Cross-Platform Model-Driven Development of Mobile Applications with $\mathfrak{mD}^2$

Henning Heitkötter
Dept. of Information Systems
University of Münster
Münster, Germany
heitkoetter@ercis.de

Tim A. Majchrzak
Dept. of Information Systems
University of Münster
Münster, Germany
tima@ercis.de

Herbert Kuchen
Dept. of Information Systems
University of Münster
Münster, Germany
kuchen@ercis.de

ABSTRACT

Mobile applications usually need to be provided for more than one operating system. Developing native apps separately for each platform is a laborious and expensive undertaking. Hence, cross-platform approaches have emerged, most of them based on Web technologies. While these enable developers to use a single code base for all platforms, resulting apps lack a native look & feel. This, however, is often desired by users and businesses. Furthermore, they have a low abstraction level. We propose $\mathfrak{mD}^2$, an approach for model-driven cross-platform development of apps. With $\mathfrak{mD}^2$, developers specify an app in a high-level (domain-specific) language designed for describing business apps succinctly. From this model, purely native apps for Android and iOS are automatically generated. $\mathfrak{mD}^2$ was developed in close cooperation with industry partners and provides means to develop data-driven apps with a native look and feel. Apps can access the device hardware and interact with remote servers.

Categories and Subject Descriptors

D.2.3 [**SOFTWARE ENGINEERING**]: Coding Tools and Techniques; H.5.2 [**INFORMATION INTERFACES AND PRESENTATION**]: User Interfaces

General Terms

Design, Documentation, Languages

Keywords

Model-driven software development, business app, mobile

1. INTRODUCTION

Mobile devices such as smartphones and tablets have dramatically increased in popularity. What propels their versatility and adaptability, however, is not the swift growth of computational power but the applications developed for them. The number of these *apps* is growing steeply. Companies begin to embrace the opportunities of apps and develop more apps with a business purpose. Currently, at least five platforms have a relevant number of users (Android, Blackberry, iOS, Symbian, and Windows Phone). Platforms in this respect subsume operating systems, source development kits (SDK), and device-specific features. Apps have to be developed *separately* for each one. Due to profound differences in programming interfaces, libraries, and programming languages, effort increases almost linearly with the number of platforms. These resources could be used more effectively. At the same time, *not* supporting platforms relevant for their customers is problematic for enterprises. Usually, at least Android and iOS support is required for *business apps*.

Cross-platform approaches are used when an application has to be developed for several platforms. Popular frameworks for cross-platform app development are currently either based on Web technology or use native components in an interpreting environment. Web-based approaches result in apps that more or less look and behave like Web sites. They are rather mature, but lack a native look & feel. The second achieve an (almost) native look & feel; but all such-like approaches have severe shortcomings with respect to abstraction level, performance, feature completeness, or bugs [9] (see Related Work). A truly native look & feel of apps is expected by consumers and important for enterprises as we learned in interviews with partners from industry. Hence, novel cross-platform approaches need to be investigated.

Model-driven software development (MDSD) works well for PC and server scenarios, e.g., enterprise applications based on Java EE (cf. [23]). The idea of MDSD is to describe a problem in a model and generate software from this representation. The additional modeling effort is offset by easy and efficient subsequent generation. Moreover, a modeling language can be an effective tool for capturing requirements. Tool support for MDSD is sophisticated [15]. Developing n apps that share the same functionality and basic behavior but are deployed to platforms that differ in interface design, usability, and development is a suitable area for MDSD. To evaluate the usefulness of MDSD in this context, we developed $\mathfrak{mD}^2$ as a prototypical framework for model-driven cross-platform development of mobile applications. Apps are described in a domain-specific language (DSL) tailored for $\mathfrak{mD}^2$. They then undergo transformation steps towards native, platform-specific code. Eventually, they are deployed as purely native apps of the selected platforms.

In this paper, we introduce $\mathfrak{mD}^2$ and describe its develop-

SAC'13 March 18-22, 2013, Coimbra, Portugal.

ment. The paper is structured as follows: the next section describes the background of our model-driven framework, its scope, and our research approach. Section 3 gives an overview of $\mathfrak{MD}^2$'s architecture. Afterwards, we present its language in Section 4 and code generators in Section 5. Section 6 describes related work. The discussion in section 7 compares $\mathfrak{MD}^2$ to other approaches and evaluates it. It also describes first results from developing real-world apps. Section 8 concludes and outlines some future work.

2. MODEL-DRIVEN MOBILE DEVELOPMENT

Model-driven development fits nicely to the problem of cross-platform mobile development. MDSD bases the development of significant parts of a program on *models*. Automatic transformations generate source code from developer-defined models. There can also be several transformations generating code for different target platforms. App developers describe their app on a comparatively high level and this specification is translated into code for different mobile platforms. There not only is a single, cross-platform code base (i.e., the model) but also an increased abstraction level and fast development cycles. Generated apps are truly native and do not suffer from any of the problems that Web-based approaches exhibit with respect to native look & feel.

