AXIOM: A Model-driven Approach to Cross-platform Application Development

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

Keywords: Model-driven Engineering, Domain-specific Languages, Mobile Application Development.

Abstract: The development and maintenance of mobile applications for multiple platforms is expensive. One approach to reducing this cost is model-driven engineering (MDE). In this paper, we present AXIOM, a model-driven approach for developing cross-platform mobile applications. Our approach uses a domain specific language (DSL) for defining platform-independent models (PIM) of mobile applications. It also defines a multi-phase, customizable transformation process to convert platform-independent models into native applications for target mobile platforms. Our approach could significantly reduce the development cost and increase the product quality of mobile applications. A prototype tool has been developed to demonstrate the feasibility of the approach. The preliminary findings are promising and show significant gains in development productivity.

1 INTRODUCTION

In recent years, there has been tremendous growth in the popularity of mobile applications targeting smart phones and tablets. With the ever-improving capabilities of these devices, mobile applications are becoming increasingly sophisticated and complicated, while also having to address challenging constraints and requirements, such as responsiveness, limited memory and low energy consumption. Furthermore, there are currently several competing mobile platforms on the market, including Google's Android and Apple's iOS. For mobile application developers, it is highly desirable for their applications to run on all major mobile platforms. Although these competing platforms are similar in capability, they differ significantly in programming languages and APIs, making it expensive to port a mobile application to different platforms.

An appealing approach to cross-platform development is model-driven engineering (MDE). In MDE, software systems are built by first defining platform-independent models (PIMs), which 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. MDE shifts the development focus away from writing code (Selic, 2003) and toward the development of models, such as those in UML and its profiles.

Despite its potential benefits and proven success in large-scale industrial applications (Object Management Group, 2011), MDE faces significant challenges to its widespread adoption including: limitations of UML (France et al., 2006; Henderson-Sellers, 2005); inadequate tool support; model transformation complexity; and apparent incompatibility with popular Agile software development methodologies such as eXtreme Programming (XP) and Scrum.

The Agile and MDE approaches to software development are each oriented toward different kinds of software. For example, MDE often targets mature middleware platforms with widely adopted common standards such as JEE, .NET, and SOA. In contrast, applications developed using Agile techniques often fit certain well-understood patterns, such as being web-based and database-driven, and using an n-tier MVC architecture. It would be beneficial if the strengths of these two approaches could be combined and their shortcomings mitigated.

In this paper, we present a novel model-driven approach to cross-platform mobile application development using a domain specific language (DSL), called *AXIOM* (Agile eXecutable and Incremental Object-oriented Modeling). Our approach defines a framework for describing the PIMs, design decisions, and implementation details of applications. We also provide tools to carry out the transformation of PIMs into native implementations across multiple platforms. A prototype tool has been developed to demonstrate the

Jia X. and Jones C..
AXIOM: A Model-driven Approach to Cross-platform Application Development.
DOI: 10.5220/0004022500240033
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 24-33
ISBN: 978-989-8565-19-8

feasibility of the approach. The prototype currently targets mobile applications in general with an emphasis on the Android and iOS platforms in particular.

2 APPROACHES TO PLATFORM-INDEPENDENCE

Platform-independence is not a new goal and the cost savings that can be realized through such independence have long been recognized. The advent of high-level languages signified an early means of providing such platform-independence through the use of greater abstractions. Languages such as C and C++ were only partially successful in achieving platform-independence because of the decision to leave some runtime aspects of the language, such as integral datatype sizing (Kernighan and Ritchie, 1988; Ellis and Stroustrup, 1990), up to compiler providers. This allowed the same source code to be compiled for many different target platforms, but did not guarantee that all of the runtime semantics would be consistent across those platforms.

Languages based on virtual machines (VMs), like Java (Gosling et al., 2000), provide true platform-independence by specifying a well defined, standardized runtime environment. Higher-level languages are converted into VM instructions, which are then executed on the target platform using the native instruction set. Because both the language and the VM are governed by specifications the behavior of the application across different platforms tends to be more consistent than when using languages without such comprehensive specifications.

