Science of Computer Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Extending a model-driven cross-platform development approach for business apps

Henning Heitkötter, Herbert Kuchen *, Tim A. Majchrzak

Department of Information Systems, University of Münster, Leonardo-Campus 3, 48149 Münster, Germany

H I G H L I G H T S

- We present the model-driven approach MD^2 for cross-platform development.
- MD^2 addresses business apps at a high level of abstraction.
- MD^2 reaches a platform-specific look and feel without compromising performance.
- A textual model is automatically transformed into native apps for Android and iOS.
- We focus on extensions: device-specific layout and extended control structures.

A R T I C L E I N F O

Article history:
Received 25 September 2013
Accepted 5 November 2013
Available online xxxx

Keywords:
Cross-platform development
Business apps
Model-driven development
Device-specific layout
Control structures

A B S T R A C T

Due to the heterogeneity of different platforms, it is an expensive endeavor to provide a mobile application (app) for several of them. Cross-platform development approaches can solve this problem. Existing cross-platform approaches have severe limitations and typically work on a low-level of abstraction. Our model-driven cross-platform approach MD^2 focuses on the domain of business apps and, hence, reaches a high-level of abstraction while maintaining a platform-specific look and feel. A textual model written in an MVC-based DSL is automatically transformed into native apps for Android and iOS. The present paper focuses on new extensions of MD^2, namely device-specific layout, extended control structures, and offline computing.

1. Introduction

Platforms for mobile devices are heterogeneous w.r.t. operating system, frameworks, and programming languages. Consequently, porting a mobile application to another device essentially amounts to an expensive redevelopment from scratch. However, to fulfill customer demands, developers would often like to provide a considered app on different platforms. Thus, approaches for cross-platform development of apps gain attractiveness. Several such approaches exist. They differ among others concerning their limitations, performance, usability, level of abstraction, and their ability to reach the desired platform-specific look and feel. They range from simple Web apps based on HTML5 [1] and JavaScript-based frameworks such as PhoneGap (aka Apache Cordova) [2] and Titanium [3] to model-driven approaches, which translate a platform-independent model automatically to native apps.

All of the existing approaches share a relatively low-level of abstraction, since they enable the development of general apps and do not focus on a specific domain. Moreover, they typically can only approach but not reach a platform-specific look and feel [4]. As we found out in a survey, a perfect platform-specific look and feel is essential for many users and a particular requirement of businesses that want to provide or use apps [5,6].

* Corresponding author.
E-mail addresses: heitkoetter@wi.uni-muenster.de (H. Heitkötter), kuchen@wi.uni-muenster.de (H. Kuchen), tima@wi.uni-muenster.de (T.A. Majchrzak).

0167-6423/$ – see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.scico.2013.11.013

Our approach MD^2 [7] focuses on the domain of business apps. It can hence reach a relatively high level of abstraction. MD^2 is a model-driven approach for cross-platform app development. Its basic ideas are described in Section 2. Section 3 gives an overview of the corresponding domain-specific language (DSL), which is used for creating models. In Section 4, we present new extensions of MD^2 such as advanced control logic and device-specific layout. In Section 5 we conclude and point out future work.

2. Basic ideas of MD^2

A *business app* is an application for a mobile device that supports a business process of an enterprise. It is either used by employees of the considered company or by its clients. Typically, clients use differing devices based on diverse platforms. However, also employees may use different platforms due to increasingly popular BYOD (bring your own device) policies. A business app is typically *data-oriented* and uses *form-based* input/output. Moreover, business apps are frequently connected to a *server*, which runs the core of the business logic and provides database access. When no Internet connectivity is available, a business app might need to work *offline* and *synchronize* with a server as soon as the connection is available again. Occasionally, business apps need to access *device-specific hardware* such as a GPS receiver. Business apps typically do not require animations as known from games. By focusing on business apps, our approach can reach a higher-level of abstraction [7].

In detail, a data-driven business app requires, among others:

- the definition of data types, and create, retrieve, update, and delete operations (CRUD) for corresponding data, locally on the device as well as on a server;
- a form-based user interface (UI) with different layouts, especially with tabular views (tabs), and with a variety of typical UI components such as form fields and buttons;
- control over the user navigation and the sequence of UI views;
- the definition of *data bindings* (mapping data to form fields) and input validation corresponding to the type and required structure of the considered input;
- means to react to events and state changes; and
- access to device-specific hardware.

