The AXIOM Model Framework
Transforming Requirements to Native Code for Cross-platform Mobile Applications

Chris Jones and Xiaoping Jia
School of Computing, DePaul University, 243 S. Wabash Ave., 60604, Chicago, IL, U.S.A.

Keywords: Model-driven Development, Code Generation, Mobile Application Development.

Abstract: The development and maintenance of cross-platform mobile applications is expensive. One approach for reducing this cost is model-driven development. AXIOM is a model-driven approach for developing cross-platform mobile applications that uses a domain specific language (DSL) to define platform-independent models for mobile applications. AXIOM uses a consistent model representation, called an Abstract Model Tree, as the basis for all model transformations and code generation. AXIOM could significantly reduce development time and cost while increasing the quality of mobile applications. In this paper we examine the AXIOM models, their underlying abstract model trees, and the structures of its different transformation rules to show how platform-specific concerns can be introduced in ways that preserve the model's platform-independence while still providing fine-grained control over the results of the transformation process.

1 INTRODUCTION

Mobile applications are increasingly sophisticated but must still address platform-specific challenges, constraints, and requirements, such as responsiveness, limited memory, and low energy consumption. The most common mobile platforms, Google's Android and Apple's iOS, are similar in capability, but differ in their programming languages and APIs, making it expensive to port applications from one to the other. From the perspective of mobile application developers, it is highly desirable that their software run on all major mobile platforms without hand writing the code each one, an approach that would be error prone and lead to difficulties in maintenance. Model-driven development (MDD) is an approach that aligns well with this desire.

MDD is a general term that refers to any approach that emphasizes software models as the primary vehicle by which applications are built. The nature of these models can vary widely, from UML in the case of MDA, to domain-specific languages in the case of proprietary products such as Canappi (Convergence Modelling LLC., 2011). The ultimate goal of MDD is to shift the development focus away from writing code (Selic, 2003) and toward the use of models as the primary representation of the target application.

One of the most comprehensive approaches to MDD is MDA (Object Management Group, 2003). Using MDA, software systems are built by first defining platform-independent models (PIMs) that capture the compositions and core functionalities of the system in a way that is independent of implementation concerns. The PIMs are then transformed into platform-specific models (PSMs), from which the native application code for each platform can be generated. Despite some early successes (Object Management Group, 2011), MDA, with its foundation of UML and OCL, has not seen significant industry adoption and the experiences of that adoption have been varied (Hutchinson et al., 2011; Aranda et al., 2012). Some common challenges include: limitations of UML (France et al., 2006; Henderson-Sellers, 2005); inadequate tool support; and model transformation complexity. More generally it has been argued that differences between modeling languages and implementation languages can result in complexities that make MDD adoption challenging (Volter, 2011). Moreover, UML-based approaches can use their models to generate native code implementations for different platforms, but often rely on the use of MOF metamodels to drive the transformation process, which forces an even greater distinction between the model and its implementation.

A second approach to MDD, and one that seems to be specific to the mobile application domain, attempts to avoid this model-implementation dichotomy by executing common industry frameworks such as

Jones C. and Jia X..
The AXIOM Model Framework - Transforming Requirements to Native Code for Cross-platform Mobile Applications.
DOI: 10.5220/0004882100260037
In *Proceedings of the 9th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2014), pages 26-37
ISBN: 978-989-758-030-7

HTML5, CSS3 and JavaScript in wrappers that act as adapters between the application and the underlying mobile OS and hardware. PhoneGap (Adobe Systems, Inc., 2011) is an example of such an implementation. However, these wrapper-based approaches rely on existing implementations using cross-platform technologies in order to achieve their goals. While they are thus platform-independent, they are not model-driven in any meaningful sense.

A third approach to MDD uses domain-specific languages (DSL). The virtues of DSLs are the mirror image of their limitations. While DSLs can concisely represent a set of concepts from a particular domain, they cannot be used to represent applications from other domains. And so a question arises: can we apply the principles of MDA to mobile applications by using a DSL to model applications in a platform-independent way and then transform those models into working implementations on different mobile devices while not being limited to a "least common denominator" subset of the platforms' capabilities?

In this paper we present the details of a practical and scalable approach to model-driven development – AXIOM (Agile eXecutable and Incremental Object-oriented Modeling) – which is suitable for developing cross-platform mobile applications. In this approach, platform-independent requirements models are represented in a DSL. The models are passed through a series of rule-based transformations followed by template-based code generation resulting in complete native implementations. By changing the transformation rules and templates we achieve the desired goal of using a single model to generate native implementations for multiple platforms.

AXIOM's major features include:

a) An entirely generative process that produces complete implementations for each native platform using a single requirements model without any manual coding in the native platform and SDK.

b) A practical and scalable solution capable of building real world mobile applications that are similar in scale and complexity to mobile applications developed manually.

c) An emphasis on platform independence that still allows full access to all of the features and capabilities for each native platform.

d) Highly reusable and customizable transformation rules for architecture, design, and refinement decisions, as well as templates for code generation that can be reused across different applications, while also being easily modifiable and customizable on a per-application and per-platform basis.

The AXIOM approach is validated by a prototype tool that implements all of the key components including model construction, model transformation, and code generation. The prototype currently generates complete native implementations for both the iOS and Android platforms. The generated code can be directly built and deployed using the native SDKs and without any additional libraries or virtual machines. While only a subset of the native iOS and Android APIs are currently supported, the prototype tool adequately demonstrates the feasibility and the potential benefits of the AXIOM approach.

