Spring 3-18-2016

# Using a Dynamic Domain-Specific Modeling Language for the Model-Driven Development of Cross-Platform Mobile Applications

Christopher A. Jones
*DePaul University, Chicago*, cjones@cdm.depaul.edu

# USING A DYNAMIC DOMAIN-SPECIFIC MODELING LANGUAGE FOR THE MODEL-DRIVEN DEVELOPMENT OF CROSS-PLATFORM MOBILE APPLICATIONS

BY

CHRISTOPHER A. JONES

A DISSERTATION SUBMITTED TO THE SCHOOL OF COMPUTING,
COLLEGE OF COMPUTING AND DIGITAL MEDIA OF DEPAUL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPAUL UNIVERSITY
CHICAGO, IL
2015

## DePaul University

College of Computing and Digital Media

## Dissertation Defense Report

I have read the dissertation written by:

Name: Christopher A. JONES SSN: 285-60-8068

***(To the advisor)***: The following dissertation title is identical to the one on the title page of the draft returned to the student. This title is approved by me and it is to be used when the final copies of the dissertation are prepared.

Title of dissertation:
USING A DYNAMIC DOMAIN-SPECIFIC MODELING LANGUAGE FOR THE MODEL-DRIVEN DEVELOPMENT OF CROSS-PLATFORM MOBILE APPLICATIONS

Advisors Initials: _____

☐ Acceptable. Candidate may proceed to make final copies.

☐ Pass, with revisions stated below.

_____

_____

_____

☐ Not Acceptable. Please explain:

_____

_____

_____

| | | |
|---|---|---|
| Advisor (Print Name) | Signature | Date |
| 1$^{st}$ Reader (Print Name) | Signature | Date |
| 2$^{nd}$ Reader (Print Name) | Signature | Date |
| 3$^{rd}$ Reader (Print Name) | Signature | Date |
| 4$^{th}$ Reader (Print Name) | Signature | Date |

USING A DYNAMIC DOMAIN-SPECIFIC MODELING LANGUAGE FOR THE MODEL-DRIVEN DEVELOPMENT OF CROSS-PLATFORM MOBILE APPLICATIONS

# Abstract

There has been a gradual but steady convergence of dynamic programming languages with modeling languages. One area that can benefit from this convergence is model-driven development (MDD) especially in the domain of mobile application development. By using a dynamic language to construct a domain-specific modeling language (DSML), it is possible to create models that are executable, exhibit flexible type checking, and provide a smaller cognitive gap between business users, modelers and developers than more traditional model-driven approaches.

Dynamic languages have found strong adoption by practitioners of Agile development processes. These processes often rely on developers to rapidly produce working code that meets business needs and to do so in an iterative and incremental way. Such methodologies tend to eschew "throwaway" artifacts and models as being wasteful except as a communication vehicle to produce executable code. These approaches are not readily supported with traditional heavyweight approaches to model-driven development such as the Object Management Group's Model-Driven Architecture approach.

This research asks whether it is possible for a domain-specific modeling language written in a dynamic programming language to define a cross-platform model that can produce native code and do so in a way that developer productivity and code quality are at least as effective as hand-written code produced using native tools.

Using a prototype modeling tool, AXIOM (*Agile eXecutable and Incremental Object-oriented Modeling*), we examine this question through small- and mid-scale experiments and find that the AXIOM approach improved developer productivity by almost 400%, albeit only after some up-front investment. We also find that the generated code can be of equal if not better quality than the equivalent hand-written code. Finally, we find that there are significant challenges in the synthesis of a DSML that can be used to model applications across platforms as diverse as today's mobile operating systems, which point to intriguing avenues of subsequent research.

# Acknowledgements

I have received support and encouragement from many people during this ten-year journey. My advisor, Dr. Xiaoping Jia, has been a great mentor and colleague. His forbearance of my often hectic work, life, and school schedules allowed me the time I needed to attempt this academic challenge. I would also like to thank the other members of my dissertation committee: Jane Huang, Adam Steele, Konstantin Läufer, and Berhane Zewdie. They've been incredibly patient as I slowly worked through my research and have always been willing to provide their guidance and feedback when asked.

My wife and best friend, Patricia Lynn Madden, has been incredibly patient and tolerant of me during this process. She quietly supported me through the countless evenings and weekends as I wrote and re-wrote the papers that gave life to this work. Although she didn't understand what I was working on, or even why I was working on it, she accepted that it was important to me, and that was enough.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | |
|---|---|
| **AMT** | Abstract Model Tree |
| **AXIOM** | Agile Executable and Incremental Object-Oriented Modeling |
| **DSL** | Domain Specific Language |
| **DSML** | Domain Specific Modeling Language |
| **DRY** | Don't Repeat Yourself |
| **MADL** | Mobile Aapplication Definition Language |
| **MDD** | Model-Driven Development |
| **MDE** | Model-Driven Engineering |
| **OMG** | Object-Management Group |

# Chapter 1

# Introduction

As of July of 2015 there were, by some estimates, over 1.6 million apps in the Google Play Store [1, 2] with another 1.4 million apps available in Apple's App Store [2, 3]. Mobile development comes with its own set of challenges including [4]:

- Likely interaction between apps running on the same device.
- Availability and interaction with sensors such as touch screens, accelerometers, and GPS.
- Rapid evolution of features and capabilities between different versions of the operating systems.
- Security stemming from the fact that many mobile devices are open in that malicious software can be installed on them just as it can be installed on more traditional computing devices.
- Testing has all of the same challenges as with more traditional applications with the added possible complexities introduced by its interaction with cellular networks.

This factors, combined with the industry's desire to quickly release new mobile software across many different mobile devices, has spurred research into mobile development practices.

One common way in which development teams have attempted to address the challenges of mobile development is through agile or ad-hoc development methodologies. However, with the explosion of mobile platforms and operating systems, such development seems to demand that greater formalism be introduced into the core agile processes [5]. Agile software development promises significant improvements in the productivity of software development projects of small to medium sizes, making them ideal for the small-scale apps that dominate the mobile platform market. As we will see in Chapter 2.4 however, there are a number of commonly acknowledged limitations of agile development processes and methods.

A second approach, model-driven development (MDD), is a software development approach for building large-scale, high-quality software systems. By focusing on models rather than code, MDD encourages platform independence and improves software developer productivity in a very different way from agile development.

Despite its potential, MDD has not been widely adopted by the software industry [6]. Mussbacher et. al. [7] identify several key problems with modern MDE practices:

- A vast array of modeling tools and languages make it difficult for organizations commit to MDE. Furthermore, there is a steep learning curve for the tools that do support MDE.
- Inconsistencies often arise between the model and the code generated from that model, thereby diminishing MDE's value.
- Code is still considered the most critical artifact of any software development effort. Models are used only to facilitate the understanding required to produce the necessary code.
- Evidence of success or failure of MDE projects focus on empirical results without identifying the underlying causes of the result. We are left with essentially anecdotal evidence.

It would be beneficial if the strengths of agile development and model-driven development could be combined and their shortcomings mitigated. While it may appear that

agile development and model-driven development are incompatible and opposite in nature, the proposed research will investigate if they can in fact be complementary to each other. We propose a novel and synergetic approach, called AXIOM (*Agile eXecutable and Incremental Object-oriented Modeling*), that attempts to bridge the gap between these two promising approaches.

AXIOM seeks to retain the key characteristics and benefits of both agile and model-driven development, while delivering improvements to software quality and developer productivity that would be difficult to achieve by either approach alone. AXIOM does this by providing a new modeling notation based on the Groovy programming language to facilitate the combining of models with code, thereby eliminating one of the objections to the use of models within an agile development effort. AXIOM's goal is to enable what Kent [8] describes as Model Driven Engineering (MDE), which is the practice of model-driven architecture advised by an appropriate methodology, in this case agile, to determine when and how the various software artifacts are produced.

To successfully bridge the gap between practitioners of agile development and MDD, AXIOM must satisfy several diverse needs. To meet the needs of agile developers, AXIOM features a modeling notation based on the dynamic language, Groovy. The use of a dynamic language as a modeling notation ensures that developers can be immediately productive without requiring the long start-up times required by more traditional modeling notations such as UML. However, rather than simply expecting developers to begin producing Groovy code that directly solves the problem at hand, the intent is to channel the development of that Groovy code into the production of models that will then be used to create the final, executable code.

AXIOM models are consistent with a subset of UML, which is the standard modeling notation for OMG's Model Driven Architecture (MDA) approach. In particular, AXIOM's notation is consistent with UML state chart diagrams. We provide a domain specific modeling language (DSML) for AXIOM's models in the form of a finite state machine. This DSML provides consistency across the various models that UML by itself lacks. AXIOM's notation is intended to enhance developer productivity be allowing

for the design of the models using a dynamic language while still retaining the benefits of visual modeling provided by UML. The notation is described in more detail in Chapter 6.

Once the AXIOM models have been defined, they are transformed into executable applications. The transformation approach is "pluggable" in that the same AXIOM models can be transformed into code that is intended to execute in different runtime environments. This allows users of AXIOM to realize improved productivity and cost savings for those applications that are intended to execute on multiple platforms. Because the models are written in a Groovy-based language, they are immediately executable. Because the emphasis is on modeling rather than on code, there is ultimately less code to write.

While AXIOM is not the first attempt to bridge MDD and agile approaches, we believe that it has the potential to succeed. Approaches such as Agile Model-Driven Development and Continuous Model-Driven Engineering either abandon the main precepts of MDD or use non-standard models. Other approaches such as xUML are not intended for code generation. Approaches that rely more heavily on formal models, such as Alloy, USE or Z, are not able to adequately support agile methodologies even though they might support model executability.

The primary focus of the proposed research is to investigate whether it is feasible to integrate agile and model-driven techniques in a coherent and complementary way using the AXIOM approach in the domain of mobile applications. The main objectives of the proposed research are to:

- Develop prototypes of tools to demonstrate the feasibility of the proposed solutions.
- Design and conduct case studies and comparative experiments to assess the effectiveness of the proposed solutions with respect to developer productivity and the software quality.

The research described in this thesis attempts to address one basic question: Can AXIOM be at least as effective as an approach using native tools and handwritten code when evaluated on:

- Developer productivity.
- Source code quality.

Work on AXIOM has been published at ICSOFT [9, 10], ENASE [11], and CCIS [12, 13]. Our research focused on developing the modeling language and prototype tools as well as on conducting experiments to demonstrate the feasibility of the AXIOM approach in the cross-platform development of mobile applications for the Android and iPhone platforms. The intent was to provide a sufficient cross-section of functionality in terms of the modeling notation and transformation tools to properly evaluate the potential of the AXIOM approach to facilitate agile model-driven engineering.

We designed and conducted both proofs-of-concept and comparative experiments to quantitatively and qualitatively assess the effectiveness of the proposed approach as compared to other software development techniques and practices. During the proofs-of-concept we evaluated AXIOM for fitness-of-use as well as fitness-of-purpose. Subsequent testing involved mid-scale comparative evaluations involving individual developers that allowed us to compare AXIOM's effectiveness against other common software development approaches in terms of code quality and developer productivity using industry standard metrics. The evaluations are described in chapter 7 and the results in chapter 8.

# Chapter 2

# Background

## 2.1 Model-Driven Engineering

Model-driven engineering (MDE), as defined by Kent [8], attempts to unify the artifacts produced by MDD with processes that define how those artifacts are to be produced. Kent subdivides these processes into *macro processes*, which determine the sequence in which the MDD artifacts are produced, and *micro processes*, which govern how the artifacts themselves are actually constructed. MDE rightly suggests that the process by which software is developed can have a profound impact on the artifacts that are actually produced. This is particularly evident when we consider agile software development processes.

MDE provides particular benefits when the target platforms for which the software is being written exhibit a high degree of variability. Mobile applications are an obvious domain since there are a variety of mobile platforms and there is no agreed upon standard that would serve to unify them in any meaningful way. To overcome this, some organizations started building their own model-driven tools to enable them to build applications that could function as both native mobile applications as well as mobile web applications [14]. These tools were grounded in existing open-source frameworks and took many of the same approach for platform independences that are still in use today including virtual machines that emphasize the use of JavaScript and HTML5.

### 2.1.1 Properties of Models

Because model-driven development, the basis for model-driven engineering, relies on models as the central representation of software, it is useful to discuss what models are and the roles that they play. One definition of a model, given by Mellor [15], is:

> "a coherent set of formal elements describing a system built for a purpose that is amenable to a particular form of analysis"

while Seidowitz [16] defines a model as:

> "a set of statements about some system under study."

For purposes of this research we define a model to be "a consistent and complete set of formal elements, visual or textual, describing a system that is amenable to analysis." Models that satisfy this definition should be:

1. **Consistent**. A consistent model is one where there are no two statements that can be made about the model that have different truth values when applied to the thing being modeled [16].

2. **Complete**. The model must totally define all important aspects of the final software. This includes the functional and extra-functional requirements as well as any constraints imposed by the target environment. When a model is not complete, it means that there is some aspect of the finished software that has not been accounted for.

3. **Formal**. Formal, or mathematical, statements are preferred to natural language statements due to their precise nature and well-defined semantics. Formal notations, such as Z, use mathematics to support advanced model checking and theorem proving as a means of model verification. The use of formal methods is a matter of some contention within the software development community, but many safety-critical applications have derived benefits from a strong degree of formalism despite the added burden on the modeler. UML provides a limited degree of formalism in the form of OCL, a key element of MDA.

4. **Visual**. Models should be visual rather than textual where possible. While textual representations can provide additional details about the content of the model, human languages are notoriously imprecise and often lead to ambiguities within the model. Notations such as UML depend on a visual representation to make the modeling process easier, more accurate, less error prone, and more understandable. Methodologies that depend on rapid prototyping do so for the same reasons. Visual languages are common and include UML and other typical diagrams in software engineering such as Entity-Relationship Diagrams (ERD), Data-Flow Diagrams (DFD) or flowcharts.

### 2.1.2 Model Driven Architecture

Model-Driven Architecture (MDA) [17–20] is a software development approach in which software systems are developed by first defining *platform independent models* (PIMs), which capture the compositions and the core functionalities of a system in a way that is independent of implementation languages or platforms. The PIMs are then transformed into implementations of the systems for different target platforms, known as *platform specific models* (PSMs). Model-driven architecture thus shifts the development focus away from writing code [18, 21, 22] and toward the development of visual models such as those in UML and its profiles[1] [23].

MDA relies on several OMG standards including UML [24], OCL [25] and MOF [26]. UML is the primary modeling notation and is augmented by OCL, which allows for additional formalism to be provided on the models. The MOF is the meta-metamodel for UML and provides the framework by which transformations from UML PIMs into PSMs takes place. By defining a model for metamodels, a *meta-metamodel*, MOF provides a mechanism to unify different metamodels using a common set of concepts by which metamodels can be related to one another. Thus the metamodels for UML and Java can be linked together because they each provide definitions for the elements of MOF. This approach is one way in which MDA provides for the transformation from UML into object-oriented languages such as Java.

[1] UML can be used informally as a communication tool or as a formal modeling notation. In the context of this proposal we refer to UML as a formal modeling notation only.

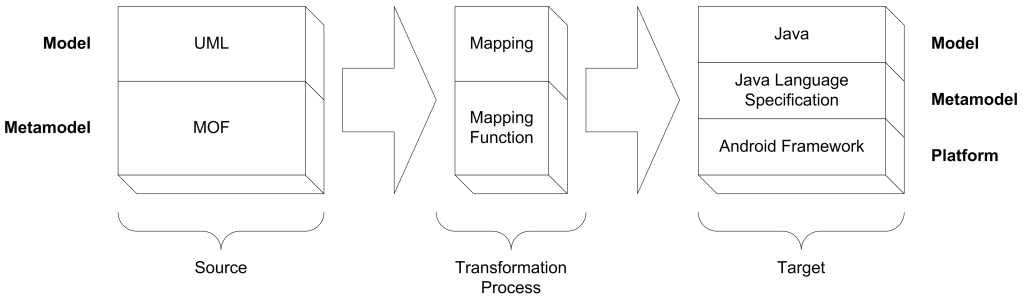



FIGURE 2.1: MDA conceptual approach. Author's image.

The basic approach to MDD within the OMG standards is as follows. A model is defined according to the rules of its metamodel. The metamodel not only provides the rules for the model, but also describes a *platform*. Platforms represent runtime environments in which the final application will be executed. We say that platforms are *realized* when they become actual executable elements within the runtime environment. *Primitive realizations* are those that can stand on their own whereas *composed realizations* are those that are themselves composed of other realized platforms, a case common in the definition of *platform stacks*.

One of the most critical elements of MDD is the transformation process, shown in Figure 2.1.

At a high level the process as described by MDA is that the source models are mapped to the target models. When the time comes to generate the platform-specific models and ultimately the deployable artifacts, the UML models are run through a transformation process that applies various rules, called *mappings*, to produce output that can ultimately be executed within the target platform. These mappings describe how elements of one model, such as UML, can be transformed into elements of another model, such as Java. The definition of these mapping rules can take many forms including *imperative*, where the mappings are represented as executable code, and *declarative*, where the mappings are encoded as rules to be applied rather then being executable themselves. Imperative-style mappings do not facilitate round-trip engineering whereas declarative-style mappings do.

OMG has defined QVT [27] as a hybrid imperative/declarative approach. Based on MOF, QVT provides two major languages that have equivalent semantics. The Core

language, which is a low-level imperative-style language and the Relations language, which is a higher-level declarative language. The major difference between the two languages is that the Core language works at a lower level of abstraction than the Relations language. Between the two languages, complex transformations can be constructed using various kinds of pattern matching, first-order predicate logic, and imperative statements.

Some key characteristics of model-driven development include:

- An emphasis on visual modeling that allows components and relationships to be defined in a more comprehensible visual form. This is critical in dealing with complex, large-scale software systems.
- The use of a high-level abstract modeling and constraint language to define platform independent models. UML allows the software design and code logic to be defined completely and separately from implementation concerns.
- UML models that are amenable to analysis of a variety of properties [28, 29]. For example, UML state diagrams can define the behavioral logic of systems and can be analyzed for concurrency related properties.
- Customizable model transformations that allow for flexibility in implementing the PIMs and the subsequent transformation to appropriate PSMs.

MDD has the potential to deliver great cost benefits in software development by automating many of the most time-consuming and error-prone aspects of software development including detailed design, coding, and testing. These savings are compounded when we have a single application that we wish to run on many different platforms. Although MDD has been proven effective and successful in many industrial enterprise applications [30] targeting mature middleware platforms with widely adopted common standards such as JEE, .NET, and SOA, there remain critical challenges to its widespread adoption in the industry [31, 32]. This is largely due to the inherent weaknesses and problems in UML and related standards, as well as inadequate tool support.

While UML is used as a common, generic notation for defining PIMs, it has a number of limitations and deficiencies [33, 34]. For example, UML models can be incomplete and/or inconsistent [35]. In addition, UML is not executable, which means that the

models must be translated to a target platform and programming language before an application can be tested. Furthermore, the Object Constraint Language (OCL) [25, 36], an important part of UML, has an awkward, non-intuitive syntax that limits its practical usefulness. Other significant obstacles include:

- A lack of adequate tool support in creating, maintaining and understanding the complex models derived from UML and related OMG standards. While visual models offer huge advantages in making complex structures comprehensible, they are also more difficult and time consuming to create and manipulate. As such, they are much more dependent on adequate tool support than simple textual models, which only require text editors.

- The difficulty in the interchange of visual models across different tools. While XMI [37, 38] is the standard for UML interchange, a recent 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% [39].

- The lack of executability in the model, which leads to long turn-around times from model to executable system. UML models are thus ill-suited for agile development processes and are generally used only for heavyweight process.

- A lack of modeling resources comparable to the extensive frameworks and libraries available to agile approaches. This means that most models need to be developed *ab initio* rather than building on known and proven solutions and utilities. This also means that knowledge may not be portable across organizations or even across projects.

- The support for so-called *round-trip engineering* is far from adequate in practice. The code generated from models is brittle and not customizable. Modifications to the generated code can make the reversal to models impossible.

- The complexity of building appropriate model transformations. Such transformations often require their own development efforts [40] that must go on in parallel with the development of the models and which may not be made to easily support multiple PSMs.

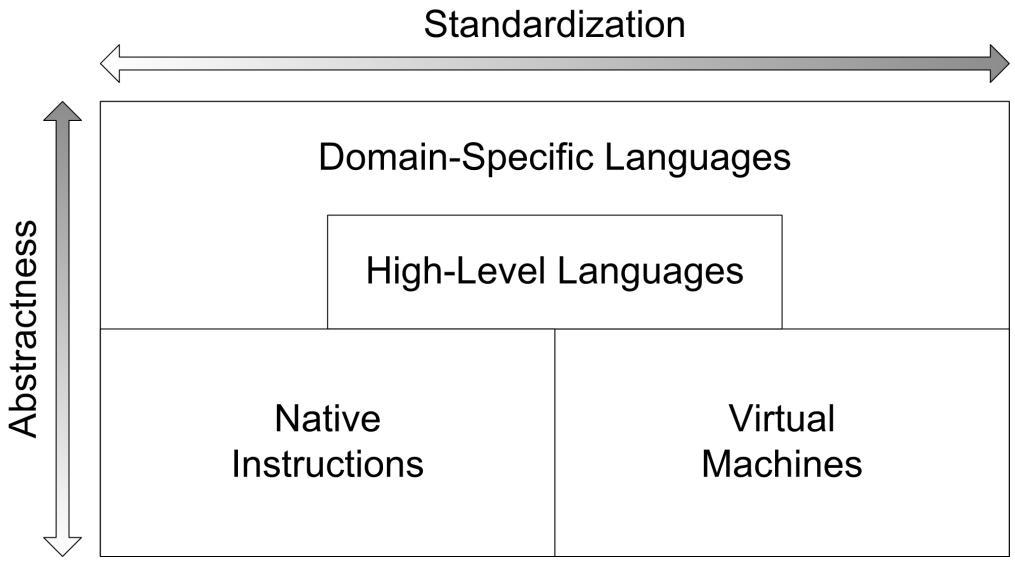



FIGURE 2.2: Approaches to platform-independence. Author's image.

### 2.1.3 Platform Independence

One of the defining characteristics of model-driven development is *platform independence*. Platform independence is concerned with ignoring the details of an application's target runtime environment until as late in the development process as possible. This allows developers to concentrate on the application's functionality without being encumbered by unnecessary details about where that application will finally execute.

Platform-independence is not a new goal and the cost savings that can be realized through such independence have long been recognized. There are two main approaches to platform independence, as shown in Figure 2.2: greater levels of abstraction or greater degrees of standardization.

The idea of platform independence has evolved gradually. Some examples of its key milestones include the advent of high-level languages along with their compilers and interpreters and the introduction of virtual machines.

