Generating App Product Lines in a Model-Driven Cross-Platform Development Approach

Jan C. Dageförde  
University of Münster  
Münster, Germany  
Email: jan.dagefoerde@ercis.de

Tobias Reischmann  
University of Münster  
Münster, Germany  
Email: tobias.reischmann@ercis.de

Tim A. Majchrzak  
University of Agder  
Kristiansand, Norway  
Email: tima@ercis.de

Jan Ernsting  
University of Münster  
Münster, Germany  
Email: jan.ernsting@ercis.de

Abstract

Within software product lines (SPLs) similar software products are created based on common features. We applied this versatile approach to cross-platform app development by extending the domain-specific language (DSL) of an established model-driven development framework. The goal was to support the formulation of coherent building blocks of business use cases, referred to as workflow elements. While the former implementation already abstracted from technical details and provided the possibility to reuse low-level features, it now enables to build business apps by combining coherent, self-contained workflow elements. Providing this support on the language level facilitates reusable component-based development. In this paper, we sketch the enhanced framework’s background and discuss SPL implications for cross-platform app development. We explain in detail how language-level modularization is realized and discuss implications, limitations, and open questions.

1. Introduction

In software product lines (SPLs), software is systematically modularized into coherent features that can be combined in different ways to assemble software products [1, p. 11]. By using a model-driven software development (MDSD) approach, the modularization of SPLs can be made explicit on a language level: The developer can specify building blocks and their interrelations using a domain-specific language (DSL) for dedicated programming [2, p. 27ff.].

In MDSD, after specifying a model for a (software) product in a DSL, a generator is responsible for automatically generating the actual application from that model [3]. During the generation process, relations of building blocks are implemented automatically, while all building blocks that are not used in the generated product are omitted. Consequently, the generator can ensure that each generated product consists only of those parts that are directly relevant to it [4, p. 181].

Developing for mobile platforms such as Android, iOS, and Blackberry typically does not bring SPL to mind. However, supporting several platforms by an app requires this app to be developed natively for each platform, to realize a Webapp, or to use a cross-platform app development framework [5]. Moreover, due to platform fragmentation, even developing for just one of them – in particular, for Android – might require multiple implementations to reach all users. In the worst case, development effort grows linearly with the number of supported platforms. Spending much effort is as undesired by practitioners as falling back to “least common denominator” solutions, for instance Webapps [6]. With increasingly complex mobile application scenarios, investing in development approaches that lead to inferior apps is no viable option. Using frameworks such as Apache Cordova [7] (a.k.a. PhoneGap) mitigates some of these disadvantages, but typically does not yield a native result despite all efforts [5]. However, exactly this is typically sought by companies [6].

To our knowledge, SPL has not been considered in this context so far. On the one hand, this is surprising given the possibilities to reduce design and implementation effort. On the other hand, mobile computing in general seems to have a tendency to leave out lessons learned from software engineering research. The research question we pursue is: How and with which means can we enhance an MDSD approach to facilitate SPL engineering? Our work is motivated by requirements of a local industry partner who develops software for the public sector in an SPL fashion. For discussions with customers about details of individual software products, they need a suitable high-level language that expresses combinations of new or existing models, while abstracting from low-level implementation details. With this request, we set out to apply SPL for app development based on a use case provided by the company. Starting from these considerations, we then identified the problem at hand in accordance with Design Science Research [8].

Using the agile software development approach Scrum, we iteratively enhanced the MDSD approach to enable SPL engineering. Our partner reviewed and evaluated the increments on a biweekly basis throughout the three months of development. This provided timely and valuable feedback to continuously evolve the resulting enhancement, but also helped validate intermediate proposals or correct approaches that required improvement.
As starting points, we had the business requirements and chose the model-driven cross-platform framework MD$^2$ [9]. While it is a research prototype [10], its inheritance of proven MDSD techniques makes it perfect for the introduction of SPL techniques. In particular, we facilitate the assembly of building blocks in arbitrary ways during the development of software products, thus enabling the creation and maintenance of SPLs. Furthermore, our second extension enables transparent development of interrelated apps using a backend for orchestration. The generator of MD$^2$ is designed in a cross-platform way, i.e., apps developed with the MD$^2$ framework are generated for prevalent mobile platforms [9]. Interrelated apps can be used on all supported platforms without additional effort.