The remainder of this paper examines AXIOM in more detail. Section 2 provides an overview of AXIOM's models and key architectural approaches. Sections 3–6 examine AXIOM's models and explore its transformation process in greater detail. Section 7 provides the initial results of our evaluation of AXIOM. Finally, Sections 8–10 offer some final thoughts about AXIOM, how it relates to other MDD approaches and its potential.

2 THE AXIOM APPROACH

AXIOM (Jia and Jones, 2011; Jia and Jones, 2012) retains the key elements of MDD such as model-centricity and the transformation of models into executable code. While UML-based approaches often use MOF (Object Management Group, 2006) metamodels to facilitate model transformation, AXIOM instead provides a DSL written in a dynamic language. AXIOM supports a subset of UML in the form of state charts and thus maintains some of the most

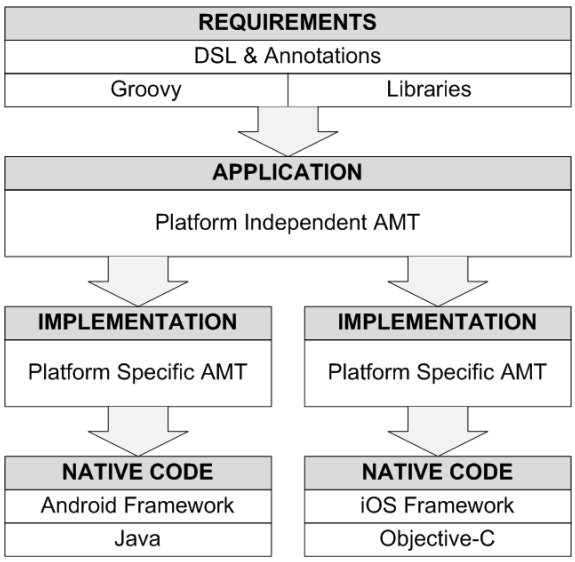


Figure 1: Model Evolution During the AXIOM Process.

powerful aspects of UML-based MDD such as model visualization, and accessibility to designers and developers who are familiar with UML and its notation.

The AXIOM approach itself is divided into three stages: *Construction*, *Transformation*, and *Translation*. Each stage emphasizes different high-level activities that serve to gradually transform the model from requirements into native source code. At each stage AXIOM emphasizes a different model. During the Construction stage the emphasis is on the requirements model. This model is canonicalized into the application model for the start of the Transformation stage. Finally, the implementation model is used during the Translation stage to produce native code for the target platform. Figure 1 illustrates the relationship between these models in the overall approach.

2.1 Abstract Model Trees

The *Abstract Model Tree (AMT)* is a common representation that unifies all of AXIOM's models. AMTs capture the logical structure and other essential elements of the models. For example, each UI screen and logical UI control of the requirements model is represented as a node in the AMT. A key feature of the AXIOM approach is that models are represented as trees rather than graphs, as in MOF. This simplifies model transformation and code generation, and makes for a versatile means of customizing transformation rule definitions.

Each node in an AMT contains a set of attributes defined as key-value pairs. In this sense the AMT is similar to an attribute syntax tree used in an attribute grammar (Knuth, 1968). However, AMTs differ from attribute syntax trees in two important aspects. First, AMTs allow for cross-node relationships and references. Such relationships are not represented as edges in the AMT, but as attributes of the nodes. Second, AMTs not only support the simple data types of traditional attribute grammars, but also support complex types such as collections and closures.

Definition 1: Abstract Model Tree
An abstract model tree, AMT, is formally defined as a 3-tuple:

$$AMT = (N, E, A)$$

where N is the set of nodes within the model, E is the set of edges connecting those nodes to form a tree, and A is a set of mappings from the nodes in N to a set of attributes in the form of key-value pairs.

2.2 Transformation Rules

AXIOM defines two types of transformations: *structural* and *styling*. We elaborate on these two kinds of transformations in Sections 5.1 and 5.2 respectively. The transformations are free of code fragments and references to the APIs of the target platform.

AXIOM's transformation rules were designed with platform-specificity in mind. Our intent was to follow a bottom-up approach to the abstraction of the different platform APIs, preserving them so that they may be used when appropriate, while abstracting the common features into the core DSL to simplify the development of cross-platform mobile applications. The transformation rules can be reused across multiple applications or customized on a per-application or even per-screen basis.

Definition 2: Transformation Rules
A transformation rule has one of the following forms:

$$LHS \rightarrow LHS' \quad (1)$$

$$LHS \rightarrow N_1, ..., N_k \quad (2)$$

$$LHS \rightarrow \varepsilon \quad (3)$$

where LHS represents a node to which the various transformation rules will be applied. The LHS can be matched based on node types and attribute values.

Rule (1) is concerned with the modification of the node's attributes. Rule (2) allows for a node to be replaced by a sequence of nodes $N_1, ..., N_k$, each of which can be the root of a subtree. Rule (3) allows for a node to be removed. All model transformations are accomplished by sets of rules in the above forms.

The model transformation process, $Transform$, accepts an AMT, M, and a rule set, R, and produces a new AMT, M'. Thus, $Transform(M,R) = M'$.

$Transform(M,R)$

```
1  Traverse M in depth-first order
2  for each node n ∈ N in M
3      if n matches the LHS of any rule r ∈ R
4          if a single match is found
5              apply r to n
6          else
7              apply r with the highest
               precedence to n
```

Model transformation is effected by a series of successive calls to $Transform$ with different rule sets, each call resulting a new model:

$$M_0, M_1, ..., M_I$$

For $k = 1, 2, ..., I$, $Transform(M_{k-1}, R_k) = M_k$, where R_k is the rule set used at the k-th phase of the transformation. M_0 is the initial source model, called the

application model. $M_{1..n-1}$ are intermediate models that represent partial transformations. M_I is the final result of the transformation process and is called the *implementation model*.