Our approach is model driven. A textual model written in a DSL is automatically transformed into native apps for the considered platforms (currently iOS and Android). Since we generate native code based on the platform-specific frameworks and operating systems, we reach a perfect platform-specific look and feel. In contrast to, e.g., Titanium, we do not suffer from performance losses due to an additional layer of interpretation [5]. Our DSL is based on the *model-view-controller* (MVC) design pattern [8]. A model consists of three clearly distinguished parts corresponding to the model, view, and controller components of this pattern. The DSL is textual rather than graphical to enable efficient modeling and processing. Automatic transformations generate the source code of apps for each of the target platforms based on these platform-independent models.

3. Concepts and DSL of MD^2

Since our DSL and its concepts have already been described in detail elsewhere [7], the following only briefly summarizes the main ideas of MD^2's DSL and its current scope. Listings 1–3 contain excerpts of the model of a simple example app for managing contacts. The following description contrasts each functional requirement derived in a top-down approach with the corresponding language constructs that allow app developers to realize that requirement when modeling an app with MD^2.

As MD^2 deals with data-driven business apps, app developers need to be able to describe the underlying data model of their app. The data part of an app's model, i.e., MVC's *model component*, not only prescribes the structure of data used within the app, but also the API used by the server back end. MD^2's DSL offers simple standard concepts of data modeling. App modelers may specify complex data types (named *entity* in MD^2) that consist of primitive attributes and of references to other entities.

Regarding the *view component*, app developers should be able to declaratively describe the user interface (UI) of their app. Data-driven apps often consist of form fields, typically arranged in structured layouts. The view of MD^2's DSL provides *container elements* representing the layout and *content elements* for the individual UI elements. Both kinds of view elements can be nested in containers. The type of the container, e.g., *FlowLayoutPane* or *GridLayoutPane*, determines how nested elements will be arranged. Tabular interfaces as often found on mobile devices can be modeled with a *TabbedPane* whose direct child containers represent the individual tabs. MD^2 offers several types of content elements such as *Label*, *TextInput* (in different variants), *Button*, or *CheckBox*. In the generated apps, they are represented by corresponding native UI elements, where available. If a mobile platform provides no exact counterpart for a language construct – for example, a date-and-time picker on Android – MD^2's code generators are able to fill this gap by referring to an MD^2-specific implementation.

```
entity Contact {
    name : string
    phone : Phone
}
```

Listing 1: MD^2 example: model.

```
FlowLayoutPane DetailView(vertical) {
    TextInput nameFld{label "Name"}
    TextInput phoneFld{label "Phone"}
    Button saveBtn("Save")
}
```

Listing 2: MD^2 example: view.

```
contentProvider Contact[] contactsStore {providerType server}
action CustomAction init {
    map DetailView.nameFld to contactsStore.name
    map DetailView.phoneFld to contactsStore.phone.number
    bind action saveContact on DetailView.saveBtn.onTouch
    call DataAction(load contactsStore)
}
action CustomAction saveContact { ... }
```

Listing 3: MD^2 example: controller.

As the component integrating model and view, the *controller component* of MD^2's DSL is central for modeling an app. Its foremost purpose is to enable app developers to control user interaction with the app. To this end, MD^2 proposes an event-based mechanism with actions being the core language element. An *action* is a sequence of *tasks*, which may either be instances of predefined task types or references to other actions. When an action gets executed, all of its tasks will be executed sequentially. To schedule an action for execution, actions have to be bound to *events* by using a corresponding task type. For example, an action that triggers a search could be bound to the event of an app user touching a certain button. A selected action will be executed at start-up and can thus be used to initialize, besides others, the event bindings. Other types of tasks besides the already mentioned event bindings will be mentioned throughout the following.

The controller component of MD^2's DSL also addresses the requirement of business apps to access and modify local and server-side data. The construct of *content providers* provides an abstraction over data sources. A content provider represents a possibly filtered view on a data source with elements of a type from the app's model component. Using a corresponding type of task, an app developer can establish a two-way data binding between a view element and an attribute path of a content provider. Further task types allow performing CRUD operations on the data represented by a content provider. The integrity of data can be ensured by binding *validators* to input elements of the UI, which is enabled by another type of task. As MD^2 strives to follow the principle of convention-over-configuration, it also derives validators implicitly if a data binding has been established.