Our framework $\mathfrak{MD}^2$ implements this idea for the domain of data-driven business apps. It was developed in close contact with industry partners. We gathered typical requirements of business apps from requirements specifications of selected enterprise products of our partners. Additional interviews with developers and managers completed the list of features typically required of apps in our context. In a top-down approach, we decided on the features of $\mathfrak{MD}^2$ from this list by prioritizing the most-often found requirements. Only then did we reconcile these features with those provided by today's mobile devices to ensure the practicability of our approach. Altogether, our approach ensures that $\mathfrak{MD}^2$ satisfies the needs of business app developers.

For data-driven business apps, developers need to

- define data types, and access create, retrieve, update, and delete operations (CRUD) for these types, locally on the device as well as on a server;
- implement the user interface (UI) with different layouts, especially UIs with tabular views (tabs), and with a variety of typical UI components;
- control the sequence of UI views;
- define data bindings and input validation;
- react to events and state changes; and
- use device features such as GPS.

We focused on code generation for tablets. This is only a matter of deliberate self-restriction, no inherent limitation. Apps generated with $\mathfrak{MD}^2$ can also be run on smartphones, they merely are not (yet) optimized for these devices.

3. OVERVIEW OF $\mathfrak{MD}^2$

Developing apps with $\mathfrak{MD}^2$ proceeds in three phases, of which only the first consists of actual work for the developer. First, the developer describes the app in a textual model. Second, a code generator for each of the supported platforms transforms the model into source code for the respective platform along with necessary structural elements such as project files. At the moment, code generators for Android and iOS are available. The third phase only requires the developer to compile the generated source code. For this, he can use the corresponding development environments such as the Android Developer Tools or Xcode for iOS. The native packages can then be run on actual devices that use the respective operating system (or in a simulator for debugging). Of these three phases, only the first phase requires cognitive effort. The code generation in the second phase happens automatically in the background; the third could be automated as well. Hence, app developers can successively enhance the model and thus proceed iteratively.

Only the first two phases involve the $\mathfrak{MD}^2$ framework, while the third phase resorts to the tools provided by each platform vendor. The framework provides several components to support or execute, respectively, these phases. We have defined a language for expressing an app as a high-level model, as needed by the first phase. It is a textual, domain-specific language (DSL). In contrast to a general-purpose programing language, it is tailored to the domain of mobile applications. The abstract syntax and the textual notation of the language have been defined and implemented using Xtext [27]. Xtext is a language development environment for the Eclipse platform. Besides a parser, it also creates an editor for the language. The editor for $\mathfrak{MD}^2$ offers features such as *syntax highlighting*, *content assistance* and *validation*. Thereby, it facilitates the specification of models.

$\mathfrak{MD}^2$ runs the second phase (code generation) fully automated as soon as the developer saves his model. The parser constructs the abstract syntax of the model and provides this set of elements and their relationships to the subsequent steps of code generation. Before invoking the code generators, the model is preprocessed to simplify subsequent generation. Next, the Android and the iOS code generator traverse the model and generate the individual apps by translating it into source code. Each code generator creates the source code of the app for the respective mobile operating system — Java for Android and Objective-C for iOS. They also create files in XML, e.g., to implement the graphical user interface (GUI) in Android or to specify the data model in iOS. Moreover, project files and settings for Eclipse (Android) and Xcode (iOS) are generated. Generated apps are bundled with (static) libraries for common features.

In addition to apps, a further code generator creates a server backend based on the data model of the application. The backend can be run on a Java EE application server. While the generated backend is fully functional, it should serve as a blueprint for the remote application programming interface (API) expected by the generated apps and is intended to be adapted to the existing server infrastructure.

The $\mathfrak{MD}^2$ language and, therefore, apps created with the framework follow the Model-View-Controller (MVC) pattern. The separation of concerns is reflected in all components: the model of an app is separated into *model*, *view*, and *controller* part. In generated apps, the controller is based on an event system, which handles all user actions as well as internal or device events. Developers specify in the model what actions should be taken if a certain event occurs. This is reflected in an event bus as the backbone of the apps.

```
1  package org.example.library.models
2  entity Book {
3    title : string
4    author : string(optional)
5    isbn : string { name "ISBN" }
6  }
7  entity LoanRequest {
8    bookRequested : Book
9    email : string
10 }
```

Figure 1: Data model of an exemplary MD² app

4. LANGUAGE AND CONCEPTS OF MD²

First, basics are introduced. Then, MD² is described differentiated by model, view and controller.

4.1 From Requirements to Specification

The domain-specific language for specifying mobile applications is a central part of MD² and forms the basis for all other components. The language specification defines the concepts that are later used by developers to model the app and that are mapped to a platform-specific implementation by the code generators. Hence, this section also serves as the documentation of the scope of MD², i.e., its set of features.

The DSL was designed in a top-down manner as specified in Section 2. Among its most important design goals was the desire to significantly raise the abstraction level necessary for specifying data-driven apps compared to manual implementation. Hence, it should describe the *problem space* of apps instead of the *solution space*, which is the target of subsequent code generation. Furthermore, the DSL should enable developers to quickly implement apps that have standard requirements or are of a prototypical nature. Still, it should be powerful enough for apps with complex UI and advanced interaction scenarios. It should support all features outlined in Section 2, not just the lowest common denominator of mobile platforms, and adhere to the architecture of Section 3. MD²'s DSL meets these requirements through four main design principles: separation of MVC components, declarative style, convention over configuration, and modularity.