AXIOM first executes $Transform$ against the initial requirements model using a set of platform-independent rules. It then executes $Transform$ again, and applies all of the rules that relate to the target platform. While it is possible that an ill-defined rule could result in non-termination because of infinite recursion, thus far the rules defined for the prototype (see Section 7.1) have been simple enough to avoid deep nesting or recursion. Future enhancements to the prototype tool could be made to address this possibility in order to support more complex rule sets.

3 REQUIREMENTS MODEL

During the *Construction* stage, business requirements, application logic, and logical user interfaces and interactions are captured as platform-independent *requirements models*. These models are represented using a DSL based on the dynamic language, Groovy. The DSL attempts to maximize the ease of modeling by allowing the requirements model to be represented in a simple, abbreviated form whenever possible.

The use of a DSL to represent functionality and requirements is not new. However, approaches such as xUML that rely on fUML and ALF (Object Management Group, 2013a) use a general-purpose language. While this provides almost limitless flexibility, it remains a least-common denominator approach; the language makes no assumptions about what it is modeling and thus must strive to be as general as possible. AXIOM fixes the target domain, mobile applications in our case, and uses that knowledge to provide a DSL that makes it simple to model behavior from that domain. Because the AXIOM DSL is written in a general purpose programming language, it has access to a rich set of libraries and frameworks that traditional MDD notations like UML do not provide.

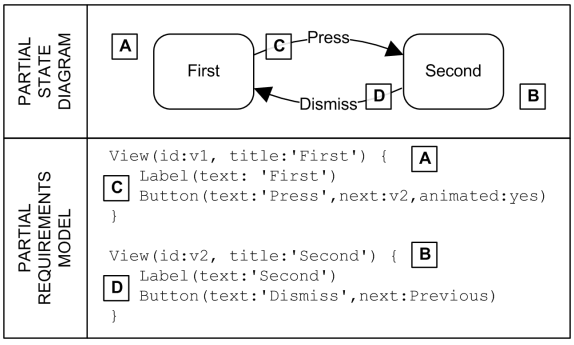


Figure 2: Partial State Diagram and Requirements Model.

3.1 Interaction Perspective

The requirements model consists of the interaction perspective, which describes how a user interacts with the final application. It captures the user interface and the application's behavior in response to user and system events. A simple example is provided in Figure 2 showing a partial state diagram and corresponding AXIOM DSL.

The interaction perspective defines the composition of the application's screens. Each screen is a view containing several types of logical UI controls. These logical controls only define their intended functions and not the actual widgets in the native platform. The names `View`, `Label`, and `Button` in the model refer to the UI elements (see A, B, C, and D in Figure 2). Other logical controls such as `ListView`, `Item`, `Panel` are also available and can represent both platform-independent as well as platform-specific widgets. AXIOM supports several different kinds of views, which makes it possible to quickly define common types of screens.

More complex applications are described as a set of related views. Each view is represented as a state in the interaction model. Transitions are defined as attributes on UI controls that trigger the transition. Optional guard conditions and actions can also be defined on the transitions.

3.2 Model Annotations

The requirements model can be decorated with annotations that are consumed by and advise the transformation process. One example is the choice of persistence framework. While that choice doesn't impact the requirements model, it can have a significant impact on the generated code. Other model annotations can be used to define alternative UI widgets to be used on platforms for which the primary widgets are unavailable. These annotations are extensible, allowing AXIOM to grow and change as new platforms and platform capabilities manifest.

4 APPLICATION MODEL

Once the requirements model has been defined AXIOM passes it through an intelligent *model builder*, which performs a series of intermediate simplifications and canonicalizations to produce an *application model*. It is this model that is processed during the

subsequent Transformation stage. An abbreviated application model AMT for the simple example of Figure 2 is shown in Listing 1.

The model builder uses a preprocessed representation of the iOS and Android APIs to expose a platform-neutral version of many of their common widgets. This allows the requirements model to specify a platform-specific widget if desired even though this constrains the target platform. The model builder also accepts optional annotations from the model that further advise the rest of the transformation process.

5 IMPLMENTATION MODEL

The *Transformation* stage carries out a series of transformations. The aim is to transform the AMT of the application model into a new AMT, the *implementation model*, which includes all the details needed to generate high quality, efficient code. The mechanism by which this transformation occurs is driven by the application of two kinds of transformation rules, *structural* and *styling*, during AXIOM's multi-pass transformation process. Each pass through the transformation process may apply one or more of either kind of transformation rule.

5.1 Structural Transformations

Structural transformations define the macro-organization of the application as the result of a series of architecture and design decisions. Some of the transformations address platform-independent issues, while others address platform-specific issues. This narrows the range of possible implementations that meet the functional, non-functional and platform needs of the application and also determine the macro-organization and interactions of the application components. This is particularly important when a multi-tier architecture is desired or when specific non-functional requirements must be satisfied. Thus structural transformations serve to define the cross-cutting concerns of the application, but do so in a way that the various intermediate models retain their platform-independent nature.

```
app1 [type:Application]
 name [type:String,value:'Modal View']
 mainview [value:v1]
 v1 [type:View]
  title [type:String,value:'First']
  name [type:String,value:'View']
  label1 [type:Label]
   text [type:String,value:'First']
  button1 [type:Button]
   text [type:String,value:'Press']
   next [type:String,value:'v2']
 v2 [type:View]
  title [type:String,value:'Second']
  name [type:String,value:'View']
  label1 [type:Label]
   text [type:String,value:'Second']
  button1 [type:Button]
   text [type:String,value:'Dismiss']
   next [type:String,value:previous]
```

Listing 1: Partial Application Model AMT.

