Towards a Reference Architecture for Model-Driven Business Apps

Sören Evers  
con terra GmbH, Münster, Germany  
Email: s.evers@conterra.de

Jan Ernisting  
University of Münster, Münster, Germany  
Email: jan.ernisting@ercis.de

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

Abstract

Model-driven development techniques have been proposed for cross-platform app development. Typically, an individual domain-specific language (DSL) is used. The MD² framework consists of a DSL for business apps and generators, which transform a domain model to native platform code. Prior research on it focused on language and general generator design; this paper accentuates the code generation stage. A reference architecture for the generated apps is proposed to accelerate the development of new generators for which architectural decisions had been made on ad-hoc basis up to now. Moreover, generators are going to expose similar structures that facilitate maintenance. Our proposal takes MD²’s characteristics into account while providing developers with greater flexibility. A code generator for a commercial cross-platform framework has been realized, demonstrating the applicability of our proposal. We not only present steps towards a reference architecture but generalize findings for use beyond the area of mobile computing.

1. Introduction

Despite the availability of cross-platform app development frameworks, developing apps for multiple platforms remains challenging [1]. For best results, separate native development for each platform leads to the most appealing and best performing results [2]. In fact, even well responsive Web apps might be considered uncanny due to their slightly imperfect resemblance of native apps [3]. Cross-platform frameworks that are not based on Web technology (such as the very popular Apache Cordova [4]) but yield native platform code are hardly recognized by developers. At least partly, this can be attributed to their current shortcomings (cf. e.g. [5]): Some of them are hard to use, most of them lack expected features, and none of them have yet achieved a critical number of users (cf. with the study of related work in Section 3).

We have closely followed the development of cross-platform frameworks. As we will argue throughout the paper, we deem solutions based on model-driven development techniques using an individual domain-specific language (DSL) [6] to be most suited. While model-driven software development (MDSD) promises rapid development, extensive reuse of models, and accessibility for non-programmers (cf. [7], [8]), many challenges remain.

We have taken a closer look at MD² [9], which is a very well documented approach [10], [11] with recent developments [5]. The general feasibility of MD² is undoubted. However, prior work focused the design of the DSL [12] and the basic construction of code generators. The code generation stage has not received much attention, though. Therefore, we identified a number of challenges:

- For each generator that is added (i.e. each new supported platform), redundant decisions regarding design and architecture have to be made.
- Adding new generators is also burdensome.
- Supporting a high number of generators obviously earns no “scientific merits” but it is required for business acceptance.
- Maintaining generators has an almost linear effort with regard to the number of generators.
- Platforms often provide best-practices for the architecture of apps. However, these best practices do not incorporate the peculiarities of the code generators.

These challenges are not static. Rapid development of the mobile platforms (such as Android, iOS, and Windows Phone), their fragmentation, and their heterogeneity add to the complexity with every new release. Moreover, in case of native development much redundancy exists: the common distinction between model, view and controller (MVC [13]) does not typically remove all of it. In summary, even with an MDSD approach such as MD², many activities keep on reinventing the wheel – over and over again, unfortunately. The aim, however, should be to scale out to other platforms at minimal cost.

Therefore, we propose a reference architecture for MD². Carefully designed, it should be capable of overcoming or at least mitigating the above sketched challenges. Moreover, it should enable rapid progress towards an industry-feasible prototype and no mere academic proof-of-concept. To underline our strive for using a theoretically sound approach, but tailoring it for practical usage, we have cooperated with an industrial partner for this paper.

The aim is not only to provide MD² with a sound reference architecture. While MD² enables a below linear scaling of app development for multiple platforms, we intend to achieve below-linear scaling of generator design, implementation and
maintenance as well. Admittedly, generators are implemented much less frequently than their generates (i.e. the apps). Nonetheless, it is a complex and laborious tasks. As long as progress in mobile computing is not decelerated (which we do not desire!), it will be required much more often than in traditional MDS development scenarios. Thusly, the full usefulness of MDS should be employed.

Our work makes several contributions. Firstly, we build the bridge between work on reference architectures, model-driven app development, and cross-platform approaches. Secondly, we propose an actual reference architecture for a well-documented academic prototype. Thirdly, we provide generalizable advice on merits and limitations of using reference architectures in our context, and in related situations.