**High-Level Languages** Early software developers would simply write assembly language code or even directly program code into the physical hardware by direct manipulation of switches. For obvious reasons these approaches did not provide any degree

of platform independence at all; the code was directly tied to the operations supported by the physical hardware. However, since computers were comparatively rare and all of a kind, the idea of "platform independence" had no real meaning. Today, even comparatively simple software may have collaborating components that are deployed to multiple, disparate operating environments. Such software may even be intended to run on many diverse platforms, leading to a need for true platform independence.

One significant evolution of platform independence was higher-level languages along with their compilers and interpreters. These innovations allowed developers to work with higher levels of abstraction than was previously possible. This required a fundamental shift in mindset away from the developer needing to understand the instructions that were available in the target runtime environment and instead toward the encoding of knowledge into the compiler or interpreter itself. This permitted developers to concentrate more on ensuring that their applications fulfilled their functional requirements and less about how they worked within the physical hardware environment. This was possible because the code that was used to represent a program was different from the machine operations that were actually used to execute that code. In fact, such high-level languages provided a greater degree of abstraction than lower-level languages such as assembly language or machine code. As long as there was a compiler or interpreter available, the source code could be compiled into the appropriate instructions for that operating environment.

Although higher-level languages were useful tools, they did not completely abstract away all knowledge of the target runtime environment because their associated compilers and interpreters were bound to those environments. For example, ANSI standard C does not completely dictate the sizes of its integral datatypes. As described by Kernighan and Ritchie:

> "Each compiler is free to choose appropriate sizes for its own hardware, subject only to the restriction that `shorts` and `ints` are at least 16 bits, `longs` are at least 32 bits, and `short` is no longer than `int`, which is no longer than `long`." [41]

This trait of C was carried over into its successor, C++ [42]. While providing flexibility to the compiler designers, these kinds of details also allowed developers to write code that could exhibit different behavior depending on the runtime environment. In addition to needing to know at least some details about the runtime environment, languages like C/C++ also required the developer to properly allocate and deallocate resources such as memory, which provided significant opportunities for mischief. Tales of memory leaks and the havoc that they caused abound in these communities.

In some circles the management of such scarce resources as memory was deemed "too important to be left to developers" and provided some of the impetus needed for the mainstream adoption of *virtual machines*.

**Virtual Machines** In 1995, Java was introduced to the software development community. Java's mantra was "write once, run anywhere" suggesting a true form of platform independence. This was accomplished through the use of a *virtual machine*, which provided a standardized runtime environment within which the Java code would actually execute. It was now up to the Java Virtual Machine (JVM) providers to handle the translation from the JVM specifications into the underlying hardware representation, as suggested by this excerpt from the Java Language Specification that describes the size of the integral types:

> "The integral types are `byte`, `short`, `int`, and `long`, whose values are 8-bit, 16-bit, 32-bit and 64-bit signed two's complement integers respectively...." [43]

The fact that Java provided a language specification that standardized elements of platform dependence into the semantics of the language itself and that those standards were enforced by each JVM enabled the same code to execute within many different runtime environments. This was a significant benefit to the software development industry and persists to this day where the same application might need to run on a variety of common hardware and operating system configurations such as desktops, laptops, mobile phones and tablet computers. Unfortunately, it turns out that while simply abstracting away some of the details of the runtime environment is beneficial, it is not sufficient.

One example, memory leaks, is a problem commonly associated with C++ applications. While such leaks are mitigated by the JVM's Garbage Collection facility, it is still possible for unwary developers to produce code that leaks memory. Even an awareness of this potential problem can be dependent on the experience and ability of the developer.

While resource management and other low-level concerns capture the motivation for increased platform independence, this is by no means the only such example. Other examples, such as user interface design, focus on the desire to build software that can run on many different platforms that have differences not only in the machine instruction set, but in their ability to interact with their users.

**User Interface Design** Another example of platform independence that has been the subject of much research is user interface design. It would be useful if we could provide a single abstract representation of an application's user interface and use it to produce multiple concrete user interfaces based on the actual capabilities of the target platform. This would allow us to use a single interface representation to provide different user experiences based on both their physical platform, such as mobile phones versus tablets, as well as the runtime environment, such as Apple's iOS or Google's Android.

A number of specialized notations have been defined for describing the user interface such as UIML [44] and XIML [45], but the notations themselves must still be converted into something that the target platform understands. Various approaches have been suggested in an attempt to facilitate such transformations including the use of transcoding and style sheets. Another approach, called TERESA [46], allows for the explicit mapping of abstract modeling elements to equivalent elements within the various target platforms, an approach that is common to many MDD transformations.

### 2.1.4 Modeling Languages and Notations

From the user interface, it is a natural progression to wanting to define a platform-independent means of representing an entire application. A number of platform-neutral software development notations are already in use including Z and UML. More broadly,

these notations fall into two main categories: *formal* and *informal*. In this context, "formal" means that the notation is rooted in mathematics and is thus subject to mathematical analysis via model checkers and theorem provers, while "informal" means that the notation cannot easily be subjected to such analysis due to some mathematical incompleteness in the notation itself.

**Z** Z [47–49] provides a modeling notation that captures and formalizes requirements into specifications called *schemas*. A model checker is then used to analyze the schemas to ensure their consistency and to point out any potential conflicts or omissions.

The initial Z language did not include references to objects or object-oriented development. Since then various object-oriented extensions to Z have been proposed [50] such as MooZ [51], Object-Z [52], OOZE [53], Z++ [54], and ZEST [55] in an attempt to bring Z's formalism to object-orientation.

**UML** UML [24] is the de facto standard modeling notation for modern software development efforts. UML uses a series of views that convey different yet complementary information about a software design. These views and their associated UML diagrams are:

- **User**. The user view consists of use cases and their scenarios and is responsible for representing high-level functional requirements. The Use Case diagram belongs to this view.
- **Structural**. This view describes the structural aspects of the software including classes and their relationships. Class diagrams and object interaction diagrams belong to this view.
- **Behavioral**. This view describes the software's runtime behavior and interactions of the various software components. Sequence, Collaboration, Activity and State Machine diagrams all belong to this view.
- **Component**. The component view describes how the various software elements will be packaged together and the dependencies between those packages. While

some of this information can be represented within the Structural view, the Component view is of a sufficiently different level of abstraction that it warrants its own view. The Component diagram comprises this view.

- **Deployment**. The Deployment view shows how the various components will be installed into the target runtime environment and describes the means by which distributed components communicate within that environment. The Deployment diagram belongs to this view.

## 2.2 Programming Languages

The use of static programming languages such as COBOL, C/C++ and Java is standard practice in modern software development. However, it is increasingly common for software to be written using dynamic languages such as PHP, Groovy and Ruby, owing in no small part to the prevalence of frameworks such as Rails and Grails, which eases the process of developing and maintaining complex applications, particularly those in the web domain.

While there are many different languages, and many different ways to categorize them, one way is to group them into *static* and *dynamic* languages. Static languages are those where the rules of the language are typically enforced at compile time; these rules must be satisfied before the code can even be executed. With dynamic languages the enforcement of some of the rules is deferred until runtime.

Although not necessarily the only difference between static and dynamic languages, typing and type checking are often two of the most significant differentiators between the two approaches. Typing rules are intended to guard against the possibility that data is used in ways that are inconsistent with the kind of information being represented. Regardless of its particular rules, each dynamically-typed language is concerned with the two principles described by Cartwright and Fagan [56]:

- **Minimal Text Principle.** The typing system should accept unannotated dynamically typed programs. If programmers are required to provide additional information, then the dynamically typed language will become harder to use than comparable languages and will indeed resemble its strongly typed counterparts.
- **Minimal Failure Principle.** The type system should be able to avoid false negatives when evaluating the run-time checks. If too many false negatives are produced, the developers will ignore the warnings, in essence disregarding the typing system altogether.

Dynamic languages often allow some typing information to be deferred until runtime. This means that a single variable can refer to many different and unrelated kinds of data during its lifetime. As long as the developer does not attempt to perform an operation on a variable that is inconsistent with its data *at the time the code is executed*, then the code will perform correctly. This is a capability that is not available in compiled languages, where each variable has a fixed and immutable type making it difficult to perform invalid operations.

It is this type flexibility and overall runtime interpretation that makes dynamic languages appealing as modeling notations because it allows a modeler to define only what they know and only when they know it, a process that is impractical with a strongly typed language. This same flexibility makes dynamic languages attractive to software developers since they are able to focus more on application functionality and less on the mechanics of actually writing code [57].

Despite the advantages of dynamic languages, there are also some significant drawbacks. One of the greatest is that defects in type usage that could be detected by static languages, may remain undetected within programs based on dynamic languages. This is because unless a particular code path is executed, the interpreter has no need to execute the instructions on that path. Thus an inadequate testing strategy may allow defects to slip into production where they may be found by a user that inadvertently triggers the faulty code path.

## 2.3 Domain-Specific Languages

Deursen [58] defines a *domain-specific language (DSL)* as:

> "...a programming language or executable specification language that offers, through appropriate notations and abstractions, expressive power focused on, and usually restricted to, a particular problem domain."

In other words, a DSL is a language that has been constructed specifically to solve problems within a restricted domain. This is in contrast to *general purpose languages* such as C++ or Java that have been constructed so that they can be used to solve problems irrespective of the domain. DSLs can be either static or dynamic and, in fact, many dynamic languages provide for the creation of *internal DSLs*, which are DSLs based on the language in which they are created.

Because DSLs are constructed to solve a specific class of problems to the exclusion of all other problem domains, they can be extremely efficient. Other benefits can include ease of expression and optimization as well as an enhanced ability to validate and test solutions. Associated costs of DSLs include the cost of its design as well as the cost of educating its users.

There are a number of DSLs in use by software engineers today such as HTML, SQL and Ant. Many DSLs exhibit common characteristics. For example, most of them are declarative, meaning that the author indicates *what* is to be done but does not indicate *how* it is to be done; that is left to the interpreter of the language. Thus, in the case of HTML, the precise approach for rendering content is driven by each browser. In the case of SQL, we indicate the data we want, but we leave it up to the database engine to determine the best approach for satisfying our request.

Another characteristic of DSLs is that they are small. This allows them to be easily understood and focuses their expressive power for the target domain. This is such a common characteristic of DSLs that they are sometimes referred to as "micro languages" or "little languages" [59].

Domain specific languages can be either *internal*, or *homogeneous*, or *external*, or *heterogenous* [60]. Internal DSLs are created by embedding them into a host language. For example, Gradle [61] is a DSL that assists us in performing build management. Gradle commands are written using Groovy and in fact can use all of Groovy's syntactic constructs and APIs. In contrast, make [62], another application for build management, is often written in C but has its own independent language, classifying it as an external DSL.

## 2.4 Agile Software Development

*Agile software development* [63] refers to a collection of processes and methods that embrace the principles articulated in the *Agile Manifesto* [64]. The best known agile processes include *eXtreme Programming (XP)* [65] and *Scrum* [66]. There is evidence showing that agile processes can deliver significantly improved productivity of software development efforts of small to medium sizes in companies with limited organizational complexity [67–69].

While different agile processes and methods vary in their particular aspects, they all embrace the key principles outlined in the *Agile Manifesto*, some of which are summarized as follows:

- Embrace changes and refactoring throughout the development process.
- Working software is the primary measure of progress.
- Working software is developed iteratively and delivered in small increments.

Agile development processes are becoming more and more prevalent in the industry. However, it is commonly acknowledged that there are a number of limitations to the applicability of agile software development processes and methods. They are most effective and most widely used in projects that fit certain well-understood architectural patterns, such as web-based, database-driven applications using an n-tier MVC architecture, where most of the quality requirements are well understood and show only limited volatility.

While agile processes and principles can be applied to software development using any language and target any middleware or platform, of particular interest to this research is one branch of agile methods using dynamic languages such as Ruby [70], Python [71] and Groovy [72], in conjunction with frameworks such as Rails [73], Django [74] and Grails [75]. This style of agile development, pioneered by Ruby on Rails, has gained a great deal of popularity, especially for web based applications. An oft-cited claim is that this combination of agility, language and framework can increase productivity by as much as ten-fold. Whether these claims are accurate or not, it is unquestionable that this approach offers benefits and advantages to organizations that can harness its potential [67].

Some key factors that affect the productivity and applicability of agile methods using dynamic languages include:

- The languages often use a very succinct syntax. They are interpreted and dynamically or optionally typed.[2] They typically support high level data types such as sets, lists, maps, iterators, generators, and closures, thus raising the level of abstraction. However, these features can result in reduced readability due to missing type information. Errors that can be detected through static type checking may not be discovered until runtime and may even slip into production code.

- The extensive use of open-source extensible frameworks and libraries for reusing prepackaged functionalities that significantly reduce development time. Examples of such frameworks include Spring [77], the Google Web Toolkit [78] and Facebook applications [79].

- The adoption of *convention over configuration*, which reduces the complexity of applications by enforcing a set of predefined conventions that eliminate the need for configuration. This can significantly reduce the amount of required code once one is familiar with the conventions. However, this approach reduces the flexibility of applications and presents serious problems when integrating with external, existing, or legacy applications.

[2] This is often referred to as *duck typing*, "a style of dynamic typing in which an object's current set of methods and properties determines the valid semantics, rather than its inheritance from a particular class." [76].

Agile development methods using dynamic languages have not been widely adopted in critical enterprise applications. One of the major obstacles in the adoption is that the runtime environments are not yet considered sufficiently mature, robust, and trustworthy as compared to JEE or .NET. Furthermore, there exist inherent risks associated with dynamic typing and the meta-programming capability of dynamic languages, which allow new kinds of defects to be introduced into applications and which can significantly impact application performance.

# Chapter 3

# A Vision for Model-Driven Engineering

Agile methodologies facilitate rapid changes only as quickly as those changes can be incorporated into the application code. As such it is desirable to have a solution that provides the following properties:

1. Model-centric.
2. Highly abstract.
3. Completely generative.
4. Tool agnostic.

## 3.1 Model-Centric

Model-centricity is the central tenet of all model-driven engineering approaches. Model-centricity means that models are the dominant element of the application and that all application code can be derived, directly or indirectly, from its model. This can be challenging given the complexity of modern software and the forces that it must balance in order to be successful.

### 3.1.1 Software Forces

There are three forces at work in any significant software development effort:

1. **Functional forces**. These are the functional requirements or basic capabilities that the software must provide to be usable by the user community for which it was created. A word processor that does not support the loading and saving of documents might be of little utility to its users regardless of how well it supports complex document layout.

2. **Extra-functional forces**. These are the non-functional, extra-functional [80][1] or Quality of Service (QoS) requirements to which the software must conform. These are the characteristics that will be exhibited by the software regardless of whether or not they have been planned for. These include properties such as performance, scalability and testability.

3. **Environmental forces**. These are the environmental or system requirements imposed by the target hardware platform and operating environment. It does little good to write an application that meets the needs of the users if it cannot be executed within the target environment.

These three forces are similar in that they must all be satisfied, but they are different in how they reveal themselves. Functional forces are the simplest to understand. Either they are satisfied or they are not. If a functional capability is not planned for, then it won't appear in the finished software product. In contrast, extra-functional forces will be present in the finished application whether they are planned for or not. For example, all software exhibits some degree of performance because all software runs on physical hardware which requires at least some amount of time to perform the instructions of the software. Like the extra-functional forces, the environmental forces will always be present. However, unlike extra-functional forces, environmental constraints do not emerge as a result of the reification of the software, but rather exist *a priori* to its development. In some ways the environmental forces are like functional forces in that we can choose our target deployment environment configuration in the same way that we

[1] Bass et. al. refer to the extra-functional requirements as *quality attributes*.

can choose which capabilities the software will provide. However, they are also similar to extra-functional forces in that we often don't know if we can meet the environmental constraints until after the software has been built and its runtime characteristics observed.

These forces can be placed on a graph as shown in figure 3.1. The software application is represented by the cube within the graph. Each vertex on the cube represents the degree for which one or more forces have been addressed. The vertices can range in value along an axis from 0 to 1, where 0 indicates that the force has not been satisfied and 1 indicates that it has been satisfied.

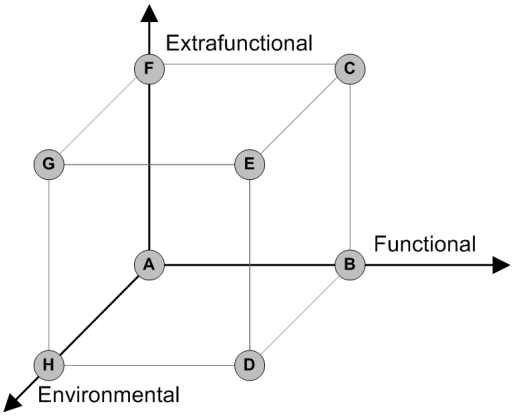



FIGURE 3.1: Software force map Author's image.

One useful property of this graph is that it can scale to different levels of detail. For example, the graph for a single capability within an application can be represented in the same way as the graph for the application as a whole. This capability allows us to compose more comprehensive application-oriented graphs from simpler, capability-oriented graphs using some form of mathematical composition.

Of the eight points shown on graph 3.1, labeled A-H, only a few are meaningful. For example, it is meaningless to discuss software that satisfies its extra-functional or environmental forces if it does not also satisfy its functional forces. This means that we can eliminate points F, G and H from the graph and simplify it as shown in figure 3.2.

Given that from a pragmatic perspective the functional forces must be satisfied before we can even begin to discuss the extra-functional and environmental forces, we can

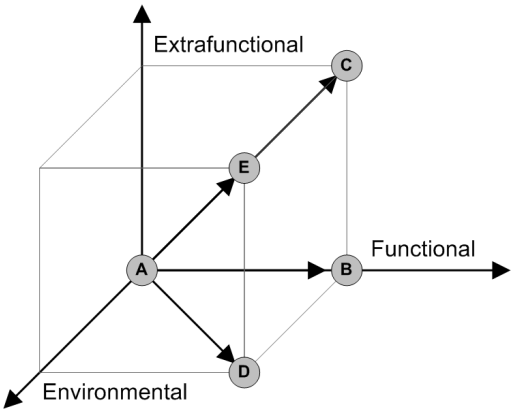



FIGURE 3.2: Simplified software force map. Author's image.

further organize the force graph, which allows us to produce maps that describe the sequence in which the various forces will be satisfied. Two such maps are shown in figure 3.3.

The maps shown in figure 3.3 are an ideal. It is rare that software engineers would be able to proceed along such an orderly route to software completion. Instead, the route typically meanders along the various axes as functionality is added, extra-functional requirements are satisfied and environmental constraints are observed.

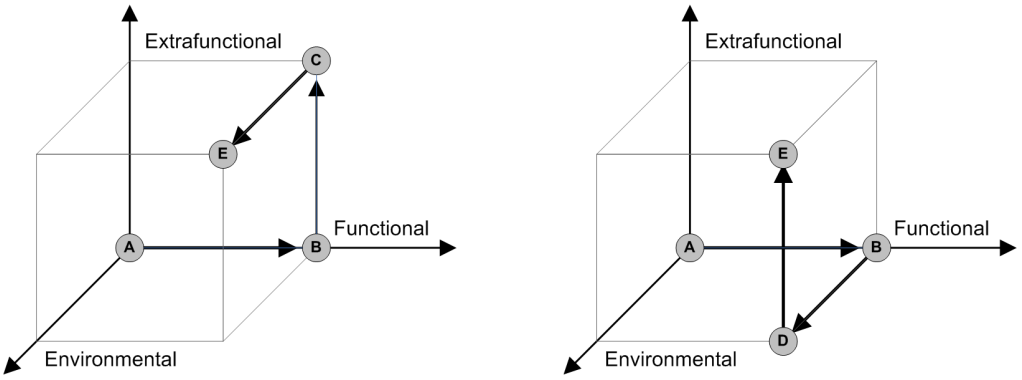



FIGURE 3.3: Software force satisfiability maps. Author's image.

To further complicate matters, while these three forces sometimes support each other, they are more often than not in stark opposition to one another. For example, writing software that performs well may require the use of additional hardware that is not available in the target environment, leading to an unsatisfied environmental constraint. When these three forces come into conflict, they must be re-balanced and brought back

into equilibrium. In the prior example where performance requires additional hardware, equilibrium demands that either:

a. The environment be expanded to include additional hardware;

b. The extrafunctional performance requirement be relaxed; or

c. The functional requirement in question be changed.

Conceptually there is a conservation principal at play that suggests that you can fix any two of the axes in the graph so long as you are willing to vary the third. Fixing all three axes introduces the risk that the three forces cannot be balanced such that all of them are satisfied.

Because finding equilibrium between the functional, extra-functional, and environmental forces is complex, it is often not possible to achieve it after only a single attempt. History is full of projects that balanced their functional and environmental forces, but failed to adequately address the extra-functional forces. The resulting application often requires significant re-work in order to address major problems such as inadequate performance or scalability.

Balancing these forces is a difficult process that is made even more difficult by traditional software development practices. For example, while it is easy to say that "business process X must complete within Y seconds of its initiation", it is difficult to ensure that this requirement will be met until after the software has been built and its performance observed. Rosa *et al.* [81] describes some of the difficulties in defining extra-functional requirements including the fact that such requirements tend to be abstract, are often stated only informally, are difficult to design into software, and are challenging to validate once the system is complete. Often such constraints go entirely undefined while in other cases they develop organically. Statements from users like "the system is too slow" are a classic example of such organic growth. Even if such response time requirements have been defined in the first place, they are often viewed as being "technical details" instead of being true application requirements.

Agile methodologies are ideal for this kind of process precisely because they attempt to provide application functionality as a series of discrete yet related chunks. Using

this approach, a functional requirement may be met for a particular environment while the extra-functional forces are also satisfied. This is because Agile methodologies do not seek to deliver an entire application at once, but instead break the delivery of the application into a series of iterations, each of which delivers some business value. This means that developers have a chance to focus on exactly the feature that they have been tasked to provide and can evaluate their implementation against the extra-functional forces in addition to the functional and environmental ones. Furthermore, when the various forces come into conflict, Agile methodologies provide a safety-valve for resolving the discrepancy, by allowing the scope of a delivery to vary based on the realities observed during development. This means that functionality that fails to satisfy the extra-functional requirements might be deferred to a later iteration if there is little elasticity in the extra-functional requirement. Conversely, if there is significant flexibility in the extra-functional requirement, then the functionality may be delivered as-is, possibly with a story defined to improve it during subsequent iterations.

Mylopoulos *et. al.* [82] describes two approaches used to help manage these difficulties. The *product-oriented* approach views the problem from the standpoint of definition. In this case, the extra-functional requirements are documented in much the same way as their functional counterparts. Once the system is complete, it is validated against both functional and quality requirements to ensure correctness. While intuitive, this approach requires that large portions of the system must be available before its extra-functional characteristics can be established and validated against the extra-functional requirements. In contrast, the *process-oriented* approach focuses on managing the design decisions made during the development process. These techniques allow software engineers to make decisions based on an impact analysis with respect to the extra-functional requirements. The end result is a system that, while still requiring validation, hopefully holds few surprises as it nears completion.