Structural transformations change the AMT by applying one or more of the following operations:

- Add a node along with its corresponding edges.
- Split a single node into multiple nodes and adjust the edges accordingly.
- Merge multiple nodes into a single node and adjust the edges accordingly.
- Add an attribute.
- Remove an attribute.

Definition 3: Structural Transformation
A structural transformation AMT results in AMT', such that:

$$AMT' = (N', E', A') \quad (4)$$

where N', E' and A' result from the application of the transformation rules from R on the original AMT's N, E and A respectively.

Structural transformations are rule-based and generally reusable. They may alter both the structure of the AMT as well as the attributes of its nodes yielding a new AMT that is functionally isomorphic to the application model, but that defines the macro-organization of the application.

Common examples of structural decisions include target platform and language, the use of architecture and design patterns, and code distribution. For example, these decisions can determine whether or not we use file-based or database-based persistence. Similarly, we might opt to generate DAOs as a means of enforcing a separation of concerns. These choices will obviously yield very different designs when the model is ultimately translated into native code.

5.2 Styling Transformations

In contrast to structural transformations, styling transformations preserve the underlying structure of the AMT, but add more information to its nodes. This results in a new AMT that is functionally and structurally isomorphic to the original application model. Styling transformations often decorate the AMT with

additional platform-specific elements to address intra-class, micro-organizational decisions.

Styling transformations change the AMT by applying one or more of the following operations:

- Add an attribute.
- Remove an attribute.

Definition 4: Styling Transformation
A styling transformation on AMT results in AMT' such that:

$$AMT' = (N, E, A') \quad (5)$$

where N and E are the same sets that were defined for the original AMT, and A' results from the application of transformation rules from R on A.

In this case the transformations are permitted to modify the attributes of any node, but are not permitted to alter the set of nodes or their edges. Examples of styling transformations include: implementation idioms and related techniques; visual layout; and theme. Styling transformations are usually not application-specific and are thus highly reusable.

One common use for this kind of transformation is platform-specific widgets. We consider three cases:

Case 1. The same basic widget exists in both platforms. Examples include text fields, labels, and buttons.

Case 2. The same widget does not exist on both platforms, but can be simulated with differing levels of effort. One example is a radio button group, which exists natively on Android, but which must be simulated or replaced on iOS.

Case 3. The widget does not exist in both platforms and cannot be effectively simulated. Examples include the ImageButton on Android and the PageView on iOS. Even though the widget could be encoded within the DSL, the application cannot be made cross-platform without changes to the transformation rules and templates to use, for example, a new widget library.

By using the annotations that were defined on the requirements model and propagated through the subsequent transformations, the styling transformations can modify the approach to widget generation and embed those decisions within the implementation model. Deferring these lower-level decisions until model transformation enables us to make selections that are appropriate for the desired characteristics of the target runtime environment. For example, while it may be a functional requirement that a list of items be sortable within the UI, we can further refine the approach to emphasize the properties of one sort algorithm over another depending on the target runtime environment and its particular constraints. Such discrimination is critical given that we must make different time-space trade-offs based on the target platform.

5.3 Organization

The implementation model can be thought of as a design of the application with the modules, classes, and their relations determined. It defines three major aspects of the overall application's organization:

- Modules. The macro-organizational aspects of the application and its resources.
- Resources. The component files that will comprise the completed application. This includes source files, but also includes whatever descriptor files are required by the target platform.
- Fragments. The fragments of content that are used to construct the final resources.

Conceptually these elements are composited, that is, modules are comprised of resources, which are in turn comprised of fragments. While the implementation model doesn't directly contain these elements, it contains the information required to generate them in the form of injection descriptors.

5.4 Injection Descriptors

Each element in the implementation model is associated with one or more *injection descriptors*, $D = \{d_1, d_2, ..., d_k\}$. It is the combination of the implementation model's organization, combined with the injection descriptors that enables AXIOM to successfully generate native code for the target platform.

Definition 5: Injection Descriptor
An injection descriptor, d_i, is a 3-tuple:

$$d_i = (\textit{target, template-ref, binding}) \quad (6)$$

where *target* refers to an implementation model element, *template-ref* is a reference to a code template that will ultimately be used to generate the code for this element, and *binding* is a map of key-value pairs that are referenced within the code templates.

6 NATIVE CODE GENERATION

During the *Translation* stage, the implementation model, M_I, is converted into native source code for

the target platform. The implementation model contains nodes that will be mapped to specific items to be included in the implementation, such as project files, class files, resource files, etc. It also includes all of the information needed to populate each item to be generated. The task is to serialize the information stored in the AMT into linear text files in the implementation.

AXIOM's code generation algorithm, *Generate*, accepts an AMT, M, and produces native code. *Generate* is template-based (Czarnecki and Helsen, 2003) and its code templates capture knowledge and information about both the programming language used in the target platform and the API of the native SDK. Each code template contains a *parametric code fragment* and an *injection point*, the location where the code fragment can be inserted. This information, along with the injection descriptors from the implementation model, drives the code generation process.