The combination of high-level languages with a standardized runtime environment provides a powerful foundation on which platform-independent applications can be built. This approach can be further refined through the use of domain-specific languages (DSLs), which expose domain-specific concepts to developers, thus providing a high level of abstraction and expressiveness within that domain. DSLs are available for specific domains like mobile and web development. Figure 1 describes several different ways in which DSLs, VMS and native instructions can interact.

DSLs can be external, meaning that their syntax is not the same as the host language, or internal, where they share the host language. This flexibility allows for DSLs that can be transformed into higher-level languages, directly into VM instructions, or even into native code for the target platform. DSLs based on languages like Ruby or Groovy are internal and ultimately generate high-level language code based on

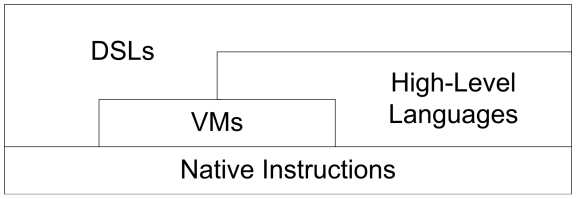

Figure 1: Approaches to platform-independence.

the host language. The benefit to this approach is that the DSL can take advantage of the power of its host language, including its compiler and associated optimizations.

A second approach is to provide a DSL that ultimately produces native VM instructions. This provides a mechanism that allows for the power of a DSL along with potential optimizations for a particular VM instruction set. However, the process of converting the DSL code into VM instructions involves the same effort as writing a compiler, potentially making it a more expensive approach than allowing the DSL to produce high-level language code for which a compiler already exists.

A third approach is to use a DSL that produces native code for the target platform. The same drawbacks exist for this approach that exist for the conversion to VM instructions. However, done well, the potential power and optimizations realized by converting directly to the native instruction set can prove valuable. This approach may also be used when few, if any, compilers exist for a target platform, such as might be the case with specialized hardware or chipsets.

3 UML AS A BASIS FOR MDE

DSLs have also been used in the form of modeling notations, the best-known of which is UML (Object Management Group, 2010), which, along with OCL (Object Management Group, 2003b), seeks to provide a common language for describing software models. The Object Management Group's (OMG) approach to model-driven engineering, MDA (Object Management Group, 2003a), relies on UML models that are then consumed and transformed into executable code.

Unfortunately, MDE in general and MDA in particular, has not seen the same industry adoption rates as Agile approaches like XP or Scrum. Some reasons for this may include:

- A lack of adequate tool support in creating, maintaining and understanding the complex models derived from UML and related OMG standards. Visual models make complex structures compre-

hensible, but are difficult and time consuming to create and maintain without strong tool support.

- The difficulty in the interchange of visual models across different tools using the XMI (Object Management Group, 2007) standard for UML interchange. One study of some of the most commonly used UML tools showed that the success rate of attempted model interchanges amongst these tools was less than 5% (Lundell et al., 2006).
- The lack of executability in UML models leads to long development cycles. UML models are thus ill-suited for Agile development processes and are generally used only for heavyweight processes.
- A lack of modeling resources comparable to the extensive frameworks and libraries available to Agile approaches. Most models must be developed from scratch rather than being built on known, proven, and previously adopted solutions.

4 AXIOM: DSL-BASED MDE

We propose to leverage the power of a DSL based on the dynamic language, Groovy, to provide a new approach to model-driven engineering. Our approach is called *AXIOM* (Agile eXecutable and Incremental Object-oriented Modeling). AXIOM retains the key elements of MDE such being model-centric and using using transformations to convert the models into executable code, but differs in the specifics. Whereas MDA relies on MOF (Object Management Group, 2006) metamodels to facilitate the transformation of UML models into executable code, AXIOM instead provides a modeling DSL written in a dynamic language. AXIOM supports a limited subset of UML in the form of class diagrams and state charts as a means if visualizing the DSL models. This allows it to maintain some of the most powerful aspects of MDD such as model visualization, while also being easily accessible to existing designers and developers who are familiar with UML and its notation.

AXIOM uses the dynamic language, Groovy, as its core modeling language and defines a DSL specifically for modeling mobile applications. The DSL provides an abstraction of the features and capabilities supported by the Android and iOS platforms. The aim of the DSL is to provide an abstract way of accessing the complete native API of each platform, and not just a limited subset of the API (often known as the lowest common denominator). By using a DSL, AXIOM also allows platform-independent models to be executable. This shortens development time and allows for the early detection and remediation of errors and anomalies. Because AXIOM is Groovy-based, it has access to a rich set of modeling elements and frameworks that UML alone does not provide.