When gaps in the solution with respect to extra-functional requirements are identified, it is often during testing when changes are difficult to implement. Such changes may require significant re-work of the implementation and may not actually address the problem at all. One significant problem is that there are limited means of modeling extra-functional requirements and even fewer means of determining the adequacy of a

solution with respect to those requirements, particularly for those requirements that are emergent only under high transaction loads or significant distribution.

Work has been done in creating frameworks for the capture, prioritization, and trade-off analysis of extra-functional requirements. Examples include a process-oriented approach called *Parmenides* [81–83] and the Architecture Tradeoff Analysis Method (ATAM) [80]. Work has also been done by Cysneiros, *et. al* [84, 85] in the inclusion of extra-functional requirements in UML use case and class diagrams.

### 3.1.2 Force Elasticity

Another challenge related to finding the equilibrium point of these three forces is that their boundaries are often not hard, but are instead fluid. For example, while a particular constraint may at first be described as a requirement, further investigation may reveal that it is instead desirable without actually being binding. For example, while database independence might be a desirable outcome for an application, it is entirely possible that an organization might relax that constraint somewhat to include only the most common database platforms in the industry. This kind of concession is often made in an effort to get the software completed and in a state where it can be used and maintained while also meting some business-driven deadline by which the software can be released.

This kind of *force elasticity* is a measure of how much the boundary for a given force may be exceeded before we actually claim that the boundary has been violated. Elasticity is a kind of tolerance and generally applies only to the lower bounds of software forces. In other words, it tends to represent the minimal quality of service that will be acceptable in a solution. For example, we often don't care how small the response time for a given process is so long as it does not exceed some threshold. Elasticity is a reflection of the realities of software development where the acceptability of a solution encompasses forces with a range of possible values that can often be negotiated based on business necessities. For example, we may decide to compensate for not meeting a particular QoS by adding value in some other way. For example, reduced performance may be acceptable if additional functionality is provided. Such a decision demonstrates

that there are often a range of equilibrium points that we will accept within any given piece of software.

## 3.2 Highly Abstract

Because of the diverse forces that must be satisfied by any given application, and because changing the software to accommodate such forces, particularly when those forces can vary based on the constraints imposed by the functional, extra-functional, and environmental dimensions, it is desirable for any model-centric approach to be highly abstract. This allows the modeler to focus on the primary dimension of functionality, and to defer some of the decisions about architecture and environment until it is time to transform the model into executable code.

One desirable side-effect of a more abstract model is that productivity can increase. If productivity is defined as the measure of output per unit of input, then we would expect that when a highly abstract model is used, the amount of input required for the same output is reduced and productivity thus increases.

The challenge, then, is to provide a highly abstract way in which mobile applications can be defined so that the developers can focus on building the applications and not on the mechanics required to construct and deploy the applications on each target platform. A DSML is ideally suited to this task since, properly crafted, it can synthesize the disparate elements of the different platforms into a single language that can be used to generate the code that will run across all platforms.

## 3.3 Completely Generative

To be effective, and in particular to accommodate agile methodologies, any model-driven approach must be completely generative. That is, the model must contain sufficient information that, when combined with necessary transformations, a complete application must result. It must not be necessary for a developer to manually modify,

enhance or instrument any of the generated code. There are both philosophical and pragmatic reasons for this.

From a practical perspective, allowing a developer to modify the generated code leads to problems of version control and the merging of these changes into subsequent revisions of the model. There is also a concern over the evolution of the model and how developer-provided source code can be managed in parallel.

Philosophically, allowing developers to introduce code outside of the model can lead to a more complex application representation, where one is never certain of just where some functionality resides. It also leads to problems of consistency and optimization. When the code is completely generated, the model and its resulting code will tend to undergo consistent transformational changes. As the transformations evolve and improve, so too will the code resulting from subsequent use of those transformations.

The final, generated code could be one of two styles: native or container-based. In a container-based model, the code generated is expected to run within some other container. This approach is one commonly when the browser provides an abstraction layer that sits between the application and the native operating system of the device. When compared to the native style of code generation, we typically find that accessing the native resources on the device is markedly slower [86, 87]. We thus prefer to generate native code rather than container-based code.

## 3.4 Tool Agnostic

Any model-driven solution should avoid new, specialized tools. In the ideal case, the model can be constructed visually or in a way that makes it look like other development code. This makes it comparatively simple to deal with more complex challenges such as the merging of model changes. It also makes it easier to move models between tools without needing to rely on approaches like XMI.

Allowing models to be tool-agnostic is another way to reduce the learning curve and costs required for MDE adoption.

# Chapter 4

# The AXIOM Approach

Both agile and model-driven development approaches offer considerable benefits in improving the productivity of software development processes and/or the quality of software products. However, each also has its own limitations. It would be beneficial to integrate the two approaches so that they complement each other and offer the best of both approaches. The challenges lie in a number of apparent incompatibilities and gaps between the two approaches:

- **The style of languages**. Agile software development embraces dynamic and scripting languages with little emphasis on static typing and analysis, while model-driven development is based on modeling notations that are visual, declarative, and amenable to a variety of techniques for ensuring qualities, such as static analysis and design by contract.

- **The role of software architecture**. Model-driven development is often characterized as architecture-centric and design-first, while agile software development is often code-centric with architecture as an emergent property resulting from iterative refactoring.

- **The trade-off between productivity and extra-functional qualities**. Agile software development generally emphasizes developer productivity over extra-functional qualities, while model-driven development prioritizes the extra-functional qualities and counts on the modeling aspects of MDD to achieve gains in developer

productivity. Agile development focuses on quality needs in the short term, whereas model-driven development attempts to address long term quality needs up front.

Despite these differences, there are also critical similarities between agile and MDD practices:

- **The focus on a single deliverable**. Agile practices tend to focus on working application code as their single deliverable, since that is what is ultimately required. MDD, too, focuses on a single deliverable: the model.
- **End-user involvement**. In agile practices such as XP, the user is an indispensable part of the software development process. They provide key insights into business functionality as well as held define the most appropriate interaction with the software. It is no different for MDD projects since the models must capture the business rules and UI design.

The ultimate goal of AXIOM is to emphasize the similarities while de-emphasizing the differences between these two approaches.

## 4.1 An Outline of the Approach

Our approach is called AXIOM (*Agile eXecutable and Incremental Object-oriented Modeling*). The AXIOM approach aims to combine MDD techniques with agile approaches to provide true agile MDE. AXIOM consists of the following key technical areas:

- **A dynamic language based modeling notation**. One of the novel aspects of the AXIOM approach is to use a dynamic language, Groovy [72], as the basis of the modeling notation. The Groovy language is augmented with a state machine mechanism based on the UML state diagram. The state machine is defined as a domain specific language (DSL) in Groovy. The specifics of the modeling notation is further elaborated in chapter 6.

- **Model transformation**. One of the trade-offs for using dynamic languages is the substantial degradation of performance of the resulting applications as compared to those developed using statically type languages. We will develop frameworks and techniques to transform the PIMs to PSMs with the goal of minimizing the performance degradation of dynamic languages and attaining the same level of performance and qualities as applications developed using the native tools and languages of the target platforms. The model transformation mechanism is further elaborated in chapter 5.

While we do not intend for AXIOM to directly support one particular style of agile development, such as XP or Scrum, we do intend that it complement the overarching principles of agile methodologies such as those espoused in the Agile Manifesto [64]. In particular, we believe that AXIOM supports agile methodologies in the following key areas:

- Working software is the primary measure of success. By allowing developers to focus on a model of the software rather than its implementation details, AXIOM employs MDA transformations to realize the final application.

- Agile processes promote constant, sustainable development. By encouraging developers to work on models, rather than on application code, we believe that AXIOM allows development teams to more rapidly produce working software at regular intervals. While we will see that AXIOM's DSML allows for less code to be written, and our analysis of AXIOM's impact on productivity supports this assertion, further work is necessary to truly gauge AXIOM's impact on agile development teams.

- Changing requirements are embraced, even late in development. Because the models are at a higher level of abstraction than the equivalent application code, we believe that AXIOM provides development teams more opportunities to spot and address risky changes, thus making it easier to incorporate those changes. Intuitively this is because we expect the AXIOM models to be significantly smaller than the required native code, thus making the evaluation of a change to be an

intellectually simpler task. From an empirical perspective, this assertion deserves more investigation and is not explored further in this research.

Similarly, while AXIOM does not directly support a particular MDD methodology, it does support the key tenets of MDD, which is that we should raise the level of abstraction and make models, rather than code, the focus of software development. By incorporating both a dynamically typed language, we believe that AXIOM raises the level of abstraction while still supporting lower degrees of abstraction when needed. Because AXIOM is model-centric, it retains the key properties of MDD such as the use of platform-independent models and the generation of appropriate platform-specific models by way of targeted model transformations.

Although we expect AXIOM to be applicable across many different application domains, this research focuses exclusively on mobile applications. Such applications provide fertile ground for experimentation because they allow us to immediately test the cross-platform capabilities of AXIOM. Such applications have not yet become as complex as many enterprise applications, which also allows us to scale up AXIOM's ability to deal with more and more complex applications while still reaping benefits during the development of comparatively simpler applications.

## 4.2 A Dynamic Language Based Modeling Notation

One of the key aspects of AXIOM is the use of a dynamic language as the core of the modeling notation. In addition to being directly executable, modern dynamic languages support many of the features found in traditional specification and constraint languages such as Z [47–49] and OCL [36]. These features include: high-level abstract data types such as sets, bags, lists, maps, and relations; high level constructs such as iterators and closures; and extensive support of object-oriented features including mix-ins, categories, and delegations. Furthermore, modern dynamic languages are highly extensible by virtue of their support for meta-programming and domain-specific languages (DSL). While statically typed languages offer some advantages, there are a number of known

incompatibilities among various type systems and conventions adopted by popular modern programming languages. It is unlikely that we can devise a strong type system that can reconcile all of the incompatibilities among the type systems of the potential target languages. Therefore, a dynamically typed language is also a natural choice for a modeling language that is intended to be platform and language independent.

The dynamic language, Groovy, along with a number of augmentations and existing frameworks, forms the core of the AXIOM modeling notation. We choose Groovy for several reasons. First, all the important features necessary for adequately defining constraints in object-oriented models are already supported. Second, Groovy is fully compatible with the Java language and JVM, a mature runtime environment with extensive libraries and frameworks, and a broad install base. Third, Groovy is highly extensible because of its meta-programming capabilities thereby allowing additional features to be added to the base language in the form of domain-specific languages. Fourth, Groovy supports compile-time transformations, which provides a foundation enabling us to build model transformation tools (see chapter 4.3). Fifth, Groovy is open source, which enables us to develop prototypes for our approach by extending and incorporating existing Groovy tools. Finally, Groovy is a popular mainstream dynamic language, which will ease the adoption of the AXIOM approach and make it more accessible to organizations and developers.

The AXIOM modeling notation will only use the subset of Groovy features that are suited to modeling. The annotation mechanism in Groovy will be used to support the stereotypes and profiles in MDD. Groovy is augmented with a textual notation of a state machine DSL.

### 4.2.1 State Machine DSL

A major augmentation to Groovy is to provide visual formalism by defining the UML state diagram as a DSL. The state diagrams define behavioral logic, which is important in many control-oriented, multi-threaded applications such as real-time control systems, mobile applications, computer games, and animations. Using state diagrams to define the behavioral logic is cognitively more natural than using linear textual code.

AXIOM's DSL closely aligns the state machine with the user interface and the interactions between its elements. For example, each view acts as a state while the UI interaction elements such as links or buttons become the transitions within the state machine and serve to drive the overall application during execution.

## 4.3 Model Transformation

One of the trade-offs of using dynamic languages is the performance of the resulting application. Anecdotal evidence [88] suggests that the difference in performance between equivalent implementations in a dynamic language and a static language can be as great as 100 times. The performance penalty is largely due to the dynamic nature of the language that makes compile time optimization impossible and the meta-programming capability that is an essential component at runtime. However, compile time optimization would be possible if the application code base is fixed. When the code base is fixed, it becomes possible to replace the dynamic and meta-programming capabilities with more efficient static code.

The proposed project will investigate a novel approach to translate an application defined in a dynamic language to a *semantically equivalent* implementation in a static language under the *fixed code base* assumption, which states the application code base is fixed and no unknown components or code will become part of the application. Our hypothesis is that the *fixed code base* assumption would enable such transformations to eliminate most, if not all, of the performance degradations inherent in dynamic languages. This assumption is supported by our basic premise that the model provides a complete representation of the application.

AXIOM provides a tool that transforms AXIOM models into equivalent implementations in statically typed target languages such as Java and Objective-C. The development of the AXIOM model transformation tool greatly benefits from the Groovy compile-time transformation framework. While the Groovy transformation framework is designed for intermediate transformations during the Groovy compilation process, we can use that same framework to generate code for different target platforms based on information captured in the AXIOM models.

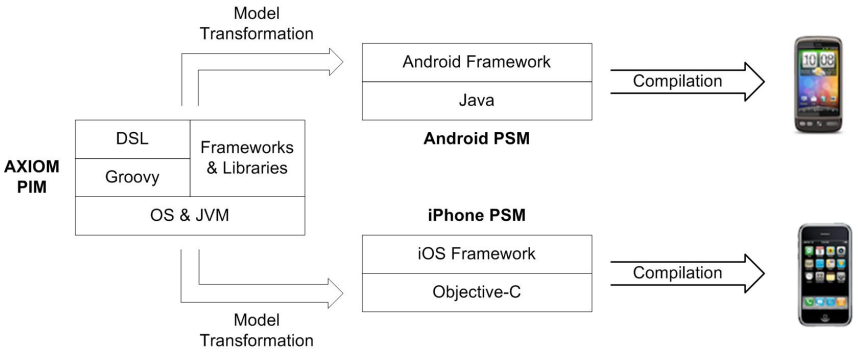



FIGURE 4.1: Using mobile frameworks within AXIOM. Author's image.

A prototype of the AXIOM model transformation tool has been developed. It transforms AXIOM models into implementations in Java and Objective-C. The model transformation tool was used in case studies to demonstrate the feasibility of the AXIOM approach by transforming AXIOM models of mobile applications into implementations for the Android (Java) and iPhone OS (Objective-C) platforms (see Figure 4.1).

## 4.4 Existing Runtime Frameworks and Libraries

One of the limitations of traditional MDD is the lack of powerful frameworks and libraries for modeling. Frameworks and libraries are always language and platform specific and are therefore unusable for platform independent notations such as UML. On the other hand, frameworks and libraries that provide similar or even equivalent capabilities and services are available across most platforms and languages. Thus defining platform independent models without the benefit of the capabilities and services provided by frameworks and libraries is a huge and unnecessary obstacle.

As shown in Figure 4.1, Groovy runs on top of the JVM and can fully leverage all the frameworks and libraries available for the JVM, which include the entire collection of Java libraries and third-party frameworks, such as EJB [89], Spring [77], Android [90], iOS [91] and others. The availability of this extensive collection runtime frameworks and libraries removes one of the major obstacles of MDD.

It must be noted that while these frameworks are available for many platforms, they may only be suitable for *Java-based* target platforms since it cannot be assumed that

every framework has an equivalent within each platform. For instance, while there may be a corresponding framework, as is the case with Spring.NET [92] or NHibernate [93], which are .NET ports of Spring and Hibernate respectively, there may not be an equivalent framework for a platform based on Objective-C. This means that the ability to use these frameworks must be tempered by the knowledge that their use may require a more complex model transformation for some target platforms.

In order to "bootstrap" AXIOM so that it had a basic understanding of the foundational elements of each platform, an ingestion process was devised. This process scans the API documentation for some of the core elements of each platform. A manual process is then used to reconcile and unify the disparate elements of the APIs where possible. This same process can be extended to include other libraries and frameworks, resulting the extension of AXIOM's capabilities without necessitating changes to its core syntax. While not central to this research, this idea has prompted subsequent research in the area of Adaptive Domain-Specific Modeling Languages, described in Section 9.3.2.

By encouraging the use of these readily available, and extremely useful frameworks, AXIOM differentiates itself from the standard OMG MDA approach, which only assumes the "least common denominator" within its models. This means that a model has no access to any pre-existing frameworks or libraries; everything must be modeled *ab initio*. In effect the burden is placed on the modeler to reinvent the wheel when it comes to ubiquitous services such as persistence, asynchronous messaging and security. In contrast, AXIOM places that burden on the transformation process, and then only in those cases where the existing framework cannot be used directly. Thus if a target platform supports Hibernate, the modeler may incorporate Hibernate directly into the model. If a new platform is desired that does not support Hibernate, it will be up to the transformation author to provide the rules that will transform the Hibernate-specific model elements into a form that can be understood by that new platform.

While at first this may be seen as a significant burden on the transformation provider, it is also the case that such transformations may be reused. Thus if a transformation must be written to incorporate Hibernate-like capabilities into an application whose target runtime environment does not support Hibernate itself, that transformation could be

reused for any application that must be transformed to execute within that same runtime environment.

## 4.5 Limitations

In addition to its potential benefits, AXIOM also has its own limitations:

- **Subset of UML**. AXIOM uses only a subset of UML from the structural and behavioral views. The user, component and deployment views are not represented and it is unclear if a complete model can be constructed without them. Part of our research is to determine what other information is required to provide a truly complete model.
- **Modeler & Developer Skill-Sets**. It is unclear as to whether or not modeler and developer skill-sets are the same. If they are not, then this may limit the usefulness of AXIOM since it will still require both skill-sets to provide a complete model.
- **Platform Limitations**. Despite our efforts at providing a common syntax for modelers, the final generation of code for the target runtime environment may still be constrained by the available frameworks and services. This means that there may be an increased burden on transformation authors in the case where there is a mismatch between the services incorporated into the model and those that are available on the target platform.
- **Non-Standard MDD**. Although it retains some of the UML diagrams that underpin the OMG MDA standard, AXIOM is not compliant with than standard. This could limit the adoption of the approach since it may be seen as "proprietary".

Despite these limitations, we expect that AXIOM can provide benefits to the software development industry by encouraging a combination of agile and MDD approaches.

# Chapter 5

# The AXIOM Architecture

The AXIOM lifecycle 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 5.1 illustrates the high-level processes that comprise the AXIOM lifecycle.

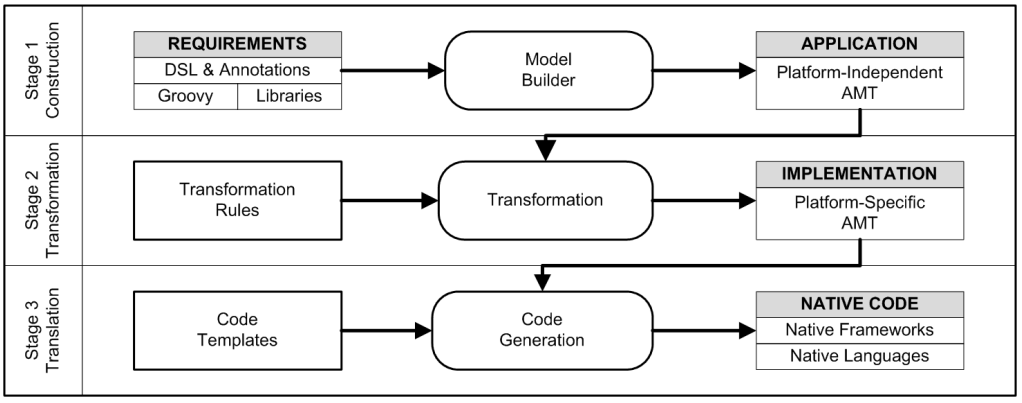



FIGURE 5.1: Stages, phases, and activities of the AXIOM approach. Author's image.

## 5.1 Abstract Model Trees

AXIOM models represent the major characteristics of an application in a platform-independent way. These platform-independent models, called *Requirements* models, describe the core functionality of the application such that it is completely independent of platform- and implementation-specific concerns.

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.

**Definition 5.1.** An *Abstract Model Tree*, *AMT*, is 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.

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 [94]. 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.

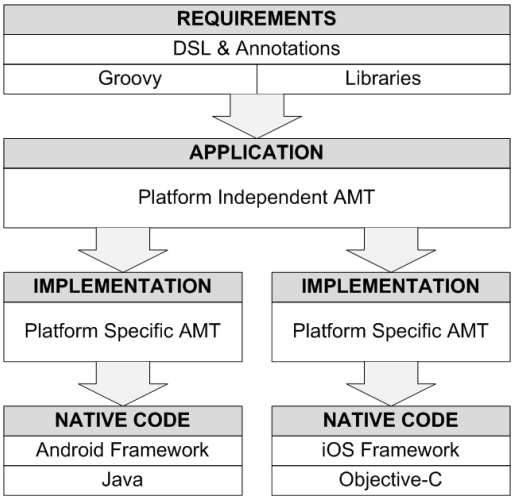



FIGURE 5.2: Model evolution during the AXIOM lifecycle. Author's image.

The AMT evolved over the course of an application's progression through the AXIOM lifecycle. Figure 5.2 shows how different AMTs are used at different times in the lifecycle. However, in each case, the definition of the AMT is strictly observed.

## 5.2 Construction

### 5.2.1 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 AXIOM's DSL, which attempts to maximize the ease of modeling by allowing the requirements model to be represented in a simple, abbreviated form whenever possible. AXIOM's DSL is further elaborated in chapter 6.

The use of a DSL to represent functionality and requirements is not new. However, approaches such as xUML that rely on fUML and ALF [95] 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.

Requirements models are declarative and attempt to completely capture the intent of an application. Because they are written in a Groovy-based DSL, they are executable for the purpose of demonstration and validation. However, requirements models by themselves are insufficient for model transformation and must be supplied with additional information such as:

a) Architecture and design decisions such as the choice of platform, language, framework, API; use of architecture and design patterns, implementation idioms and techniques.

b) Customizations: fine-grained decisions on various aspects of the application, including styles, themes, looks and feels, etc.

These augmentations are almost always platform-specific. They are introduced to the Requirements model during the actual transformation process, which is discussed in detail in Chapter 5.3.

To illustrate the basic elements of the AXIOM approach, consider this simple and incomplete example. An application security system defines users, who belong to various roles. Each role has privileges that determine what the holders of those privileges can actually do within the application. Users want to be able to manage users and roles and their associations as well as manage the privileges held by each role. For simplicity, we don't allow for the creation of new privileges.

The application for this simple set of requirements requires a UI model, to expose those objects and services to the application users, and a behavioral model, to drive the application. While the notation has not been completely defined, the samples in the following sections provide a basic introduction as to their purpose and some of their mechanics.

### 5.2.2 Interaction Perspective

AXIOM's notation allows for the representation of user interfaces in a platform-neutral way. In AXIOM, the emphasis is on *views*. Each view describes its visual and navigational elements, but little more.