Our work makes several contributions. Firstly, it combines the benefits of SPLs with those of cross-platform app development. Secondly, it provides detailed insights on how MDSD can facilitate the application of SPL techniques. Thirdly, it describes the enhancement of an existing approach. Fourthly, it extends the knowledge by a thorough discussion of implications. In this regard, we argue that model-driven cross-platform development of business apps lends itself as a novel application domain for SPL.

This paper is structured as follows. In Section 2, we explain the background of business app development using MDSD. Section 3 motivates modularization and explains how we propose it to be done. Section 4 describes how workflows can be coordinated across applications. We then comprehensively discuss our findings to draw a conclusion in Section 5.

2. Model-Driven Business App Development

Model-driven development allows deriving artifacts from a model and is also known as MDSD [4]. Models can be of graphical or textual nature. Artifacts can be mere software configurations and range up to fully-fledged information systems. As we will also underline in this paper, model-driven development nicely lends itself to be combined with SPLs [11].

Business apps constitute a subset of applications that are run on mobile devices such as smartphones. They exhibit the following key characteristics (cf. [12]):

- form-based, i.e., are designed using text-based forms;
- data-driven, i.e., process given inputs to present adequate data to their users; and
- are typically linked to at least one backend information system.

This definition is rather narrow and excludes e.g., graphically-intensive apps such as games. Nevertheless, the majority of apps developed by businesses for use by their partners and customers adhere to it.

MD$^2$ is a model-driven approach for cross-platform business app development that features a textual DSL (cf. [13], [10]). Archetypes, i.e., models expressed in MD$^2$-DSL, comprise descriptions of model-view-controller components (MVC, explained below). Currently, iOS and Android are the target platforms for the generated apps. Additionally, MD$^2$ generates a Java backend for data storage and exchange. Since it provides RESTful interfaces, the backend can easily be replaced with existing ones (possibly written in other languages, but providing an identical interface); alternatively, the Java blueprint can be used to wrap access to other systems. In contrast to other MDSD approaches, MD$^2$ allows modelers to realize both read and write access with existing information systems rather than restricting them to a consumer role (cf. [12, p. 6f.] for a discussion of existing approaches).

MD$^2$-DSL (realized with Xtext language tools [14]) adheres to the concept of convention-over-configuration, i.e., its language elements expose reasonable defaults that can be adjusted when needed. Therefore, MD$^2$ archetypes are expanded in a preprocessing phase using Xtend [15]. Xtend is also used for the generation phase in which distinct generators transform the completed MD$^2$ archetype into compilable code for apps and the backend. The exact steps taken in the generation process are described in [10].

MD$^2$-DSL heavily draws from the well-established MVC pattern [16], [17]. As shown in Figure 1, controllers manage both the model that describes the inherent domain model and the views responsible for the user interface. The views present information which is expressed in the model (represented by the dashed line). However, the model and the views contain no reference to their controller, except for the abstracted observable role of the controller in the view (cf. observer pattern [16]), which is visualized by the dotted line. Established MVC Web frameworks encompass Spring MVC [18], Apache Struts [19], and Google’s relatively new Angular JS [20].

As a result, the MVC pattern enables MD$^2$ app developers to specify different aspects of the app separately, resulting in one archetype. Therefore, each component of the MVC pattern has its own set of files in MD$^2$. When developing the archetype, this concept allows using code on a low level.

Reusing code is possible in two ways that are typically present in non-SPL software development already. Firstly, many elements of one part of the archetype have to be specified only once and can be reused in other components of the MVC pattern. For instance, the archetype elements that describe the model and the views of an MD$^2$ app have to be referenced within the controller. Thereby, representation and alteration of data within the app is managed. Moreover,
actions are handled based on events thrown by the user interface of the app. This management and handling can easily be replicated if required elsewhere. In the same way, each archetype element within one of the low-level components (model and view) can be reused multiple times within itself and within the controller.

Secondly, code reuse is enabled by creating multiple files for the same MVC component. Thus, the files of MD\textsuperscript{2} can be structured in a way that groups specific features of an app. Having a library of such files prepares for a modularization of common features on a file level, resulting in the possibility of generating different apps from arbitrary combinations.

However, these fine-grained kinds of reuse do not get close to the intentional modularization that is given in SPLs [1, p. 11ff.]. The common software architecture is a starting point, but not enough to specify a SPL. Therefore, we extended MD\textsuperscript{2} to improve and emphasize modularization to aim towards SPLs in the narrower sense. These efforts are outlined in the following.