AXIOM supports customizable model transformations and code generation. It permits both kinds of MDE: *completely generative*, where all of the code comes from the model, and *partially generative*, where nearly complete code is generated with some parts to be completed manually. The model transformations can be customized through the use of annotations on the model as well as developer-customizable code templates for patterns and idioms. The aim is to allow the generated code to be optimized for performance and other quality requirements through techniques supported by native platforms including multithreading, memory management, and application lifecycle management.

It should be noted that while we emphasize the development of mobile applications for our initial research and in this paper, AXIOM is by no means limited to such applications. As we will see, AXIOM's basic DSL-centric approach is suited to a variety of applications.

5 AXIOM INTENT MODELS

Applications are first defined as platform-independent *intent models* using AXIOM's DSL. Intent models describe the core functions, user interfaces, and interactions of the application in a way that is completely devoid of references to implementation-specific aspects of any platform. The intent model is composed of two core perspectives: the *interaction perspective*, visualized using UML state diagrams, and the *domain perspective*, visualized using UML class diagrams.

Consider a simple application that associates users with roles, perhaps as part of a broader application security component. We want the ability to associate each user with multiple roles, from which they will ultimately derive their application privileges. To provide these capabilities we must be able to manage both user and role information as well as manage the associations of users to roles. In the next few sections we examine how AXIOM represents the key elements of this simple application.

5.1 Interaction Perspective

The interaction perspective describes the user interface and the application's behavior in response to user and system events. Figure 2 shows the interaction perspective of our simple application with the corresponding AXIOM DSL shown in Figure 3.

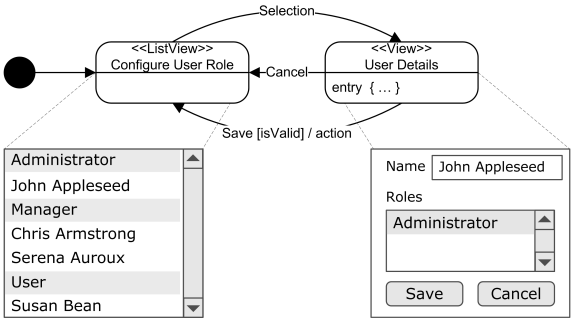

Figure 2: Graphical depiction of interaction perspective.

```
app(name:'Configure User Role') {
  def roles = [ ... ]
  def users = [ ... ]

  ListView(id:user) {
    roles.each {role -> Section(title:role) {
      users.findAll{it.role == role}.each {
        user -> Item(text:user.name,
          next: [event:Selection,
          target:detail, data:user])}}}
  }

  View(id:detail, title:'User Details') {
    entry(data) = {user = data}
    Panel(orientation:'horizontal') {
      Label(text:'Name')
      Text(id:name_value,text:data.name)
    }
    Panel(orientation:'horizontal') {
      Label(text:'Roles')
      Selection(id:role_value,
        options:roles, selected:user.role)
    }
    Panel(orientation:'horizontal') {
      Button(id:btn_cancel, text:'Cancel',
        next:[event:Click, target:user])
      Button(id:btn_save, text:'Save',
        next:[event:Click, target:user,
        guard:isValid(), action: {
          user.name = name_value.text
          user.role = role_value.selected
        }])
    }
  }
}
```

Figure 3: Partial DSL of interaction perspective.

The interaction perspective defines the composition of the two screens. The first screen is a list view with several sections. The names `ListView`, `Section`, and `Item` in the model refer to the UI elements. The second screen is a view containing several types of *logical* UI controls including labels, buttons, a text field, and a selection. The logical UI controls in the intent model only define their intended functions and not the actual widgets that implement these functions.

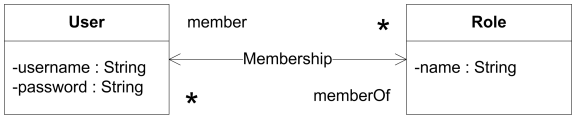

Figure 4: Graphical depiction of domain perspective.

```
@Entity
class User {
  String username
  String password

  @Relation
  hasMany = [memberOf : Role];
}

@Entity
class Role {
  String name

  @Relation
  hasMany = [member : User];
}
```

Figure 5: Partial DSL of domain perspective.