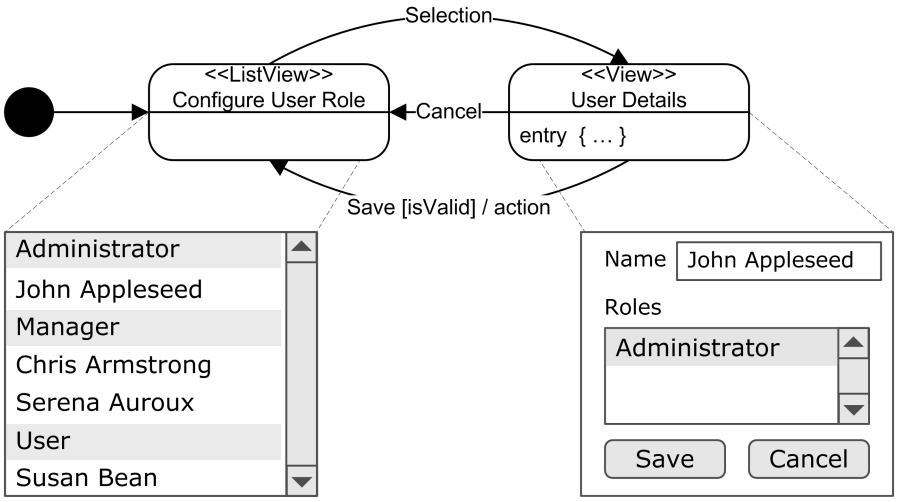



FIGURE 5.3: Requirements model with interaction perspective. Author's image.

Figure 5.3 depicts the simple two-screen application described above represented by a state diagram. The requirements model is shown in Listing 5.1. The AXIOM UI model defines the composition of the two screens. The screens are comprised of several widgets and demonstrate different layouts. The logical UI controls in the requirements model define only their intended functions and not the actual widgets that implement those functions. These logical controls are mapped to concrete implementations during the Translation stage.

The basic transition table for this two-state application is shown in Table 5.1. Each view in the UI model corresponds to a state in the interaction model. Transitions between the states, and hence their associated views, are defined using the `next` attribute of the UI control that triggers the transition. Optional guard conditions and actions can also be defined on the transitions.

| From State... | To State... | By Transition... | Guarded By... | Action... |
|---|---|---|---|---|
| Configure User Role | User Details | Selection | — | — |
| User Details | Configure User Role | Cancel<br>Save | —<br>isValid | —<br>action |

TABLE 5.1: Transition table for simple application

This approach of combining view with an FSM provides a simple mechanism for navigating through the application. Each imperative action within the UI can result in a corresponding transition within the FSM. This example leaves out some significant elements such as the binding of the view to persistent data. These enhancements are part of the scope of this work.

### 5.2.3 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.

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.

## 5.3 Transformation

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

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

  ListView(id: 'Configure User Role') {
    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 : 'Role')
      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
                    }])
    }
  }
}
```

LISTING 5.1: Requirements model of a simple application.

process. Each pass through the transformation process may apply one or more of either kind of transformation rule.

AXIOM's transformation process adheres to the basic OMG MDA vision for PIM-to-PSM-to-code conversions. The core of the AXIOM transformation process is a Groovy

*Builder*. Like AXIOM's notation, a builder is an internal DSL. There are a number of existing Builders within Groovy such as those for XML or HTML. In our case, the builder is called the `AxiomBuilder`. The `AxiomBuilder` can be engaged on-demand and as often as needed. This facilitates MDD based on agile methodologies where teams follow iterative and incremental development.

The purpose of the `AxiomBuilder` is to transform AXIOM models into source code for the target platform. We do this by providing an abstraction of the platform and making that available to the `AxiomBuilder` so that we effectively have an implementation of the Strategy design pattern. Each of these strategies encapsulates the mappings required to transform AXIOM models into code for the target platforms. These transformation rules effectively act as "plugins" and advise the overall transformation process. This approach is shown in Figure 5.1, where the same AXIOM models can be transformed into either iOS or Android applications.

All model transformations begin with the AXIOM Requirements models. As shown in Figure 5.1, the process of transforming the intent models into executable code is divided into three phases: structural transformation, decorative transformation, and code generation. Each phase transforms one AMT into another. Figure 5.4 illustrates the changes introduced by each phase of the transformation process.

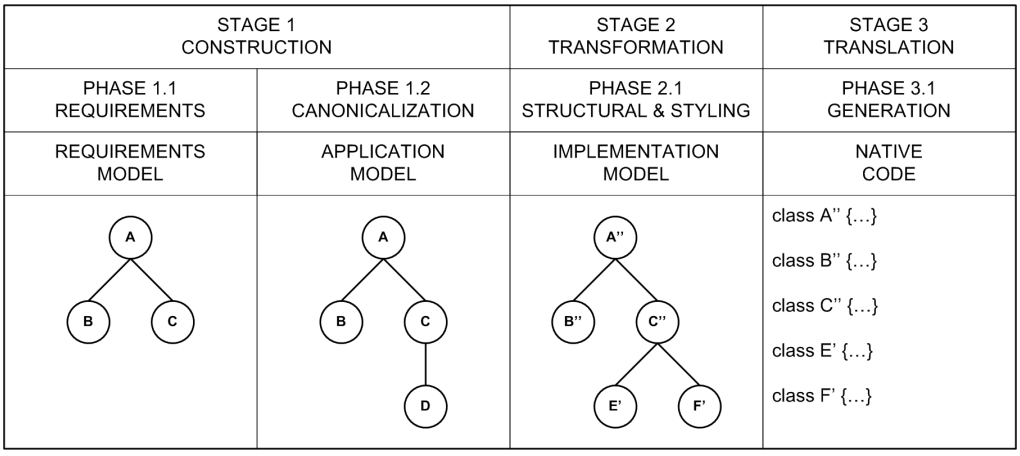



FIGURE 5.4: Transformations of AXIOM models. Author's image.

### 5.3.1 Transformation Rules

AXIOM defines two types of transformations: *structural* and *styling*. We elaborate on these two kinds of transformations in Sections 5.3.1.1 and 5.3.1.2 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 5.2.** A *transformation rule* has one of the following forms:

$$LHS \rightarrow LHS' \tag{5.1}$$

$$LHS \rightarrow N_1, ..., N_k \tag{5.2}$$

$$LHS \rightarrow \varepsilon \tag{5.3}$$

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 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.

#### 5.3.1.1 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.

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. Specifically, these transformations change the AMT by:

- Adding nodes along with their corresponding edges.
- Splitting single nodes into multiple nodes and adjusting the edges accordingly.
- Merging multiple nodes into a single node and adjusting the edges accordingly.
- Adding attributes.
- Removing attributes.

**Definition 5.3.** A *structural transformation* on an abstract model tree, $AMT$, results in a new abstract model tree, $AMT'$, such that:

$$AMT' = (N', E', A') \tag{5.4}$$

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, or design model, 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 JPA, Spring JDBC, or Active Record for the persistence. Similarly, we might opt to generate DAOs as a means of enforcing a separation of concerns within the generated

code. These choices will obviously yield very different designs when the model is ultimately translated into native code.

#### 5.3.1.2 Styling Transformations

In contrast to structural transformations, Styling transformations preserve the underlying structure of the design model's AMT, but add more information to the AMT's nodes. This results in an AMT that is not only functionally isomorphic to the original application model, but that is structurally isomorphic to the design model. Styling transformations not only use the injection descriptors but also decorate the design model with additional platform-specific elements to address intra-class, micro-organizational decisions.

Styling transformations change the AMT by:

- Adding attributes.
- Removing attributes.

**Definition 5.4.** A *styling transformation* on an abstract model tree, $AMT$, results in a new abstract model tree, $AMT'$, such that:

$$AMT' = (N, E, A') \tag{5.5}$$

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 highly reusable.

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

**Case 1** The same 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. For example, radio button groups exist natively on Android, but must be simulated or replaced on iOS.

**Case 3** The widget does not exist in both platforms and cannot be effectively simulated. Examples include Android's ImageButton and iOS's PageView. Even though the widget could be encoded within the DSL, the application cannot be made cross-platform without changing the transformations 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 overall 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 characteristics 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.

### 5.3.2 Organization

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

- Modules. The macro-organizational aspects of the application.

- Resources. The component files that will comprise the completed applications. Such files may include source files, but also whatever descriptors 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 that is required to generate these elements during the code generation process in the form of injection descriptors.

### 5.3.3 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 5.5.** An *injection descriptor*, $d_i$, is a 3-tuple:

$$d_i \quad = \quad (\textit{target, template-ref, binding}) \tag{5.6}$$

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

## 5.4 Translation

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 [96] 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 $n \in N$ in $M$
3 **for** each injection descriptor $d_i$ of $n$
4 Retrieve the template $d_i[template\text{-}ref]$
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 5.2 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.

One key difference between AXIOM and other MDD transformation approaches is that we do not require a formal metamodel mapping to the OMG's Meta Object Facility (MOF). One key purpose of MOF is to drive XML Metadata Interchange (XMI), an OMG standard for allowing UML models to be represented in a portable format that can used across different modeling products and tools. In essence MOF attempts to provide the same "write once, read anywhere" properties as Java. Because the AXIOM models are just Groovy source code, any text editor has the ability to open the models, which means that existing software development tools can manipulate these models. This also allows AXIOM to provide transformations and translations for technologies without MOF models.

```
1  package ___PACKAGE___;
2  /**IMPORT INJECTION POINT**/
3  public class ___VIEWNAME___
4  extends ___SUPERCLASS___ {
5    /**DECLARATION INJECTION POINT**/
6    @Override
7    public void onCreate(Bundle state) {
8      super.onCreate(state);
9      /**ONCREATE INJECTION POINT**/
10   }
11   /**METHOD INJECTION POINT**/
12 }
```

LISTING 5.2: Partial template for Java view implementation.

Figure 5.5 shows the screen shots of the application generated from the sample application in Listing 5.1. In this case the AXIOM model and its corresponding transformations are for the iOS version of the application. An equivalent Android implementation could also be generated.

If instead of a mobile application we desired a web application, then we would require new structural and decorative transformation rules as well as an appropriate set of templates for use during the code generation phase of model transformation. We briefly discuss the use of AXIOM in support of alternate domains in Chapter 9.3.1.

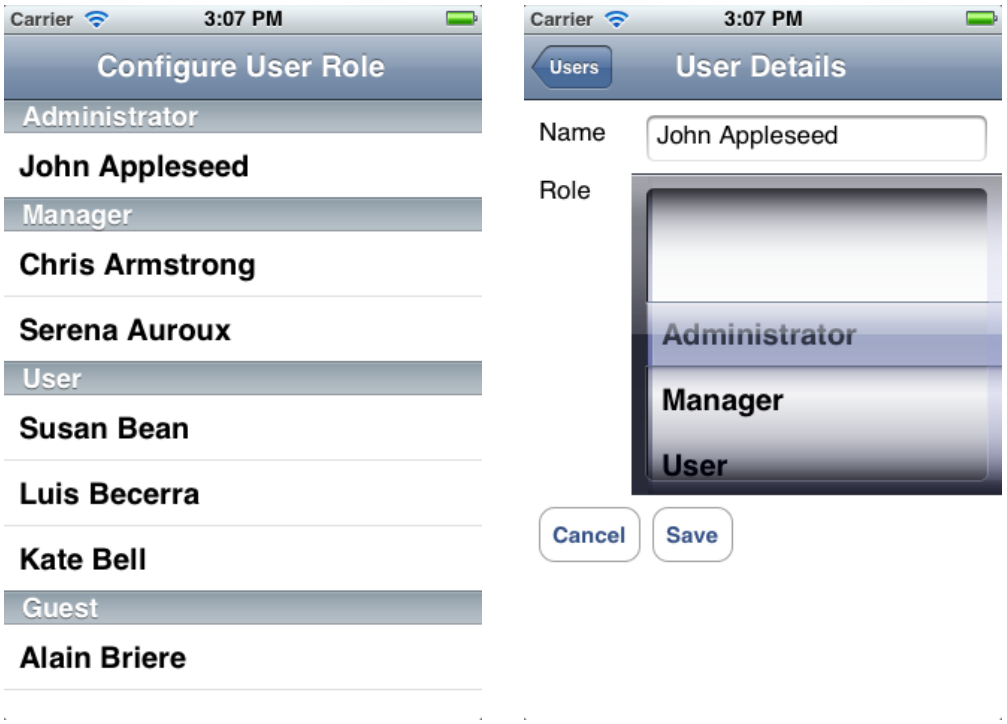



FIGURE 5.5: Screen shots of generated application on iOS. Author's image.

# Chapter 6

# The AXIOM Notation

The AXIOM notation is the actual modeling language. Requirements models are created in model definitions files and are then processed as described in 5. Intuitively, AXIOM model objects form a tree, organized into progressively more granular elements the deeper in the tree those objects are located.

In this chapter we discuss the basic AXIOM language elements and their impact on the final, native code.

## 6.1 MADL Files

Each application is defined within a single *Mobile Application Definition Language*, or *MADL*, file. MADL is synonymous with the AXIOM notation. However, whereas AXIOM describes our entire approach to model-driven mobile application development, MADL refers only to the notation.

MADL files use a `.madl` extension and can be used to generate multiple native applications with the help of a model configuration file.

### 6.1.1 Model Configuration Files

Model configuration files define certain key pieces of information about the application itself as well as about the target platforms for which code will ultimately be generated. Listing 6.1 provides a simple example:

```
1  developer {
2    name   = "App Team"
3    org    = "App Inc"
4    domain = "apps.com"
5  }
6  platform {
7    ios {
8      version = 7
9    }
10   android {
11     version = 4.4
12   }
13 }
```

LISTING 6.1: Simple MADL configuration file, version 1.

This simple example defines some high-level information about the application provider in the `developer` section, as well as information about the target iOS and Android platforms in the `platform` section. It is possible to use a single model configuration file for multiple applications to help ensure consistency.

## 6.2 Objects, Attributes, and Attribute Values

There are several commonly-used objects within MADL:

**Application** Represents the application as a whole. There can only be one of these objects for each application and it must be the root object in the model.

**Top-Level Views** UI elements that act as screens within the application. Each view acts as a state within the applications state machine. Only view objects can be children of the application object.

**Widgets** Logical UI controls that provide the main functionality of the user interface. Widgets include components such as labels or buttons and are use to trigger state transitions between views.

**Containers** Groupings or collections of widgets. Groups are a special kind of widget and effectively allow modelers to implements the *Composite* design pattern.

Each object within MADL follows the same syntax:

```
ObjectType ( Attributes ) {
  Children
}
```

Each `ObjectType` must be one of those supported by MADL. `Children` are other, nested objects. For widgets there will be no children and the curly braces can be omitted.

An object's `Attributes` are a comma-delimited list of key-value pairs in the form of:

```
key 1 : value 1, key 2 : value 2, ..., key N : value N
```

Each MADL object type has a set of unique keys. Although arbitrary key-value pairs may be provided, unless they map to the pre-defined attributes for the specified object type, they will be ignored.

Besides being simple key-value pairs, attributes can also be objects. In such cases, their value can be a collection of attributes themselves. For example, the `font` attribute (of type `Font`) demonstrates some of the flexibility of MADL attributes. It accepts a 3-tuple of:

```
[font-name, font-style, font-size]
```

Attribute values can be provided in any order. If no values are provided default values will be used.

## 6.3 Applications

The root of each MADL file is the `app` element. This is the basis for the abstract model tree that we defined in Chapter 5.1. Each application is rooted in an `app` node with a set of related properties, along with a set of views, which represent the various screens in the mobile application.

Consider the classic "Hello, World" example shown in Listing 6.2. In this example the application has the name of "Hello, World" and contains a comment placeholder identifying a single top-level view. Views are described in more detail in the next section.

```
1 app(name:'Hello World') {
2   // top-level view
3 }
```

LISTING 6.2: Requirements model of "Hello, World" application, version 1.

## 6.4 Top-Level Views

Each application can have one or more top-level views. Views act as the states in the application's state diagram, with the first view being the start state when the application is first executed. A simple, one-view example is shown in 6.3:

```
1 app(name:'Hello World') {
2   View {
3     // widgets
4   }
5 }
```

LISTING 6.3: Requirements model of "Hello, World" application, version 2.

### 6.4.1 View Types

There are several types of views supported by MADL, each of which supports a commonly used mobile interface style. Table 6.1, describes these view types and the platforms that support them.

TABLE 6.1: MADL View Types.

| View Type | Description | Platform | |
|---|---|---|---|
| | | Android | iOS |
| Expandable List View | Groups list data by categories that can be open (expanded) or closed. | ✓ | |
| List View | Displays a scrollable list of items. | ✓ | ✓ |
| Navigation View | Displays the application's main navigation options. | ✓ | ✓ |
| Page View | Displays content page-by-page. | | ✓ |
| Popup | Displays a popup window. Popups must be defined within another view type. | ✓ | ✓ |
| Tabbed View | A view that contains other views. Each nested view receives its own tab on the parent view. | ✓ | ✓ |
| View | A generic view with no special functionality. | ✓ | ✓ |

### 6.4.2 View Properties

Like other objects within MADL, views have properties. One of the most useful is the `id` property, which uniquely identifies each view within the application. The `id` is used by widgets to navigate between views, and thus act as the state transitions within the application's state machine. As shown in Listing 6.4, views can have additional properties such as titles, backgrounds, and orientations.

```
1 app(name:'Multi-View')  {
2   View(id:view1, title:'View 1') {
3     // widgets, with transition to view2
4   }
5   View(id:view2, title:'View 2') {
6     // widgets, with transition to view1
7   }
8 }
```

LISTING 6.4: Requirements model of multi-view application, version 1.

### 6.4.3 Navigating Between Views

Each mobile platform supports a variety of graphical widgets. These will be discussed more in Section 6.5, but here we wish to examine a key property of some widgets, which is the support of navigation through the views of the application.

Select widgets such as `Button`, `Item`, and `Image` have a property called `next`. This property refers to the unique ID of a view within the application and is used to navigate the user to that view. Listing 6.5 shows the same two-view application from Listing 6.4, but includes `Button` widgets that trigger navigation between those views.

```
1 app(name:'Multi-View')  {
2   View(id:view1, title:'View 1') {
3     Button(text:'Show View 2', next:view2)
4   }
5   View(id:view2, title:'View 2') {
6     Button(text:'Show View 1', next:view1)
7   }
8 }
```

LISTING 6.5: Requirements model of multi-view application, version 2.

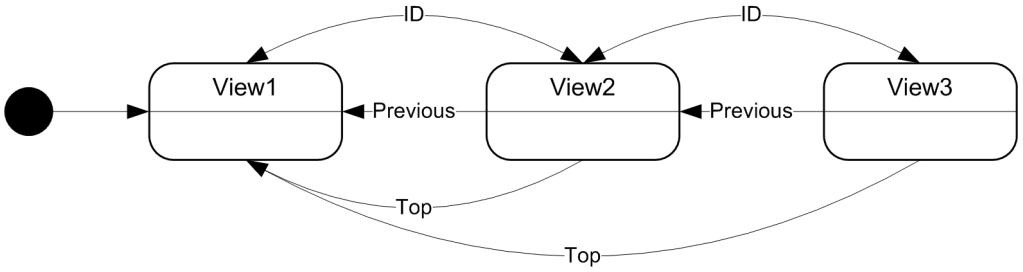
View navigation can be specified either by using a view ID or by using the `Previous` and `Top` keywords. The *view ID* sends the user to the view with the specified ID, the `Previous` keyword sends the user to the most recent view, and the `Top` keyword sends the user to the top-most, or initial application view. The applicability of these different navigational types is shown in Figure 6.1.

### 6.4.4 Passing Data Between Views

In many cases, one view will need to use the data from another view. For example, we might use one view to acquire some data and a second view to actually process that data. For such a design approach to work, we must be able to pass data from the source view to the target view. To accomplish this, we use a variation of the `next` attribute that accepts not only a view ID but the data to be provided to that view:

```
next: [to: view-id, data: expression]
```

The `data` element of the list can contain any data that the developer needs to provide to the view including complex types such as objects, lists, and maps. The target view can then refer to that data using an implicit variable called 'data'.




| From State... | To State... | Available Navigation Types | | |
|---|---|---|---|---|
| | | ID | Previous | Top |
| View1 | View2 | ✓ | | |
| View1 | View3 | ✓ | | |
| View2 | View1 | ✓ | ✓ | ✓ |
| View2 | View3 | ✓ | | |
| View3 | View1 | ✓ | | ✓ |
| View3 | View2 | ✓ | ✓ | ✓ |

FIGURE 6.1: Availability of navigation types for view display. Author's image.

## 6.5 Widgets

Applications and views by themselves don't provide much in the way of functionality. In order to implement a user interface, each view must incorporate *widgets*. For example, we can add a `Label` widget that displays static text as shown in 6.6:

```
1 app(name:'Hello World') {
2   View {
3     Label(text: 'Hello, World!',
4           font: [Bold, 36], color: Red)
5   }
6 }
```

LISTING 6.6: Requirements model of "Hello, World" application, version 3.

In this case, the `Label` requires the `text` attribute to be provided, but we have also provided several optional attributes such as `font` and `color`.

### 6.5.1 Model Configuration and Widget Mappings

As we discussed in Chapter 5, one of AXIOM's challenges is to be able to represent platform-specific widgets in a platform-independent way. Within MADL this is facilitated using the *design* section of the model configuration file, as shown in Listing 6.7.

```
1  developer {
2    name   = "App Team"
3    org    = "App Inc"
4    domain = "apps.com"
5  }
6  platform {
7    ios {
8      version = 7
9    }
10   android {
11     version = 4.4
12   }
13 }
14 design {
15   android {
16     Selection {
17       type = 'RadioGroup'
18     }
19   }
20 }
```

LISTING 6.7: Simple MADL configuration file, version 2.

In this example we have overridden the default drop-down list that would ordinarily be selected with a `RadioGroup` implementation instead. It is possible to make this change at the application level, as shown here, or to only specific instances of the `Selection`.

## 6.6 Actions

Applications of any real usefulness will typically need to respond to user actions and perform related tasks. Listing 6.8 shows a simple application and how it can react to actions initiated by the user's interaction with the widgets. Specifically, interactions with the various widgets in the application will cause a label's text to be updated to reflect that interaction.

The `Label` defined on line 4 has an attribute called `id`, which allows us to assign it a unique ID within the model. This ID can be used to refer to that widget from anywhere within the model. The various `Button`, `Slider`, `Switch` and `Selection` widgets each define a closure for the `action` attribute. Whenever the user interacts with a widget, the closure for its `action` will be executed. In this example, each `action` closure is defined to update the text of the label with ID 'label'. This example also demonstrates the use of template string expressions such as `$value` on lines 14 and 21. This is little more than the use of a Groovy-String, or G-String, to evaluate an expression.

In addition to explicitly-defined actions, like those shown in Listing 6.8, MADL also supports the concept of *implicit actions*. An implicit action is one that is defined by virtue of the relationships between widgets. For example, consider the following snippet:

```
Switch(id: sw1)
Label(text: 'Switch is ' + (sw1.on ? 'On' : 'Off'))
```

Notice that the `Label` uses the value of the `Switch` widget to display its current state. However, `Label` widgets don't support the `action` attribute – they are entirely passive controls and do not perform actions by themselves. However, AXIOM will identify the relationship between these two widgets and generate an implicit action for the `Switch`, effectively transforming the snippet above to the following:

```
Switch(
  id: sw1,
  action: {label1.text = 'Switch is ' + (sw1.on ? 'On' : 'Off')}
)
Label(id:label1)
```

Implicit actions are a convenient way for the application developer to focus on the result that they want and not on the mechanics of realizing that result.