The preceding explanations cover the core functionality of MD^2. These language constructs enable app developers to realize a large range of business apps with MD^2. Additional features already implemented in MD^2, for example in the area of workflows or access to device-specific features such as GPS, are out of scope of this article. The following section covers desirable extensions for expanding the application area of MD^2.

4. Extensions of MD^2

Despite its very promising application in the scenarios we already implemented [7,9,10], there is much room for extending language, tools, and generators. We will describe the conceptualization of two extensions that will greatly increase the versatility of MD^2.

4.1. Specification of control logic

In the current implementation, the dynamic aspects in the controller are limited to controlling the user interaction by modeling what action to execute in response to which events. While this enables complex user-induced control flow and has proven sufficient for many common scenarios, the app itself does not make decisions, except in limited areas such as workflows and validation. For example, it would not be possible for the app itself to choose one of several possible navigation paths depending on the value of some form element. These limitations are due to the design decision that business logic resides on back end servers, whose responses implicitly control the behavior of apps. This architecture integrates well with a business environment in which enterprise applications already provide the business logic and companies strive to keep it secured. There are, however, situations where app developers want to model application logic directly in MD^2 so that apps perform offline computing.

There are several DSLs that include behavioral constructs, for example, the Mawl language for form-based Web services [11]. However, our goal is to embed respective constructs into an existing language, namely, MD^2. Thus, other DSLs only help in identifying useful approaches to achieve this goal. Spinellis gathered design patterns for DSLs [12], of which the *piggyback pattern* might be helpful in our scenario. It describes the reuse of an existing language for common elements. Xtext [13], the language development environment used by MD^2, for example already provides the expression language Xbase (including loop and *if* expressions), which we could integrate. However, while such a complete set would increase the expressiveness of MD^2 assuming adequate means of integration with existing language constructs, it would also significantly

```
action CustomAction gcdEuclid[not calcStore.b == 0] {
    calcStore.tmp = calcStore.a % calcStore.b
    calcStore.a = calcStore.b
    calcStore.b = calcStore.tmp

    call gcdEuclid
}
```

Listing 4: Exemplary use of planned extension to calculate the gcd.

increase its complexity. An adequate solution needs to integrate well with the event-based paradigm already used by MD^2, but also with the object-oriented paradigm supported by the target platforms.

Instead of including imperative-like if or loop constructs into MD^2, the following three measures already serve to capture the requirements:

1. support the conditional execution of actions;
2. expand the language with respect to Boolean, relational, and arithmetic expressions; and
3. enable recursive calls of actions.

These extensions make MD^2 Turing-complete while maintaining its succinct syntax and simplicity.

We propose to extend the syntax of action definitions with an additional guard condition. The conditional expression may reference the state of view elements and of content providers. At runtime, the action can only be executed if the condition holds.[1] Listing 4 demonstrates the planned extension with an implementation of the Euclidean algorithm for calculating the greatest common divisor (gcd) of two natural numbers. As it implements a well-known algorithm, it serves to demonstrate the usefulness of the proposed language extension. Depending on the available expressions, other simple calculations should be easily representable as well, while more complex logic can still reside on servers.

The implementation of these constructs in the code generators will not prove difficult: Their counterparts are readily available in the target languages, so that no custom implementation of the recursive semantics is necessary. A further extension should introduce actions with arguments in order to enable recursion over parameters (instead of global content providers).

4.2. *Device-dependent layout*

A main challenge of developing apps for multiple platforms and devices is to adjust the layout accordingly. A differing appearance of typical elements such as text boxes and buttons is not the only challenge. Screen orientation (horizontal or vertical), aspect ratio and resolution differ even on devices that run the same mobile operation system. This not only applies to the differences between tablets and smartphones but also to devices that are similar on the first look. Therefore, a *device-dependent layout* is needed.

The proliferation of Web applications has lead to adaptive designs. The concept is commonly referred to as *Responsive Web Design* [14]. A notable example of a Web development framework for adaptable applications is Twitter bootstrap [15]. It allows applications to almost freely scale. Subject to size of the view area, element appearance and layout are adapted. For example, tables might be redrawn from a column-based layout to form distinctive tables for each former row. Due to a low level of abstraction, this approach is not universally feasible, though.