### 3. Modularization of Apps for Software Product Lines

In this section, we introduce a new layer to the MD\textsuperscript{2} framework. We first give a brief background, before subsequently describing changes on the language level and particularities of the control flow.

#### 3.1. Background and Opportunities

To leverage opportunities of MDSD for the application to the concepts of SPLs, we propose a modularization of the archetype’s architecture on the language level. The modularization is component-based and carried out in the context of business processes. This enables developing software-intensive systems from a set of core modules [1, p. 13f.]. The following example of an industry partner serves as a reference: A citizen is able to report an observed (public) deficiency using a mobile app by sending a description, the location, and, optionally, a picture to the local authorities for processing (cf. Figure 2). A typical scenario would be reporting a newly emerged pothole. Based on a smartphone picture, authorities could decide whether immediate action is required. This example will be used throughout the paper to illustrate our proceeding and the results.

#### 3.2. Language-Level Modularization into Workflow Elements

Pursuing this idea of language-level modularization in the context of SPLs, the need arises to specify multiple apps from different building blocks on the language level, instead of on file level. We aimed for building blocks that encapsulate coherent high-level business process steps in order to make the language conceivable for persons in a business environment who are not necessarily familiar with (or at least trained for using) programming languages. Consequently, the level of abstraction is increased, so that these building blocks can be combined to represent business processes or workflows. To emphasize this, we name these building blocks workflow elements.

Following our use case, building blocks could be the detection of the current location or capturing a photo that supports the report of the deficiency. However, the former implementation of the MD\textsuperscript{2} framework did not allow expressing the modularization into workflow elements. Furthermore, it is not accounted for by the concepts underlying the MVC pattern, either. Therefore, we introduce the workflow elements as a new archetype element within the controller component of MVC to structure the behavior of an app. Each workflow element serves as an autonomous controller including a coherent part of business logic.

However, a single workflow element is not sufficient to represent complex business processes. Hence, in a next step, we specify a way to interconnect different workflow elements to follow a common control flow. After checking back with the industry partner, we extend the archetype’s architecture and add a new Workflow layer to our MVC pattern. This way we deliberately separate the modeling of the technical features in the archetype’s controller from the business process design in the new workflow layer. This layer is built on top of the controller, as illustrated in Figure 3, and configures the composition of workflow elements and their interconnections using a simple and straightforward language.

Workflow elements are connected via a concept that we name workflow events. Each workflow element can fire
different workflow events based on internal conditions, for example depending on form data entered by the user. In this way, the workflow events fired in the context of the workflow elements represent an interface for a later interconnection with other workflow elements. In the new workflow layer, the effect of such a workflow event can be specified, which can be either the invocation of another workflow element or the termination of the whole workflow instance.

Invoking a workflow element means passing the control flow to its controller. As a consequence, in turn, the controller loses control by firing a workflow event, because then the so-called workflow handler is responsible for coordination across workflow elements. Listing 1 shows the new structure of workflow elements in the MD[^2]-DSL and how workflow events can be fired within a workflow element. Furthermore, Listing 2 shows how the fired events are handled within the workflow layer, enabling the construction of larger processes.

At this point a combination of multiple building blocks to an app, as needed for an SPL, is still only possible on a file level. To solve this issue we introduce another archetype element, the App. This element enables the specification of multiple workflow elements in order to include them into an app. In contrast to the former process of app generation, where every aspect of the controller files was combined into one app, now only the defined workflow elements are utilized when generating the respective apps. In this way, an app developer can define many workflow elements encapsulating certain business process steps, which are then combined in arbitrary ways to form various apps.

Furthermore, the developer can define which workflow elements may be started manually by a user. If the keyword `startable` is added to a workflow element, a button is generated into the app’s user interface. Each of these buttons functions as an entry point to the workflow and will start a new workflow instance at the respective workflow element. In addition, the text on the button can be set. Listing 3 illustrates how the app element can be used within the MD[^2]-DSL.

Due to the proximity of the app element to the new workflow layer within the app design process, this new archetype element is placed within the workflow layer. To generate an additional app with specific features, it is sufficient to create a new file for the workflow layer, which defines the needed workflow elements within the app element as well as their interconnection by handling the fired events.