This paper is structured as follows. In Section 2, we introduce the status quo of MDS, the prototype we use as the starting point for building the reference architecture. Section 3 draws the general background of our work with a special focus on related approaches. In Section 4, construction of the reference architecture is described. An application scenario in Section 5 demonstrates its feasibility. In Section 6, we discuss our work before we draw a conclusion in Section 7.

2. Status Quo of MDS

While our work on reference architectures is applicable in general (as will specifically be argued in Section 6), motivation and implementation of it is bound to the cross-platform development framework MDS. We briefly focus on its destined area of application, before a sketch of how MDS was organized prior to transitioning to the proposed reference architecture is given.

In the context of information systems, business apps typically expose the following characteristics [5]:

- they are usually required to interact with at least one backend information system,
- their graphical representation and user interactions are form-based, and
- they are data-driven, providing relevant information to its users.

MDS features a textual DSL that is used to describe an archetype of model-view-controller (MVC) components [13], [14]. For that purpose, underlying domain models are represented as models; views can be described meticulously or default to conventions provided by the framework; and controllers are employed to express the behavior of the app [9]. All these parts are expressed in MDS-DSL rather than a verbose programming language. Using a custom DSL allows targeting multiple mobile platforms, currently iOS and Android, with just one MDS archetype that is transformed into runnable code by means of code generation. In addition, MDS provides a ready-to-use Java backend, which is designed as an interchangeable, loosely-coupled Web service so that other backend platforms can be used alternatively.

The transformation of an MDS archetype to executable artifacts takes place in distinct phases (cf. [10]) and utilizes the Xtext language [15] and Xtend generation [16] tooling:

1) Describe archetype using MDS-DSL;
2) automatically enhance this archetype and generate platform code in the following stages:
   a) expand complex elements, such as facilities for the automatic generation of views for given model elements, into simpler language elements, e.g. view elements such as labels and input fields;
   b) process the input model to complete defaults for unconfigured settings (MDS follows the convention-over-configuration paradigm); and
   c) fan-out to the distinct platform generators that output an ObjectiveC project for the iOS app, a Java project for the Android app, and a Java project for the backend; and finally
3) compile the generated code in the corresponding platform development environments.

The first and second generation stage help to reduce the effort required to be performed by each generator as the diversity of language elements, that need to be dealt with, is reduced. We also refer to these two stages as preprocessing.

3. Research Background and Related Work

The background of our research can be highlighted from two directions. Firstly, it follows prior work that was conducted on MDS. This has been done in the preceding section. Secondly, it draws from work on reference architectures in MDS in general. An overview of this will be given in the following along with a discussion of related work, both in a narrower and in a broader sense.

Related approaches of MDS have been extensively discussed in the respective papers on its development [9], [10], [11], [5]. Repeating this here would not contribute to this paper, particularly because other cross-platform development approaches without a foundation in MDS were discussed. However, those related approaches that are based on MDS need to be assessed with regard to their reliance on a reference architecture:

- applause [17] targets major platforms (iOS, Android, and Windows Phone) and stipulates build strategies to add support for additional targets.
- AXIOM [18] archetypes undergo distinct transformations, too. Yet, it offers no insights into the organization of code generators.
- Xmob [19] fails to provide any tools.

Transpilers, i.e. tools that take code for one platform and compile it for another, are out of scope for this comparison. An example for such a tool is J2ObjC [20]. Arguably, compilers can use a reference architecture. Nevertheless, the usage is conceptually different to what we present here.
Reference architectures, in general, are extensively studied. Papers that contextually relate to ours are rather scarce, though. Martínez-Fernández et al. present a case study on reusing architectural knowledge [21]. Their work does not deal with mobile apps; the positive findings are encouraging, though. Another perspective that is found in the literature is variability [22], [23] – an aim that we also pursue.

For our future work, articles such as the one by Angelov et al. [24] could become relevant. They try to classify types of reference architecture. Moreover, they argue that some architectures are more effective than others. If such advice can be used to improve our architecture, it will be well accepted. For a discussion of room for improvement, cf. also Section 6.