```
app(name: 'Actions') {
  View {
    Label(text: 'Simple Actions')
    Label(id: label, width: '*')
    Button(
      text: 'Button 1',
      action: {label.text = 'Button 1: Touched'}
    )
    Button(
      text: 'Button 2',
      action: {label.text = 'Button 2: Touched'}
    )
    Slider(
      action: {label.text = "Slider: ${value}"}
    )
    Switch(
      action:
        {label.text = 'Switch is ' + (on ? 'On' : 'Off')}
    )
    Selection(
      options: ['One', 'Two', 'Three'],
      action: {label.text = "Current selection: ${value}"}
    )
  }
}
```

LISTING 6.8: Simple actions.

### 6.6.1 View States

Actions provide significant flexibility in terms of being able to manage the user experience of an application. However, by themselves they also make the model much harder to understand and maintain. Consider Listing 6.9 as an example.

The actions in this case are a more complex sequence of changes rather than the simpler, single changes that we've looked at until now. Since there is no limit on the complexity of an action, it is clear forcing the action code to be inline with the widget definition does not lead to easily understandable or maintainable models. To address this problem, MADL provides *view states*.

View states are a mechanism by which the logic of an action can be placed into an external state definition and referenced using a widget's `next` attribute. If we consider each view to be a state within the mobile application, then view states are like sub-states.

Listing 6.10 demonstrates the use of view states to make the code from Listing 6.9 more readable. Figure 6.2 shows the state diagram corresponding to this simple application.

```
app('App State') {
  View(id: view1) {
    Label(id: l1, text: 'Hello', width: '*')
    Text(id: t1, prompt: 'enter text')
    Row {
      Button(id: b1, text: 'Enable', action: {
        t1.enabled = true; b1.enabled = false;
        b2.enabled = true
      })
      Button(id: b2, text: 'Disable', action: {
        t1.enabled = false; b1.enabled = true;
        b2.enabled = false
      })
    }
  }
}
```

LISTING 6.9: Complex actions, version 1.

The new model defines two `state` objects. These objects are similar to views in that they have unique identifiers that are used to navigate between them. The first `state` is always considered the initial state and will be executed when the view is first rendered. Each `state` can define closures representing the major lifecycle events of that state. The `onEntry` closure will be executed whenever that state becomes active.

View states are activated using the `next` attribute of the widgets that comprise the view's UI. In this example, the view changes state in response to the user interactions with the two buttons defined on the view.

#### 6.6.1.1 Guard Conditions

There is one additional refinement that we can make to the model in Listing 6.10 and that is to shift the transition between states from the view widgets to the states themselves. At the same time we can impose guard conditions on when those transitions are followed by using the `when` keyword within the affected view state as shown in Listing 6.11.

```
app('App State') {
  View(id: view1) {
    Label(id: l1, text: 'Hello', width: '*')
    Text(id: t1, prompt: 'enter text')
    Row {
      Button(id: b1, text: 'Enable', next: s1)
      Button(id: b2, text: 'Disable', next: s2)
    }

    state(id: s1) { // initial state
      onEntry {
        t1.enabled = true; b1.enabled = false;
        b2.enabled = true
      }
    }
    state(id: s2) {
      onEntry {
        t1.enabled = false; b1.enabled = true;
        b2.enabled = false
      }
    }
  }
}
```

LISTING 6.10: Complex actions, version 2.

### 6.6.2 Events

MADL supports a variety of built-in events to deal with common mobile interactions. The application shown in Listing 6.12 uses two events, `onShake` and `onTouch`, to react to interactions with the mobile device itself, rather than to react to interactions with specific graphical widgets. Table 6.2 provides examples of commonly-used events.

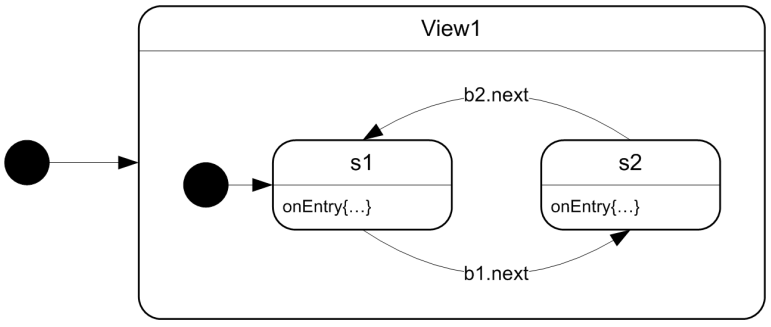



FIGURE 6.2: View states. Author's image.

```
app('App State') {
  View {
    Label(id: l1, text: 'Hello', width: '*')
    Text(id: t1, prompt: 'Line 1')
    Text(id: t2, prompt: 'Line 2')
    Button(id: b1, text: 'Done')

    state(id: s1) { // initial state
      onEntry { b1.enabled = false }
      when { t1.text && t2.text } next s2
    }
    state(id: s2) {
      onEntry { b1.enabled = true }
      when  { !t1.text || !t2.text } next s1
    }
  }
}
```

LISTING 6.11: Complex actions, version 3.

Each event is defined as a closure, which allows it to interact with all of the widgets within the view, as well as implementing any required business logic.

Events can also define closures to be executed before or after the event itself. Listing 6.13 demonstrates the use of the `doAfter` closure attached to the `onTap` event. In this case the `doAfter` closure changes a label in the view based on whether or not the user taps or double-taps the display. Notice that the first `doAfter` block has a time delay of 2.5 seconds, which defers the execution of the closure for that period of time. The second `onTap` event (which could also have been defined using the `onDoubleTap` event) performs its `doAfter` script immediately after its own execution.

TABLE 6.2: MADL Events.

| **Event** | **Is triggered when...** |
|---|---|
| onDoubleTap | The device's screen is tapped twice in quick succession. |
| onOrientationChange | The device's orientation changes between portrait and landscape. |
| onShake | The device is shaken. |
| onTap | The screen is tapped. An optional parameter can be used to indicate how many taps are expected for the closure to be executed. |
| onTouch | The device's display is touched. |

```
app('Shake and Break 01') {
  View(statusBar: no) {
    Image(id: home, file: 'home.png')
    onShake {
      home.file = 'homebroken.png'
      play 'glass.wav'
    }
    onTouch {
      home.file = 'home.png'
    }
  }
}
```

LISTING 6.12: Event handling, version 1.

```
app('Tap Gesture 2a') {
  View {
        Label(id: label1)
        Label(id: label2)
        onTap {
          label1.text = 'Single tap detected'
        } doAfter delay:2.5.second, { label1.text = '' }
        onTap(taps: 2) {
          label2.text = 'Double tap detected'
        } doAfter { label2.text = '' }
  }
}
```

LISTING 6.13: Event handling with lifecycles.

#### 6.6.2.1 Events and View States

MADL allows events to be combined with view states, thereby permitting an application to be behave differently for the same event depending on its current state. Listing 6.14 demonstrates this capability.