---

**Figure 3. Architecture of MD[^2] archetype components.**

**Listing 1. Structure of a workflow element.**

```java
1 WorkflowElement LocationDetection {  2   defaultProcessChain LocationProcessChain  3     onInit {  4       initMappings,  5       initButtons  6   }  7     action CustomAction initMappings {  8       // Mappings of model to view elements  9       <...> 10   } 11     action CustomAction initButtons {  12       bind action FireEvent{LocationEvent} on 13       LocationVerifyView.Buttons.Next2. onClick 14       bind action FireEvent{WithoutPhotoEvent} on 15       LocationVerifyView.Buttons. SkipUpload.onClickListener 16     } 17     action CustomAction saveComplaint {  18       // Save content providers  19       <...> 20   }
21 processChain LocationProcessChain {  22     step LocationDetection:  23       view LocationDetectionView  24       proceed {  25         on LocationDetectionView.Navbar.Next. onClick 26         then saveComplaint } 27     step LocationVerify:  28       view LocationVerifyView  29       reverse on LocationVerifyView.Buttons. Previous.onClick 30   }
```

**Listing 2. MD[^2] archetype describing a workflow based on the use case.**

```java
1 WorkflowElement LocationDetection 2   fires LocationEvent { start MediaCapturing } 3   fires WithoutPhotoEvent { start SubmitComplaint } 4   </WorkflowElement MediaCapturing 5   fires MediaCapturedEvent { start SubmitComplaint } 6   </WorkflowElement SubmitComplaint 7   fires SubmitEvent { end workflow }
```

**3.3. Control Flow Granularities**

Until now, it was possible to specify fine-granular actions in the controller. Controllers express the business logic of apps. Instead of having the components for business logic specifications unstructured on the highest hierarchical level, the new archetype element of workflow elements structures the behavior of the archetype within the controller. Before introducing the concept of workflow elements, we already used a new archetype element called workflow, which allowed for an easy concatenation of views on a conditional basis. This is an improvement in usability of the DSL, since transitions between views get more transparent to the modeler.

---


[4] The MD[^2] framework is a modeling language that allows for the specification of workflows and the generation of code based on these specifications.
In view of the former workflow component, the question of the granularity of the new workflow elements arises. In the context of business processes, the workflow elements represent coherent sub-processes, which therefore may comprise not only of one single data form, but of multiple views and actions. The combination of these sub-processes should be possible not only by software developers, but by (business) domain experts. For this reason, the workflow elements encapsulate technical details like data storage and user interface behavior. We decided to keep the concept of the workflow, since it is still necessary to be able to concatenate multiple views within a workflow element. However, due to the naming conflict with the workflow elements, the workflow component is now called process chain.

Other controller elements, like content providers or remote connections (these MD²-specific concepts are elaborated in [12, p. 8]), are not specific to individual workflow elements and therefore remain on the highest hierarchical level of the controller DSL. In this way, they can be referenced within the workflow elements. An example of a controller using the two new archetype elements, workflow element and process chain, is shown in Listing 1.

For the design of the textual language for the new workflow layer, we used a flow-oriented structure comparable to that of Business Process Model and Notation (BPMN) as orientation. Drawing a comparison of our introduced concepts to the BPMN standard, a workflow element can be matched to a sub-process activity of BPMN. They are structured similarly in terms of granularity, since a sub-process activity is only an abstract representation of more detailed processes underneath [21]. The workflow element behaves similarly, as the concept of process chains can be used to represent a more detailed level of the logic of business processes. Therefore, this design emphasizes the separation of the modeling process for business experts on the one hand and programmers on the other. In contrast, this separation is not explicated in common graphical process or workflow modeling approaches (cf. BPMN or ConcurTaskTrees [22]).

Figure 2 shows a BPMN implementation of our use case. It can be separated into three steps: First the citizen reports the deficiency. Then the complaint is reviewed by the local authorities, which may result in maintenance activities. Finally, some feedback is redirected to the citizen. The first part of this BPMN model can be translated in an app by creating three distinct workflow elements, location detection, media capturing, and submit complaint.

To achieve the component-based approach of SPLs, workflow elements can easily be added or deleted by adapting the workflow definition. An example for an optional workflow element in our use case could be a step for inserting personal data. While some local authorities might require this kind of information for the reduction of prank reports, others might want to exclude it from the app due to data privacy issues.

4. Workflow Coordination across Applications

Starting from apps that are constructed by combining multiple building blocks and describing their control flow, we have expanded on this idea further: using the extended MD²-DSL, you can also generate a set of related apps that, in sequence, perform a workflow together. A central backend is used to orchestrate the workflow by coordinating control flows to the appropriate apps.