A rough relation is given to papers who argue for reference architectures as a mean to overcome heterogeneity. An example is the work by Cavalcante et al. [25]. Their target domain is the Internet of Things (IoT). Arguably, the hardware and software heterogeneity they have to cope with is even greater than that found in smartphones, tablets, and mobile platforms. Although it is hardly possible to draw complementary insights from their work, it also underlines the strengths of reference architectures. Interestingly, the discussion of heterogeneity links to the research on enterprise architectures. While this topic is out of scope of our paper, integrating different architectures to a “reference enterprise architecture” [26] is a notable idea.

References architectures can be used to enable software product lines [27] (SPLs). We have considered SPL concepts for $MD^2$ already; thus, we will follow this idea.

Unsurprisingly, there is no related work in a narrow sense. Neither the combination of MDSD and app development nor of MDSD and reference architectures is unique, but we could not identify any approaches that explicitly discuss all three topics. Due to our positive findings, we expect papers similar to ours to surface soon (cf. also with Section 6).

4. Reference Architecture

Based on the background of our work, we sketch steps towards a reference architecture for $MD^2$. As argued earlier, the very architecture is tailored to cross-platform development frameworks, but the path we take allows for a generalization.

4.1. Motivation

For the development of $MD^2$ code generators the application of a prototype-driven approach is proposed as discussed in [9, p. 530] and [28, p. 27]. Prototypical apps for all target platforms are built as a blueprint for the code to be generated. However, platforms often provide best-practices for the architecture of apps which do not incorporate the peculiarities of the code generators that are typically structured with the DSL and the application scenario in mind.

During the implementation phase it turned out that architectural decisions for the generated apps had to be done from scratch which often needed many iterations to turn the app blueprint into a code structure that was well-generatable. The resulting architectures for the different platforms were very similar. To avoid this overhead in the future, we devise an architecture that takes the characteristics of $MD^2$ into account, but does not restrict developers too much concerning the implementation of the actual platform-specific features.

A general strength of MDSD is its ability to enable reuse at the domain level. That reduces costs by using an automated process and it increases software solutions’ longevity as the app model can easily be adapted to new software platforms [29, p. 18]. However, these strengths only come to fruition if the effort for the implementation of the code generators is reasonable compared to the number of generated apps. As a result, an MDSD approach is typically only economically viable, if the number of generated apps is high or if the effort for the implementation of new code generators is low. If only very few apps are generated, it might make more sense to natively implement the apps for the different target platforms, unless generator development is facilitated.

Code generators are the only part of the framework that is dependent on the number of supported target platforms. Since a cross-platform approach such as $MD^2$ often aims at supporting many platforms, it makes most sense from an economical point of view to tackle the implementation complexity of the code generators. The reference architecture is an attempt to lower the implementation effort for new code generators and thus leverages the advantages of the model-driven approach to also make it feasible for smaller numbers of generated apps and a high number of supported platforms.

$MD^2$ focuses on data-driven business apps. They serve a specific business purpose and mainly deal with form-based input, processing, and display of data using standard user interface (UI) elements. Their well defined domain predestines them to be represented in a reference architecture.

4.2. Towards a Set of Components for the Reference Architecture

The DSL-based approach, as applied in $MD^2$, has a couple of advantages such as providing a syntax that is close to the notations used by domain experts, or the possibility to realize code analysis and verification on a domain level. However, the development of a DSL also has some disadvantages. The development effort is large because a complex language processor must be implemented, and language extension is hard to realize because most language processors are not designed with extension in mind [30, pp. 330f].

A goal is to keep code generators as simple as possible, since the development effort for a code generation framework mainly depends on the number of supported platforms. One way is to reduce the number of language constructs that
have to be supported by actual code generators, so that implementing new code generators is sped up and less error prone due to the lower complexity. Furthermore, by dealing with low-level elements, it is easier to assure that all generators interpret the model in the same way, i.e. to obtain a similar behavior throughout all apps. For example, it is easy to assure that an if-statement or an event behaves the same on all platforms. In contrast, implementing a complex element that has a lot of implicit logic may be error-prone across platforms.

Examples for complex language elements used in MD² are conditional events. They define an action that is executed as soon as a certain Boolean condition is satisfied, e.g. whether all input fields of a dialogue mask are correctly filled in. Such an element can be decomposed into more basic language elements, only comprising events that are triggered by user interactions with UI elements and an element that allows the conditional execution of code.