The names `View`, `Panel`, `Label`, `Button`, `Text`, and `Selection` in the model also refer to the UI elements. Each view in the UI corresponds to a state in the interaction model. Transitions are defined as the `next` attribute of the UI control that triggers the transition. Optional guard conditions and actions can also be defined on the transitions.

5.2 Domain Perspective

The domain perspective describes business entities. These are typically persistent and are transferred and referenced among different parts of the application. Figure 4 shows the domain perspective for our simple example with the corresponding AXIOM DSL shown in Figure 5.

AXIOM's domain perspective is defined using a notation that is based on the GORM (Rocher et al., 2009) framework. GORM provides for persistence and relationship management between persistent objects. This makes it suitable for defining both standalone domain objects as well as for incorporating persistence when required.

Each entity describes the properties and relationships of a domain object. This allows for the precise definition of the entity's properties. GORM supports both field and cross-field validations on its properties although we have not yet incorporated that notation into AXIOM.

Entities can also have relationships, indicated by the *@Relation* annotation. The nature of each rela-

tionship is used to manage the lifecycle and validation required to maintain it. Each such relationship defines a role name, which is used to define additional properties of the entity, as well as its cardinality and navigability. Cardinality is reflected using single- or multi-valued properties defined using GORM's *hasOne* and *hasMany* property names respectively. Navigability is defined by the presence or absence of a property that allows for navigation to the reciprocal entity in the relationship. In this example each entity has a reference to the other associated with its *hasMany* property, indicating that these entities support bi-directional navigability.

We assume that any class annotated by `@Entity` will be persisted although the precise persistence mechanism is not encoded within the model. That decision will be made during the structural transformation phase of the AXIOM's transformation process.

5.3 Intent Models and Transformations

Intent models are declarative and capture the intent of applications completely. They are also executable for the purpose of demonstration and validation. However, intent models must be informed by additional decisions in order to produce high quality and finished applications:

- **Structural.** These include architecture and design decisions such as choice of platform, language, framework, API; the use of architecture and design patterns and implementation idioms and related techniques. These decisions typically have a significant impact on the code that is ultimately generated as well as on the organization of that code. This is particularly true when a multi-tier architecture is desired or when specific non-functional requirements must be satisfied. Structural transformations are discussed in more detail in Section 6.1.
- **Refining.** These include decisions about various aspects of the application such as styles and themes. This might also include intra-class decisions such as algorithm selection. In general these decisions, while significant, do not have as great an impact on the generated code as the structural decisions although they can certainly affect how well the finished application meets its requirements. Refining transformations are discussed in more detail in section 6.2.

Structural and refining decisions almost always affect the platform-specific model. While structural decisions typically have a much broader impact on the finished application than refining decisions, they both serve to narrow the range of possible implementations that meet the functional, non-functional and platform needs. All structural and refining decisions are made during the transformation process.

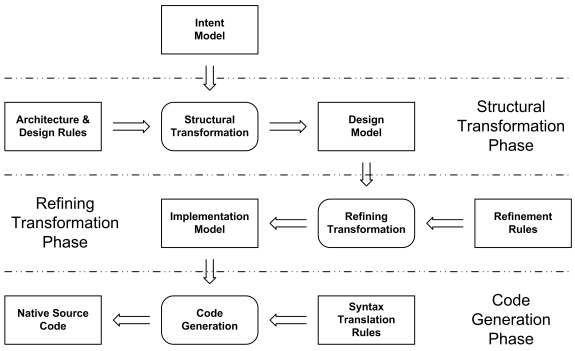

Figure 6: AXIOM transformation process.

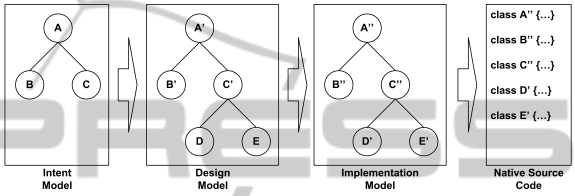

Figure 7: Evolution of AXIOM AMTs.