Adaptive graphical user interfaces are no new idea.[2] However, adapting to users (e.g., based on roles [17]) is different in concept to what we want to achieve. Nevertheless, the perspective on usability (cf., e.g., [18]) is a precondition for our work and we can draw from that direction of research. In particular, advanced work on automatically generated user interfaces subject to user abilities [19] is very helpful. The idea to *functionally* represent user interfaces instead of defining how they look like is conceptually similar to what we have in mind. Semantic grouping is done via container types, which is a strategy that we deem the perfect choice for our purpose.

Behan and Krejcar present work on "adaptive graphical user interface" particular in the domain of mobile devices [20]. Their usage of the MVP pattern can be seen as a best practice, as it separates distinct tasks in the design of interfaces. Their focus on the Android platform is a first step but only partly helpful when trying to achieve comparable results in a cross-platform scenario. Coutaz combined model-driven techniques with interface design and coined the term "Interface Plasticity" [21]. Analogously to our strive for a high level of abstraction, Coutaz demands interface components to be "meta-described", which allows them to be "transformed" [21, p. 6].

The basic idea for an adaptive device-dependent layout is quite simple and illustrated in Fig. 1. Two container elements A and B comprising of an arbitrary number of nested elements could be identified as *logic sub-structures*. This means that they can be placed independently despite some semantical connection. Their content, however, needs not be realigned. As shown in the picture, both containers might be shown on a high resolution tablet screen in landscape layout but do not fit on a smartphone screen. If B depends on A, option 1 (access through A) is preferable. If both are semantically on the

[1] Of course, one could also choose to place the guard directly in front of individual tasks.
[2] Taking account of "user style" for GUI design was discussed as early as 1977 [16].

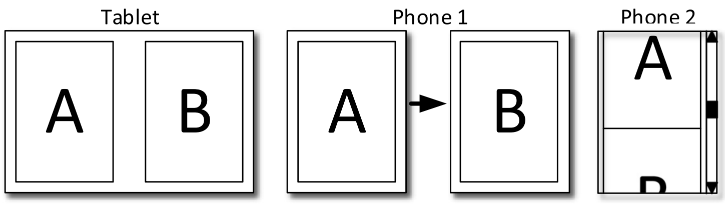

Fig. 1. Layout options.

```
FlowLayoutPane ContactMainWindow {
    FlowLayoutPane ListView(vertical) { ... }
    FlowLayoutPane DetailView(vertical, DEPENDSON ListView) {
        ATOMIC {
            TextInput familyNameFld{label "Family name"}
            TextInput givenNameFld{label "Given name"}
        }
        ...
    }
}
```

Listing 5: Snippet of the tentative syntax for the layout extension.

same level, option 2 (vertical fit) represents a more natural adaptation. A solution that tackles this problem should be as automated as possible without patronizing the developer. It also has to be very flexible. At the same time, the goal of having an appealing layout in any possible situation should be addressed with the same high-level of abstraction that we use in other aspects of MD^2.

Our approach towards an adaptive layout utilizes the nesting of container elements that express semantic connection of contained elements. The hierarchy makes it possible to render views by recursively processing the structure of interface elements. Besides automated analysis of this “semantic nesting”, it might be necessary to take influence on the layout process. In general, the layout algorithm may proceed automatically, be set up manually, or combine both. While cumbersome manual action should be avoided, a completely automated process does not provide the desired flexibility. We, therefore, propose to adapt the layout automatically guided by keywords to control the layout process if necessary. The first keyword, `dependsOn`, specifies a semantic connection between containers. In the example of Listing 5, it is used to ensure that option 1 is used when rearranging the layout.

The subsequent question is whether developers have to specify explicitly which elements must be kept together, or which elements are splittable. Based on our experience in designing business apps, we deem the first option to be much more feasible, as a loose coupling between UI elements is more common. It is in most cases sufficient to render them near to each other. Thus, we propose to introduce the `atomic{}` container to denote elements that deviate from this norm and need to be placed as if they were one element. For example, Listing 5 assumes that the fields for given and family name should always be kept in one row and thus uses the keyword.

The proposed changes only slightly add more complexity to our DSL and are sufficient to address most layout requirements. However, implementing the automation poses some challenges that will have to be addressed.