As defined by the general *MVC architecture*, the model of an app has to be split into three packages, one for each MVC component. A package only uses those parts of the language that deal with the respective component. The description of the app is thus on a first level structured into separate MVC components according to functional criteria and can, on a second level, be partitioned into structural components. For example, each screen of the user interface could be described in a separate file of the view component. These structural means enable a clear separation of concerns and provide the flexibility to structure the model to fit the app.

The language is *declarative* in the sense that an app developer specifies *what* the app should accomplish instead of prescribing *how* to accomplish it algorithmically [16]. Thus, all components focus on the problem space. For example, the view model describes what the user interface should look like, i.e., its GUI elements and their composition, but not how to build it in a sequence of imperative statements.

Wherever possible, the language follows the principle of *convention over configuration* by assuming default settings for certain aspects. For example, developers only need to specify the individual user interface fields of an entity if the desired presentation deviates from some automatically generated view on this entity. Due to MD²'s *modularity*, recurring aspects can be specified once and reused several times. This applies, amongst others, to view elements and to styles.

In the following, we briefly describe central elements of the language, beginning with the part for describing the data model. Figure 1 displays an example data model in MD². Like all of the following excerpts, it is part of a simple library app, in which users can request to loan a book.

4.2 Model

Entities are the main language element. Like most elements, they have a name immediately following the respective keyword (line 2 in Figure 1). They have attributes of predefined types such as `String`, `Integer`, and `Date` (lines 3–5) and single- or multi-valued references to other entities (8). Attributes and references can further be described with a name (5), which otherwise defaults to a readable version of its identifier. They can be specified as *optional* (4) instead of being required by default. Type-specific parameters can be used to restrict the range of permitted values. Besides entities, MD² allows the definition of enumeration types. In summary, the data part of the language offers the typical elements expected to describe the data model of an application. This helps developers while learning the language. At the same time, unnecessary complexity is avoided.

4.3 View

The view part specifies the UI of the app (Figure 2). The language provides two kinds of view elements: *individual content* and *container*. Content elements such as labels, form fields, and buttons (lines 3–13 in Figure 2) are grouped and arranged inside of container elements, which are called *panes* in MD². Panes are associated with a certain layout, e.g., a flow or grid layout, that determines how contained elements are arranged (2). Containers may be nested. A special kind of container often required by business apps is a pane with tabs (16). Each of its children containers will be displayed as a tab, accessible via a platform-specific tab bar. Most view elements can be parametrized to achieve a desired look. For example, labels can be styled (4) and layout parameters influence how children are arranged. View elements can be defined once and referenced by other view elements (17), thus enabling modularity and code reuse. MD²'s view part is easy to use since it utilizes well-known terms and offers them in a concise, declarative way.

A special feature of MD² is the ability to quickly create a default representation for an entity type. To achieve this, a developer can insert an *AutoGenerator* (5). He has to specify the entity type via a *content provider* (explained further below), which MD² can infer the entity type from. It then generates a standard representation that includes labels and input fields for all attributes of the type. Additionally, it creates data bindings between the view elements and the data. AutoGenerators allow developers to quickly implement an app as long as they do not have special requirements.

4.4 Controller

The controller component of MD²'s language brings together model and view and allows to describe the behavior of the app. It is also used to specify general properties such as the view to show and actions to be performed at startup (lines 5, 6 in Figure 3). *Actions* are a central element of the controller. Custom actions (8–22) execute other actions, most notably standard action types supplied by MD². When

```
1  package org.example.library.views
2  FlowLayoutPane MainView(vertical) {
3    Label bookHeader { text "Book"
4      style HeaderStyle }
5    AutoGenerator bookInfoPart {
6      contentProvider bookProvider }
7    Button loadBookBtn ("Load book from ISBN")
8
9    Label loanHeader { text "Request Loan"
10     style HeaderStyle }
11   TextInput loanEMail { label "Email address"
12     tooltip "Please provide your email..." }
13   Button loanBtn ("Loan this book")
14 }
15 ... // InfoView
16 TabbedPane TabbedView {
17   MainView -> Main
18   InfoView(tabTitle "Info")
19 }
20 style HeaderStyle {
21   fontSize 20 textStyle bold }
```

Figure 2: View model of an exemplary MD² app

instantiating a standard action type, parameters control the behavior of the specific instance. Examples are event binding, CRUD operations, data mapping, navigation within the UI, and accessing device features such as GPS. The action concept is the sole language element that does not follow a declarative language style because actions describe the behavior of the app, albeit on a high abstraction level.

Actions can be bound to events (11–14). MD² has three kinds of events: those resulting directly from user interaction with GUI elements, global events, and conditional events. User interface events occur for example when a user touches a view element or if she uses a gesture, e.g., swiping. Global events refer to changes in the state of the app, such as losing the connection to a remote server. Conditional events examine the internal state of the app and occur when a certain condition becomes true. This may be a failed validation or a more complex expression combining the state of several model and/or view elements. For this purpose, MD² includes a concise language for boolean expressions.