6 AXIOM TRANSFORMATION PROCESS

A critical component of MDE is model transformation, which converts the abstract PIM of an application into executable code. The structural and refining decisions are introduced during AXIOM's three-phase transformation process that includes: *structural transformation*, *refining transformation*, and *code generation* as shown in Figure 6.

All AXIOM models are represented as *abstract model trees* (AMTs) corresponding to the logical structure and elements of the models. For example, each UI view and logical UI control in the intent model is represented as a node in the AMT. The AMT is similar to an AST, but allows for cross-node relationships and references. Each node in the AMT supports attributes both in the form of simple name-value pairs as well as more complex types such as collections and closures.

Each phase in an AXIOM model transformation reconfigures the AMT. Figure 7 illustrates the changes introduced by each phase of the transformation process.

6.1 Structural Transformations

The structural transformation phase applies architecture and design rules to the intent model and may alter both the structure of the AMT as well as the attributes of its nodes. This yields a new AMT that can be structurally different from the original intent model. Structural decisions often impact the macro-organization of the components in the application and their interactions. The structural transformations are rule-based and generally reusable. The result of the structural transformation is a *design model* that is functionally isomorphic to the original intent model, but that also defines the macro-organization of the application driven by the platform-specific technology and API. The design model can be mapped to a design of the application with the modules, classes and their relations determined.

Some common examples of structural decisions include target language and platform, framework selection, and code distribution. Thus transforming the intent model into an iOS-based mobile application will yield very different code than transforming it into a JEE-based web application. Similarly, the decision for a web application to run within a single JEE container will yield different code than one that will be deployed into different physical tiers.

6.2 Refining Transformations

During the refining transformation phase, the structure of the design model AMT is preserved but the attributes of the nodes may be changed. This results in an output tree that is not only functionally isomorphic to the original intent model, but that is also structurally isomorphic to the design model. Refining transformations decorate the design model with additional platform-specific elements to address intra-class, micro-organizational decisions. The resulting *implementation model* maintains the macro-organization of the design model, but includes all the necessary details needed to generate high quality, efficient code. The decorations applied during the refining transformation phase are usually not application specific, and are highly reusable.

Examples of refining transformations include algorithm selection, visual layout and theme. This means that while it may be a functional requirement that a given list of items be sortable, we can refine the approach to emphasize the characteristics of one sort algorithm over another. This becomes critical when we consider that we must make different time-space tradeoffs based on the target platform.

6.3 Code Generation

During the code generation phase, the implementation model is converted into native source code for the target platform. Code generation is based on a set of platform-specific templates that are application-independent and reusable. The code generation is completely automated and highly customizable.

7 BENEFITS

AXIOM has several notable benefits for software development. First, the DSL provides higher levels of abstraction that enable the construction of the platform-independent intent models while also permitting a high degree of expressiveness. Because the DSL is written in Groovy, modelers gain instant access to existing Java-based frameworks and libraries, which saves the effort that would otherwise be required to model them. The DSL also grants the AXIOM intent models a degree of executability that facilitates rapid development and verification, an approach that aligns perfectly with the principles of modern Agile approaches, which emphasize a rapid turnaround from concept to completion.

Second, the fact that AXIOM is encoded in a textual model rather than a graphical one ensures a degree of tool independence; all that is required is a text editor. Related problems such as concurrent model development, model versioning, and model merging can be addressed through existing source code control systems.

Third, even though the models look more like a programming language, AXIOM supports a limited subset of UML models, thus retaining some of the visual expressiveness of UML.

Finally, the AXIOM transformation rules and templates can be used across entire families of applications and technologies rather than being specific to a particular application domain as is often the case with many model compilers. In addition, while many model compilers are “black box” in the sense that a change to the generated code often requires a change to the compiler’s code, AXIOM attempts to take a “white box” approach by externalizing the various transformation rules and templates. This approach allows for the reuse of the templates and transformation rules across different applications rather than binding them to only a single application.