To master the complexity of code generators, we suggest that the DSL should incorporate two levels of abstraction: A higher level of abstraction that deals with more domain-specific, complex features such as conditional events, and a lower level of abstraction. The preprocessing stage of MD² transforms all high-level elements to the strict subset of low-level elements. The code generators only have to deal with the smaller subset of low-level elements. The advantage of processing high-level logic by the preliminary model-to-model transformation is that the DSL can be extended with new language features that are built on top of existing low-level elements, without requiring changes to implementations of the actual code generators.

That preserves the advantages of DSLs, while eliminating said disadvantages. While modelers can use more abstract concepts, generators benefit from reduced complexity and can be more light-weight. However, it comes at the cost of a complex preprocessing. Heitkötter et al. [9] distinguish the problem space from the solution space. The problem space describes what the app should do (behavior), whereas the solution space describes how the problem is realized algorithmically. Currently, code generators mainly deal with this gap. The idea of this approach is to shift the bridging of this abstraction gap up to a higher layer – namely the preprocessing. This is sensible considering the economies of scale, as preprocessing is platform-independent and therefore has to be implemented only once. Nevertheless, the leaner code generators still have to be realized for each platform.

The reference architecture described in this paper is based on a minimal set of language elements that are used to resemble all MD² language constructs.

4.3. Introduction of the Reference Architecture

The following architecture is developed for the MD² framework, but should be generalizable with respect to its general applicability towards the generation of business apps.

Each generated app consists of static code that comprises those parts of the code that is identical among all generated apps on a platform, regardless of the concrete model. The static code is often organized in libraries that realize the frame or infrastructure of an MD² app. It emerged from the prototype-driven approach in which an app was build first, afterwards identifying which parts of the code are independent from actual apps and which parts are app-specific. The app-specific classes that are generated from models are referred to as generated code.

In Figure 1, the proposed reference architecture is shown as a simplified class diagram. The descriptive nature of the MD²-DSL allows encapsulating each core element of the language in separate classes. To preserve maintainability and extensibility of the generator in case of language changes, a design goal was to have few interrelations between the classes and to hide system characteristics behind standardized interfaces.

The following patterns are used. Facades are used to hide system-specific characteristics and to provide a common interface that is used by other components. An example is the ViewManager that provides a common interface to create and switch between views in MD² apps. The implementation of the view itself is modelled abstractly as a hierarchy of widgets. It is highly platform-dependent, and therefore, cannot be reflected in the reference architecture. Wrapper classes are used to facilitate the code generation. DataTypes and WidgetWrappers bridge native interfaces to interfaces that are designed with the requirements of the template-based code generation in mind. Similar concepts share a common interface. For example, all actions implement an execute method. Event handlers only have to deal with the abstract concept of an action to execute. That allows extending the model if new actions, validators, events, or data types are introduced in MD². References to all widgets and content providers are managed in registries.

The MVC architecture of MD² is reflected in the reference design, which is separated into three areas representing the view, the model, and the controller. Each language element is assigned to one of these areas. The ContentProvider is at the border of the controller and the model. It provides persistence operations on model elements. The ViewManager overlaps the controller and the view. It is the only component that knows how the view is organized internally. The DataMapper is placed at the intersection of all three areas. Mappings directly synchronize the view and the model and are registered from within an action. Furthermore, the central role of actions in MD² becomes evident. They contain the control logic of the app and are the only component that has references to all other controller elements, registries, as well as to the ContentProvider, ViewManager, and DataMapper.

To a large extent, components’ functionality is independent
from actual apps modelled in MD$^2$. The only components that depend on the actual app model are entities and enums that specify the data model of the app; content providers that use these entities and specify filters and a URI of the backend service; as well as the custom actions that specify the control logic or behavior of the application.

These app-specific components have to be generated. They are annotated with a white circle in the model.

We propose two different ways to generate objects. A common pattern is to provide an abstract super-class containing the static code parts. The generator builds classes that inherit from this abstract class and implement the app-specific code. Another approach is to generate a “recipe” on how to build a specific component. The actual component is then created during runtime in a factory that is provided as static code by the library. The chosen approach depends on the element that is generated and the target platform. In case of the application example based on MD$^2$, the first approach is used for custom actions, enums, entities, and content providers. The second approach is typically used for the views, if they are built from a description file (for example XML).