5. Conclusions and future work

We have developed the model-driven approach MD^2 for the cross-platform development of business apps. In contrast to other approaches, it reaches a high level of abstraction and hence offers high productivity while avoiding performance penalties. In the present paper, we have focused on extensions of MD^2. First, we described how to add recursion to the DSL, which allowed us to reach Turing completeness also for offline computations. Moreover, we investigated means of using the nesting structure of containers in the user interface for a device-specific layout. In the future, we would like to broaden the scope of MD^2 further. Another area for future work is the inclusion of individual, manually written code, for example, using the Generation Gap pattern [22, pp. 571ff.]. This would enable a different approach to the inclusion of complex business logic or platform-specific features. Moreover, we will evaluate MD^2 in even more real-world scenarios and compare it to other cross-platform approaches with regard to both developer and user experience.

References

[1] HTML5, http://www.w3.org/TR/html5/, 2013.
[2] Apache Cordova, http://cordova.apache.org/, 2013.
[3] Appcelerator, http://www.appcelerator.com/, 2013.
[4] H. Heitkötter, S. Hanschke, T.A. Majchrzak, Evaluating cross-platform development approaches for mobile applications, in: Proc. 8th WEBIST, Revised Selected Papers, in: LNBIP, vol. 140, Springer, 2013, pp. 120–138.
[5] H. Heitkötter, T.A. Majchrzak, U. Wolffgang, H. Kuchen, Business Apps: Grundlagen und Status quo, 4, Förderkreis der Angewandten Informatik an der WWU Münster e.V., 2012.

[6] T.A. Majchrzak, H. Heitkötter, Development of mobile applications in regional companies: Status quo and best practices, in: Proc. 9th WEBIST, SciTePress, 2013, pp. 335–346.
[7] H. Heitkötter, T.A. Majchrzak, H. Kuchen, Cross-platform model-driven development of mobile applications with MD^2, in: Proc. SAC '13, ACM, 2013, pp. 526–533.
[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[9] H. Heitkötter, T.A. Majchrzak, Cross-platform development of business apps with MD2, in: Proc. DESRIST '13, in: Lect. Notes Comput. Sci., vol. 7939, Springer, 2013, pp. 405–411.
[10] H. Heitkötter, T.A. Majchrzak, H. Kuchen, MD2-DSL – eine domänenspezifische Sprache zur Beschreibung und Generierung mobiler Anwendungen, in: ATPS '13, in: LNI, vol. 215, GI, 2013, pp. 91–106.
[11] D. Atkins, T. Ball, G. Bruns, K. Cox, Mawl: A domain-specific language for form-based services, IEEE Trans. Softw. Eng. 25 (1999) 334–346.
[12] D. Spinellis, Notable design patterns for domain-specific languages, J. Syst. Softw. 56 (2001) 91–99.
[13] The Eclipse Foundation, Xtext, http://www.eclipse.org/Xtext/, 2013.
[14] E. Marcotte, Responsive Web Design, http://alistapart.com/article/responsive-web-design, 2013.
[15] Twitter Bootstrap, http://twitter.github.io/bootstrap/, 2013.
[16] W. Feeney, J. Hood, Adaptive man/computer interfaces: information systems which take account of user style, SIGCPR Comput. Pers. 6 (1977) 4–10.
[17] P. Akiki, A. Bandara, Y. Yu, RBUIS: simplifying enterprise application user interfaces through engineering role-based adaptive behavior, in: Proc. EICS '13, ACM, 2013, pp. 3–12.
[18] A. Mejía, R. Juárez-Ramírez, S. Inzunza, R. Valenzuela, Implementing adaptive interfaces: a user model for the development of usability in interactive systems, in: Proc. CUBE '12, ACM, 2012, pp. 598–604.
[19] K. Gajos, D. Weld, J. Wobbrock, Automatically generating personalized user interfaces with Supple, Artif. Intell. 174 (2010) 910–950.
[20] M. Behan, O. Krejcar, Adaptive graphical user interface solution for modern user devices, in: Proc. ACIIDS'12, Springer, 2012, pp. 411–420.
[21] J. Coutaz, User interface plasticity: model driven engineering to the limit!, in: Proc. EICS '10, ACM, 2010, pp. 1–8.
[22] M. Fowler, Domain-Specific Languages, Addison-Wesley Pearson Education, 2011.