Generate(M)

```
1  Traverse M in depth-first order
2  for each node n ∈ N in M
3      for each injection descriptor d_i of n
4          Retrieve the template d_i[template-ref]
5          for each parameter, p, in the template
6              Substitute d_i[binding][p] for p
7          Inject the instantiated code to d_i[target]
           at the injection point specified by the
           code template.
8  for each item in the native implementation
9      Aggregate the code fragments into a linear
       source code file
```

Listing 2 shows a partial code template used to generate the Java source for the views in the requirements model. This template's placeholders such as `___PACKAGE___` correspond to keys within the injection descriptor being applied to the node in the AMT. Javadoc-like placeholders such as `/**IMPORT INJECTION POINT**/` indicate additional injection points that are associated with their own code templates. These injection points are associated with their own injection descriptors and will be processed during the execution of the *Generate* function

AXIOM has a knowledge of many of the core widgets of both the iOS and Android platforms. This knowledge was derived through a separate process whereby the API was consumed and a map built defining the widgets, their classes, their available properties and the associated getters and setters. When AXIOM generates native code, the map for the target platform is consulted. Any property not located in the map is ignored. A modeler could thus provide both Android and iOS properties on the model but only the properties of the target platform would be incorporated. This is in keeping with AXIOM's goal of preserving a modeler's ability to be as platform-specific or platform-neutral as desired.

Each platform has its own default configuration, which include aspects of UI design including font size, style, and color. These defaults act as a kind of CSS style when they are applied during the code generation process. They can be easily modified to meet new and changing needs, making them potentially application-independent and reusable.

7 PRELIMINARY EVALUATION

7.1 Approach

A proof-of-concept prototype tool has been developed to demonstrate the feasibility of AXIOM. The prototype targets two popular mobile platforms: Android and iOS. The prototype can transform AXIOM models into native implementations in Java for Android and Objective-C for iOS. The generated application source code is then compiled using the native SDKs on the target platform to produce executable applications. The design of the generated code follows the common MVC architecture.

Using the prototype tool, we assessed more than 100 test cases, each of which models a working mobile application that can be successfully built and deployed on iOS and Android devices. The test cases demonstrate functionality that is common to many mobile applications including screen navigation and the use of appropriate widgets – some cross-platform, others not. Figure 3 shows the comparative frequencies of the source lines of code by platform and provides some basic descriptive statistics. The strip plot uses almost-transparent data points, so the darker the area, the more points are concentrated there.

The sample applications were developed by Masters students from DePaul's Software Engineering program. These individuals were all experienced soft-

```
package ___PACKAGE___;
/**IMPORT INJECTION POINT**/
public class ___VIEWNAME___
extends ___SUPERCLASS___ {
  /**DECLARATION INJECTION POINT**/
  @Override
  public void onCreate(Bundle state) {
    super.onCreate(state);
    /**ONCREATE INJECTION POINT**/
  }
  /**METHOD INJECTION POINT**/
}
```

Listing 2: Partial Template for Java View Implementation.

ware developers, although there were significant differences in their expertise in developing mobile applications. None of them had used AXIOM before and were provided training on the DSL.

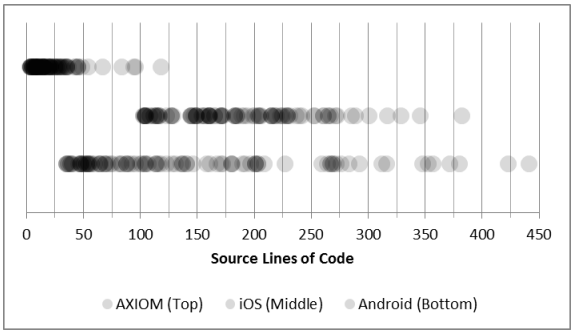
Our assessment of AXIOM focused on two kinds of metrics: representational power, including relative power; and information density, including compression ratio.

7.2 Representational Power

Representational power is concerned with how much code in one language is required to produce the same application in another language. This provides a coarse-grained indication of the relative effort expended by a developer to produce an application using different languages. For our evaluation, this involved comparing the source lines of code (SLOC) of the AXIOM requirements models to the generated source lines of code for both iOS and Android.

For the comparative evaluation of the SLOC, we used CLOC (Danial, 2013) with Groovy as the source language for AXIOM. The Android and iOS platforms were accounted for using Java and Objective-C respectively. We eliminated from consideration all generated code that served to automate the building of the generated applications, such as Ant scripts for the Android applications. The SLOC counts do not include "non-essential" lines of code such as comments, opening or closing block delimiters such as braces, or closing element tags.

While source lines of code are not ideal in terms of representing application complexity because of the potential size differences introduced by developer ability, in this case we felt the metric to be appropriate. First, the applications were straightforward enough that developer ability was likely not a significant factor. Second, we had a limited number of developers perform the actual coding, which helped to control for some of the inherent variation in ability. Third, had we chosen to analyze story or function points, we would likely have seen significant clustering of the data owing to the comparative simplicity of the applications. By focusing on SLOC we were able to automate some parts of the analysis as well as see relative differences in the sizes of the different representations of the applications. As we begin to analyze larger scale applications, story or function points can provide additional input into the analysis.



Statistics ($n = 104$)	AXIOM	iOS	Android
Minimum	3.00	103.00	35.00
Median	15.00	171.50	106.00
Mean	21.65	184.19	140.65
Maximum	118.00	382.00	441.00

Figure 3: Strip Plot and Statistics of SLOC by Platform.