This extension is especially useful if distinct parts of one business process have to be performed by actors in distinctive organizational roles. Consider our running example: with the findings of the last section, we created an app that enables citizens to capture and describe an observed deficit and send it to the local authorities. It would now be useful to describe the authorities’ app, which receives and handles a citizen’s claim, as part of the same archetype. Both apps are in fact a part of the same workflow, each executed by another role (cf. Figure 2).

Consequently, our second extension of the MD² framework tackles exactly this: An MD² archetype can describe any number of apps comprising workflow elements (cf. Listing 3). Listing 4 is an extension of the workflow in Listing 2, but also contains the workflow elements of the local authorities’ app and therefore spans across two apps. Furthermore, Listing 4 shows that the workflow can also be passed back to the initial app (or to others): After reviewing the complaint, the workflow is passed back to the citizen app, giving a feedback about the status of the maintenance process.

The existing part of the archetype (cf. Listing 2) that specifies the workflow handling has to be changed only slightly. Now, it no longer ends the workflow after the SubmitComplaint fires a SubmitEvent, but invokes the next workflow element (ReviewComplaint). The boundaries of the apps are specified using App entries in the archetype,
Listing 4. MD² archetype describing a workflow that spans across two apps. The boundaries of individual apps are not visible here, but in the app part.

```plaintext
1 WorkflowElement LocationDetection
2   fires LocationEvent { start MediaCapturing <- }
3   fires WithoutPhotoEvent { start <- SubmitComplaint }
4 WorkflowElement MediaCapturing
5   fires MediaCapturedEvent { start <- SubmitComplaint }
6 WorkflowElement SubmitComplaint
7   fires SubmitEvent { start ReviewComplaint }
8   fires CancelWorkflow { end workflow }
9 WorkflowElement ReviewComplaint
10  fires ReviewEvent { start ReceiveFeedback }
11 WorkflowElement ReceiveFeedback
12  fires EndOfProcessEvent { end workflow }
```

thus implying app context changes between two workflow elements.

Consequently, the control flow leaves the citizen app as soon as the SubmitEvent is fired and passes the control to the app of the local authorities. Note that the authorities’ app does not have a startable element: the workflow is always started by a citizen and the control of the workflow remains there until a workflow element of another app is invoked by firing an event.

As with single-app archetypes, throwing any event closes the control flow of the current workflow element and invokes the workflow handler. This may result in starting another workflow element within the same app. The control flow logic of the workflow handler within each app is automatically generated. Consequently, it is responsible for workflow element changes within this app. However, if the subsequent workflow element belongs to another app, the workflow handler passes the data to the generated backend. In addition, the backend is provided with an identifier of the workflow instance and the names of the fired workflow event and the current workflow element (cf. Figure 4).

For orchestration, the backend contains all information about the workflow, e.g., which workflow element in which app has to be started following a certain workflow event being fired. To be able to continue a workflow instance when the control flow is directed to another app, that app has to inform the user about this pending workflow instance. Of course, it is not possible to force the user to directly handle the workflow’s inherent issue, since the user might be busy, possibly even because working on another workflow instance.

For that reason the backend maintains a list of all workflow instances that need to be continued in each app. Correspondingly, every app implements a view displaying this list and updating it whenever changes arise. By choosing an entry from this list, the user can continue the workflow instance (cf. Figure 4). In this case, first, the data, saved for this workflow instance within the backend, is restored locally within the content providers (cf. [13]). Then the control flow is restarted by calling the onInit actions of the workflow element and then continuing with the default process chain.

As a result, the MD² framework can generate multiple apps from workflow elements that are used jointly to fulfill a workflow with the aid of a coordinating backend. Due to loose coupling, the backend supports apps running on arbitrary platforms. Moreover, if different parts of a workflow are to be performed by actors with distinctive organizational roles, the MD² archetype can be designed such that one app per role is generated. If a role’s app continues a workflow started by others they can use the list of incoming workflow instances, similar to the worklist in workflow management systems [23], [24].

5. Discussion and Conclusion

In the following, we first present related work. On that basis we highlight findings before naming limitations. The resulting open questions are discussed to provide an outlook before we finally draw a conclusion.

5.1. Related Work

Related work can be studied from several perspectives: approaches similar to MD², the combination of MDSD and SPL, and work related to SPL in app development. As already mentioned as part of the introduction, we are not aware of any work that combines SPL, MDSD, and cross-platform (app) development. Generally work related to MD² has been discussed in the respective papers on technological aspects [13], [25] and is not repeated here.