A central part of MD² is the data-driven nature of the framework. The connection of the app to local and remote data sources is provided with *content providers* as the corresponding language element (24–30). A local content provider refers to a database on the device administered by the app. Remote content providers refer to a server that provides access via a specified interface (25, 31). Content providers specify the type of data they provide (24), which refers to an entity defined in the data model or a primitive type and could be multi-valued. Filters allow to define a query string to select only objects that match certain criteria (26). The language offers various action types for invoking CRUD operations on content providers.

The controller is also responsible for combining model and view through *data mappings*. Mapping a view element to an attribute of a content provider (17) creates a two-way binding between the GUI field and the object managed by the content provider. For fields mapped to a data element, MD² implicitly assumes default validators as derived from the data model definition. For example, fields mapped to required attributes will be checked for non-nullness. Developers are free to attach additional or replace implicit validators. Several standard validators are included to be able to ensure well-formed input. Examples are validators to check if input can be interpreted as an integer or date value and regular expression validators (18–21). Both mappings and validators are automatically inferred for auto-generated parts of the view but can be overridden and customized.

```
1  package org.example.library.controllers
2  main {
3    appName "Library Application"
4    ... // version information etc.
5    startView TabbedView.Main
6    onInitialized init
7  }
8  action CombinedAction init {
9    actions bindEvents mapFields }
10 action CustomAction bindEvents {
11   bind action loadBook
12     on MainView.loadBookBtn.onTouch
13   bind action sendLoanRequest
14     on MainView.loanBtn.onTouch
15 }
16 action CustomAction mapFields {
17   map MainView.loanEMail to loanProvider.email
18   bind validator RegExValidator
19     // simplified and erroneous regex
20     (regEx "[a-z]+@[a-z]+\\.[a-z]+")
21     on MainView.loanEMail
22 }
23 ... // actions loadBook, sendLoanRequest
24 contentProvider Book bookProvider {
25   providerType someServer
26   filter first where Book.isbn equals
27     MainView.bookInfoPart[Book.isbn]
28 }
29 contentProvider LoanRequest loanProvider {
30   providerType someServer }
31 remoteConnection someServer {
32   uri "http://..." }
```

Figure 3: Controller model of an exemplary MD² app

The controller model also defines navigation paths through the app and restrictions on the users' navigation. This is especially important for a GUI made up of tabs, in which users may try to change tabs at all times. Managing complex navigation scenarios like this is enabled by MD²'s *workflow concept*. It accompanies simple actions for switching views that are also part of the language. A workflow consists of several steps, each associated with a container element of the view to be displayed whenever the step is active. Additionally, a step may define conditions that have to be fulfilled before the user of the app is allowed to go forward or backward. An app may, e.g., require its users to enter a valid address before leaving a tab. If an app is associated with a workflow, the workflow controls the users' navigation. The language provides several actions for navigating through the workflow that can be bound to events, such as a user *swiping* left.

4.5 Implementation of the Language

The language has been implemented as a single Xtext project. The grammar of MD²'s DSL has been defined in the grammar language of Xtext, which is an attribute grammar [14] based on rules in Extended Backus-Naur Form (EBNF) [24]. Xtext derives an implementation of the abstract syntax of the language from the grammar. This implementation is based on the Eclipse Modeling Framework (EMF) [20], a framework for implementing formal (meta-) models as Java classes. The attributes of the grammar are used to tailor the abstract syntax as implemented in EMF. In addition to the grammar definition of MD²'s language, several Java methods validate values of more complex language elements. Xtext moreover generates an Eclipse editor for the language with helpful features such as syntax highlighting, code completion, and validation. These features allow app developers to quickly specify the model of their app and help them during modeling. Furthermore, an Eclipse wizard sets up the initial MD² project structure.

As the first step of the code generation phase, a model-to-model transformation preprocesses (thus modifies) the in-memory representation of the MD² model, which has been created by the parser. As this step deals with simplifying the model for the following code generation, it operates

on the level of the language. Subsequent steps operate on the modified model. Preprocessing serves two main purposes. Firstly, it expresses implicit semantics explicitly by using appropriate language elements. For example, it inserts validator bindings for requirements derived from the data model, e.g., a not-null validator for fields mapped to a required attribute. Secondly, it expands shorthand notations; most importantly, it replaces the AutoGenerator element with a corresponding set of view elements, together with data mappings and validators. In this context, it is also responsible for resolving references to reused view elements. Preprocessing simplifies the subsequent generation because the code generators thus deal with less variation and fewer advanced concepts.

5. CODE GENERATORS OF MD²

The following subsections describe the implementation of generators for Android and iOS. Due to space restrictions, we concentrate on the conceptual mapping.

5.1 Towards Code Generation

The responsibility of the code generation phase is to create the source code of apps out of the preprocessed model. It must be compilable to a native app without modification and the suchlike packaged app must be directly installable and runnable on the respective mobile platform. This requires transforming the declarative model written in MD²'s language into source code according to the target SDK.