```
1  app('Shake and Break 01') {
2    View(statusBar: no) {
3      Image(id: home)
4      state(id: s0) {
5        home.file = 'home.png'
6      }
7      state(id: s1) {
8        home.file = 'homebroken.png'
9        onTouch(next: s0)
10     }
11     onShake {
12       play 'glass.wav'
13     } next s1
14   }
15 }
```

LISTING 6.14: Event handling with view states, version 1.

# Chapter 7

# Evaluation

A proof-of-concept prototype tool has been developed to demonstrate the feasibility of AXIOM. The prototype tool transforms AXIOM models into native implementations for the Android and iOS platforms. The design of the generated code follows the common MVC architecture. 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.

## 7.1 Approach

Using the prototype tool, we conducted two kinds of analyses: small-scale and mid-scale. The small-scale tests evaluated the initial AXIOM Requirements model against the code generated from that model. The mid-scale tests compared the code generated from the Requirements model to hand-written code provided by experienced software developers.

### 7.1.1 Small-Scale Experiments

In these experiments we assessed more than 100 small-scale tests, each of which models a working mobile application that can be successfully built and deployed on iOS and

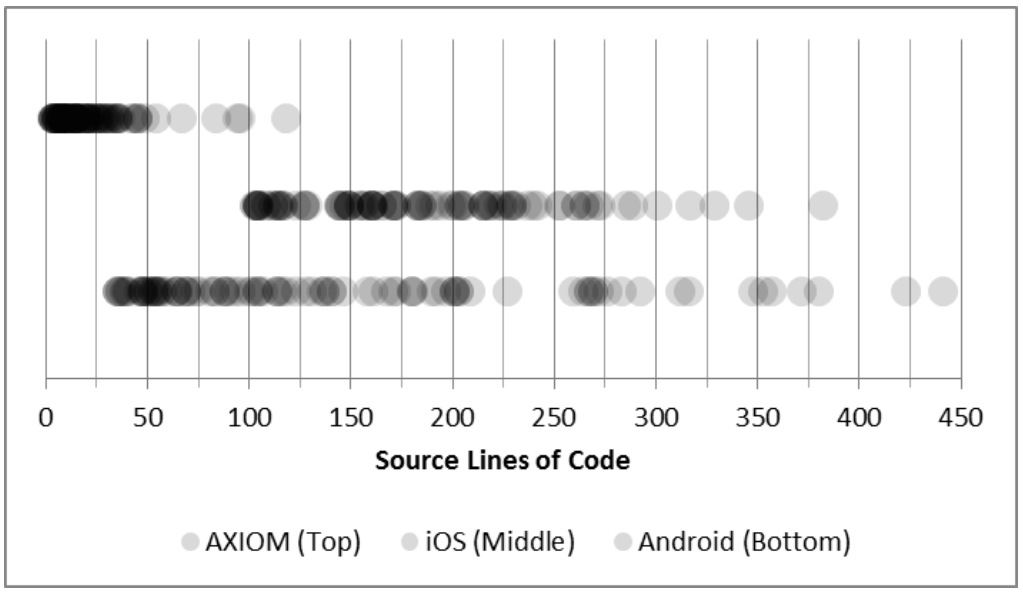



FIGURE 7.1: Strip plot of SLOC by platform. Author's image.

Android devices and which demonstrate functionality common to many mobile applications including screen navigation and assorted widgets – some cross-platform, others not.

The strip plot in Figure 7.1 shows the comparative frequencies of the source lines of code by platform and provides some basic descriptive statistics. The plot uses transparent points for each data point, 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 students were all experienced software developers, although there were significant differences in their mobile application development expertise. None of them had used AXIOM before but they were trained in its DSML.

### 7.1.2 Mid-Scale Experiments

In these experiments, five mid-sized applications were developed featuring a variety of navigation and user interactions. Table 7.1 describes the applications and some aspects of their structure and complexity.

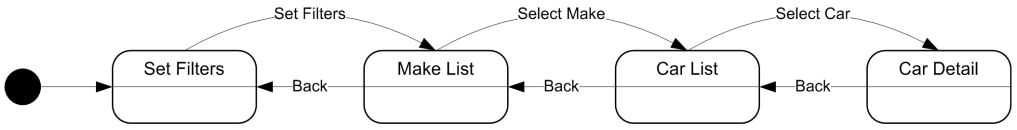
TABLE 7.1: Description of Mid-Scale Applications.

| App. | Description | Screen Count | Transition Count |
|---|---|---|---|
| CAR | Shop for cars by makes and models. | 6 | 8 |
| CVT | Unit conversions for weight, volume, etc. | 8 | 7 |
| EUC | Data about EU member countries. | 3 | 2 |
| MAT | A memory game where players must match pictures. | 1 | 1 |
| POS | A simple restaurant point-of-sale system. | 8 | 9 |

The native-code versions of these applications were developed by a Masters student from DePaul's Software Engineering program. The AXIOM models were developed by the same student with significant training and input from the authors. To ensure consistency, each application had a pre-defined set of requirements that needed to be met by both the AXIOM and hand-written implementations.

#### 7.1.2.1 CAR

The CAR application allows a user to browse for cars. They can define some simple filter criteria. Searching based on that criteria will send them to a list of matching automobiles. Selecting one of the automobiles provides more details. A finite state machine and transition table for the CAR application is shown in Figure 7.2.



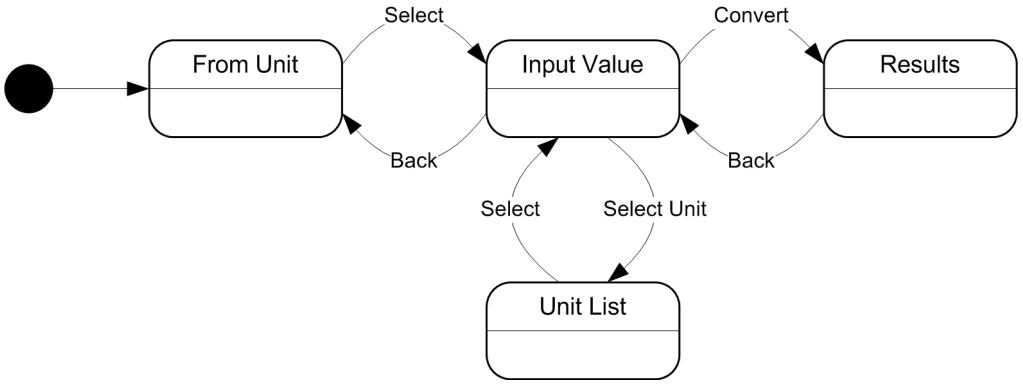

| From State... | To State... | By Transition... | Guarded By... | Action... |
|---|---|---|---|---|
| Set Filters | Make List | Button | — | — |
| Make List | Car List | Selection | — | — |
| Car List | Car Detail | Selection | — | — |
| Car Detail | Car List | Back | — | — |
| Car List | Make List | Back | — | — |
| Make List | Set Filters | Back | — | — |

FIGURE 7.2: Transition table for CAR application. Author's image.

The screen captures in Figure A.1 provide a side-by-side comparison of the AXIOM-generated and natively implemented screens. In most cases the screens are nearly identical save for stylistic issues such as font style and size.

#### 7.1.2.2 CVT

The CVT application allows a user to perform assorted unit conversions. The user first selects the kind of conversion to be performed. They then provide the value to be converted as well as the unit to which it will be converted. The final screen shows the results of the conversion. A finite state machine and transition table for the CVT application is shown in Figure 7.3.




| From State... | To State... | By Transition... | Guarded By... | Action... |
|---|---|---|---|---|
| From Unit | Input Value | Selection | — | — |
| Input Value | Results | Button | — | — |
| Input Value | Unit List | Button | — | — |
| Unit List | Input Value | Selection<br>Back | —<br>— | —<br>— |
| Results | Input Value | Back | — | — |
| Input Value | From Unit | Back | — | — |

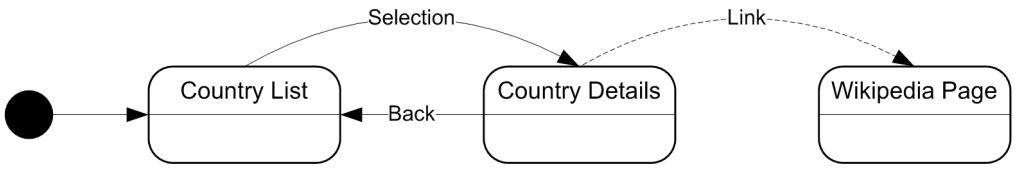
FIGURE 7.3: Transition table for CVT application. Author's image.

The screen captures in Figure A.2 provide a side-by-side comparison of the AXIOM-generated and natively implemented screens. As shown in the screen captures, style and layout are again somewhat different between the two applications. In the case of the

native Android application, a calculator widget is invoked when the user is prompted to enter their value, something that is not currently generated by the AXIOM transformation and translation rules.

#### 7.1.2.3 EUC

The EUC application displays data about European Union countries. The user first selects the country of interest. They are then shown the details of that country. They may choose to follow an optional link to a Wikipedia page about the selected country. The finite state machine and transition table for the EUC application is shown in Figure 7.3.



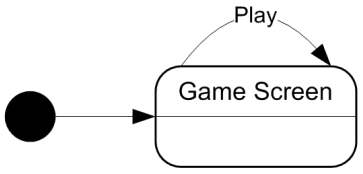

| From State... | To State... | By Transition... | Guarded By... | Action... |
|---|---|---|---|---|
| Country List | Country Details | Selection | — | — |
| Country Details | Wikipedia Page | Link | — | — |
| Country Details | Country List | Back | — | — |

FIGURE 7.4: Transition table for EUC application. Author's image.

The screen captures in Figure A.3 provide a side-by-side comparison of the AXIOM-generated and natively implemented screens. As shown in the screen captures, style and layout are again somewhat different between the two applications. This application is different from the others in that interaction with the generated application causes the built-in browser to be opened and thus for the application context to change.

#### 7.1.2.4 MAT

The MAT application is a simple memory game where the user attempts to locate and remember matching pairs of numbers. The finite state machine and transition table for the MAT application is shown in Figure 7.5.




| From State... | To State... | By Transition... | Guarded By... | Action... |
|---|---|---|---|---|
| Game Screen | Game Screen | Selection | — | — |

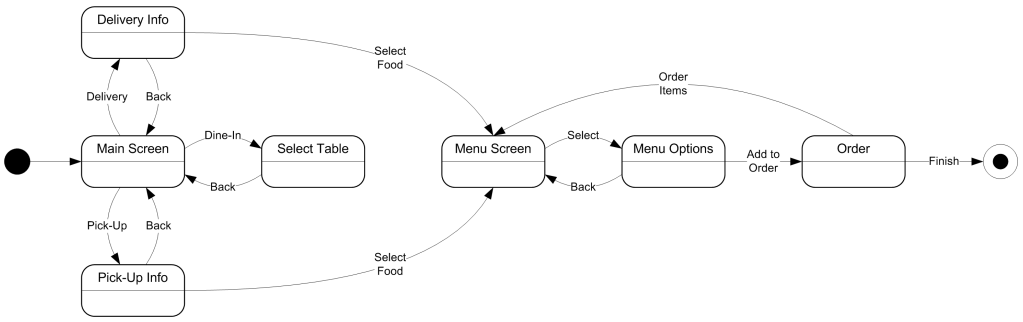
FIGURE 7.5: Transition table for MAT application. Author's image.

The screen captures in Figure A.4 provide a side-by-side comparison of the AXIOM-generated and natively implemented screens. As shown in the screen captures, style and layout are again somewhat different between the two applications.

#### 7.1.2.5 POS

The POS application is a simple point-of-sale system for ordering food from a restaurant. The user is offered the choice of dining in, carrying out, and delivery. Once they select their option they can either select a table, in the case of eating in, or browse the menu. For each item on the menu, the user can select their options and add it to their order. Once their order is complete, they can click the "Finish" button to place their order. The finite state machine and transition table for the POS application is shown in Figure 7.6.

The screen captures in Figure A.5 provide a side-by-side comparison of the AXIOM-generated and natively implemented screens. As shown in the screen captures, style and layout are again somewhat different between the two applications.




| From State... | To State... | By Transition... | Guarded By... | Action... |
|---|---|---|---|---|
| Main Screen | Select Table | Dine-In | — | — |
| | Delivery Info | Delivery | — | — |
| | Pick-Up Info | Pick-Up | — | — |
| Delivery Info | Main Screen | Back | — | — |
| | Menu Screen | Select Food | — | — |
| Pick-Up Info | Main Screen | Back | — | — |
| | Menu Screen | Select Food | — | — |
| Menu Screen | Menu Options | Select | — | — |
| Menu Options | Main Screen | Back | — | — |
| | Order | Add to Order | — | — |
| Order | Menu Screen | Order Items | — | — |
| | Finish | Finish | — | — |

FIGURE 7.6: Transition table for POS application. Author's image.

### 7.1.3 Analyses

Each set of experiments was subjected to different analyses. Both the small- and mid-scale experiments were analyzed quantitatively, as described by the metrics in Section 7.2. In addition, the mid-scale experiments were of sufficient scale to make it feasible to perform an additional qualitative analysis as described in Section 7.3.

## 7.2 Quantitative Analysis

Both sets of experiments defined metrics for *representational power* and *information density*.

### 7.2.1 Representational Power

Representational power measures how much code in one language is required to produce the same application in another language. This provides a rough indication of the relative effort expended by a developer to produce an application using different languages.

Our evaluation compared the source lines of code (SLOC) of the AXIOM Requirements models to the generated SLOC for both iOS and Android. For the comparative evaluation of the SLOC, we used CLOC [97] with Groovy as the source language for AXIOM. The Android and iOS platforms were accounted for using Java and Objective-C respectively. The SLOC counts do not include "non-essential" code such as comments or block delimiters such as braces.

While SLOC is 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 the inevitable 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 see relative differences in the sizes of the different representations of the applications.

Our analysis of SLOC assumes that developer productivity measured in *source lines-of-code per person-hour* (SLOC/PH) is roughly constant regardless of languages used. Research by Jiang [98] suggests that while language generation can significantly affect developer productivity, 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 our assumption to be reasonable.

Like Jiang, Kennedy [99] identifies language as a significant component of productivity. Kennedy's relative power metric, $\rho_L$, based on SLOC, measures the relative expressiveness of one language to another.

**Definition 7.1.** Kennedy's Relative Power Metric is given by:

$$\rho_{L/L_0} = \frac{I(M_{L_0})}{I(M_L)} \tag{7.1}$$

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

### 7.2.2 Information Density

Information density is a measure of a language's conciseness. Languages with high information densities have more compact representations. To evaluate comparative information densities, we created ZIP files using *gzip*, which is based on the DEFLATE algorithm, excluding all files that were not generated by AXIOM. We then compared the compression ratios, $CR_L$, derived using:

$$CR_L = \frac{\text{Uncompressed } I(M_L)}{\text{Compressed } I(M_L)} \tag{7.2}$$

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 $CR_L$. We then used the compression ratios to derive the *language density*.

**Definition 7.2.** Language Density, $\delta$, is defined as:

$$\delta_{L/L_0} = \frac{CR_{L_0}}{CR_L} \tag{7.3}$$

Language density is similar to Kennedy's relative power metric, but it relates 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 even though their SLOC measurements are the same.

## 7.3 Qualitative Analysis

For the qualitative analysis we used SonarQube[1] [100], an open-source, static code quality analysis tool that is popular with software developers. SonarQube uses a set of plugins to perform static source code analysis using common plugins such as FindBugs [101, 102].

Within SonarQube we used the *Android Lint* plugin to analyze the Android code. It provides a more specialized analysis of the code and supporting files than would the standard Java analysis, which was a reasonable second choice. For the iOS code analysis we used an open-source Objective-C plugin [103].

### 7.3.1 Source Code Organization

One way to begin our qualitative analysis is with source code organization. In other words, how is the source code allocated by file, function, and class. These metrics can be useful in discussing later topics such as complexity.

SonarQube uses a different algorithm for calculating the SLOC than CLOC does, which leads to discrepancies in the SLOCs when compared to the quantitative analysis. In particular SonarQube:

- Identifies a new SLOC by the presence of a carriage return.
- Ignores all comment lines.
- Treats trivial lines, such as those containing curly braces, as its own SLOC, something we eliminated for the quantitative analysis.

[1] SonarQube was formerly known as “Sonar”.

- Does not include the XML files in the analysis. This is a limitation of the plugin. Thus this part of the analysis is limited only to the Java code. Since much of the XML documents tends to be generated by an IDE rather than by hand in either approach, the elimination of those files' contributions to the SLOC is not significant.

Since we are referring to other SonarQube metrics during our analysis of the experimental results, we will use the SonarQube SLOC to ensure consistency.

### 7.3.2 Issues and SQALE

SonarQube performs its analysis by applying individual rules to the source code based on the language being analyzed. For example, one set of rules will apply to Java code while another set will apply to Objective-C code. In the cases of projects with multiple languages, SonarQube can analyze each language according to its own set of rules and then aggregate the results into a single quality report.

The rules are organized according to the SQALE Quality Model [104, 105], which is used to organize the non-functional requirements that relate to code quality and technical debt. This organization is defined by characteristics, sub-characteristics, and requirements. For our purposes, the eight SQALE "characteristics" are of primary interest:

1. Testability
2. Reliability
3. Changeability
4. Efficiency
5. Security
6. Maintainability
7. Portability
8. Reusability

Each rule is given a severity indicating how much of an impact a violation of the rule may have on the finished application. SonarQube identifies five severity levels. From least critical to most critical these are:

1. Info
2. Minor
3. Major
4. Critical
5. Blocker

To some extent these severities are arbitrary, but in general the more severe the issue or rule violation, the greater the chance that the code causing the issue contains a latent defect. The most common issue identified above, "Hardcoded Text", is a minor issue; it is undesirable, but will not stop the application from actually executing (although it does impact it's ability to be effectively internationalized). False positives and negatives are, of course, possible, which is why static analysis is often followed up with some kind of code inspection.

Given the number of issues identified for an application, and given the count of the application's lines of code, we can now calculate the *issue density*, which is just a more generic form of defect density:

**Definition 7.3.** Issue Density is defined as:

$$\text{Issue Density} = \frac{\text{Issue Count}}{\text{LOC}} \tag{7.4}$$

### 7.3.3 Code Duplication

Another quality measure is the amount of code duplication that occurs within the code base. Such duplication is usually the result of so-called "copy-paste" reuse, where a

useful block of code is copied to a new location. This is often the result of individual developers not being able or willing to refactor the code in such a way as to encourage DRY and SOLID [106–108] principles. In general the goal is to minimize the amount of redundant code found in a given code base. SonarQube uses PMD [109] to analyze the source code for duplicate code using an implementation of the Karp-Rabin [110] string matching algorithm.

### 7.3.4 Complexity

The final set of qualitative metrics that we analyze involve code complexity. SonarQube provides an analysis of cyclomatic complexity for the overall application as well as by function, class and source file. While arguably this complexity isn't as important for generated code, since all maintenance is intended to be done via the model rather than by changing the generated code directly, code that is more complex than necessary will tend to be less performant than equivalent, simpler code. The code generation templates required to build such structures will typically be more complicated than equivalent, simpler templates, which can make the maintenance and extension of the templates more difficult as well as being more likely to inject defects into the generated code.

# Chapter 8

# Evaluation Results

## 8.1 Quantitative Results

Each metric in the quantitative analysis 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.

Table 8.1 summarizes the results of our analysis of representational power and information density based on the median SLOC from the small- and mid-scale tests. To simplify the analysis, we treat all generated code for each platform as a single "language" even

TABLE 8.1: Median small-scale case metrics.

| Test Cases ($n = 104$) | of language $L_0$ of iOS | Android | AXIOM |
|---|---|---|---|
| Representational Power | | | |
| Source LOC | 171.50 | 106.00 | 15.00 |
| Relative Power | 11.43 | 7.07 | 1.00 |
| Information Density | | | |
| Compression Ratio | 12.01 | 16.71 | 1.88 |
| Language Density | 6.39 | 8.89 | 1.00 |

though the generated code may comprise several different languages. For example the Android "language" includes XML and Java while the iOS "language" includes XML and Objective-C.

### 8.1.1 Representational Power

The reduction in the size of the AXIOM Requirements models compared to the size of the generated code represents a significant reduction in development time and hence an increase in developer productivity. Since the median relative power of AXIOM is between 6.59 and 11.43 that of iOS and between 7.07 and 7.39 that of Android, we conclude that AXIOM is more representationally powerful than either iOS or Android. Similarly, the median compression ratio for AXIOM's models, 1.88 for the small-scale tests and 4.43 for the mid-scale tests, is significantly smaller than either iOS or Android, suggesting that AXIOM's DSL is more compact. The median iOS and Android language densities suggest that both languages may involve greater complexity and wordiness to represent the same model than AXIOM although the values seem to gradually converge as the applications get larger. We believe that at that scale, the percentage of the model that is concerned with the mobile-ness of the application is outweighed by the percentage of the application that is concerned with the business logic.

Table 8.2 shows a more detailed analysis of the individual representational powers for the mid-scale experiments. The preliminary results show that the AXIOM-generated code is often comparable to, if not smaller than, hand-written Android and iOS code. In the cases of the CVT and POS applications, there was significantly less hand-written code than generated code. These differences in SLOC result in similar differences in the languages' representational powers, as shown in figure 8.1 and information densities for those tests. In the case of the CVT application, AXIOM produced 8 views whereas the hand-written version used only 2. Similarly, for the POS application there were 9 hand-written Java files for the Android platform, but 13 for the AXIOM-generated code.

TABLE 8.2: Comparison of mid-scale experiment representational power metrics.

| Metric | Comparison of AXIOM to a Generated $L_0$ of | | Comparison of AXIOM to a Handwritten $L_0$ of | | |
|---|---|---|---|---|---|
| | iOS | Android | iOS | Android | AXIOM |
| Source LOC | | | | | |
| CAR | 435 | 488 | 594 | 889 | 66 |
| CVT | 1,384 | 1,125 | 431 | 538 | 365 |
| EUC | 726 | 506 | 559 | 913 | 46 |
| MAT | 293 | 311 | 193 | 245 | 64 |
| POS | 1,953 | 1,849 | 677 | 957 | 165 |
| Relative Power | | | | | |
| CAR | 6.59 | 7.39 | 9.58 | 14.34 | 1.00 |
| CVT | 3.79 | 3.08 | 1.18 | 1.47 | 1.00 |
| EUC | 15.78 | 11.00 | 12.15 | 19.85 | 1.00 |
| MAT | 4.58 | 4.86 | 3.02 | 3.83 | 1.00 |
| POS | 11.84 | 10.89 | 4.10 | 5.80 | 1.00 |

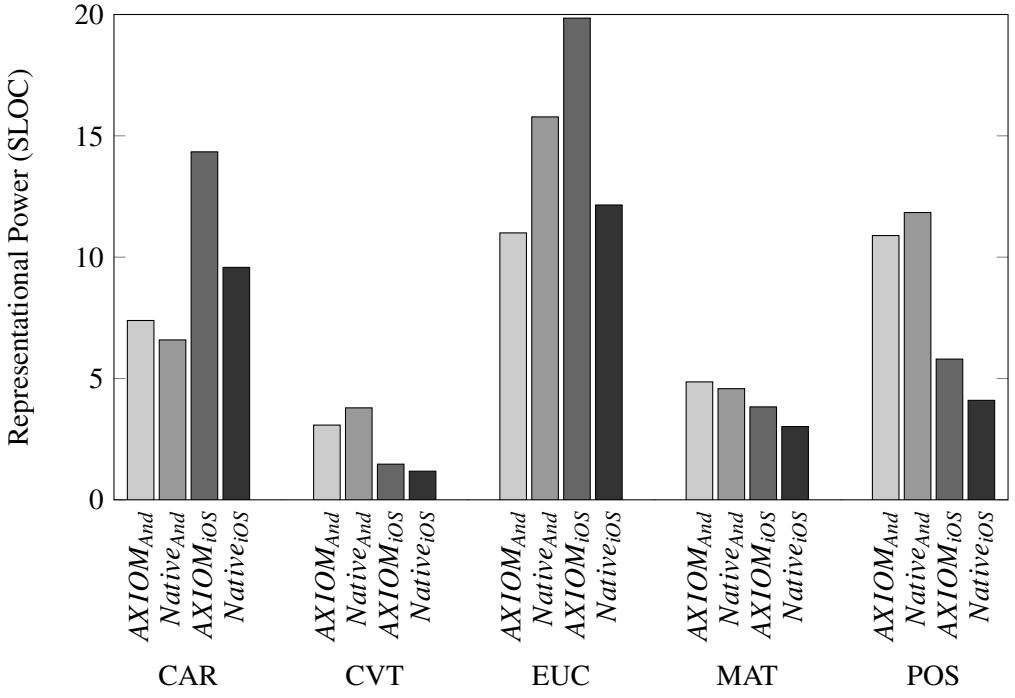



FIGURE 8.1: Comparison of mid-scale experiment relative power. Author's image.

Further refinement of the prototype could make the translation more parsimonious by reducing instances of duplicated code and incorporating more concise syntactic structures, thereby bringing the amount of generated code closer to that of the hand-written code. For example, there are ways within Groovy to use closures to shrink the models, but the AXIOM prototype does not currently use such structures in its translation, which can lead to bloated code. Further syntactic refinements are discussed in chapter 9.3.

### 8.1.2 Information Density

The compression ratios and language densities for the mid-scale experiments shown in Table 8.3 and Figure 8.2 suggest that as the size of the application increases, AXIOM becomes more and more comparable to the native languages in terms of its ability to succinctly represent the models. For example, the median language density for the small-scale Android experiments was 8.89 but for the mid-scale experiments it decreased to 1.83 for the generated code and 2.20 for the hand-written code. The iOS platform experiences a similar reduction. These reductions are not surprising given that AXIOM uses Groovy as its core syntax. Because Groovy is in the same language family as both Objective-C and Java, we expect that the core characteristics of its compression ratio and information density will be similar.

TABLE 8.3: Comparison of mid-scale experiment information density metrics.

| Metric | Comparison of AXIOM to a Generated $L_0$ of | | Comparison of AXIOM to a Handwritten $L_0$ of | | |
|---|---|---|---|---|---|
| | iOS | Android | iOS | Android | AXIOM |
| Compression Ratio | | | | | |
| CAR | 9.18 | 10.82 | 9.53 | 10.59 | 4.82 |
| CVT | 6.87 | 7.54 | 9.95 | 12.87 | 7.33 |
| EUC | 9.69 | 9.03 | 10.72 | 11.47 | 4.94 |
| MAT | 10.30 | 16.27 | 10.96 | 15.83 | 4.43 |
| POS | 7.35 | 8.97 | 10.17 | 10.71 | 5.93 |
| Language Density | | | | | |
| CAR | 1.90 | 2.24 | 1.98 | 2.20 | 1.00 |
| CVT | 0.94 | 1.03 | 1.36 | 1.76 | 1.00 |
| EUC | 1.96 | 1.83 | 2.17 | 2.32 | 1.00 |
| MAT | 2.33 | 3.67 | 2.47 | 3.57 | 1.00 |
| POS | 1.24 | 1.51 | 1.72 | 1.81 | 1.00 |

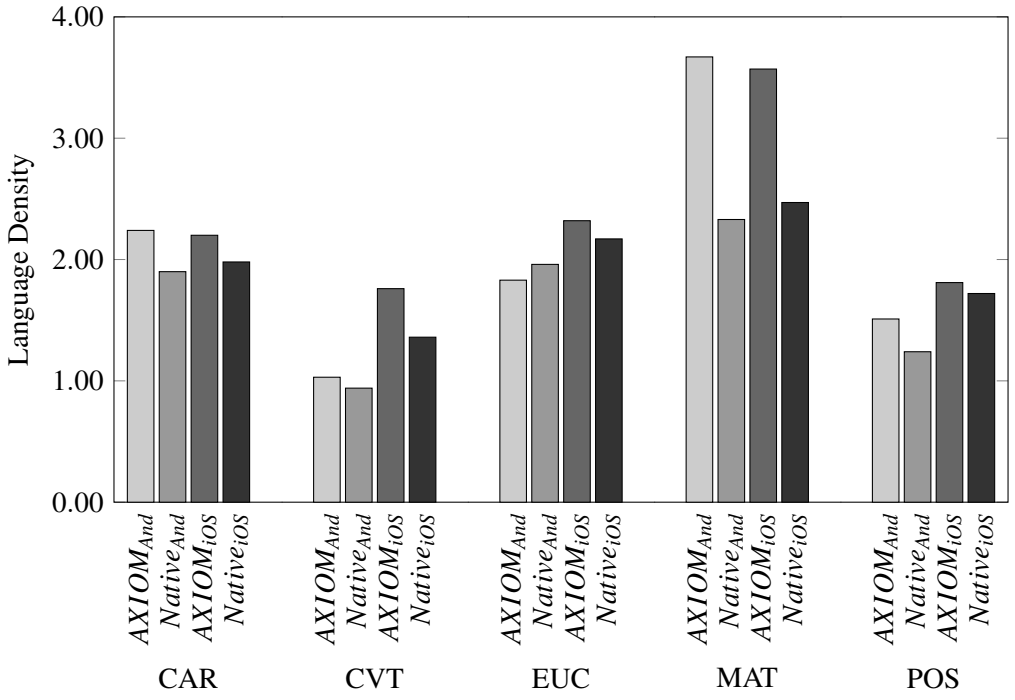



FIGURE 8.2: Comparison of mid-scale experiment language density. Author's image.

This observation suggests that there are some key factors that need to be considered when designing a DSML:

1. There is a certain minimum amount of code that is needed to "bootstrap" even the simplest programs and that base code is comparatively complex. This is why even on the small-scale experiments we see a significant improvement in information density. As more and more application logic is added, that code begins to erode the information density improvements since the syntactic representations between the different languages tends to converge. This in turn has a direct impact on developer productivity, which we discuss further in chapter 9.1.

2. The truly significant benefits of the adoption of this kind of DSML will be reaped only when the DSML is capable of concisely representing the domain-specific concepts of the business in addition to the domain-specific concepts of mobile applications. That is, while there is certainly benefit in allowing a mobile application to be concisely modeled, the current AXIOM language emphasize the modeling of the UI and interaction elements while leaving the implementation of the application logic at a comparatively low level of detail. The net effect is

that while there are productivity benefits to be seen, they are not as great as they might be if a separate DSML was constructed to handle the application logic more effectively.

## 8.2 Qualitative Results

For the qualitative analysis we used SonarQube[1] [100], an open-source, static code quality analysis tool that is popular with software developers. SonarQube uses a set of plugins to perform static source code analysis using common plugins such as FindBugs [101, 102].

Within SonarQube we used the *Android Lint* plugin to analyze the Android code. It provides a more specialized analysis of the code and supporting files than would the standard Java analysis, which was a reasonable second choice. For the iOS code analysis we used an open-source Objective-C plugin [103].

To keep the analysis meaningful, we eliminate from consideration many of the ancillary files produced either during code generation or by the developers who produced the hand-written code. These files include many XML files that are important to the overall ability for the application to be executed on the target platform, but which are often generated by the various IDEs and supporting tools. We choose instead to focus only on the Java or Objective-C code, which is where the most qualitative issues will be found and where developer productivity will most likely be impacted by AXIOM's code generation capabilities.