Our analysis of SLOC makes two assumptions:

Assumption 1
Developer productivity measured in *source lines-of-code per person-hour* (SLOC/PH) is roughly constant regardless of languages used. Research by Jaing (Jiang et al., 2007) suggests that while language generation significantly affects developer productivity, the differences between languages in the same generation are less pronounced. Since we focus on platforms using Objective-C and Java, both of which are 3GL, we believe this assumption to be reasonable.

Assumption 2
The native applications produced by the prototype tool are comparable in size and complexity to the same applications developed manually. An admittedly subjective review of the code generated by AXIOM within the context of our working test cases suggests that it is consistent with industry best-practices such as separation of concerns and the corresponding creation of appropriate abstraction layers.

As described by Jiang (Jiang et al., 2007), the language used to implement the final software can have a significant impact on productivity. Kennedy (Kennedy et al., 2004) also identifies language as a significant component of productivity. Kennedy's relative power metric, ρ_L, compares the relative expressiveness of two languages using SLOC.

Definition 6: Kennedy's Relative Power Metric
Kennedy's relative power metric is given by:

$$\rho_{L/L_0} = \frac{I(M_{L_0})}{I(M_L)} \tag{7}$$

where $I(M_{L_0})$ is the number of lines of source code required to implement model M in native code and $I(M_L)$ is the number of lines of code required to implement M in AXIOM.

7.3 Information Density

Information density is a measure of a language's expressiveness. Languages with high information densities require less space for their particular representation and thus have higher "signal-to-noise" ratios.

To evaluate comparative information densities, we created compressed ZIP files using *gzip*, which is based on the DEFLATE algorithm (Deutsch, 1996). As with the SLOC analysis, we excluded all files that were not actually generated by AXIOM. We then compared the compression ratios, CR_L, derived using:

Definition 7: Compression Ratio

$$CR_L = \frac{\text{Uncompressed Source Size of L}}{\text{Compressed Source Size of L}} \quad (8)$$

where L is the language in question. While compression ratios will vary from model to model, a large number of samples can serve to provide a typical value for CR_L in the aggregate.

Definition 8: Language Density
As part of our analysis, we introduce the concept of *language density*, δ, which we defined as:

$$\delta_{L/L_0} = \frac{CR_{L_0}}{CR_L} \quad (9)$$

Although language density is similar to Kennedy's relative power metric, it instead describes the relative density of one language to another based on their respective compression ratios. This is different than measuring how many lines of code are required in different languages for similar representations since one language might use a verbose syntax and the other a very concise one.

7.4 Results

The results of the analyses of representational power and information density that were carried out against the trial code samples are summarized in Table 1.

To simplify the analysis, we treat all generated code for each platform as a single "language" even though the generated code may comprise several different languages. For example the Android language includes XML and Java whereas the iOS language includes XML and Objective-C.

Each of the previously discussed metrics requires one or two languages depending on whether or not it is nominal or ratio. Nominal metrics, such as compression ratio, refer to language L_0 which can be any of iOS, Android or AXIOM. Ratio metrics, such as relative power or language density, compare two languages, L_0 and L. L_0 is the base language and is either Android or iOS. L is the target language, which is always AXIOM for our purposes.

Under assumptions 1 and 2, the reduction in the size of the AXIOM requirement models compared to the size of the generated applications represents a significant reduction in development time, and hence an increase in developer productivity. The median size of the AXIOM requirements model is about 14% of the size of the generated applications for Android and about 9% of the size of the generated applications for iOS. Since one line of AXIOM is equivalent to about 11 lines of iOS code and about 7 lines of Android code, we conclude that AXIOM is more representationally powerful than either iOS or Android.

In evaluating the information density we see that the median compression ratio for AXIOM's models is 1.88, 15% the size of the median compressed iOS model and 11% the size of the median compressed Android model, suggesting that AXIOM's DSL has a much higher "signal-to-noise" ratio than either of the other two languages. Similarly, the median iOS and Android language densities are 6.39 and 8.89 respectively, suggesting that while both languages may involve greater complexity, redundancy or wordiness to represent the same model, the Android model can be compressed further than the iOS model.

It is possible that these results only apply to these comparatively simple trials and that larger-scale applications may yield significantly different results. It is also possible that these results are due in part to natural variability in developer ability. The AXIOM code was not reviewed for optimality so there might have been more efficient implementations than those provided. Although there was variability in terms of mobile application development experience, AXIOM embeds much of that domain knowledge in its DSL, reducing its overall impact.

Table 1: Comparison of Median Test Case Metrics.

Test Cases ($n = 104$)	of language L_0 of iOS	Android	AXIOM
Representational Power			
Source LOC	171.50	106.00	15.00
AXIOM as a %	8.70	14.15	100.00
Relative Power	11.43	7.07	1.00
Information Density			
Compression Ratio	12.01	16.71	1.88
AXIOM as a %	15.65	11.25	100.00
Language Density	6.39	8.89	1.00

8 DISCUSSION

AXIOM is completely generative. Developers need not edit the generated code to incorporate additional logic because all such logic is specified within the model. While partially generative solutions are feasible, the deviation of the final code from the source model because of the hand-written developer-contributed code makes such an approach less attractive than its fully generative counterpart. Additionally, since AXIOM models are source code, they can be managed using existing software development tools and techniques such as IDEs and source code management systems and do not require specialized software to support concurrent model development.