The development of the generators of MD² was based on a *reference implementation* as proposed by [19]. The reference implementation consisted of prototypical apps for Android and iOS that served as a "blueprint [...] for code to be generated" [19, p. 27]. It was kept minimal as far as domain functionality, i.e., parts differing between apps, was concerned and concentrated on implementing architectural elements on both mobile platforms, which would not change for different apps. Thus, we identified static, generic parts, which are identical for all apps on a platform regardless of the model. In contrast, dynamic content depends on the actual app and its MD² model. It, thus, needs to be generated.

The set of static parts for apps of one mobile operating system is part of the *target platform*. This term refers to the environment in which generated code will be executed, consisting of the operating system and its interfaces, external libraries, and MD²'s static content. The dynamic content resorts to the elements of the target platform, as it will often instantiate or subclass its elements and use its interfaces to access platform-provided functionality. The existence of a defined platform for each mobile system, especially the MD²-specific elements, notably simplifies code generation compared to generating each app from scratch. MD² defines target platforms for Android, iOS, and the backend server.

Generated apps implement the architecture outlined in the previous sections. Each code generator has to account for particularities of the respective mobile platform and find a suitable counterpart for all concepts of MD²'s language. It has to make the declarative concepts explicit by expressing their semantics with means of the target platform, mainly the object-oriented and imperative programming language endorsed by the mobile platform. A particular challenge are *concept mismatches* between Android and iOS. If the equivalents of a language concept on each platform differ significantly, the code generator has to bridge the gap in order for them to behave similar on a fundamental level while still respecting the look & feel of the mobile platform. Examples for such differences include the life cycle concept as well as the presentation of tabbed applications. Hence, the generation step allows MD² to support more than just the lowest common denominator of Android and iOS. The comparison in figure 4 shows that both apps have a native user interface while offering the same functionality.

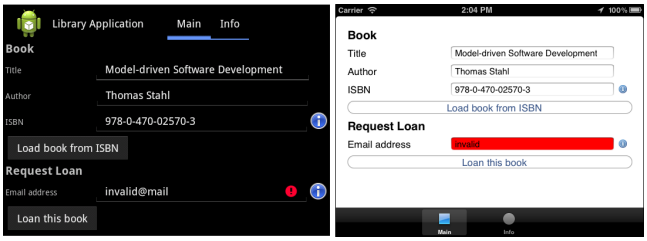

Figure 4: Comparison of Android (left) and iOS library app (right), as generated by MD² from the model in Figures 1–3 (screenshots slightly modified to save space)

5.2 General Implementation

MD² registers a build participant in Eclipse that is called when a MD² project is built after saving a model file. After preprocessing, it executes all three code generators. Generators have been implemented with Xtend [26], a Java-like programming language with a modernized syntax, additional features such as lambda expressions, and template expressions. The latter are especially helpful for generating readable code, because they facilitate working with multi-line and highly dynamic strings while maintaining formatting.

As MD²'s generators are responsible for transitioning from a declarative model to object-oriented, imperative code, their implementation follows the structure of the target application. Generally speaking, one part of a code generator is responsible for generating one kind of classes. Hence, it does not necessarily cover only one part of the language, as there can be no one-to-one correspondence. Rather, it has to integrate information from several sources. Nevertheless, there is a high degree of locality because of the separation of concerns that is present in language and architecture due to the MVC pattern. This considerably simplifies code generators.

5.3 Android Generator

The MD² library that is part of the Android target platform implements central parts of the overall architecture. This includes the event bus, workflow management, and data mapping. For each, the library defines the central class that controls the respective behavior and it offers classes to be instantiated in generated code. Furthermore, it provides base classes such as `Entity`, `CustomAction`, and `ContentProvider` that will be extended by generated app-specific classes to reflect the specifications of the model. The library also defines additional GUI elements for those content elements offered by MD² that have no direct correspondence in Android, for example input fields for time stamps. The Android generator targets Android version 3.0 and upwards.

The code generator transforms the *data model* into plain (old) Java objects (POJOs) with annotations to control the

serialization into JavaScript Object Notation (JSON). JSON is used to store data locally and for backend communication.

The *view model* is represented as Android-specific layout files in XML. In order to create a native look & feel, the generator adheres to Android's design guidelines [1]. For most content elements, corresponding Android UI components such as `Button`, `Text Field` or `Date Picker` are available. Complex elements of the view model are built up from basic Android components. For example, MD^2's input fields that allow users to select a time stamp combine date and time pickers from Android. Regarding container elements, Android's `Linear Layout` provides a good match for MD^2's flow layout. A `Table Layout` is used to emulate MD^2's grid layout (Android's `Grid View` is supposed to only be used for displaying dynamic data in a grid). Tabbed layouts are implemented via an `Action Bar`, which is Android's preferred concept for this kind of UI. The individual tabs are implemented and added to the action bar as so-called fragments, which are exchangeable modules of an user interface. Each top-level view container of the model is implemented as a separate activity as recommended by Android's guidelines.

