Design of Adaptive Domain-Specific Modeling Languages for Model-Driven Mobile Application Development

Xiaoping Jia and Christopher Jones
School of Computing, DePaul University, Chicago, IL 60604, U.S.A.

Keywords: Model-Driven Development, Domain-Specific Modeling Languages, Cross-platform Development, Mobile Application Development.

Abstract: The use of a DSL is a common approach to support cross-platform development of mobile applications. However, most DSL-based approaches suffer from a number of limitations such as poor performance. Furthermore, DSLs that are written *ab initio* are not able to access the capabilities supported by the native platforms. This paper presents a novel approach of using an adaptive domain-specific modeling language (ADSML) to support model-driven development of cross-platform mobile applications. We will discuss the techniques in the design of an ADSML for developing mobile applications targeting the Android and iOS platforms, including meta-model extraction, meta-model elevation, and meta-model alignment. Our approach is capable of generating high performance native applications; is able to access the full capabilities of the native platforms; and is adaptable to the rapid evolutions of its target platforms.

1 INTRODUCTION

Mobile applications are popular and are becoming increasingly sophisticated. In addition to some unique constraints and requirements, such as high responsiveness, limited memory, and low energy consumption, mobile application development faces particular challenges with a short time-to-market, rapid evolution of technologies, and competing platforms. Currently, there are several competing mobile platforms on the market, including Google's Android and Apple's iOS, which are similar in capabilities, but drastically different in their programming languages and APIs. It is highly desirable and often necessary for a mobile application to run on all major mobile platforms. However, it is very expensive to port mobile applications from one platform to another.

One approach for supporting *cross-platform* mobile application development is to use programming languages and virtual machines that are available on different platforms, such as HTML5 and JavaScript (Apache Cordova, 2015; Appcelerator, 2015) While this approach is adequate for certain types of applications, it is less than satisfactory with some serious shortcomings, including slower response times when compared to equivalent native applications (Charland and Leroux, 2011; Corral et al., 2012). This approach also suffers from significant limitations, such as being able to use only a small subset of the features supported by the underlying platforms. Canappi (Canappi, 2011) uses a *domain-specific language* (DSL) to define and generate cross-platform mobile applications as front-ends to web services, but does not define the web services themselves. Other DSL-based approaches include Mobl (Hammel et al., 2010) and md^2 (Heitkötter et al., 2013).

A promising approach that may offer some solutions to the challenges of mobile application development is *model-driven development* (MDD) (Vaupel et al., 2014), and using *domain-specific modeling languages* (DSML) to represent the platform-independent models (PIM) (Jones and Jia, 2015; Jones and Jia, 2014). While DSMLs can concisely represent the model entities of various target platforms, they often adopt the *least-common denominator* approach to be able to model applications in a platform-independent manner, and then transform the models into implementations on different mobile platforms. One limitation of this approach is that harnessing the full capabilities of the target platform would either lead to language "bloat", or render the language platform-specific. Consequently, DSMLs that are designed *ab initio* are not capable of harnessing the full power of the native APIs for the underlying platforms. Currently, there is no satisfactory solution to develop-

Jia X. and Jones C..
Design of Adaptive Domain-Specific Modeling Languages for Model-Driven Mobile Application Development.
DOI: 10.5220/0005557404130418
In *Proceedings of the 10th International Conference on Software Engineering and Applications* (ICSOFT-EA-2015), pages 413-418
ISBN: 978-989-758-114-4

ing cross-platform mobile applications that is capable of both delivering high performance and providing full access to the API features supported by the underlying platforms.

In this paper, we present a novel approach of designing *adaptive domain-specific modeling languages* (ADSML) to support model-driven development of cross-platform mobile applications. We will discuss the techniques in the design of an ADSML for developing mobile applications targeting the Android and iOS platforms, including meta-model extraction, meta-model elevation, meta-model alignment, and meta-model unification. Our approach will be able to address the following challenges in cross-platform development of mobile applications: a) generating high performance native applications; b) accessing the full capabilities of the native APIs of the underlying platforms; and c) adapting to rapid evolutions of the target platforms.

2 DSML-BASED MDD

The AXIOM project (Jia and Jones, 2013; Jia and Jones, 2012; Jia and Jones, 2011) has successfully demonstrated the feasibility and effectiveness of using a DSML to support model-driven development of mobile applications. An internal DSML hosted in Groovy is used to represent mobile applications in the form of *abstract model trees* (AMTs). A model represented as an AMT is passed through a series of transformations, resulting in new AMTs and ultimately in native Android and iOS code. Preliminary experiments have shown significant reductions in the source code size, improvement in developer productivity, with the quality of generated native code comparable to handwritten native code produced by experienced mobile application developers.