In particular, cross-platform development approaches based on Web technology are related because they fulfill similar aims. Their versatility is facilitated by the ease-of-use of HTML5, CSS3, and JavaScript. While coding is very
straightforward, reusage is typically limited to (sometimes extensive) copy-and-paste. Some of the approaches that are closer to ours from a technological point of view are worth consideration.

Xamarin [26] and Smartface [27] feature fully-fledged programming languages (C# and JavaScript) to create apps. Smartface, however, requires developers to take into account the respective target platforms. This limits their applicability to additional platforms as their apps’ source gets convoluted with platform specific code to deal with respective style guides. Xamarin takes a twofold approach to app generation: iOS apps are output as native apps, whereas the Android ones are executed in a runtime.

WebRatio [28] is a commercial tool that runs in a hybrid environment (cf. [12, p. 6f.]). It features a graphic DSL expressed in Interaction Flow Modeling Language (IFML) [29]. WebRatio’s tool suite has been made capable of describing processes across apps by combining IFML with BPMN. This is a feature similar to the app-spanning workflows described in this paper. IFML archetypes are amenable to reuse model subsets as components. Similar to MD, IFML can be used to describe views and their corresponding bindings to user-triggered events. Moreover, both allow expressing business logic and connecting to backend systems. Despite some conceptual differences, it would be very interesting to apply SPL concepts to IFML.

Two further approaches that are rather similar to MD are applause [30] and Xmob [31]. Both are based on DSLs but, in comparison, pose some limitations. In principle, applause should be applicable to the very approach described in this paper, but it does not show current development progress. Similarly, to judge the applicability to Xmob, it is required to see more work published on it. In general, it has to be noted that virtually all approaches based on Web technology do not yield truly native results. As already argued in the introduction, this is desired in many business app scenarios.

Several papers discuss the combination of SPLs and MDSD. It is even proposed as a paradigm, coined “Model-Driven Software Product Line Engineering (MD-SPLE)” [32] or “Model-Driven Software Product Line (MD-SPL)” [33]. Along with work on the actual combination of both paradigms (such as [34]), these papers underline the feasibility of our thoughts. Besides that, our contribution is complementary and reports experiences that underline the joint paradigm.

DeltaJ [35], a flexible approach to implement SPLs in Java, can be used to focus on Android apps and a corresponding backend. Due to its technical background, it targets developers, whereas the MD2-DSL can be mastered by at least technology-affine domain experts. DeltaJ enables for more versatile apps by providing product lines of them. In that setting, it does not focus on cross-platform development, though. The ability to provide native code for platforms other than Android could be achieved through transpiling. However, any current approaches for app development are rather limited. For example, J2ObjC [36] only supports non-UI code. In contrast, providing product lines and supporting multiple platforms can both be done in one step through the extensions made to MD2.

The approach by Quinton et al. [37] applies SPLs to app development. They aim to cope with complexity resulting from differences in software and hardware features. As their product derivation process is based on a meta model, their work is rather complementary to ours. Similar work has been presented by Rosa and Lucena [38], who use reusable software components to cope with hardware heterogeneity. The paper by Albassam and Gomaa [39] confirms the applicability of SPL concepts to mobile computing, even though their focus on gaming is quite exclusive.

Summing up, several approaches exist that share some functionality or ideas with our work, even thought we could not identify any closely related work. The following elaborates how other work on SPL could be integrated with MD and how our findings might be beneficial for others.

5.2. Findings and Implications

In view of the related approaches, our work leads to several findings regarding app product line development. In addition, we identified implications on SPL research and practice.

Since we do not allow variation through configuration of internal details of workflow elements, it is debatable whether our approach fully fits to SPL. However, it “is also possible to achieve variation by replacing components with different components embodying the particular variant desired. Similarly, products with less capability may be achieved by leaving out entire components” [1, p. 65]. Consequently, the MD2 approach is applicable to SPL, as workflow elements can be exchanged for others providing varying functionality. The exchange is facilitated as their interfacing events are specified in the workflow component of the archetype. As an alternative, workflow elements can be left out completely in the configuration of the interrelations.