While the UI is thus defined separately in XML, the code generator creates a *controller* class in Java for each fragment and activity. These controllers react to the Android-specific life cycle. They are responsible for initializing the view and for registering event listeners to the central event bus implemented in MD^2's Android library. Most functionality of the controller component is already provided by that library and needs to be instantiated and adapted by the app at hand. An application class manages app-wide initializations, e.g., of content providers, and executes start-up actions. The generated code contains a subclass for each custom action specified in the model. Such a class contains the statements corresponding to the tasks that ought to be performed by the action. They either call other generated actions or actions defined in the library. Further classes generated as part of the controller component deal with workflow steps and content providers. Data mapping and validation are handled by referring to appropriate actions of the library, which resort to functionality implemented in MD^2's library for Android.

In addition to generating the already mentioned XML and Java files for the different MVC components, the Android code generator copies static content and creates Eclipse-specific project files. The generated code can be opened as an Eclipse project, allowing developers to compile it using the standard Android developer tools.

5.4 iOS Generator

MD^2's library for the iOS platform is responsible for functions similar to the Android library and has a similar design. A particularity of Objective-C compared to Java is the separation of declaration and implementation into header and implementation file. This is reflected in the code generator, as each generated class is split into these two files. The iOS generator targets iOS in version 5.1. As iOS apps should follow a MVC approach anyway, the separation as prescribed by MD^2's architecture is straightforward to implement.

The iOS code generator uses Apple's *Core Data* framework [5] to implement the *data model* on iOS. Therefore, it generates an Objective-C class for each entity and a single XML file defining entities and their relationships.

Each top- or tab-level container element of the *view model* is transformed to a view class in Objective-C that builds up the respective user interface based on the UIKit framework from iOS' *Cocoa Touch*. Instead of using them directly, all UIKit view elements are wrapped in widgets that enhance their functionality and encapsulate often needed functions to simplify the generated UI classes. These widget classes are provided by the iOS MD^2 library, just as the layout classes that implement the arrangement of view elements in iOS.

In contrast to Android, the *controller logic* on iOS requires no distinction between tabs and other views. Each generated view is accompanied by a corresponding controller that initializes view and data. A base class for controllers provides most of this functionality, leading to thin view-specific controllers. An iOS `Tab Bar Controller` manages tabbed applications to achieve an iOS-specific look & feel (see Figure 4). As on Android, a central initialization class is responsible for setting up the app after start-up. It connects controllers to views, prepares content providers, and configures the app. At last, it transfers control to the initial actions defined in the model. Custom actions are implemented as subclasses similar to Android. A list of statements handles all tasks that are part of the action by calling the appropriate generated or library-provided action. Content providers, event handling, workflow management, and validation are entirely implemented in MD^2's iOS library and only need to be instantiated and configured by generated code.

The iOS code generator also generates a project file with the specific format expected by Xcode. The implementation of this part required considerable effort due to the non-standard syntax of Xcode project files[1], their complicated format, and because therein all generated files have to be listed and logically grouped in directories.

5.5 Backend Generator

Since the backend is only used for storing and retrieving data, the backend generator deals with the data model only. It creates a Java EE 6 application that can be run on an application server. The application provides a Web service that implements a REST-based interface [7]. Thereby, the backend defines the API by which remote content providers access data on servers according to the data model. Any server adhering to this API can be used as the target of a remote provider connection. The generated backend can serve as a quick starting point for a custom implementation of the server, but is also fully functional on its own with respect to all CRUD operations. The generator creates an implementation of the data model based on the Java Persistence API (JPA) as well as Enterprise Java Beans (EJB) for accessing and manipulating the corresponding database.

The actual Web service layer uses the Java API for RESTful Web Services (JAX-RS) for specifying and implementing its interface. It provides methods for retrieving data via HTTP `GET` requests, creating and updating via `PUT`, and deleting via `DELETE`. The entity name is part of the request path, and filter queries are to be passed in the query portion of the path. Data is transferred in JSON format.

Additionally, the backend generator creates an Eclipse project file and several Java EE configuration files in XML that allow rapid deployment of the application.

[1]Xcode project files neither use XML nor a similar format, but Mac OS X property lists (*p-lists*).

6. RELATED WORK

In the wide sense, our approach has to be compared to other approaches for cross-platform app development. In the narrow sense, it has to be distinguished from other MDSD app development frameworks. Finally, some tools exist that can have some similarities while following different ideas.

Cross-platform approaches fall into four different categories [9]: mobile Web app, hybrid apps, runtime environments, or generative approaches. *Web apps* are built with Web technologies (HTML, CSS, and JavaScript) and are accessed via mobile browsers. They lack access to device-specific features, except for a limited set that will be made available with HTML5 [11]. *Hybrid apps*, for example created with Apache Cordova [2] (formerly known as *PhoneGap*), package a Web site with a native component that provides access to device features. Both, Web and hybrid apps, look and behave like Web sites, because the browser engine is responsible for rendering the UI. $\mathfrak{mD}^2$ apps instead have a native look & feel. There are some approaches such as mgwt [18] or vaadin TouchKit [21] that try to mimic a native UI with Web technologies. However, they always face the fundamental limitations of running in a Web environment. *Almost*, but not exactly resembling a native app can be seen as a kind of *uncanny valley* [8]. Hence, $\mathfrak{mD}^2$ is a better match if customers expect a native look & feel.