The DSML supports capabilities that are common across our target platforms as well as platform-specific capabilities when desired. However, the mappings of the DSML elements to the native platforms is static and not easily adaptable. It requires changes to the DSML itself when there are changes to the underlying native API. Similarly, extending the AXIOM DSL to a new platform, such as the Windows Mobile OS, requires significant effort to align the DSML to the new native platform. The DSML would be more effective if it could evolve and adapt, automatically incorporating new elements of its target platforms and APIs. This is the motivation for an *adaptive domain-specific modeling language (ADSML)*.

3 ADAPTIVE DSML

A *domain-specific modeling language* (DSML) is a domain-specific language (DSL) designed to represent models for model-driven development (MDD).

3.1 Meta-Models and Mappings

The definition and design of a DSML is based on a *meta-model*. We introduce the following definitions.

Definition 1: Meta-model
A *meta-model*, MM, is formally defined as (C,R), where C is the set of *classes* in the meta-model, and R is the set of *relations* among the classes.

Each class contains a set of attributes and methods. A *model entity* in a meta-model refers to any entity contained in the meta-model, which can be a class, an attribute, or a method. We use $E(MM)$ to denote the set of model entities in meta-model MM.

We use MM^* to denote a platform-independent meta-model, and use MM_a to denote the platform-specific meta-model for platform a. For example, to support cross-platform development of mobile applications on Android and iOS, we will deal with platform-specific meta-models, MM_{Android} and MM_{iOS}, and platform-independent meta-model MM^*.

Definition 2: Meta-model Mapping
A *simple mapping*, $\Phi[MM_a, MM_b]$, from meta-model $MM_a = (C_a, R_a)$ to another meta-model $MM_b = (C_b, R_b)$ is a function from $E(MM_a)$ to $E(MM_b)$, i.e., every model element in MM_a is mapped to a unique model element in MM_b.[1] At the class level, $\Phi[MM_a, MM_b]$ is a function from C_a to C_b.

We can also define *complex mappings*, where some classes in C_a are mapped to auxiliary classes that need to be added to the native platform.

A platform-independent meta-model MM^* supports both Android and iOS if there exist two mappings Φ_{Android} and Φ_{iOS} that map MM^* to MM_{Android} and MM_{iOS}, respectively. A DSML for representing PIMs of mobile applications can be designed based on the meta-model MM^*. We call a DSML designed based on a fixed meta-model and fixed mappings a *static* DSML.

[1] For the sake of brevity, we will only include the class level mapping in this paper. Mappings at the attribute and method level can be defined similarly.

3.2 Characteristics of Adaptive DSML

We propose the concept of *adaptive DSMLs*, which possess the following characteristics:

- The platform-independent meta-model, on which the DSML is based, is adaptive and is not defined *ab initio*. The platform-independent meta-model is constructed automatically from the native platform-specific meta-models, so that it can evolve over the time to accommodate the evolution of the native platforms.
- As the native platforms change and evolve, the mappings from the platform-independent to platform-specific meta-models are constructed automatically. The transformation and code generation tools are also adaptive to automatically accommodate the changes.
- While the basic syntactic structure of an adaptive DSML is fixed, its vocabulary may include entities in the platform-independent as well as the platform-specific meta-models. The interpretation of the vocabulary is dependent on the target platform and the current mappings among the meta-models.

We will discuss some of the techniques to implement adaptive DSML in the next several sections.

Due to the adaptive nature of the language, it is more practical to implement adaptive DSML as an internal DSL in a host language with strong support for DSL. In our prototype, the adaptive DSML for modeling mobile applications and targeting the Android and iOS platforms is implemented as an internal DSL of Groovy. Its basic syntactic rules are defined by the host language. Our adaptive DSML uses the Groovy Builder pattern to construct the object hierarchies that represent models of mobile applications. The Builder pattern offers a simple and intuitive syntax to express the model compositions. Each component is expressed as follows:

ComponentName (*attributes ...*) {
    ... nested child components ...
}

The components can be nested to form a hierarchy, with the root component representing the application. The component names allowed are the names of the classes in the platform-independent meta-model MM^* as well as the names of the classes in the platform-specific meta-models MM_{Android} and MM_{iOS}. In other words the component names supported by the adaptive DSML can evolve over the time.

3.3 Meta-Model Evolution

A *meta-model evolution* is triggered when the API of one or more of the native platforms have been updated, such as changes or new releases of the API. The meta-model evolution process involves:

- Meta-model extraction (section 4),
- Meta-model elevation (section 5), and
- Meta-model alignment (section 6).