### 8.2.1 Source Code Organization

Table 8.4 shows the basic organization of the source code in the various experiments. In most cases the AXIOM prototype generates a number of artifacts that is comparable to that produced by the developers with the hand-written code. In some cases the number of artifacts can be significantly fewer. We look at the qualities of these generated artifacts in later sections.

[1] SonarQube was formerly known as "Sonar".

In many cases the SLOC of the AXIOM-generated and hand-written code using native tools are almost the same. However, as shown by Figure 8.3, in two of the experiments there is a wide divergence: CVT and POS. In both cases, the basic difference was in the approach taken to model the code. For example, in the case of the CVT application, both the AXIOM model and hand-written code used a series of "if" statements to perform the conversions. However, in the case of the AXIOM model, these calculations spanned multiple views – one-per-unit-type (length, power, volume, etc.). This lead to a significant amount of redundant code, which is born out when we look at the SonqrQube analysis of code duplication in chapter 8.2.3. Some efforts were made to optimize the model further, but this exposed some architectural limitations in the prototype.

In the case of the POS application, a similar approach was taken, where multiple redundant views were generated where it was not actually necessary. In the case of the hand-written code, the developers took an approach that allowed the application to take better advantage of its data-driven nature.

TABLE 8.4: Comparison of source code organization for mid-scale experiments.

| Experiment | OS | Style | SLOC | Files | Funcs. | Classes | Stmts. |
|---|---|---|---|---|---|---|---|
| CAR | Android | AXIOM | 253 | 3 | 9 | 6 | 121 |
| | | Native | 379 | 7 | 21 | 8 | 171 |
| | iOS | AXIOM | 372 | 9 | 23 | 7 | 232 |
| | | Native | 349 | 13 | 31 | 9 | 199 |
| CVT | Android | AXIOM | 1,015 | 8 | 59 | 9 | 533 |
| | | Native | 326 | 2 | 9 | 2 | 231 |
| | iOS | AXIOM | 1,452 | 21 | 115 | 19 | 816 |
| | | Native | 403 | 7 | 32 | 5 | 255 |
| EUC | Android | AXIOM | 254 | 4 | 11 | 8 | 106 |
| | | Native | 260 | 5 | 16 | 5 | 123 |
| | iOS | AXIOM | 666 | 11 | 27 | 9 | 467 |
| | | Native | 205 | 11 | 20 | 7 | 114 |
| MAT | Android | AXIOM | 178 | 1 | 28 | 1 | 75 |
| | | Native | 99 | 2 | 6 | 2 | 41 |
| | iOS | AXIOM | 221 | 5 | 10 | 3 | 143 |
| | | Native | 137 | 7 | 15 | 4 | 66 |
| POS | Android | AXIOM | 999 | 13 | 88 | 25 | 405 |
| | | Native | 448 | 9 | 28 | 9 | 177 |
| | iOS | AXIOM | 2,037 | 31 | 152 | 29 | 1,289 |
| | | Native | 655 | 29 | 79 | 21 | 299 |

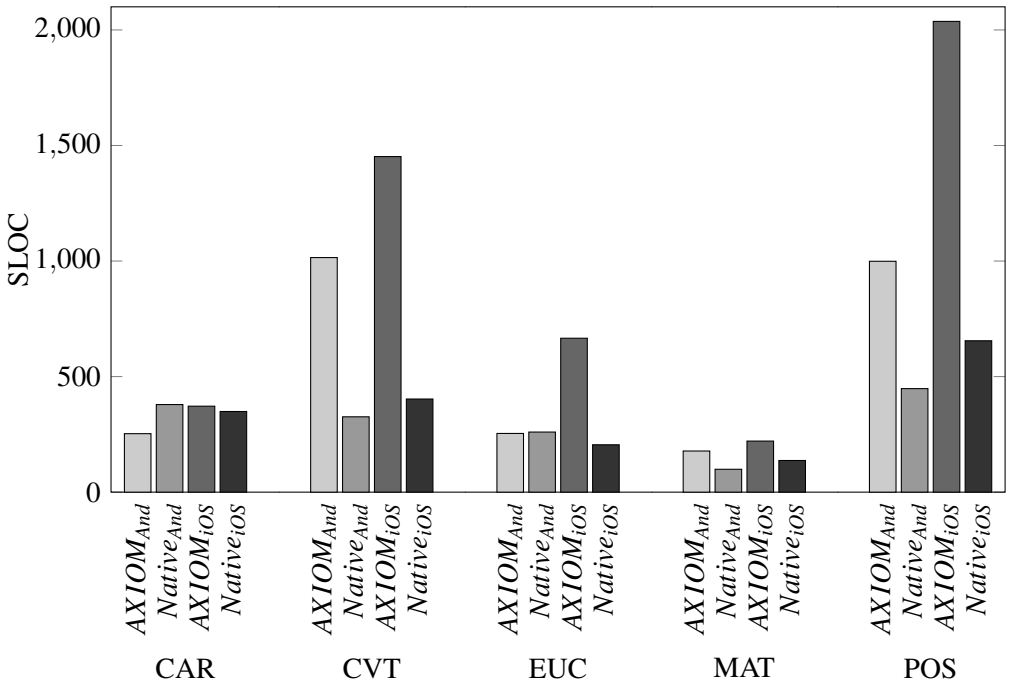



FIGURE 8.3: SLOC comparison for mid-scale experiments. Author's image.

### 8.2.2 Issue Density

As described in chapter 7.3.2, SonarQube defines five different severity levels. However, for this analysis we condense the number of issue severities to only two categories: major and minor. Major issues are those that are categorized as "major", "critical", and "blocker" by SonarQube. Minor issues are those that SonarQube classifies as "minor" and "info". OCLint uses three issues severities: "major", "minor", and "info".[2] For this analysis a major issue includes those that OCLint defines as "major" while a minor issue includes those OCLint defines as "minor" or "info." Table 8.5 shows these mappings.

In most cases, the AXIOM-generated code does not fare as well as its hand-written counterpart. When comparing the generated and hand-written code by application for each platform, we find that at best the AXIOM-generated code has an issue density that is 138% that of the hand-written code while at worst it is about 550% that of the hand-written code. This is consistent with the fact that AXIOM is template-based since any issues present in the template, or in the algorithm that translates it, will be injected into the generated code. This is a side-effect of all template-based approaches. However,

[2]There is a defect in the OCLint [111] plugin that is used to ingest the results of the Objective-C analysis into SonarQube so the analysis was completed manually.

TABLE 8.5: Mapping of SonarQube to OCLint issue severities.

| Analysis Severity | SonarQube Severity | OCLint Severity |
|---|---|---|
| Major | Blocker, Critical, Major | Major |
| Minor | Minor, Info | Minor, Info |

as we will discuss in chapter 9.2, this side-effect can also be harnessed to improve the overall code quality of any application that uses the template.

Table 8.6 shows the distribution of the issues found during an analysis of the Android implementations of the CAR application. As we might expect, the AXIOM code contains more instances of specific issues, such as "unused ID" or "hardcoded text" because its templates and the overall code generation process inject these issues.

TABLE 8.6: Distribution of issues for the Android CAR implementation.

| Rule | AXIOM | Handwritten |
|---|---|---|
| Hardcoded Text | 31 | 11 |
| Unused id | 14 | 4 |
| Target SDK attribute is not targeting latest version | 0 | 1 |
| Overdraw: Painting regions more than once | 0 | 1 |
| The launcher icon shape should use a distinct silhouette | 3 | 0 |
| Dynamic text should probably be selectable | 2 | 0 |
| Missing allowBackup attribute | 1 | 0 |
| Image without contentDescription | 1 | 1 |
| Icon densities validation | 1 | 0 |
| Nested Layout Weights | 1 | 0 |
| ScrollView size validation | 1 | 1 |
| Unused resources | 1 | 2 |
| Total Issues | 56 | 21 |

Table 8.7 describes the static code analysis issues identified by SonarQube, in the case of Android, and OCLint, in the case of iOS, for each of the mid-scale tests. Figure 8.4 shows the calculated issue densities from Table 8.7.

TABLE 8.7: Comparison of mid-scale experiment issue densities.

| Experiment | OS | Style | SLOC from Table 8.2 | Issue Count Major | Issue Count Minor | Issue Density |
|---|---|---|---|---|---|---|
| CAR | Android | AXIOM | 253 | 0 | 56 | 0.11 |
| | | Native | 889 | 0 | 21 | 0.02 |
| | iOS | AXIOM | 349 | 0 | 61 | 0.14 |
| | | Native | 594 | 0 | 32 | 0.05 |
| CVT | Android | AXIOM | 1,125 | 0 | 79 | 0.07 |
| | | Native | 538 | 0 | 19 | 0.04 |
| | iOS | AXIOM | 1,384 | 0 | 190 | 0.14 |
| | | Native | 431 | 0 | 29 | 0.07 |
| EUC | Android | AXIOM | 506 | 0 | 80 | 0.16 |
| | | Native | 913 | 0 | 79 | 0.09 |
| | iOS | AXIOM | 726 | 0 | 196 | 0.27 |
| | | Native | 559 | 0 | 25 | 0.04 |
| MAT | Android | AXIOM | 311 | 0 | 40 | 0.13 |
| | | Native | 245 | 0 | 9 | 0.04 |
| | iOS | AXIOM | 293 | 0 | 53 | 0.18 |
| | | Native | 193 | 0 | 25 | 0.13 |
| POS | Android | AXIOM | 1,849 | 0 | 258 | 0.14 |
| | | Native | 957 | 0 | 46 | 0.05 |
| | iOS | AXIOM | 1,953 | 0 | 292 | 0.15 |
| | | Native | 677 | 0 | 33 | 0.05 |

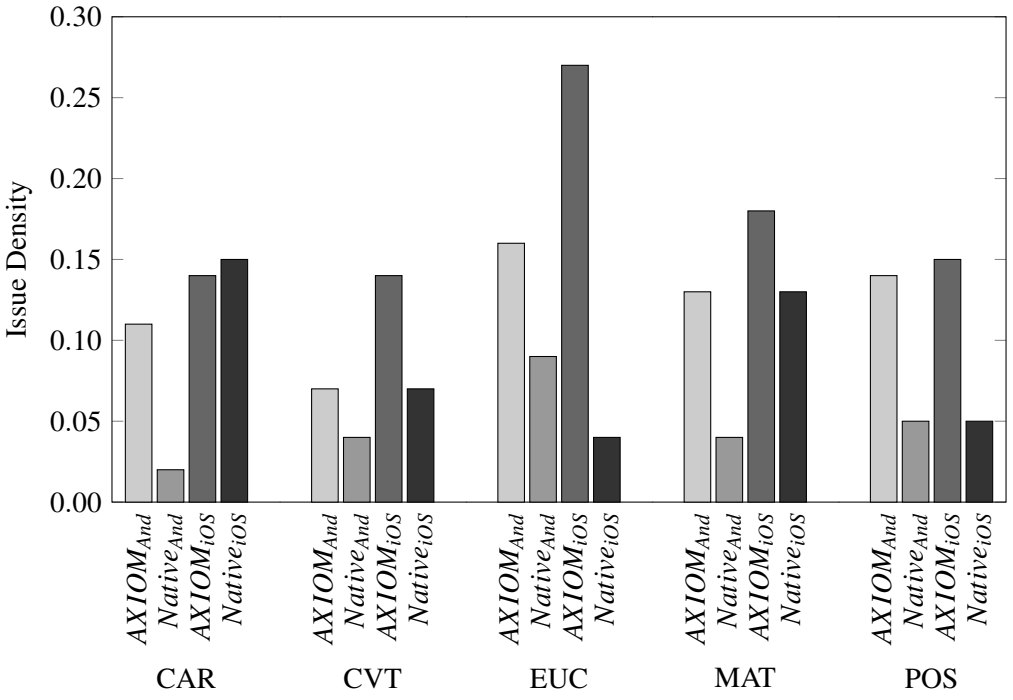



FIGURE 8.4: Issue densities for mid-scale experiments. Author's image.

### 8.2.3 Code Duplication

As shown in Table 8.8 and Figure 8.5, there is little duplication although the generated code contains more duplication than does the hand-written code. This is partially an artifact of the prototype tool rather than something inherent to the approach. For example, the prototype sometimes naively generates redundant views for the display of similar information rather than providing a single, generic view. Furthermore, each of these views will produce code from the same template, thus resulting in significant duplication. This duplication is also partially an artifact of the modelers ability to translate the applications requirements into a simple, unambiguous requirements model. Finally, these results also reflect some limitations in the AXIOM DSL itself which, if corrected, would reduce the amount of code produced during the translation stage. These limitations and some of the approaches that can be taken to address them are discussed in chapter 9.

AXIOM's generated code often contains slightly more duplicated code than the hand-written code. This is a side-effect of the template-based code generation and can be exacerbated by a poorly implemented model. In the case of the CAR application, for example, the application model was written in such a way as to cause redundant views to be generated. However, in the EUC application, an application that is even more data intensive than the CAR application, there is no code duplication at all, because the model was written much more efficiently.

Further investigation suggests that the prototype's code generation algorithm is still relatively naive and assumes that the modeler has defined the model in an optimum way. Historically such assumptions have generally proven incorrect, leading to enhancements such as code optimizers.

Because such unnecessary code duplication can be significant source of application bloat, this suggests that the preprocessing and normalization steps defined by the approach need to be augmented to include additional optimization steps to minimize the model before the more complex transformation and translation stages begin.

TABLE 8.8: Comparison of code duplication for mid-scale experiments.

| Experiment | OS | Style | Duplication | | | |
|---|---|---|---|---|---|---|
| | | | Total % | Lines | Blocks | Files |
| CAR | Android | AXIOM | 5.4 | 18 | 2 | 1 |
| | | Native | 0 | 0 | 0 | 0 |
| | iOS | AXIOM | 0 | 0 | 0 | 0 |
| | | Native | 0 | 0 | 0 | 0 |
| CVT | Android | AXIOM | 21.1 | 265 | 41 | 7 |
| | | Native | 27.4 | 110 | 8 | 1 |
| | iOS | AXIOM | 21.5 | 483 | 7 | 7 |
| | | Native | 0 | 0 | 0 | 0 |
| EUC | Android | AXIOM | 0 | 0 | 0 | 0 |
| | | Native | 0 | 0 | 0 | 0 |
| | iOS | AXIOM | 0 | 0 | 0 | 0 |
| | | Native | 0 | 0 | 0 | 0 |
| MAT | Android | AXIOM | 0 | 0 | 0 | 0 |
| | | Native | 0 | 0 | 0 | 0 |
| | iOS | AXIOM | 0 | 0 | 0 | 0 |
| | | Native | 0 | 0 | 0 | 0 |
| POS | Android | AXIOM | 6.1 | 84 | 5 | 3 |
| | | Native | 0 | 0 | 0 | 0 |
| | iOS | AXIOM | 16.5 | 541 | 16 | 9 |
| | | Native | 2.3 | 32 | 2 | 2 |

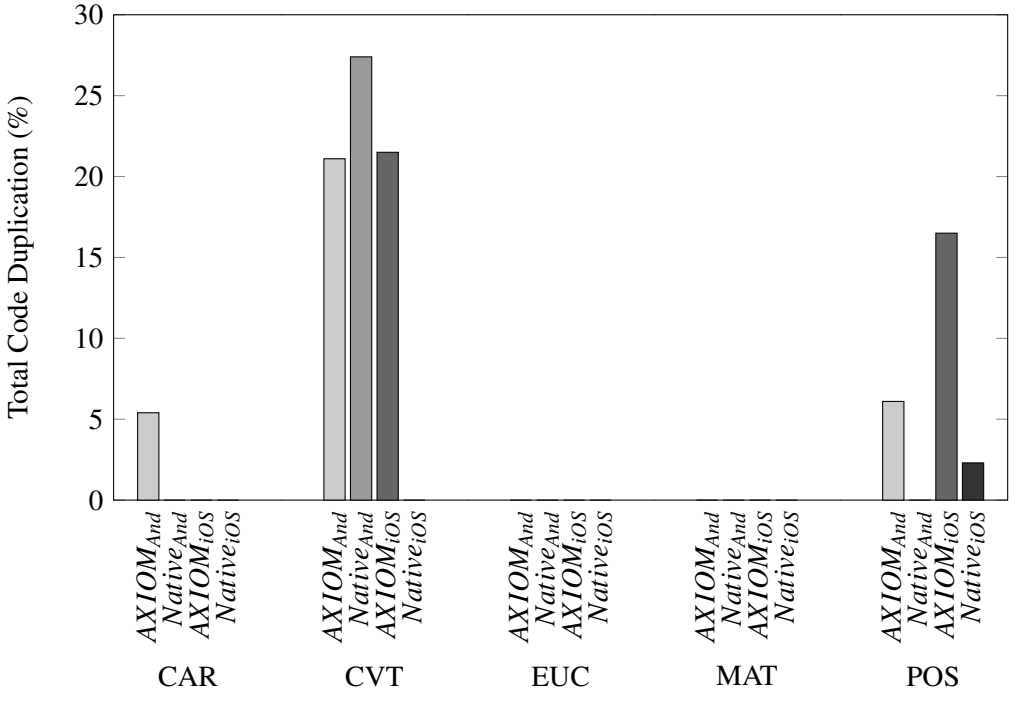



FIGURE 8.5: Complexity comparison for mid-scale experiments. Author's image.

### 8.2.4 Complexity

Table 8.9 and corresponding Figure 8.6 show that, in most cases, the AXIOM-generated code has a much higher cyclomatic complexity when viewed at the application level. However, when viewed at the function, class and file levels, the complexity is often less then the complexity of the hand-written code. Since the AXIOM prototype tends to generate more code than human developers, it stands to reason that the overall application complexity would be consistently higher. Also, as has been mentioned before, any complexity present in the template from which the final, translated code is generated will be passed into the generated code. That also means that these complexity values can be improved by enhancing the translation templates and transformation rules to be more parsimonious with the generated code.

TABLE 8.9: Comparison of complexity for mid-scale experiments.

| Experiment | OS | Style | Cyclomatic Complexity | | | |
|---|---|---|---|---|---|---|
| | | | Overall | Function | Class | File |
| CAR | Android | AXIOM | 26 | 2.9 | 4.3 | 8.7 |
| | | Native | 53 | 2.5 | 6.6 | 7.6 |
| | iOS | AXIOM | 41 | 1.8 | 5.6 | 4.6 |
| | | Native | 63 | 2.0 | 6.8 | 4.8 |
| CVT | Android | AXIOM | 198 | 3.4 | 22.0 | 24.8 |
| | | Native | 91 | 10.1 | 45.5 | 45.5 |
| | iOS | AXIOM | 215 | 1.9 | 11.2 | 10.2 |
| | | Native | 115 | 3.6 | 22.6 | 16.4 |
| EUC | Android | AXIOM | 23 | 2.1 | 2.9 | 5.8 |
| | | Native | 33 | 2.1 | 6.6 | 6.6 |
| | iOS | AXIOM | 92 | 3.4 | 10.0 | 8.4 |
| | | Native | 34 | 1.7 | 4.6 | 3.1 |
| MAT | Android | AXIOM | 33 | 1.2 | 33.0 | 33.0 |
| | | Native | 11 | 1.8 | 5.5 | 5.5 |
| | iOS | AXIOM | 16 | 1.6 | 4.7 | 3.2 |
| | | Native | 24 | 1.6 | 5.5 | 3.4 |
| POS | Android | AXIOM | 148 | 1.7 | 5.9 | 11.4 |
| | | Native | 53 | 1.9 | 5.9 | 5.9 |
| | iOS | AXIOM | 279 | 1.8 | 9.6 | 9.0 |
| | | Native | 115 | 1.5 | 5.4 | 4.0 |

It is also important to note that because AXIOM is model-driven and the models operate at a higher level of abstraction than does the hand-written code, the complexity and its impact on software maintenance, should be less of a factor for the AXIOM approach than with the hand-written approach. This obviously does not change the impact that the additional complexity might have on other important application factors like runtime performance or application size.

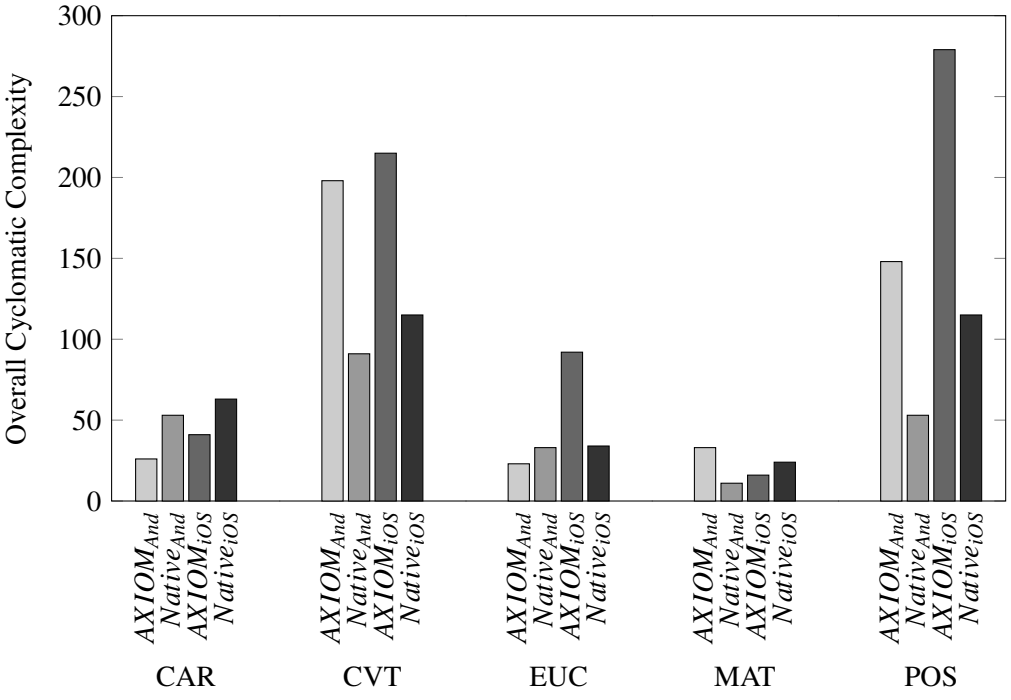



FIGURE 8.6: Complexity comparison for mid-scale experiments. Author's image.

# Chapter 9

# Discussion

This research attempts to answer two main questions about model-driven development for mobile platforms using the AXIOM approach:

1. Is developer productivity with AXIOM significantly different when compared to writing native mobile applications?
2. Does the code created using AXIOM exhibit similar quality to equivalent hand-written code?

## 9.1 AXIOM's Impact on Developer Productivity

Developer productivity can be influenced by many factors including team size, development language, techniques used, and development methodology. Based on research done by Jiang [98] and others [112–114], the two most significant factors in overall productivity are the average team size and development language, accounting for approximately 25% of the variability of the Normalized Productivity Delivery Rate (PDR)[1] Therefore in evaluating AXIOM's ability to deliver on its productivity goals, we focus on these two points.[2]

[1] PDR is defined as the normalized work effort, essentially the hours spent on the project, divided by adjusted function points, which is the functional size of the project in points multiplied by an adjustment factor.

[2] Jiang's research did not find development methodology to be a significant factor in productivity. However, at that time the Agile movement had not yet gained the momentum it has today.

We did not establish the function points of the various small- and mid-scale applications. They were relatively limited in scope and did not exhibit the complexity of projects that typically warrant function-point analysis. The lack of such an analysis does not affect our assessment since for each application the overall complexity would be the same regardless of the implementation technique (AXIOM, or native hand-written code). This means that any productivity improvements would necessarily result from changes in the work effort rather than because of changes in the scope of that effort.

Jiang's model for software productivity is given by equation 9.1[3]:

$$\begin{aligned} ln(PDR) \quad = \quad & 0.357 * log(TeamSize) \\ & - 0.463 * I(3GL) \\ & - 1.049 * I(4GL) \\ & - 1.021 * I(ApG) \\ & - 0.138 * I(MR) \\ & - 0.219 * I(Multi) \\ & - 0.269 * I(PC) \\ & - 0.403 * I(OO) \\ & - 0.447 * I(Event) \\ & + 0.821 * I(OO : Event) \\ & - 0.276 * I(Business) \\ & - 0.024 * I(Regression) \\ & + 1.015 * I(Business : Regression) \\ & + 2.651 \end{aligned} \tag{9.1}$$

This model was constructed by analyzing the project database of the International Software Benchmarking Standards Group (ISBSG) [115]. This database contains metrics and descriptive information about each of its over 6,700 development and enhancement projects[4]. These projects include 100 types of applications across 30 industry verticals spanning 26 countries. This project database has been used as the basis for other analyses of software productivity such as those done by Liu [116], Jeffery [117], and Lokan [118]. This breadth of projects makes Jiang's model well-suited for our analysis.

[3] Jiang's original equation refers to $log(PDR)$ rather than $ln(PDR)$, but the text of the paper refers to $ln$.
[4] Jiang's work was based on release 10 of the ISBSG database, which held data on only 4,100 projects.

Jiang's model captures several different properties such as team size, the type of language (3GL, 4GL, Application Generator), the platform (mid-range, multi-platform, or PC), development techniques used (OOAD, event modeling, regression testing, or business area modeling). Factors separated by a colon are applied if both of the techniques were used. With the exception of team size, each factor, $I(X)$, takes a boolean value: 1 if $X$ was used and 0 if it was not.

In our experiments, many things were held constant. We have already discussed application complexity. Similarly, the team size was always 1. We did not use an application generator[5] and thus disregard that factor. Furthermore, we did not use any form of event or business modeling or formal regression testing. Finally, we consider all applications as being for the "PC" platform, a decision we explain in the subsequent analyses below. These simplifications reduce the equation to that shown in equation 9.2:

$$\begin{aligned} ln(PDR) &= -0.463 * I(3GL) \\ &\quad - 1.049 * I(4GL) \\ &\quad - 0.269 * I(PC) \\ &\quad - 0.403 * I(OO) \\ &\quad + 2.651 \end{aligned} \tag{9.2}$$

In equation 9.2, we emphasize that there is one dominant and different factor remaining: the type of language being used to develop the application. For the handwritten, native applications, the $ln(PDR)_{Native}$ is given by equation 9.3:

$$\begin{aligned} ln(PDR)_{Native} &= -0.463 * I(1) && \text{Java or Objective-C} \\ &\quad - 1.049 * I(0) && \text{No 4GL} \\ &\quad - 0.269 * I(1) && \text{PC Platform} \\ &\quad - 0.403 * I(1) && \text{Object-Oriented} \\ &\quad + 2.651 \\ ln(PDR)_{Native} &= 1.516 \\ PDR_{Native} &= 4.554 \end{aligned} \tag{9.3}$$

[5] We treat AXIOM as a 4GL rather than an application generator. While it does indeed generate code, it requires up-front development in the DSML MADL first.

For native development we consider both Java and Objective-C as 3GL languages. Because each of those languages is only used for a single platform, we consider both Android and iOS to both be so-called "PC" platforms.

If we apply similar reasoning to the AXIOM applications, we derive the value of $ln(PDR)_{AXIOM}$ shown in equation 9.4:

$$\begin{aligned} ln(PDR)_{AXIOM} &= -0.463 * I(0) && \text{No Java or Objective-C} \\ &\quad -1.049 * I(1) && \text{AXIOM is 4GL} \\ &\quad -0.269 * I(1) && \text{PC Platform} \\ &\quad -0.403 * I(1) && \text{Object-Oriented} \\ &\quad +2.651 \\ ln(PDR)_{AXIOM} &= 0.093 \\ PDR_{AXIOM} &= 1.097 \end{aligned} \tag{9.4}$$

We thus find that $PDR_{AXIOM}$ is slightly more than 4-times greater than $PDR_{Native}$. However, as shown by Equation 9.1, there are many factors that contribute to development productivity in Jiang's model. What assurances do we have that AXIOM is the single-most important contributing factor in our productivity analysis?

According to Jiang and others, the average team size explains 17.3% of the variance in $ln(PDR)$. Development language explains another 7.8% of the variance when the languages are of different generations, that is 3GL vs. 4GL. These two factors explain a total of 25.1% of the overall $ln(PDR)$ variance. However, the fact that the team size was held constant, as were the other key factors, suggests that the type of language, native code or AXIOM, can account for almost all 100% of the variability in $ln(PDR)$ in our mid-scale analyses. We thus find that $PDR_{AXIOM}$ is approximately four times greater than $PDR_{Native}$, suggesting that AXIOM can provide significant increases in productivity compared to development using existing tools for native platforms.

But does the actual, measurable output of the mid-scale experiments agree with the expected result of Jiang's model? If we compare the expected $PDR_{AXIOM}$ to the representational power metrics from Table 8.2, shown in Figure 9.1, we find significant variance. For the five mid-scale experiments the median representational power is 4.10 for iOS and 5.80 for Android, which is in line with Jiang's model's predictions. One

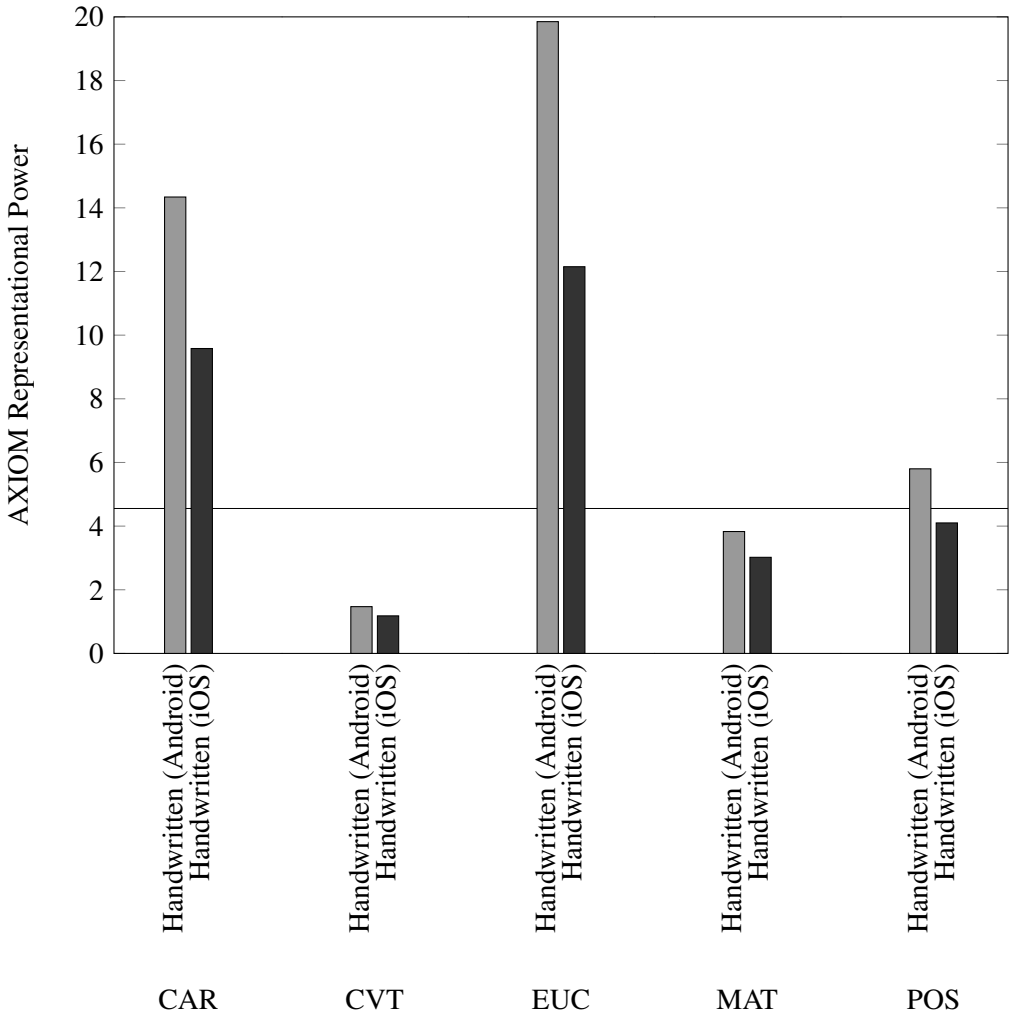



FIGURE 9.1: Representational power of AXIOM compared to handwritten code. Author's image.

of the experiments, CVT, required an amount of AXIOM code that was only slightly less than the amount of hand-written code needed to implement the application. In that particular example, the AXIOM model was fairly naive, resulting in code bloat.[6]

If we consider the impact that AXIOM has on applications involving a team size of more than one developer, then we can infer that the productivity benefits still persist. Based on the small- and mid-scale experiments, AXIOM has generally been observed to reduce the number of source lines of code to be written when compared to the equivalent native application. We can thus deduce that fewer developers would be required to

[6] A more efficient model was devised, but the AXIOM prototype was unable to process the semantically more complex model.

construct any given application using AXIOM and thus that the productivity benefits would remain.

AXIOM can also improve developer productivity because it emphasizes up-front modeling and because the transformation rules and templates can be changed and reused. 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. 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 and without being required to start *ex nihilo*.

This quantitative analysis suggests that AXIOM significantly affects developer productivity, owing to its cross-platform nature and DSML, which enables a more concise representation of an application than the native code can. This is reflected in the comparative information density values from Table 8.3 as well. However, some cautions are in order.

First, it is unclear if Jiang's model scales effectively to smaller applications such as those in our mid-scale experiments. Second, Jiang's model does not explicitly deal with mobile applications. We have identified each platform as though it was equivalent to general PC development, but it is uncertain if this is the case. Nevertheless, we expect that the coefficients associated with the platform, while possibly different, would contribute equally in both cases and thus not have a material impact on the overall productivity difference described by Jiang's model. Third, the coefficients in the model may be different today than in 2007 when the model was developed. Advances in tools, such as IDEs, and techniques, such as agile development, may have affected the impact of those factors on the model, causing the variation associated with the development language to be less important relative to other factors.

There are other reasons why the actual developer productivity might be different from that suggested by Jiang's model. First, as is the case with all such productivity assessments, there exist significant differences between the capabilities of individual developers and it is extremely difficult to control for them. In this evaluation, we attempted

to control for mobile development experience on the various platforms, but did not further refine this admittedly coarse-grained selection process. Second, the DSL was in somewhat a state of flux during the time that the mid-scale evaluations were being conducted. This meant that the developers who produced the AXIOM models might have encountered problems that caused them to produce less than optimal code. Similarly as new features of the language were introduced, the developers may not have gone back and incorporated them into their already complete code.

We have found evidence of this in at least two of the mid-scale experiments. For example, in an earlier version of the CAR application, the MADL file was significantly longer because the modeler provided a naive implementation of the application. Based on a Sonar analysis, the resulting source code was approximately 1,300 lines long. After reviewing the model and applying classic software engineering techniques such as refactoring, the model and generated code came down to their present levels as shown in Table 8.4. A similar problem was discovered with the EUC applications with even more dramatic results.

This suggests that one of the challenges with this kind of approach to model-driven engineering is striking a balance between model flexibility and model simplicity. By allowing the full syntax of the Groovy language to be available within the model, we increase the burden on the modeler to apply intelligent software development best practices in order to prevent the final code from being too bloated. While AXIOM performs some initial pre-processing of the model to eliminate as many sources of inefficiency as possible, it cannot realistically compensate for a completely inefficient model while also guaranteeing that the model's semantics will be preserved after that optimization. This challenge is the same as that of many compiler optimizers.

Nevertheless, there are several way in which AXIOM's DSL could be improved to further enhance developer productivity such as increased abstraction and implementation patterns. These same techniques will also provide significantly more power to the modeling notation and will thus enable modelers to avoid the use of Groovy syntax for common modeling constructs.

### 9.1.1 Increased Abstraction

As described in Section 2.1.2, AXIOM is platform-independent, but not particularly abstract so raising its level of abstraction is one important way in which AXIOM could be extended. As always, the goal is for AXIOM to allow modelers to focus more on the structure and behavior of the application and less about defining individual views and transitions.