AXIOM's transformation rules and templates can be used across multiple applications by externalizing the various transformation rules and templates so that they can be reused during the transformation of other application models. From a practical perspective, this means that it will likely take longer to develop the rules and templates for new technologies than it might to simply use their APIs directly, but once they have been created, they are usable by any other application that requires them. For one-offs or proofs-of-concept this up-front cost may be significant enough that other, more common approaches, such as incremental prototypes built with hand-written code, may prove to be more economical.

Because AXIOM's transformation process divides the transformations into two discrete types, structural and styling, and because those transformations can be applied at either the application or view scope, it is possible for us to overcome the "least common denominator" problem that arises with some cross-platform development efforts. AXIOM was designed with platform-specificity in mind, even as it attempts to provide platform-independent abstractions that can help simplify the modeling process. Thus, AXIOM is not constrained to work with only the small subset of features that are common across all platforms. Because the application model defers low-level implementation decisions until structural and styling transformations have produced the implementation model, it is possible, through the use of the transformation rules and appropriate code templates, to generate virtually any kind of code output.

AXIOM can scale to mobile applications that are similar in size and complexity to those that are developed manually. This is because the process of model transformation and code generation is one of composition from smaller, simpler elements and can thus work at different scales with equal facility. At present the AXIOM prototype is constrained to the amount of memory available to the JVM during the transformation process, but that is a limitation on the implementation of the prototype and not on the approach itself.

Like most code generators, AXIOM can improve developer productivity. Because it emphasizes up-front modeling and because the transformation rules and templates can be changed and reused, developers can quickly see the impact of any given change. For these productivity gains to be realized, the templates and transformation rules must be designed and implemented up front. These rules and templates need not provided by the development team itself, any more than they currently build all of their own frameworks. For example, the third-party provider of a persistence framework could provide the templates and transformation rules that they believe best reflect the use of their framework. If application-specific changes are required, they can be made as the application is modeled without starting *ex nihilo*.

While AXIOM's DSL-based approach could certainly be used to model other styles of applications, its DSL has been constructed specifically for the mobile application domain. Similarly, the rules and templates that generate the native source code could be re-written to generate other styles of applications and even source code in other languages. In addition, the AXIOM prototype currently uses only a subset of the iOS and Android APIs. All of these are limitations of the prototype and not of the approach in general.

Our preliminary results with respect to AXIOM's representational power and conciseness are promising. While the results have been measured on a small subset of all possible mobile applications, those applications reflect common requirements such as navigation across multiple screens as well as the use of a variety of user interface widgets. Some of the capabilities are easy to model in a platform-independent way, while others are not. Thus far we have not found any inherent limitations in the approach, but it is possible that further testing on more complex applications might reveal unexpected properties of the DSL or our algorithms that reduce AXIOM's effectiveness.

9 RELATED WORK

Various frameworks and tools have been developed to support MDA-style MDD including AndroMDA (Bohlen et al., 2003) and the Eclipse Foundation's Generative Modeling Technologies (The GMT Team, 2005) and ATL Transformation Language (The ATL Team, 2005). AXIOM is based on MDA as well, although it is not based on MOF.

Executable UML (xUML) uses UML models as

the primary mechanism by which applications are built (Mellor and Balcer, 2002). However, the process of writing a model compiler can require significant effort. There are publicly available xUML compilers such as xUmlCompiler (xUML Compiler, 2009), but each compiler targets specific technologies for its code generation processes. More recent approaches such as fUML (Object Management Group, 2013b) have improved on this approach by incorporating ALF (Object Management Group, 2013a), a platform-independent imperative language. Despite using only a subset of the full UML, fUML still suffers from many of the same limitations.

Mayerhofer (Mayerhofer et al., 2013) describes xMOF as a means of specifying the behavioral semantics of models so that they can be incorporated into MOF-based transformation processes. AXIOM avoids MOF in favor of developer-driven semantics in the code templates and transformation rules.

Cuadrado (Cuadrado et al., 2012) describes a process whereby a meta-model is used to generate an intermediate language that ultimately produces Java bytecode that directly references the native platform API. AXIOM relies instead on pre-defined mappings of objects and their properties.

Cross-platform mobile applications can use languages and virtual machines that are common across different platforms, such as HTML and JavaScript (Appcelerator, Inc., 2011; The JQuery Project, 2011; Adobe Systems, Inc., 2011). Canappi (Convergence Modelling LLC., 2011) uses a DSL to define and generate cross-platform mobile applications as front-ends to web services. Unlike AXIOM, many of these approaches do not allow access to native APIs or customizable code generation.

Mobl (Hammel et al., 2010) is a DSL that targets mobile applications. However, it does not address the model-driven aspects of MDD. Thus while the DSL code may indeed be transformed into executable code, the models themselves are not major artifacts of the software development process.

md^2 (Heitkötter et al., 2013) is similar to AXIOM in principle, but differs in its orientation. AXIOM takes a developer-centric, bottom-up approach to its DSL design, while md^2 was developed top-down and with a business-centric focus. Both approaches generate native code though with differences in the role of the developer in advising the transformation process.

Bi-directional transformations such as those researched by Anjorin (Anjorin et al., 2013) allow changes to the implementation to be incorporated into the model to support round-trip engineering. AXIOM emphasizes one-way transformation from model-to-platform in an attempt to eliminate the need for manual changes to the generated code.

AXIOM is partly based on the ZOOM (Liu and Jia, 2010; Jia et al., 2007; Jia et al., 2008) project.

10 CONCLUSION

AXIOM is a model-driven approach for developing high quality, cross-platform applications. AXIOM uses a DSL to represent a platform-independent requirements model. That model is canonicalized to become an application model represented as an abstract syntax tree. The application model is changed into an implementation model through the application of structural and stylistic transformation rules. This model, along with reusable code templates, is used to produce native code for the target platform.