Approaches such as Titanium [3] that use a *separate runtime environment* are, in principle, able to build up the UI with native components. On each target platform, an interpreter at runtime interprets the source code of the app written in a scripting language. However, since these imperative languages have a low abstraction level, developers need to write a large amount of code compared to the concise and declarative $\mathfrak{mD}^2$ models. Furthermore, such approaches were found to lack features and to have performance problems [9]. All aforementioned categories lack a direct native integration into the platform, because they use some kind of intermediate layer. *Generative approaches* create completely native apps out of a common code base. Besides model-driven approaches, examined below, this category includes tools that translate general-purpose programming languages. The cross-compiler XMLVM [25] transforms Android apps written in Java into Objective-C, but is still in very early stages and has a low abstraction level. J2ObjC [12] also translates Java into Objective-C, but does not consider platform-specific functionality or the UI, so that only business logic can (partly) be shared among Android and iOS. J2ObjC might be worthwhile to integrate with $\mathfrak{mD}^2$.

The model-driven *applause* [6, 4] provides a DSL for specifying apps. applause is based on Xtext and consists of code generators for iOS, Android, Windows Phone 7 and Google App Engine. However, it appears like there has not been much development progress lately. The developers acknowledge that applause's status is prototypic and far from productive usage [4]. applause is different to $\mathfrak{mD}^2$ in particular due to our domain-specific focus (on business needs). Moreover, applause is mostly restricted to displaying information.

AXIOM [13] also generates apps from a DSL. However, since AXIOM's DSL is based on the programming language Groovy and resembles features of UML, it has a rather technical appeal. Its features were derived in a bottom-up manner from the functions provided by mobile devices, rather than from business requirements. In contrast to $\mathfrak{mD}^2$, its transformations have several intermediary steps requiring additional user decisions and are hence not fully automated.

While we will not got as far as to either suggest Web apps *or* native apps and nothing in between [8], no cross-platform framework yet replaces native app development without caveats [9]. $\mathfrak{mD}^2$ in contrast has a limited application domain but provides truly native apps in every aspect.

While our approach abstracts from classical programming, it is a good idea to make it accessible to people with no technical background. Thus, work such as [10] is also related.

7. DISCUSSION

We evaluated $\mathfrak{mD}^2$ in projects with our industry partners. One project was an insurance tariff calculator as a real-world proof-of-concept, another one the library application that served as a simplified example throughout this paper.

We found $\mathfrak{mD}^2$ to be suitable for developing mobile business apps, as was the goal of our structured, top-down approach. Our own experience and feedback we got from partners prove that the scope of $\mathfrak{mD}^2$ is useful and allows to develop a large category of simple and complex apps. The increased abstraction level frees app developers from tedious low-level programming in Java or Objective-C, respectively, and from having to delve deeply into the specifics of each SDK. $\mathfrak{mD}^2$ enables a quick development process; an initial version and subsequent iterations require considerably less time and effort compared to separate native development. The DSL that forms the core of $\mathfrak{mD}^2$ provides several benefits: it follows a clear architectural separation, has a declarative style simple to understand, requires to specify only what is necessary, and lends itself to a modular definition of apps. Furthermore, being a textual language, it avoids problems typically surfacing when using graphical DSLs [17] and is particularly well-suited for collaborative development.

The library app demonstrates how quickly requirements can be transformed into working Android and iOS apps connected to a server. The actual implementation as a $\mathfrak{mD}^2$ model took less than half an hour. The tariff calculator is a comparatively large application with extensive server communication and more complex interaction scenarios. $\mathfrak{mD}^2$ enabled rapid development of initial executable versions that implemented parts of the requirements. Subsequent iterations gradually included more use cases into the model, providing immediate feedback to the developer. To illustrate the savings from modeling with $\mathfrak{mD}^2$ compared to manual implementation, we surveyed the number of lines of code (LOC) of models on the one hand and generated apps on the other hand. The latter should reasonably approximate, if not underestimate, the LOC that a manual implementation would require. While LOC are not a perfect indicator of development effort, we argue that a comparison of LOC for the same application nevertheless provides insights. The library application required a model of 109 lines, from which the Android generator generated 1359 LOC Java and 183 LOC XML (iOS: 1081 LOC Objective-C, 21 LOC XML; backend: 478 LOC Java, 57 LOC XML). For the tariff calculator, similar magnitudes of difference apply: 709 LOC in $\mathfrak{mD}^2$ lead to 10110 LOC Java and 2263 LOC XML for Android; 3270 LOC Objective-C and 64 LOC XML for iOS; and 1923 LOC Java and 57 LOC XML for the backend. This does not even reflect

the additional savings due to the platform-specific libraries provided by $\mathfrak{MD}^2$, which implement common functionality and simplify the generated code. Even after discounting for the difficulties with the LOC measure, these exemplary numbers clearly show that $\mathfrak{MD}^2$ significantly reduces the LOC to be written or, with $\mathfrak{MD}^2$, modeled.