All elements that provide interfaces to native platform functionality are marked with a black square. Implementing these elements is expected to account for the major development effort of the code generators. The reference architecture only specifies a very scanty, minimal interface for them along with explanations of the components’ expected functionality. Examples are the DisplayMessage action that is supposed to use the standard output for text messages and the WidgetWrapper that abstracts from the actual widget implementations of the target platform. DataStores provide data from a local database, a remote data source, or that originate from system-specific features such as location information based on GPS. If the required platform functionality is not supported natively, it has to be implemented. For example, events indicating the loss of the Internet connection may not be featured on all platforms.

The reference architecture aids in estimating the development effort for new code generators, because elements that require most development time become evident. An experienced developer with profound knowledge of the target platform is taken into a position to easily oversee the actual work. Even if she has little experience with the MD$^2$ framework: implementing the remaining components is relatively straightforward.

### 4.4. Separation of Concerns in Content Providers

In the original implementation of MD$^2$, content providers mainly served two purposes:

- Provide a standardized interface to manage data in the form of entity instances and to interact with these data. Changes in a content provider were not persisted until they were explicitly saved using load, save, and delete operations.
- Provide a data source. That can be the access to a backend, a local data base, or access to system functionality for example to get the current location using a mobile device’s GPS.

We propose to separate these two concerns. The management of entity instances as well as MD$^2$-specific operations such as the observation for attribute changes is provided by a content provider. It operates on an abstract data source that manages one entity type. Multiple content providers might exist for the same entity type. The actual data source is
provided in a separate layer in the form of a data store. The data access architecture is visualized in Figure 2. It shows various local and remote data sources at the example of a customer entity. Multiple content providers use the same remote data store, but manage different instances of the customer entity.

The benefit of providing a separate layer for the data access is a clear separation of generated code from static, platform-specific code. The content provider contains app-specific code, such as filters on the data source, and thus, has to be generated. The data source on the other hand is independent from concrete apps. They are connected using an interface with load, save, and delete operations. Per default, each generator should support one store to connect with the MD² backend and one store that offers local storage means. However, the proposed architecture allows replacing the actual data source or service without the need to adapt the code generators and to rerun the code generation process. The generated content provider code stays unchanged. That allows for example to connect to Web services with a different interface than the default generated MD² backend.

4.5. Required Components and their Interfaces

As follows, the required classes that have to be provided by the generated apps are proposed and the functionality of each element as intended by this reference architecture is briefly outlined. The reference architecture is tailored to the MD² language; however the elements described should be generalizable for arbitrary business apps. Using the same method and attribute names throughout all platforms helps to understand the structure of generators and generated apps. Internally, the classes can be implemented completely different on the various platforms.

It is important to mention that we only present a simplified sketch of our reference architecture to outline the general idea and to give an impression on how it can support in the development of code generators. The actual reference architecture comprises all elements of the MD² language and provides a specification of the interfaces that represent a minimal set of methods and attributes which should be available on all platforms to fully support the code generation from MD² models. The description of the interfaces is omitted in this paper.

The Controller triggers the start-up of the MD² application. It creates the instances of all components, registers all content providers and widgets in the according registries, and executes the initial Action of the app.

Actions are the only imperative component in MD². They are executed when triggered by an event or when called directly from within another action. Each action has to implement an execute method that contains the control logic or behavior of the action. Furthermore, actions have to provide an equals method which is used by the event handlers to ensure that an action is bound to an event only once, so that the same action is not executed multiple times if an event is triggered.

The DataMapper bypasses the actions and allows direct communication between the view and the model. Once a mapping is registered, the input fields and the mapped model attribute are synchronized on each change of either of the components.

Validators implement a method to validate input values and might have additional constraints such as min and max values. Each validator allows defining a customized message that is displayed, if the input is invalid. Furthermore, each validator type should provide a default message that is displayed if no custom message is set. Validators are bound to widget wrappers.

For each event type, supported by MD², a distinct EventHandler class is implemented: it registers and unregisters events of the specified type. That allows hiding the actual event implementation behind a common interface. It either wraps a native platform event or – if no native counterpart exists – it implements the event behavior. Typically, widget-specific events that present clicks or changes of UI elements are supported natively. Other events, for example to inform about an Internet connection loss, are not supported on all target platforms. If an event is triggered, the handler is in charge of executing the bound Action. The view implementation is highly system-specific. On certain systems, views are implemented programmatically. Other platforms follow a descriptive approach, e.g. views are described in XML files. To handle the variety, the actual system functionality is hidden behind facets. That allows changing the actual view implementation independently of the controller components.