It constructs an updated and unified platform-independent meta-model MM^* and a set of updated transformation rules to transform models in MM^* to each of the platform-specific meta-models, e.g., MM_{Android} or MM_{iOS}. The updated MM^* will become the new basis of the adaptive DSML.

4 META-MODEL EXTRACTION

Meta-model extraction is a process that extracts platform-specific meta-models from the native API of the target platforms. The primary source of input for meta-model extraction is the API documentation of the native platform in HTML. We extract the following information for each class:

- The name, inheritance, subtype, and use relation.
- The textual description of the class (in English).
- For each attribute of the class, the name, type, and textual description.
- For each method of the class, the name, signature, and textual description of the method and each parameter.

The initial platform-specific meta-model extracted from the native API of platform a is denoted as MM_a^0, which is a complete and accurate representation of the native API, and serves as the starting point of the subsequent model elevation and model alignment.

For programming languages that support reflection, part of the information can also be extracted from the binary code of the libraries. However, all information obtainable from the binaries that is useful in our analysis should also be obtainable from the API documentation. The textual descriptions and some other useful information are not available in the binaries. Hence, we choose not to use the binaries as the sources of meta-model extraction.

5 META-MODEL ELEVATION

Meta-model elevation simplifies and elevates the level of abstraction of platform-specific meta-models.

Meta-model elevation is carried out through the following operations on the meta-model:

1. Visibility analysis: to determine the *public* portion of the native APIs;
2. Tagging key architectural elements: to identify the key elements of the meta-models;
3. Pattern-based transformation: to simplify the meta-models through a series of semantics preserving transformations of the meta-models.

The result of meta-model elevation is a set of elevated platform-specific meta-models for each native platform. For native platform a, the elevated meta-model is denoted as MM_a^1.

5.1 Visibility Analysis

The API of a native platform typically consists of two parts: the *public* part, which is intended to be used directly by developers; and the *protected* part, which is intended for customizing and extending the native API. We will limit our DSML to support the *public* API only. It is more sensible to customize and extend the native API using the native languages supported by the platform. The additional classes for the customized or extended API can be incorporated in the DSML as auxiliary classes.

We define a set of rules to identify the classes, attributes, and methods that are intended solely for the purpose of customizing and extending the native API. These entities will be removed from the meta-model.

5.2 Tagging Key Architectural Elements

Tagging is the process of attaching tags, which are simple keywords or strings, as meta-data to model entities in a meta-model. Tagging associates model entities with key concepts in the domain. A model entity may be tagged with multiple tags.

In a given domain, there are usually a number of *key architectural elements* (KAE) that are typically present and play important roles in every application in the given domain. For example, in the domain of user interfaces (UI) of mobile applications, we can identify the following key architectural elements:

- *View*: a self-contained unit of UI, e.g., a single screen, a popup, etc. Typically contains a hierarchy of view objects.
- *Transition*: a connection between two views, the source, and the destination of the transition. It is also commonly associated with a trigger event.
- *Action*: code to be executed in response to certain event. It can also be associated with a source object that triggers the event, or a transition.

One or more tags can be associated with a key architectural element to indicate its potentially different roles and subtypes. We define a set of rules to determine whether a model entity is tagged as being associated with a key architecture element.

5.3 Pattern-based Transformation

We start by identifying common design patterns and idioms. When a pattern is recognized, it is replaced with a more concise but equivalent representation. Some of the patterns and idioms being considered are:

- *The Property Pattern*. APIs often define the getter and setter methods associated with a *property* following a number of well-established naming conventions. When such a pattern is recognized, the getter and setter method can be replaced with the associated property.
- *The Enumeration Pattern*. APIs of mobile platforms often use integers rather than enumerations for the sake of better performance. However, enumerations are often safer and easier to use. When such a pattern is recognized, we replace the integers with enumerated types.

For each of the transformations of the meta-model, we also introduce code generation rules that reverse the transformation at code generation time.

6 META-MODEL ALIGNMENT

Meta-model alignment establishes *alignment relations* among the entities in different platform-specific meta-models. Entities in different meta-models are considered *aligned* if the entities are considered to have the similar functions or behaviors, or play a similar role in their respective platform.

Definition 3: Alignment Relation
Given two platform-specific meta-models $MM_a = (C_a, R_a)$ and $MM_b = (C_b, R_b)$, an alignment relation among the meta-models, denoted as $\Xi[MM_a, MM_b]$, is a relation that contains all the pairs of *aligned entities* (e_1, e_2), where $e_1 \in E(MM_a)$, and $e_2 \in E(MM_b)$. At the class level, $\Xi[MM_a, MM_b]$ is the relation of *aligned classes*, between C_a and C_b.