AXIOM also has some limitations. First, AXIOM only honors a subset of the available UML diagrams, specifically class diagrams and state charts. Other UML diagrams may provide additional benefits, but

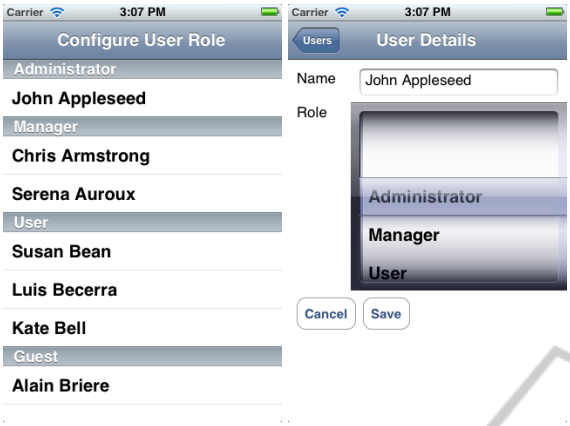

Figure 8: Screen shots of generated application on iOS.

are currently unrecognized. Second, AXIOM cannot easily make up for the limitations of a given platform, a challenge for any MDE approach. Finally, AXIOM deviates from the standard OMG definition of MDA by using a DSL for its representation rather than a MOF-based metamodel.

8 THE PROTOTYPE AND PRELIMINARY RESULTS

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. AXIOM models can be transformed 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. Figure 8 shows the screen shots of the iOS application generated from the sample application described in Figure 2. An Android implementation can also be generated.

Using the prototype tool, we conducted preliminary analyses to assess the effectiveness of AXIOM. Using a small set of working examples ($n = 29$), we compared the sizes of the AXIOM intent models and the generated source code on both iOS and Android platforms. Our assumptions are that: a) developer productivity measured in *lines-of-code per person-hour* (LOC/PH) is roughly constant regardless of languages used; and b) 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 is that it is consistent with industry best-practices such as separation of concerns and the corresponding creation of appropriate abstraction layers.

Under these assumptions the reduction in the size of the AXIOM intent models compared to the size of the generated applications would translate into a significant reduction in development time, hence an increase in development productivity. Based on our preliminary studies, shown in Table 1, the median size (in LOC) of the AXIOM intent models is 7% of the size of the generated applications for Android and about 10% of the size of the generated applications for iOS.

Similarly, Kennedy's relative power metric (Kennedy et al., 2004), ρ_L, also based on LOC, measures the impact of AXIOM on developer productivity. Kennedy's relative power metric is given by:

$$\rho_L = \frac{I_N(P)}{I_A(P)} \quad (1)$$

where I_N and I_A are the lines of native and AXIOM code respectively that are required to implement application P.

These early results suggest a potentially significant increase in productivity when compared with manually developed applications using standard development tools on native mobile platforms.

9 RELATED WORK

9.1 Model-driven Engineering

There are different approaches to MDE. Some of these closely follow the MDA standard while others amend either the MDA process or its deliverables.

AndroMDA (Bohlen et al., 2003) is an open-source, UML-based, template-driven MDA framework. It accepts UML models in an XMI format and uses them for code generation. AndroMDA is not a modeling environment and is thus limited by the quality of the XMI output produced by other tools.

The Eclipse Foundation provides UML-based technologies that support MDD in terms of both model construction and model transformation. These projects include Generative Modeling Technologies (GMT) (The GMT Team, 2005) and the ATL Transformation Language (ATL) (The ATL Team, 2005).

Executable UML (xUML) is an approach to software development that uses UML models as the primary mechanism by which applications are built (Mellor and Balcer, 2002). Like AXIOM, xUML advocates the benefits of UML executability. One significant challenge of xUML is that the process of writing a model compiler may require as much effort as

Table 1: LOC of intent model vs. generated code.

$(n = 29)$	AXIOM	iOS	Android	AXIOM as % of iOS	AXIOM as % of Android	ρ_L compared to iOS	ρ_L compared to Android
Median	19.5	168.0	279.0	10.3	7.0	8.6	14.3
Average	27.2	222.1	328.5	11.5	7.7	8.2	12.1
Minimum	8.0	126.0	77.0	5.8	3.2	15.8	9.6
Maximum	98.0	539.0	646.0	31.9	15.2	5.5	6.6

producing the original models. There are examples of publicly available xUML compilers such as xUml-Compiler (xUML Compiler, 2009), but each compiler targets a specific set of technologies for its code generation processes.