To deal with the variety of different widget types on the platforms, the reference architecture uses the concept of widget wrappers that correspond to view elements provided by the DSL. For example, MD² provides the abstract concept of Boolean inputs that are represented as checkboxes on some platforms and switch controls on others such as iOS. The wrappers abstract from the concrete native widgets and provide unified getter and setter methods for all widgets.
The advantage of this approach is that developers only have
to deal with the variability of the target platform in one
place. Other components of the code generator can rely on
a common interface.

Internally, MD² uses its own data types. The job of
the WidgetWrapper is to translate the widget value that
typically uses a platform-specific data type into the according
MD² DataType.

MD² comes with its own set of Validators. The target
platform might not support validators at all or does not
support validators in the way they are intended in MD².
The task of the widget wrapper is to apply the validator
functionality to the actual widget. This might be achieved,
for example, by overwriting the widgets native validator
function or by configuring the native validators accordingly.

A special feature of the MD² language is its ability to
interact with view elements that are not currently displayed.
Examples are the comparison of two values of different
views, and the possibility to get or set values, disable or
enable view elements, or to check the validity of a value in
a view currently not displayed. However, often views which
are currently not displayed are not kept in memory. That
means, the current state and value of a widget is lost once a
view is left. The widget wrappers as proposed are registered
at start-up of the app for all widgets and stay registered over
the entire app lifetime. All operations are performed on the
WidgetWrapper instead of the actual widget. This allows
the ViewManager to destroy the current view.

A WidgetRegistry manages all instances of a widget,
or more precisely instances of widget wrappers. It provides
methods to add a widget and to retrieve a registered widget
with a given ID that is assigned at modeling time. The ability
to remove a widget is not required as all widgets are only
registered once on start-up. At runtime, the MD²-DSL does
not allow adding and removing widgets.

The ViewManager is responsible to build the views on
start-up and to enable navigation between views. It is a facade
that hides the platform-specific way on how the views are
actually managed. Once, at start-up, this component creates
WidgetWrappers for all widgets, registers the widget
wrappers in the WidgetRegistry, and stores the actual view
instances in an internal data structure.

ContentProviders manage instances of MD² entities.
They provide methods to set and get the values of an attribute
of the currently managed entity as well as the entire entity.
By assigning the value from one content provider to another
the value is implicitly copied. The content provider has to
implement the handling logic for the change event of the
MD²-DSL. Every time a new value is set, the new value is
compared with the stored value. If they differ, a change event
is fired for the attribute in question. In case the entity is reset
or the content is replaced, all attribute values are compared
with the old values and events are fired accordingly.

The content provider abstracts from the actual data source.
It handles the requests and translates the content that is
returned e.g. from a backend service or a local database
to MD² entities. The store provides an interface for the
system functionality to store, request, and delete data. Content
providers delegate load, save, and remove tasks to their
associated DataStore.

Each data store provides data for exactly one Entity,
Enum, or simple MD² DataType. They all inherit from
a common Type interface that provides methods to create
recursive copies. Furthermore, the interface specifies methods
for comparison and the retrieval of string representation.

The reference architecture assumes that the underlying
DSL is a data-type driven language such as MD². Modelers
start by determining the required data type; other elements
such as the input widgets are inferred from the data types.
This has two implications. A way to explicitly convert
values (e.g. to assign an integer data type to a float) has
to be provided. MD² deals with this problem by offering
implicit type conversions. In addition, data types provide
string representations. When assigning any data type to a
field of type string, it is automatically converted into a string.
The underlying architecture of generated apps has to be
designed to support such auto-conversions of types.

Data types have to be implemented similarly among all
target platforms to be able to achieve a uniform behavior of
the generated apps. However, type conversions, string rep-
resentations, and default values are implemented differently
among languages. Moreover, platforms do not always have
native counterparts for all MD² data types. For example,
many languages only provide a date object while MD²
explicitly differentiates between date, time, and datet ime,
which have different string representations. Only by the
type of the wrapper class, they can be distinguished to
allow different string representations for the three data types.
Therefore, the reference architecture provides own elements
for all MD² DataTypes that provide a common interface
with methods for comparisons, type conversions, and to get
the string representation.