Defining coherent parts of software in the form of workflow elements enables to modularize a software product into building blocks. These building blocks can then be recombined for the development of other products within the same SPL by changing the way they are combined. While we have shown the general feasibility of this idea in cross-platform app development, the concept might be extended successfully to other domains. As proposed by related approaches [37], [38], the problem with device fragmentation could be overcome this way. Consider an example: future apps will not only run on smartphones and tablets but also on devices such as Smart TVs and even in car-bound systems. These systems will offer different hardware capabilities. Instead of distinguishing e.g. whether a camera is present and having the app behave accordingly, product lines of the app could address systems with a missing hardware feature, with inferior equipment, and
with state-of-the-art gadgets. This would not only offer better app quality but also relieve developers from hard-wiring switch constructions into their programs.

Using a DSL simplifies expressing the exact combination, i.e. the control flow between these workflow elements, by making the modularization explicit on the language level. Implementing the actual interactions between workflow elements is transparent to the developer. Instead, MD²’s generators are responsible for creating the corresponding source code from the archetype. Furthermore, the idea of workflow elements as building blocks can be used to describe the interaction among two or more apps. A backend, which is also generated from the archetype, takes care of the coordination of the control flow across apps. As a result, MD² can also be used to develop cross-platform apps that achieve the same workflow together, without requiring any manual implementation for coordination and data exchange.

A general question concerns the use of textual DSLs versus graphical models of other approaches. A textual layer appeases developers, who are used to writing program code. Specifying the model by writing it comes natural to them and does not require using any specific modeling tool. Instead, they can resort to their text editor of choice.

With an intentionally high level of abstraction, a textual model becomes accessible to non-programmers, too. We aim to facilitate understanding by a business audience. At the same time, we enable developers to express interrelations without having to worry about implementation details, even considering data exchange across the boundaries of an app. Deriving a graphical representation of the model for communication purposes – e.g. to discuss a workflow – is possible: MD² provides a structured meta model, thereby allowing for other syntaxes. Purely graphical models could of course be transformed to a textual notion. However, we deem the usage of textual models to be more efficient here.

In combination, the findings and the discussion points enable us to make a proposal: The enhancement of MD² in the described form enables the development of app product lines that may consist of multiple interconnected apps. The feature diagram (cf. [40]) in Figure 5 illustrates the features of our two related example apps, where each feature corresponds to a workflow element in the example archetype. In this use case, new products of the SPL can be created by defining an app that leaves out optional features in the workflow, or by creating additional workflow elements and including them in the archetype. Furthermore, if the use case is extended by more features in the future, they can be implemented as additional workflow elements and added to the workflow of corresponding apps with little effort.

5.3. Limitations

Due to the novelty of our work, limitations are unavoidable. Two kinds of limitations have to be considered: those that apply to our work and those inherent to the use of SPL concepts. Firstly, the limitations of this paper are discussed, since they can be mitigated in future work. The enhancements made to MD² required profound changes, including work on the generators. Interim versions only provide limited platform support; the original scope is to be reestablished soon.

Despite working in close contact with an industry partner, a sound concept, and elaborated testing, there is little empirical backing of our work. Admittedly, we are refining an academic prototype. Nevertheless, empirically assessing the feasibility of the approach would be highly beneficial.

At the moment, code size minimization could be improved considerably. Each app is provided with the full data model, which theoretically is not necessary since partial models are sufficient in some instances. However, at least the business logic is provided for each app in its tailored form, thus leaving out dispensable parts.

We outlined that workflow elements can be considered as similar to subprocess activities in BPMN and that elements of an MD² workflow archetype can be mapped to those in BPMN process models. Nevertheless, MD² cannot be seen as a workflow management system, since these typically comprise many features beyond workflow enactment [41], which MD² cannot provide. For example, MD² does not include any way of monitoring the execution of workflows.

In that context, it should also be noted that our current implementation does not take care of actual role management or authorization. Therefore, if different apps are used for distinct roles, the security of this approach currently relies on the deployment process: Apps pertaining to certain roles must be distributed to members of these roles exclusively.

Besides these considerations, general aspects of SPLs have to be taken into account. To achieve modularization for SPLs, suitable architectures need to be developed, concerning both the product line and the products [1, p. 77]. In this case, MD² is already developed as an open source platform and provides a ready-to-use architecture for the creation of app product lines. Nevertheless, work on MD² is not finished. Furthermore, for the product architectures, the individual

![Figure 5. Feature diagram of the example SPL consisting of two interoperating apps.](image)
workflow elements need to be constructed in a way that arbitrary recombinations are possible where needed. This is not a simple endeavor, and it is unclear whether known patterns still apply in the context of MDSD for mobile apps.