There have also been attempts to introduce more formalism into MDE. Examples include Alloy (Jackson, 2002), UML Specification Environment (USE) (Gogolla et al., 2007; Kuhlmann and Gogolla, 2008), Z (Clarke et al., 1996; Hamilton et al., 1995) and its object-oriented extensions like MooZ (Meira and Cavalcanti, 1990), Object-Z (Smith, 2012), OOZE (Alencar and Goguen, 1991), Z++ (Lano, 1991), and ZEST (Cusack and Rafsanjani, 1992).

The overall process of MDE has also been examined for ways to improve on its ability to deliver applications. Continuous Model Driven Engineering (CMDE) as defined by eXtreme model-driven design (XMDD) (Margaria and Steffen, 2008) uses process modeling as its means of eliciting the necessary requirements and behavior. Agile Model Driven Development (AMDD) (Ambler, 2009) shares the notations and tools commonly used in MDD but retains code as the central focus of the development effort. AMDD has been executed in combination with the MIDAS framework (Cáceres et al., 2004; Cáceres et al., 2003) as a means of implementing web-based applications.

Each of these approaches takes a slightly different approach to MDE and thus has its own challenges. Many of the approaches are deeply rooted in UML and thus suffer from UML's shortcomings including the lack of first-class support for UI design. Approaches that rely on the creation of custom model compilers or transformations simply shift the development burden from the application and its models to the transformation framework. Most formal approaches were never designed for MDE and thus do not provide true model executability. Approaches that change the overall MDD process either lose model-centricity or are rooted in notations that deviate from mainstream UML.

AXIOM encourages the development of executable models using a DSL that supports interaction, UI and domain design while also retaining the widely adopted graphical notation associated with UML class and state diagrams.

9.2 DSL-based Development

One approach for cross-platform mobile application development is to use languages and virtual machines that are common across different platforms, such as HTML and While this approach is adequate for certain types of applications, it has known shortcomings and limitations. Canappi (Convergence Modelling LLC., 2011) uses a DSL to define and generate cross-platform mobile applications as front-ends to web services. Unlike AXIOM, it allows neither access to native APIs nor customizable code generation.

WebDSL and Mobl (Visser et al., 2010; Hammel et al., 2010) are two DSLs that target web and mobile applications specifically. WebDSL is similar to Ruby's Rails and Groovy's Grails in that it allows for the rapid development of applications using a custom DSL. However, neither Mobl nor WebDSL addresses the model-driven aspect of the development process. Thus while the DSL code may indeed be ultimately transformed into executable code, the models themselves are not executable and are not considered major artifacts of the software development process.

AXIOM is partly based on the ZOOM (Liu and Jia, 2010; Jia et al., 2007; Jia et al., 2008) project as well as on OMG's MDA. AXIOM retains some key parts of UML, such as state and class diagrams, but unlike MDA, AXIOM defines a domain-specific modeling notation and a transformation framework that is not based on MOF.

10 FUTURE WORK

Our research into the AXIOM approach is in its early stages yet, but the preliminary results are promising. We intend to continue refining the approach so that it can work with even more complex models. One key area of work that remains is the further development

of the rule-based transformations and the associated templates. In particular these transformations must be able to handle cases where given functionality is supported to different extents on different platforms. Some of those challenges have already been encountered and addressed in the user interface, but other such challenges remain such as the implementation of persistence.

Another area that remains to be addressed is the introduction of non-functional requirements into the models and the various decisions that advise the structural and refining transformations. Such architectural concerns are central to the ability to model and transform an application for a particular platform. For example, it would be desirable for the model transformations to choose algorithms that are appropriate for each target platform's memory and persistent storage characteristics.

These enhanced models will be used to drive comparative experiments to determine if the early benefits seen in the preliminary results continue to manifest as the scale and complexity of the applications increases, particularly in the areas of developer productivity, generated source code quality, and the runtime performance, efficiency and defect densities of the executable application.

11 CONCLUSIONS

AXIOM is a model-driven approach for developing high quality, cross-platform applications. We have successfully demonstrated its feasibility in developing cross-platform mobile applications for Android and iOS platforms. AXIOM uses a DSL to provide a high level abstraction of mobile platforms. Applications are represented as intent models using the DSL and are then augmented with structural decisions and refined with other platform-specific elements during a multi-phase transformation process to produce source code for native applications.