Another non-trivial task is to deal with values that are not
set, e.g. the value of an empty input field. MD² assumes
that every value, except Boolean, can be null. However,
strongly typed languages such as Java and C do not allow
setting primitive values such as integers to null. Possible
solutions are to internally represent integers as objects instead
of primitives or to introduce an explicit Boolean flag as a
DataType attribute that stores the information on whether
the value is set or not. Furthermore, the check on whether
a value is set might be different for different data types. A
unique interface hides all these platform-specifics.

An Entity object represents an MD² entity. Besides the
operations inherited from Type, it has to provide methods
to get and set attribute values. In untyped languages such as
JavaScript it is also necessary to manage the data types of
all attributes internally.
The work on the reference architecture was driven by theoretical considerations. Therefore, we employed the enhanced MD² in a real world scenario, in which we were supported by a partner from practice.

We applied the reference architecture to the commercial product map.apps [31]. Map.apps is a JavaScript-based Web client that strives to facilitate the development of Web apps with a focus on geo-data and map-based technologies. Map.apps allows composing basic elements organized in so-called bundles to higher-level applications. The core of the client is a JavaScript implementation of the OSGi specification to provide a modular and service-based framework to develop dynamic Web map applications [32]. It provides a large set of default bundles and a programming interface that allows implementing new bundles. Geo-related apps can be composed using this set of bundles.

However, often customers require personalized dialogue windows to display data acquired from a remote service or to integrate parts of their business processes into geo-related contexts. The map.apps standard approach is to develop customized bundles that provide the required functionality. In our prototypical application scenario, we showed that an MD²-based approach is an efficient alternative to the manual development of customized bundles for such applications.

The idea of the OSGi approach is to partition a program into bundles that realize a certain task and provide their functionality to other bundles through a well-defined interface. The OSGi architecture enforces a clean design to be able to decompose software into separate components using interfaces. As follows, we discuss how the components described in the reference architecture can be partitioned into bundles. An overview of the used bundles and their dependencies is illustrated in Figure 3.

The core idea is to separate static code that does not vary between the different apps and provide it as an own bundle called md2_runtime. This runtime bundle is then used by the generated apps. The actual app is provided in a light-weight bundle that only consists of the generated code files and a controller class that configures and creates a new instance of the static code of the runtime bundle. The runtime bundle is implemented as a service factory in OSGi terms. A service factory creates a new instance for each distinct consuming bundle. This ensures that multiple generated MD² apps can be installed in the same map.apps instance at the same time without using the same event handlers and widget registries.

In Section 4.4, we proposed a layer architecture for data access. To access business services or the local storage, so-called data stores are used as an intermediate layer between content providers and actual data sources. These data stores are provided in separate bundles. A remote_store allows accessing the generated MD² default backend, a local_store provides a local database, and a location_store provides a geo-coding service. The md2_runtime uses the data store bundle that is currently registered in the OSGi runtime. By replacing the store bundle, the data store that is used can be changed. That opens various opportunities. Besides simple use cases such as exchanging the geo-coding service by registering a different location_store, more complex use cases could be realized without changing the actual generated md2_app bundle or the md2_runtime bundle. For example, it is possible to provide different stores depending on the log-in status of a user in the map.apps platform.

Besides these MD²-specific bundles, the md2_runtime bundle depends on various map.apps core bundles, for example to programmatically create windows or to automatically generate views and their according UI elements (dataforms) from plain JavaScript objects.

### 6. Discussion

The design and implementation of the reference architecture allows for an analysis of the taken steps and the outcome. Therefore, a discussion of findings is presented with the aim of giving generalizable advice. The unavoidable limitations of our work are explained afterwards. They directly lead to open questions, which are presented along with an outlook.

#### 6.1. Findings and Implications

As a first finding, the realization of a reference architecture for MD² can be considered a success. As a second finding, reference architectures seem to be appropriate in a number of similar cases. Thirdly, some implications arose.

The application scenario has shown that MD² is not limited to the generation of mobile apps, but can be used for arbitrary data-driven business applications. The high-level of abstraction has proven to be beneficial in this regard. For example, low level concepts such as GPS functionality can be transferred to non-mobile environments only with difficulty. In fact, they cannot even be applied when devices...
are mobile, but lack specific sensor features. Raising the level of abstraction by providing the location functionality via data stores, a transfer is enabled.