AXIOM provides a practical solution to MDD. It separates the complexity of the transformation process from the definition of the rules and templates that drive that process. This allows new rules and templates to be defined without changes to complex transformation frameworks or model compilers. It also allows the rules and templates to be modified in accordance with changing technologies, best practices, and organizational standards.

Our initial results are promising. In small-scale tests we have seen significant benefits in terms of representational power and information density when compared to hand-written native iOS and Android code. This reflects the AXIOM DSL's concise and mobile-centric syntax.

AXIOM has the potential to scale to large mobile applications, which, when combined with its completely generative nature, enables cost-effective cross-platform mobile development. Its models begin entirely platform-independent and through a series of successive transformations acquire platform-specific elements. This approach allows the platform-specific models full access to any and all native APIs for the target platform. The transformation process itself is fixed, but the rules and code templates that are used by the process can be changed at will, making AXIOM an extremely flexible approach to MDD.

REFERENCES

Adobe Systems, Inc. (2011). Phonegap. http://www.phonegap.com/.

Anjorin, A., Saller, K., Rose, S., and Schrr, A. (2013). A framework for bidirectional model-to-platform transformations. In Czarnecki, K. and Hedin, G., editors, *Software Language Engineering*, volume 7745 of

Lecture Notes in Computer Science, pages 124–143. Springer Berlin Heidelberg.

Appcelerator, Inc. (2011). Appcelerator. http://www.appcelerator.com/.

Aranda, J., Damian, D., and Borici, A. (2012). Transition to model-driven engineering: What is revolutionary, what remains the same? In *Proceedings of the 15th International Conference on Model Driven Engineering Languages and Systems*, MODELS'12, pages 692–708, Berlin, Heidelberg. Springer-Verlag.

Bohlen, M., Brandon, C., et al. (2003). AndroMDA. http://www.andromda.org/docs/index.html.

Convergence Modelling LLC. (2011). Canappi. http://www.canappi.com/.

Cuadrado, J. S., Guerra, E., and de Lara, J. (2012). *The Program Is the Model*: Enabling transformations@run.time. In Czarnecki, K. and Hedin, G., editors, *SLE*, volume 7745 of *Lecture Notes in Computer Science*, pages 104–123. Springer.

Czarnecki, K. and Helsen, S. (2003). Classification of model transformation approaches. In *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, pages 1–17, Anaheim, CA, USA.

Danial, A. (2013). CLOC. http://cloc.sourceforge.net/.

Deutsch, P. (1996). DEFLATE Compressed Data Format Specification version 1.3. RFC 1951 (Informational).

France, R. B., Ghosh, S., Dinh-Trong, T., and Solberg, A. (2006). Model-driven development using UML 2.0: Promises and pitfalls. *Computer*, 39(2):59–66.

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.

Henderson-Sellers, B. (2005). UML - the good, the bad or the ugly? perspectives from a panel of experts. *Software and System Modeling*, 4(1):4–13.

Hutchinson, J., Rouncefield, M., and Whittle, J. (2011). Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA. ACM.

Jia, X. et al. (2007). Executable visual software modeling:the ZOOM approach. *Software Quality Journal*, 15(1).

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., Liu, H., et al. (2008). A model transformation framework for model driven engineering. In *MSVVEIS-2008*, Barcelona, Spain.

Jiang, Z., Naud, P., and Comstock, C. (2007). An investigation on the variation of software development productivity. *International Journal of Computer and Information Science and Engineering*, pages 461–470.

Kennedy, K., Koelbel, C., et al. (2004). Defining and measuring the productivity of programming languages. *The International Journal of High Performance Computing Applications, (18)4, Winter*, 2004:441–448.

Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145.

Liu, H. and Jia, X. (2010). Model transformation using a simplified metamodel. In *Journal of Software Engineering and Applications*, volume 3, pages 653–660.

Mayerhofer, T., Langer, P., Wimmer, M., and Kappel, G. (2013). xMOF: Executable DSMLs Based on fUML. In *Proceedings of the 6th International Conference on Software Language Engineering*, volume 8225, pages 56–75.

Mellor, S. J. and Balcer, M. J. (2002). *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. Foreword By-Ivar Jacoboson.

Object Management Group (2003). MDA guide. http://www.omg.org/mda.

Object Management Group (2006). OMG's MetaObject Facility. http://www.omg.org/spec/MOF/2.0/PDF/.

Object Management Group (2011). Success stories. http://www.omg.org/mda/products_success.htm/.

Object Management Group (2013a). Concrete syntax for a UML action language: Action language for foundational UML (ALF), version 1.0.1. Specification. http:// www.omg.org/spec/ALF/1.0.1/PDF.

Object Management Group (2013b). Semantics of a foundational subset for executable UML models (FUML), version 1.1. Specification. http://www.omg.org/spec/FUML/1.1/PDF.

Selic, B. (2003). The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25.

The ATL Team (2005). ATL Transformation Language. http://eclipse.org/atl/.

The GMT Team (2005). GMT Project. http://www.eclipse.org/gmt/.

The JQuery Project (2011). JQuery mobile framework. http://www.jquerymobile.com/.

Volter, M. (2011). From programming to modeling - and back again. *IEEE Software*, 28(6):20–25.

xUML Compiler (2009). xUML Compiler- Java Model compiler Based on "Executable UML" profile. http://code.google.com/p/xuml-compiler/.