In general, it has yet to be shown whether the complexity of supporting many platforms and, on top of that, diverse hardware can really be mastered using SPL concepts. Nevertheless, it can help to refrain from a “least common denominator” style of programming in which the worst, most inconvenient, or weakest feature determines the quality realized for all targets. In fact, software products may differ in quality in accordance with their capabilities (or those of the targets). However, with regard to the immense complexity posed by the context-dependability of apps (cf. [42], [43]) there is a need for additional effort to cope with a theoretically endless number of app products. This strictly is no limitation of SPLs, but one that has to be kept in mind nonetheless.

These limitations do not harm the value of our work. In fact, they provide the subsequent tasks for future work.

5.4. Open Questions and Outlook

The above considerations directly lead to open questions and to future tasks. With regard to the application of SPLs to model-driven cross-platform app development, we will need to investigate whether further improvements can be made. SPL is a rich field in its own regard and our work in this context is not yet exhaustive. Moreover, as already initiated with this paper, we intend to provide generalizable feedback with the aim of also contributing to research on SPL.

Our contributions to MDSD will continue. Besides the development of additional generators, hardware heterogeneity (e.g., differing screen sizes of mobile devices) is a likely field for our future activities. Thus, we will apply our existing work as documented in this paper to enable a more faceted portfolio of app products from a common source.

Improved generation from existing conceptual models in business is another topic of interest. Particularly with regard to workflows as well as process descriptions and models, re-usage of existing models could prove very useful for companies. Transforming models that have not been made for software development is no straightforward task, though.

On the conceptional level, the idea of workflows spanning across app borders needs to be given more attention. The initial ideas suggest that complex workflows can be created, in which arbitrary graphs with app products being individual nodes can be realized. However, currently only linear control flows are supported. Furthermore, at any given time, only one workflow element may have the control flow of a workflow instance. As a result, parallelizing or otherwise complicated business process structures cannot be represented. Consequently, future work will tackle how arbitrary graphs can be realized language- and generator-wise.

We would also like to become able to provide a higher level of automation in apps. For example, incoming workflows could be pushed to applicable apps instead of waiting for a user (by means of an app) to ask for it. Enabling workflows with mixed pull-/push-functionality is another task that requires profound theoretical work first, though.

As argued to be a limitation, security features could (and probably should) be implemented. Authentication and authorization need to be addressed by suitable concepts and find their way into the code generators. It might even be discussed whether parts of them should be realized on the language level – without giving up its declarative nature and without burdening developers with unnecessary manual tasks.

As a middle- to long-term task, interchangeability between a DSL such as that of MDSD and BPMN or IFML would be a very interesting endeavor. This task also aligns with the integration of existing conceptual models proposed above. It is, however, rather to be seen as a general open question than an easily addressable piece of future work.

5.5. Conclusion

In this paper, we have introduced software product lines to model-driven cross-platform app development – and vice versa. We have sketched the background of MDSD, which is based on MDSD techniques, and illustrated how modularized apps should adhere to SPL concepts. Modularization on the language level led to a profound enhancement of MDSD, allowing for more versatility and even better reusability. Moreover, it has become possible to span workflows beyond the boundaries of single apps.

Following our research question, we confirmed that model-driven development approaches can be enhanced to facilitate SPL engineering using the language-level modularization approach. As our enhancements were continuously validated or readjusted in reviews by our industry partner during the iterative implementation, the results have shown to be very convenient for the development of business app SPLs. A product manager of our industry partner stated that the language-level modularization would “increase productivity in product planning activities with customers”, as this approach enables building applications from a set of pre-defined components, without “worrying about implementation details together with a customer”. Obviously, this only serves as a qualitative evaluation of our research. Consequently, a quantitative analysis of the benefits should follow. Nevertheless, our partner’s manager and we hope that our work will “offer a leading edge”.

The discussion of insights from our work has not only shown that some of our results are generalizable. It has also become apparent that tasks remain and new challenges arose. We propose to continue research on the combination of MDSD and SPL, as this seems promising for the development of apps for a variety of mobile devices. Against this
background, we will continue to contribute to MD². Moreover, we will consider further applications for SPL to areas in which it has not yet been considered.

Acknowledgments

We thank our industry partner con terra GmbH for the encouragement and support, particularly with regard to providing real-world use cases. We express our gratitude for the commitment of the students who helped implement the enhancements for MD² during their project seminar.

References