According to France and Rumpe, code generators should be well-understandable by developers in the sense that they are no black boxes [33]. They found that current code generators do not make explicit the architectural choices that are made by the generators when they produce code. However, no solutions to tackle this problem were proposed until now. The reference architecture is a contribution towards the research in this area.

Up to now, the manifested use of the MVC pattern within the MD²-DSL did not offer much support in implementing generators. We construe the reference architecture as a tangible instantiation of the MVC pattern. Specifically, we find resemblance of the pattern in the general structure of our reference architecture (cf. Figure 1). Overall, our proposal provides more specific guidance for constructing code generators. Essentially, the reference architecture realizes France and Rumpe’s proposal to develop so called “glue” interfaces that can be used to integrate with foreign code [33].

On that note, the reference architecture allows cutting an edge when it comes to controlling the efforts required for realizing and maintaining code generators. Additionally, it could, for example, aid the process of evaluating the suitability of novel technologies (e.g. programming languages or platforms) for set contexts by constructing according generators. Thus, we lowered the barrier to apply MD² to other platforms.

On a more general level, drawing a comparison to e.g. the work of Graciano Neto et al. [34] is possible. Our proposal provides a case for the equivalence of reference architectures and meta models, i.e. MD²’s MVC nature consequentially influences both its DSL and, thus its inherent meta model, as well as the reference architecture.

6.2. Limitations

Due to the great number of newly introduced features and changes to the model, the existing Android and iOS generators do not work anymore. They will need to be upgraded – or probably rewritten. This is no limitation of using a reference implementation, though. In fact, it is owed to the fact that these generators were designed without having a reference architecture in mind. Therefore, rewriting these provides the opportunity to extract additional feedback on the architecture’s suitability with regard to MD² and in general.

Views are the most-platform specific parts of apps; typically, data models only differ slightly and the controller merely needs to take care of platform-particularities of logic implementation. It might, thus, turn out that not all properties of other platforms were taken into account. Therefore, minor changes to the reference architecture could be required as long as backing empiricism is still scarce.

In the course of the work presented in this paper, we used the reference architecture for one exemplary platform. Its true merits will be shown when building several new generators and working with them for a while. Work on this is already initiated. Moreover, we will seek additional industry advice.

Once several generators implement the reference architecture and an at least two-digit number of apps has been realized, quantitative assessment should follow. In the long-run, approaches such as MD² should be backed by profound empirical evidence of their feasibility (cf. [23], [35]). The lack of quantitative evidence thereby is the main current boundary of our work despite the technological progress. This deficiency needs to be tackled in future endeavours.

These limitations do not impair the value of our work. In fact, they have to be kept in mind for future work and lead to open questions.

6.3. Outlook and Future Work

The discussion of insights and the limitations allow presenting an outlook. Some future tasks can be derived directly, other require more elaborate motivation.

As argued in the motivation, a critical mass of users is required for long-term feasibility. To reach a critical mass, the feature-set of MD² needs to be sound and a number of generators is required. Moreover, an initial number of apps as case studies and examples will need to be in existence along with a comprehensive documentation. Achieving this status will be one of the main tasks of future work. It will, of course, require many small steps. One of the steps of particular importance is to reach a level of supported target platforms that will be confirmed to be suitable by our partners from industry. The reference architecture’s explicitness benefits and, ideally, enables this undertaking as additional platforms are now realizable with reduced efforts.

We will also continue to contribute technologically to MD². From the earliest design on, the focus has been on applying sophisticated concepts with a thorough theoretic background to novel problems. This work will be continued. In particular, we will revisit the reference architecture in due time.

7. Conclusion

In this paper we have introduced a reference architecture for the model-driven cross-platform app development framework MD². After introducing the relevant background, we motivated and discussed the construction of a reference architecture. The latter did not only enhance MD², as was demonstrated in a real-world application scenario. It also has merits beyond the realm of mobile computing.

While there now exists a reference architecture for MD², neither work on the framework nor on the application of reference architectures in general is finished. Our work will continue in both directions.
Acknowledgments

We would like to thank our industry partner con terra GmbH for the encouragement and support. Particularly helping us focus on real-world use cases has proven very beneficial.

References

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.