Meta-model alignment is carried out through the following steps:

1. Similarity analysis: to discover the similarities between the model entities in different platform-specific meta-models.

2. Entity alignment: to determine the alignment relation among the entities in different platform-specific meta-models.
3. Meta-model unification: to construct a platform-independent meta-model by unifying the aligned entities in the platform-specific meta-models.

6.1 Similarity Analysis

We first analyze the similarity among the classes in $MM_a = (C_a, R_a)$ and $MM_b = (C_b, R_b)$. For each pair of classes $c_1 \in C_a$ and $c_2 \in C_b$, we define a class similarity function $\theta_0(c_1, c_2) \in [0,1]$. The similarity function is calculated based on the following factors:

- the tags of each class and the relation with other classes;
- the similarity of the attributes and methods belong to the respective classes;
- the similarity of the words in the class names;
- the similarity of the textual descriptions of the classes.

We will use the text analysis techniques developed in Natural Language Processing (NLP) research (Noyrit et al., 2013) to calculate the semantic similarity between words and textual descriptions.

We first establish a threshold T_0^- for two classes to be considered *possibly* similar. For each pair of classes c_1 and c_2 that $\theta_0(c_1, c_2) \geq T_0^-$, we perform further similarity analysis on the attributes and methods of the classes. The attribute and method similarity functions are calculated using the following factors:

- the tags of each attribute or method and the respective types or signature;
- the similarity of the words in the attribute or method names;
- the similarity of the textual descriptions of the attribute or method.

6.2 Entity Alignment

Entity alignment is the key step in model alignment to compute the alignment relation $\Xi[MM_a, MM_b]$ among the model entities in MM_a and MM_b.

We start with a relation of *anchored alignments* $\Xi_0[MM_a, MM_b]$, which consists of pairs of aligned model entities that are manually identified and verified. The computed alignment relation must be a super set of the anchored alignment relation, i.e., $\Xi_0[MM_a, MM_b] \subseteq \Xi[MM_a, MM_b]$.

We then establish the alignment threshold T_0^+ for similarity functions θ_0, to compute the alignment relation:

- For a pair of classes $c_1 \in C_a$ and $c_2 \in C_b$, if $\theta_0(c_1, c_2) \geq T_0^+$, we consider classes c_1 and c_2 to be aligned.

A similar process will be applied to the attributes and methods.

For an alignment relation, we define the following metrics to measure its degree of success in aligning the meta-models:

- *Aligned class ratio*: the ratio of the classes that are aligned with some class over the total number of classes.

$$AC_a = \frac{|\text{domain}(\Xi_{cla}[MM_a, MM_b])|}{|C_a|}$$

$$AC_b = \frac{|\text{range}(\Xi_{cla}[MM_a, MM_b])|}{|C_b|}$$

- *One-to-one ratio*: the ratio of the number of one-to-one class alignment pairs over the total number of class alignment pairs.
- *Degree of alignments*: the maximum number of classes that are aligned with any class.

A successful alignment relation should satisfy the following requirements:

- The aligned class ratio for both meta-models should be sufficiently high;
- A majority of the alignments should be one-to-one, i.e., the one-to-one ratio should be near 100%
- The degree of alignments should be fairly low, typically 3 or less.

We will also apply the techniques in ontology alignment (Shvaiko and Euzenat, 2013; Granitzer et al., 2010) in entity alignment.

6.3 Meta-Model Unification

A unified platform-independent meta-model, MM^*, can be derived from the platform-specific meta-models MM_a, MM_b, and the alignment relation $\Xi[MM_a, MM_b]$. Meta-model unification produces two mappings: $\Phi[MM^*, MM_a]$ and $\Phi[MM^*, MM_b]$.

If we have a one-to-one alignment between classes c_1 and c_2, a unified class $U(c_1, c_2)$ is added to the unified meta-model MM^*. The class $U(c_1, c_2)$ can be referred to using the name of c_1 or c_2 or a new unique name. All these names are considered aliases. The mappings are updated as follows:

- $U(c_1, c_2) \mapsto c_1$ is added to $\Phi[MM^*, MM_a]$
- $U(c_1, c_2) \mapsto c_2$ is added to $\Phi[MM^*, MM_b]$

