (2011)
12. von der Maßen, T., Lichter, H.: Modeling Variability by UML Use Case Diagrams. In: REPL@RE 2002. pp. 19–25 (2002)
13. Nebeling, M., Husmann, M., Zimmerli, C., Valente, G., Norrie, M.C.: XDSession: Integrated Development and Testing of Cross-device Applications. In: EICS 2015. pp. 22–27. ACM (2015)
14. Nebeling, M., Zimmerli, C., Husmann, M., Simmen, D.E., Norrie, M.C.: Information Concepts for Cross-device Applications. In: DUI@EICS 2013. pp. 14–17 (2013)
15. Object Management Group: MDA Guide Revision 2.0 (2014), http://www.omg.org/cgi-bin/doc?ormsc/14-06-01
16. Object Management Group: Unified Modeling Language (2015), http://www.omg.org/spec/UML/2.5/
17. Paternò, F.: ConcurTaskTrees: An Engineered Approach to Model-based Design of Interactive Systems, pp. 483–501. Lawrence Erlbaum Associates (2002)
18. Paternò, F., Santoro, C.: A Logical Framework for Multi-device User Interfaces. In: EICS 2012. pp. 45–50. ACM (2012)
19. Prehofer, C., Wagner, A., Jin, Y.: A Model-based Approach for Multi-Device User Interactions. In: MODELS 2016. pp. 13–23 (2016)
20. Santosa, S., Wigdor, D.: A Field Study of Multi-device Workflows in Distributed Workspaces. In: UbiComp 2013. pp. 63–72. ACM (2013)
21. Satyanarayanan, M.: Pervasive Computing: Vision and Challenges. IEEE Personal Communications 8(4), 10–17 (2001)

22. Scharf, F., Wolters, C., Cassens, J., Herczeg, M.: Cross-Device Interaction: Definition, Taxonomy and Applications. In: AMBIENT 2013. pp. 35–41 (2013)
23. Schobbens, P., Heymans, P., Trigaux, J.C.: Feature Diagrams: A Survey and a Formal Semantics. In: RE 2006. pp. 139–148 (2006)
24. Sindre, G.: A Look at Misuse Cases for Safety Concerns. In: Situational Method Engineering: Fundamentals and Experiences, IFIP, vol. 244, pp. 252–266. Springer (2007)
25. Sindre, G., Opdahl, A.L.: Eliciting Security Requirements with Misuse Cases. Requirements Engineering 10(1), 34–44 (2004)
26. The Open Group: ArchiMate® 3.0 Specification (2016), `http://pubs.opengroup.org/architecture/archimate3-doc/`
27. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. Morgan Kaufmann (2011)
28. Wolters, D., Gerth, C., Engels, G.: Modeling Cross-Device Systems with Use Case Diagrams. In: CAiSE'16 Forum. CEUR Workshop Proceedings, vol. 1612, pp. 89–96. CEUR-WS.org (2016)
29. World Wide Web Consortium (W3C): SPARQL Query Language for RDF (2008), `http://www.w3.org/TR/rdf-sparql-query/`
30. Yang, J., Wigdor, D.: Panelrama: Enabling Easy Specification of Cross-device Web Applications. In: CHI 2014. pp. 2783–2792. ACM (2014)

Dennis Wolters received his B.Sc. degree in Computer Science in 2010 and his M.Sc. degree in 2012 from Paderborn University, Germany. He is currently a Ph.D. candidate and teaching assistant at the Chair of Database and Information Systems at the same university. His research interest include mobile solutions, service-oriented computing, and model-driven software development with a focus on engineering cross-device systems.

Christian Gerth received his Ph.D. in Computer Science, with a dissertation on business process model change management from the Paderborn University in 2012. From 2007 till 2010, he worked as a research assistant in the Business Integration Technologies group at IBM Research - Zurich, Switzerland. From 2011 till 2014, he contributed his respective knowledge to various industrial software engineering projects of the s-lab-Software Quality Lab in Paderborn, Germany. Since 2014, he is Professor of Information Systems and Software Engineering at Osnabrück University of Applied Sciences. His research interests include model-driven software development with a focus on mobile apps and web applications, business process management, and requirements engineering.

Gregor Engels received his Ph.D. in Computer Science in 1986 from the University of Osnabrück, Germany. Between 1991 and 1997 he held the position of Chair of Software Engineering and Information Systems at the University of Leiden, The Netherlands. Since 1997, he is Professor of Informatics at the Paderborn University, Germany. He is chairperson ofthe Software Innovation Lab, the university part of the technology transfer institute Software Innovation Campus Paderborn (SICP). His research interests are in the area of model-driven software development, software architecture, and software quality assurance.

Received: September 30, 2016; Accepted: April 18, 2017.