Besides the development process, we also evaluated the apps generated by $\mathfrak{MD}^2$. They exhibit a truly native look & feel, because, as native applications, they directly use the native UI elements. Additionally, great care was taken to ensure that all $\mathfrak{MD}^2$ concepts are transformed in a platform-specific manner. For example, the layout of tabbed interfaces differs considerably in generated Android and iOS apps in order to ensure a native experience for users.

As a novel and complex approach to cross-platform development, $\mathfrak{MD}^2$ currently has some limitations. Its scope is deliberately focused on data-driven business apps with a UI mainly consisting of form fields. Regarding business logic, $\mathfrak{MD}^2$ supports typical scenarios, others have to be handled on the server up to now. In its current state, it provides access only to GPS, but not to other device-specific features.

8. CONCLUSIONS

This paper introduced $\mathfrak{MD}^2$, our approach for model-driven cross-platform app development. At first, general concepts were explained. We then described language and generators in detail. Based on the study of related work, we discussed our framework. While not being a general purpose tool that can be used for any kind of app, it has proven to be feasible for typical *business apps* even in its prototypic state.

We made several contributions. Firstly, a detailed introduction into possibilities and challenges for developing apps in a model-driven way was given. Secondly, our novel approach has been introduced. Thirdly, we highlighted details of $\mathfrak{MD}^2$ that are relevant beyond its application in our framework; the insights we got may prove helpful both for other MDSD and further app development approaches. Fourthly, we gave an abstract set of features needed by typical business apps which at the same time are reflected in $\mathfrak{MD}^2$'s language. Fifthly, we discussed our approach based on real-world projects and showed for which scenarios it is feasible.

Our work is not finished. Development of $\mathfrak{MD}^2$ will continue by stepwise expanding the class of supported applications, aiming at eventually covering most apps that base on standardized interface elements. Moreover, we will assess the framework theoretically and evaluate it with corporate partners. Another goal is to better understand in which situation what kind of framework support is needed.We would like to incorporate non-linear workflows and integrate the *generation gap pattern* [22] for advanced actions, conditions, etc. This would allow modifications to generated apps that are not lost on regeneration. At the moment, we do not provide any such means, because we intended to fully automate the platform-specific parts. A more advanced aim is to allow the specification of more business logic in models.

9. ACKNOWLEDGMENTS

We would like to thank viadee Unternehmensberatung GmbH for supporting the development of $\mathfrak{MD}^2$. Implementation was mainly carried out by master students Sören Evers, Klaus Fleerkötter, Daniel Kemper, Sandro Mesterheide, and Jannis Strodtkötter, whose effort is highly appreciated.

10. REFERENCES

[1] Android Open Source Project. Android design, 2012. http://developer.android.com/design/index.html.
[2] Apache Cordova, 2012. http://incubator.apache.org/cordova/.
[3] Appcelerator, 2012. http://www.appcelerator.com/.
[4] applause, 2012. https://github.com/applause/.
[5] Apple Inc. iOS Data Management, 2012. https://developer.apple.com/technologies/ios/data-management.html.
[6] H. Behrens. MDSD for the iPhone. In *Proc. of OOPSLA*, 2010.
[7] R. Fielding and R. Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2):115–150, 2002.
[8] M. Fowler. CrossPlatformMobile, 2011. http://martinfowler.com/bliki/CrossPlatformMobile.html.
[9] H. Heitkötter, S. Hanschke, and T. A. Majchrzak. Comparing cross-platform development approaches for mobile applications. In *Proc. 8th WEBIST*, 2012.
[10] H. Höpfner et al. Towards a target platform independent specification and generation of information system apps. *Softw. Eng. Notes*, 36(4):1–5, 2011.
[11] HTML5, 2012. http://www.w3.org/TR/html5/.
[12] J2ObjC, 2012. https://code.google.com/p/j2objc/.
[13] X. Jia and C. Jones. AXIOM: A model-driven approach to cross-platform application development. In *Proc. 7th ICSOFT*, 2012.
[14] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2:127–145, 1968.
[15] I. Kurtev et al. Model-based DSL frameworks. In *OOPSLA '06*, 2006.
[16] J. W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming (GULP-PRODE'94)*, 1994.
[17] M. Mernik, J. Heering, and A. Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 37(4):316–344, 2005.
[18] mgwt, 2012. http://www.m-gwt.com/.
[19] T. Stahl and M. Völter. *Model-driven software development*. John Wiley & Sons New York, 2006.
[20] D. Steinberg et al. *EMF: Eclipse Modeling Framework, 2nd Edition*. Addison-Wesley Longman, 2009.
[21] vaadin TouchKit, 2012. https://vaadin.com/touchkit.
[22] J. Vlissides. Pattern hatching: Generation gap, 1996. http://www.research.ibm.com/designpatterns/pubs/gg.html.
[23] J. White, D. C. Schmidt, and A. Gokhale. Simplifying autonomic enterprise java bean applications via mdd. In *Proc. 8th MoDELS*, 2005.
[24] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, Nov. 1977.
[25] XMLVM, 2012. http://www.xmlvm.org/.
[26] Xtend, 2012. http://www.eclipse.org/xtend/.
[27] Xtext, 2012. http://www.eclipse.org/Xtext/.