The potential benefits of AXIOM include significant cost savings in software development owing to dramatic increases in productivity. AXIOM supports executable models, which enable iterative and incremental development and allow early validation of the applications. Product quality can be significantly improved due to the reduced amount of hand-written code. The highly customizable transformation process offers a high degree of control over code generation.

Our preliminary findings in terms of the potential gains in productivity are promising. We intend to further enhance the prototype tool to provide more comprehensive support of mobile platforms. This will enable us to conduct more extensive comparative studies and experiments using AXIOM. We plan to collect and analyze data in a number of different aspects, including developer productivity, the source code quality of generated applications, and the performance, efficiency and defect density of generated applications.

REFERENCES

Alencar, A. J. and Goguen, J. A. (1991). OOZE: An object oriented Z environment. In *ECOOP'91*, pages 180–199.

Ambler, S. (2009). Agile model driven development (AMDD): The key to scaling agile software development. http://www.agilemodeling.com/essays/amdd.htm/.

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

Cáceres, P., Daz, F., et al. (2004). *Integrating an Agile Process in a Model Driven Architecture*.

Cáceres, P., Marcos, E., et al. (2003). A mda-based approach for web information system development. In *Proceedings of Workshop in Software Model Engineering*.

Clarke, E. M., Wing, J. M., et al. (1996). Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28:626–643.

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

Cusack, E. and Rafsanjani, G.-H. B. (1992). Zest. In *Object Orientation in Z*, Workshops in Computing, pages 113–126. Springer.

Ellis, M. A. and Stroustrup, B. (1990). *The Annotated C++ Reference Manual*. Addison Wesley.

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.

Gogolla, M., Büttner, F., et al. (2007). USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34.

Gosling, J., Joy, B., et al. (2000). *The Java Language Specification, 2nd Edition*. Addison Wesley.

Hamilton, D., Covington, R., et al. (1995). Experiences in applying formal methods to the analysis of software and system requirements. *Industrial-Strength Formal Specification Techniques, Workshop on*, 0:30.

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

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.

Jackson, D. (2002). Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290.

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

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

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.

Kernighan, B. and Ritchie, D. (1988). *The C Programming Language, 2nd Edition*. Prentice Hall.

Kuhlmann, M. and Gogolla, M. (2008). Modeling and Validating Mondex Scenarios Described in UML and OCL with USE. *Formal Aspects of Computing*, 20(1):79–100.

Lano, K. (1991). Z++, an object-orientated extension to z. In *Proceedings of the Fifth Annual Z User Meeting on Z User Workshop*, pages 151–172, London, UK. Springer-Verlag.

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

Lundell, B., Lings, B., et al. (2006). UML model interchange in heterogeneous tool environments: An analysis of adoptions of XMI 2. In *MoDELS 2006, Genova, Italy*, pages 619–630.

Margaria, T. and Steffen, B. (2008). Agile it: Thinking in user-centric models. In Margaria, T. and Steffen, B., editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 490–502. Springer.

Meira, S. R. L. and Cavalcanti, A. L. C. (1990). Modular Object-Oriented Z Specifications. In Nicholls, J., editor, *Z User Workshop*, Workshops in Computing, pages 173 – 192, Oxford - UK. Springer-Verlag.

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 (2003a). MDA guide. http://www.omg.org/mda.

Object Management Group (2003b). UML 2.0 OCL. http://www.omg.org/docs/ad/03-01-07.pdf.

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

Object Management Group (2007). XML model interchange (XMI), version 2.11. http://www.omg.org/spec/XMI/2.1.1/.

Object Management Group (2010). Unified Modeling Language. http://www.omg.org/spec/UML/2.3/.

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

Rocher, G., Ledbrook, P., et al. (2009). GORM - standalone GORM. http://www.grails.org/GORM+-+StandAlone+Gorm.

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

Smith, G. (2012). Object-Z. http://itee.uq.edu.au/ smith/objectz.html.

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

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

Visser, E. et al. (2010). WebDSL. http://webdsl.org/home.

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