If we have a one-to-many alignment between class c_1 and classes $c_{2,1} \ldots c_{2,n}$, and none of $c_{2,1} \ldots c_{2,n}$ is aligned with any other class, n unified classes $U(c_1, c_{2,1}) \ldots U(c_1, c_{2,n})$ will be added to the unify model meta-model MM^* The class $U(c_1, c_{2,i})$ can be referred to using the name of c_1 or $c_{2,i}$ or a new unique name. All these names are considered aliases. The mappings are updated as follows:

- $U(c_1, c_{2,1}) \mapsto c_1$, ... , $U(c_1, c_{2,n}) \mapsto c_1$ are added to $\Phi[MM^*, MM_a]$
- $U(c_1, c_{2,1}) \mapsto c_{2,1}$, ... , $U(c_1, c_{2,n}) \mapsto c_{2,n}$ are added to $\Phi[MM^*, MM_b]$

The mappings $\Phi[MM^*, MM_a]$ and $\Phi[MM^*, MM_b]$ produced during the unification process can be used to derive transformation rules from the MM^* to MM_a and MM_b.

A similar approach is used to unify the aligned attributes and methods for each pair of aligned classes, which will produce the attribute and method level model mappings.

7 CONCLUSION

Domain-specific modeling languages make the development of applications for a particular domain much simpler than hand-written approaches. However, DSMLs are often "frozen" as static mappings from DSML elements to native language elements.

An adaptive domain-specific modeling language uses information about the target platforms and APIs to evolve its syntax and capabilities. Our approach extracts a meta-model for each target platform. These platform-specific meta-models undergo a process of elevation, where an appropriate subset of the extracted meta-model is selected for further analysis. Similarity analysis aligns the meta-models by mapping one platform to the other. Finally, these mappings are unified into a platform-independent meta-model on which the DSML can be based.

Our approach enables access to the full capabilities of the native platforms and is thus capable of generating high performance native applications. It is also adaptable to rapid evolutions of the target platforms. This adaptability depends on effective ontology and tag management since it is based on the derivation of semantically useful information from the documentation of the native platform and its APIs.

We are currently developing an adaptive version of the AXIOM DSML to demonstrate the feasibility and effectiveness of the techniques proposed in this paper.

REFERENCES

Apache Cordova (2015). https://cordova.apache.org/.

Appcelerator (2015). http://www.appcelerator.com/.

Canappi (2011). http://www.canappi.com/.

Charland, A. and Leroux, B. (2011). Mobile application development: Web vs. native. *Communications of the ACM*, 54(5):49–53.

Corral, L., Sillitti, A., and Succi, G. (2012). Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science*, 10:736 – 743. MobiWIS 2012.

Granitzer, M., Sabol, V., Onn, K. W., Lukose, D., and Tochtermann, K. (2010). Ontology alignment - A survey with focus on visually supported semi-automatic techniques. *Future Internet*, 2(3):238–258.

Hammel, Z., Visser, E., et al. (2010). mobl: the new language of the mobile web. http://www.mobl-lang.org/.

Heitkötter, H., Majchrzak, T. A., and Kuchen, H. (2013). Cross-platform model-driven development of mobile applications with md2. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 526–533, New York, NY, USA. ACM.

Jia, X. and Jones, C. (2011). Dynamic languages as modeling notations in model driven engineering. In *ICSOFT 2011*, pages 220–225, Seville, Spain.

Jia, X. and Jones, C. (2012). AXIOM: A model-driven approach to cross-platform application development. In *ICSOFT 2012*, pages 24–33, Rome, Italy.

Jia, X. and Jones, C. (2013). Cross-platform application development using AXIOM as an agile model-driven approach. In *Communications in Computer and Information Science*, volume 411, pages 36–51. Springer Berlin Heidelberg.

Jones, C. and Jia, X. (2014). The AXIOM model framework: Transforming requirements to native code for cross-platform mobile applications. In *ENASE 2014*, pages 26–37, Lisbon, Portugal.

Jones, C. and Jia, X. (2015). Using a domain specific language for lightweight model-driven development. In *Communications in Computer and Information Science*. Springer Berlin Heidelberg.

Noyrit, F., Gérard, S., and Terrier, F. (2013). Computer assisted integration of domain-specific modeling languages using text analysis techniques. In *MODELS 2013, Miami, FL, USA, September 29 - October 4, 2013. Proceedings*, pages 505–521.

Shvaiko, P. and Euzenat, J. (2013). Ontology matching: State of the art and future challenges. *IEEE Trans. Knowl. Data Eng.*, 25(1):158–176.

Vaupel, S., Taentzer, G., Harries, J. P., Stroh, R., Gerlach, R., and Guckert, M. (2014). Model-driven development of mobile applications allowing role-driven variants. In *MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceedings*, pages 1–17.