```
1 ListView(id: CountryListView, title: 'European Union') {
2   EUMemberCountries.each { c ->
3     Item(text: c.name,
4          detailText: c.capital,
5          image: "${c.key}_flag.gif",
6          next: [to:CountryDetailView, data:c])
7   }
8 }
```

LISTING 9.1: ListView via iteration.

One promising area of improvement is in the brevity of AXIOM's syntax. AXIOM exhibits many of its Groovy roots, and much of the code uses native Groovy syntax. For example, the code snippet in Listing 9.1 generates a `ListView` of countries.

This code is comparatively low level and continues a time-honored approach for dynamically constructing visual elements from a data store by essential binding key properties of a row of data to the corresponding properties in the graphical widget. The code could be much more succinct, while being equally expressive, if we had a more abstract notation that took advantage of a `ListView`'s most common properties:

1. They tend to be bound to a particular data type, in this case, a Country.
2. They often show only a small subset of information, such as an icon, a title, or both.
3. They usually lead to a more complete, detailed representation when they're clicked.
4. The item that was clicked must be provided to the detail page so that it can retrieve the additional information for further display.

If we consider such commonalities, then we might further refine AXIOM to allow for the construction of a `ListView` like that of Listing 9.2.

```
1 ListView(id: CountryList,
2          title: 'European Union',
3          itemType: Country,
4          itemTextField: "capital",
5          itemIconField: "flag",
6          next: CountryDetail)
```

LISTING 9.2: ListView via convention.

In this example, we define field names to act as the parameterization of the original closure. We also made the passing of the selected data between the list and the detail views implicit rather than explicit. We could also incorporate sensible defaults to streamline the writing of the code while still allowing developers to provide explicit values where needed. For example, we might make the `next` property optional and assume a convention that an item on a `ListView` will always transition to a related detail view whose name is based on the `itemType`. In this case such a view might be called `CountryDetail`. Such patterns are common in frameworks such as Rails [73] and Grails [75] and can further enhance developer productivity by allowing them to focus only on those aspects of the application that most require custom code.

### 9.1.2 Implementation Patterns

Another way in which AXIOM could be extended is through the incorporation of common architectural and implementation patterns. At present AXIOM is limited by a one-size-fits-all code generation strategy, which may be fine for small-scale applications, but which will not scale effectively to larger, more complex software. It would be useful if common patterns and platform idioms could be incorporated into its DSML.

For instance, if we consider the example of the `Country` data type, we could easily conceive that we might want to generate a master-detail interaction to view and manage a set of countries. Such a syntax might appear like that of Listing 9.3.

One can easily imagine the constructive use of sensible default values for `listRenderer` and `detailRenderer` based on the underlying `type` provided to the `ListAndDetail`. As with any such approach, the goal is to provide the developer with as much or as little assistance as they desire. Configuration provides the desired flexibility. Convention reduces the need for such flexibility.

```
1 ListAndDetail(type: Country,
2  listRenderer: CountryListRenderer,
3  detailRenderer: CountryDetailRenderer)
```

LISTING 9.3: Increased abstraction via pattern.

Another example would be in the mechanism by which data is persisted. At present AXIOM works only with local storage on the mobile device. If the language could be extended to allow persistence either on the local device or through integration with an external web service, its flexibility would be dramatically improved. Furthermore, with an appropriate degree of abstraction built into the AXIOM DSL, the generation of the services that provide remote data persistence could be automated as well. Such an approach begins to incorporate elements of an Architecture Description Language (ADL) [119] into AXIOM.

## 9.2 AXIOM's Impact on Code Quality

As we showed in Chapter 8, AXIOM's prototype generates code that is in many cases, according to the SonarQube analysis, of slightly lesser quality than the equivalent hand-written code. For example, the AXIOM-generated code often exhibits a significantly greater issue density than the equivalent hand-written code. However, the issue density by itself is only one aspect of the analysis. We should also consider the estimated time to remediate the issues once they have been found. The time needed to remediate each issue varies according to a number of factors such as the development language, complexity of the remediation, developer, and development methodology. However, the approach used to remediate the issues, and the impact of that remediation, is markedly

different between the hand-written and AXIOM-generated approaches and that difference can make it much more efficient to address issues using code generation than in the traditional hand-written approach.

By way of example, suppose that we wish to deal with the most common issue identified at the top of Table 8.6, that of "Hardcoded Text." If we assume five minutes[7] to address each instance of the issue in the hand-written code, then it will take 55 minutes to address those issues in the current code base. However, there is no guarantee that new instances of the issue will not be introduced during subsequent development, requiring additional time to remediate. Finally, the fixes do not propagate to other applications, so they must be updated separately. In essence, each fix is independent of all others and each new line of code is an opportunity to inject issues.

In the case of AXIOM-generated code, it is likely that the remediation will take longer than five minutes. The appropriate translation templates must be updated, which is likely to take longer than the five minutes required to address the issue that we assumed for the hand written code. However, once the templates have been updated, the issue has now been updated for all instances of the code, both present and future. Whenever that template is used to generate code, the new translation rules will be used and the issue will not be re-introduced. In the case of the CAR code, this translates into a savings of 155 minutes less the time needed to update the template. If this issue exists in other applications, as it doubtless will, then the fix will be applied once the new template is used during translation, again resulting in potentially significant time savings.

Because the fixes are cumulative, AXIOM represents a better long-term investment for the remediation effort. In addition, as industry best practices evolve, those practices can be incorporated into the code templates more readily than if we needed to refactor each application by hand.

[7] The SQALE [104] metrics tend not to go below this five minute remediation threshold.

## 9.3 Future Work

In addition to increasing AXIOM's abstraction level and incorporating common patterns and idioms, there are other promising areas of future research including using AXIOM for other application domains and making the language more adaptable to changes in the mobile domain.

### 9.3.1 Alternate Application Domains

The AXIOM DSML has been constructed specifically for the mobile application domain. However, there is no effective limitation to the overarching AXIOM approach. This means that AXIOM can be extended to other domains, such as web applications. Constructing such a domain requires the same steps as for any other domain-specific language:

1. Construct the DSL syntax.
2. Define the templates that represent the code to be generated in the native solution.
3. Define the injection descriptors and how they will map from the AXIOM model elements to the placeholders within the templates.

### 9.3.2 Adaptive Domain-Specific Modeling Languages

One intriguing avenue of research has resulted from this work: adaptive domain-specific modeling languages (ADSML) [120, 121]. One of the challenges that results from the use of a domain-specific modeling language, especially in a domain that evolves as rapidly as mobile technologies, is how to quickly extend the language to accommodate new features and capabilities. We proposed a new kind of DSML that is able to dynamically and automatically adapt its syntax to incorporate new linguistic features and frameworks.

# Chapter 10

# Related Work

## 10.1 General Model-Driven Engineering

Other software development approaches share some common characteristics with the AXIOM approach. In many cases we have drawn upon the strengths of these approaches as inspiration for AXIOM. This chapter describes some of these approaches and how they relate to AXIOM's fundamental goals of being agile, model-driven and formal. We review these approaches to differentiate them from AXIOM and from our vision of what a true fusion of agile development with MDE would look like.

In general we classify these general-purpose MDE approaches into three major categories:

- **UML-Based Approaches**. These are approaches that closely follow the MDA standard as described by the OMG by relying on UML and its profiles, OCL, and other related MDA extensions.
- **Process-Driven Approaches**. These approaches attempt to take the core MDA concepts, but addresses them more from a process perspective rather than by rigorously following the MDA standard and approach. This is often done in an effort to incorporate agility into processes that otherwise rely on more "heavyweight" UML.

- **Formal Approaches**. These approaches are based on formal models and have little or nothing in common with MDA. However, these models provide similar benefits as MDA's OCL and may prove more intuitive to use.

It is important to note that while AXIOM is currently being exercised within the mobile application domain, we believe that the underlying approach is not limited to that domain and that the same approach could be used to build other kinds of applications.

### 10.1.1 UML-Based Approaches

Some MDA implementations closely follow the OMG's MDA standards including their reliance on UML, OCL, and MOF. We call these "classic MDA approaches".

AndroMDA [122] is an open-source MDA framework. It accepts UML models, generally in an XMI format, and uses them to generate custom components. AndroMDA uses a plugin or "cartridge" system. These cartridges are used to generate the various components, which allows AndroMDA to evolve its generative capabilities without requiring a constant overhaul of the core AndroMDA framework.

AndroMDA uses UML stereotypes as one means by which it can determine which cartridges can be used for code generation. For example, classes marked with the `<<Service>>` stereotype use different cartridges than those marked with an `<<Entity>>` stereotype.

In addition to stereotypes, AndroMDA requires several other rules to be followed within the model to ensure both syntactic and semantic completeness so that the code generation process completes smoothly. Many of these rules are straightforward and closely resemble the "convention over configuration" trend of many software frameworks such as Ruby on Rails.

Finally, AndroMDA combines the concepts of *templates*, for producing code, with *metafacades*, which are facades that shield template and cartridge builders from the complexity of interacting with the underlying MOF metamodel that is used to represent

the various entities within the models being consumed. This approach allows the templates to be built in such a way that they can flexibly adapt to changes in the model and metamodel without requiring changes to the underlying templating language.

Because AndroMDA uses traditional MDA, it is tightly coupled to UML with all of its associated benefits and problems. Because it is not itself a modeling environment, it is also limited by the quality of the XMI output produced by each vendors' tools. Finally, because it uses MOF in the form of its metafacades, it incurs a penalty in terms of model representation and transformation because of the overhead involved in converting the model representation into the corresponding metamodel AST.

While AndroMDA does satisfy the tenets of MDA, it is not suited to projects using agile methodologies. First, it requires UML models to be built in different tools and then exported via XMI so that the code can be generated. The implication is that we require two completely different skill sets: modelers, to build the models and developers to provide any custom code needed by the cartridges to produce their code. Finally, there is no support for UI as a first-class concept, a shortcoming often associated with UML.

The Eclipse Foundation provides multiple technologies that support MDD across the various aspects of MDD including model construction and model transformation. These projects include Generative Modeling Technologies (GMT) [123], the ATL Transformation Language (ATL) [124] and others. As is the case with AndroMDA, these technologies are all based on UML. This makes them a standard MDD approach but also means that they have all of the limitations that are inherent to that approach. These projects have an advantage over AndroMDA in that the tools can produce and consume the various models whereas AndroMDA is intended to consume the models produced elsewhere.

AXIOM differs from the Eclipse approach both in terms of its approach of using a DSL for all aspects of modeling rather than just for the transformation process. AXIOM also provides for the UI as a first-class model.

Executable UML (xUML) is an approach that uses UML models as the primary mechanism by which applications are built [125]. Like AXIOM, xUML advocates the benefits of UML executability. xUML defines three different models for specifying software:

- **Data.** This is represented using classes as well as their attributes, associations and constraints. The UML class diagram is the notation used for this model.
- **Control.** This model is all about the states of the various data as well as the events, transitions and procedures that are used to move them through their respective lifecycles. A UML statechart is used to express this model.
- **Action.** These are the fundamental operations within the application. Actions are referenced from the control model and are triggered whenever an object changes state. The UML action language is used to define all actions.

Once the various xUML models have been defined, they are processed by a *model compiler*. This compiler is responsible for taking the xUML models, which are the equivalent of the MDA's PIMs, and transforming them into executable code by applying a set of decisions about the target runtime environment. In other words, the model compiler decorates the PIM with runtime configuration information, thereby creating a PSM, and then transforms that PSM into executable code.

xUML is consistent with our definition of MDA, but it has some significant problems. The first is that it is based on UML, and thus possesses all of UML's aforementioned limitations. The second problem is that the process of writing a model compiler may prove to be as complex, if not more so, than producing the original models. There are examples of publicly available xUML compilers such as xUmlCompiler [126], but each compiler targets a specific set of technologies for its code generation processes. Furthermore, xUML, which relies on fUML [127] and ALF [95], uses a general-purpose language. While this provides significant flexibility, it remains a least-common denominator approach; the language makes no assumptions about what it will model.

AXIOM differs from xUML by supporting agility in addition to executability, through dynamic typing and on-demand static checking and analyses. Another key difference is AXIOM's support for UI models as a first-class models. AXIOM also fixes the target domain to mobile applications and provides a DSL to simplify models in that domain. Because the AXIOM DSL is written in a JVM-based language, it has access to any library that is available to the JVM.

XSI-Mobile [128] is a UML profile used to represent mobile concepts. It is based on and extends the XIS [129, 130], a UML profile that can be used to define PIMs for interactive applications, but which lacks the concepts common to mobile applications such as gestures. It defines various views along with navigation rules and widgets. The widgets are then associated with gestures and used to trigger actions. Custom operations can be defined as stubs, with the developer then filling in the missing details after code generation. This approach suffers from the same basic issues as UML in general and MDA in particular.

Mayerhofer [131] 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.

### 10.1.2 Process-Driven Approaches

There have been attempts to unify agile methodologies with MDA in the past by constructing hybrid approaches. While some such approaches are little more than the informal application of agile thought processes to MDD [132], others involve a wholesale modification of the underlying MDD premises and approach.

#### 10.1.2.1 Agile Model Driven Development

*Agile Model Driven Development (AMDD)* is an iterative software development process using UML. While AMDD [133] shares the notations and tools commonly used in MDD, one notable deviation from the typical MDD approach is that AMDD keeps the code as the focus of the development effort. While AMDD supports more sophisticated code-generating models, its author assumes that only 5-10% of business application models can benefit from such treatment. Instead, AMDD models are strictly passive, serving as a communication medium between developers and the business community, but are not used to generate code. Since the vast majority of the code in AMDD is written by hand, it does not rely on code generation or on the round-trip engineering capabilities of UML tools. It is the developer's responsibility to manually maintain the

consistency between the model and code. Unfortunately, this falls into the well known trap of software evolution that such artifacts will rapidly become inconsistent with one another; the code will be maintained while any other documentation will be completed hastily at the end of project development cycle assuming it is ever completed at all.

In short, AMDD treats models mainly as way of understanding the problem and its potential solutions. It is not concerned with the executability of the models, nor is it concerned with generating code from the models. Therefore, it is fundamentally different from the AXIOM approach.

AMDD has been executed in combination with the MIDAS framework [134, 135] as a means of implementing web-based applications.

#### 10.1.2.2 Continuous Model Driven Engineering

*Continuous Model Driven Engineering (CMDE)* creates "holistic models" by combining model-driven design, extreme programming, and process modeling. Together these elements constitute eXtreme model-driven design (XMDD) [136]. XMDD goes beyond traditional MDD by using process modeling as its means of eliciting the necessary requirements and behavior. This process model, or *service logic model*, is the central model of XMDD and undergoes successive refinements until its abstractions can be mapped to existing services. Within XMDD this is referred to as the "One Thing Approach." The approach of using BPM provides several useful capabilities such as simulation and actual process execution via BPM engines. The process models can be decorated as needed with temporal constraints, symbolic type information and cross-cutting concerns.

While CMDE seeks to address the representational differences between what business users understand and what developers must build, it only partially succeeds in this goal. Process models may indeed be more intuitive for business users to understand, but it is not clear that such models lend themselves to the actual realization of the finished application, which often requires code generation, not only by an MDD transformation, but from software developers as well. In those cases, the differences between the requirements captured by the model and the implementation captured by the code may be

significant. In addition, this approach does not emphasize the use of UML and is thus unable to capitalize on the significant numbers of modelers and developers who have been trained in its use.

### 10.1.3 Formal Approaches

Before there was MDA, there were formal modeling notations. These notations allows us to subject models to formal analysis and verification in an effort to discover inconsistencies or errors. The approaches described in this section may support MDD, but they often do not provide complete solutions by themselves. One particularly valuable aspect of many formal approaches is model checking, which allows a modeler to determine whether their models make logical sense. We hope to provide similar capabilities within AXIOM to support formal verification of its models.

Alloy [137] is a formal specification and modeling notation based on first-order logic that can be used for model-driven development. Alloy is designed for the specification and verification of models.

Research has been done in transforming UML model into Alloy models, most notably a tool called UML2Alloy [138] but this is a challenging process because of the notational and semantic differences between the two languages. For example, UML distinguishes between sets, scalars and relations whereas Alloy does not. Even if the UML models are successfully transformed into equivalent Alloy models, Alloy itself does not support model executability and was not designed for agile development.

UML Specification Environment (USE) [139, 140] is a formal specification and modeling notation that can be used for MDD. In addition to the graphical UML representation, USE also provides a complementary human readable textual representation and is in that respect similar to AXIOM. USE is designed for the specification and verification of models and some of its model verification tools could potentially be adapted for AXIOM models. However, USE does not support model executability and was not designed for agile development.

Like Alloy and USE, Z is a formal specification language. Z has seen only limited adoption and then primarily for extremely complex or safety-critical applications [141, 142]. While Z can be used for representing the functional requirements of an application, there is not enough information captured by the schemas to adequately represent other critical concerns of the software such as its components and their organization and interaction. These concerns are crucial for object-oriented software because they allow modelers to define useful and usable abstractions that will ultimately be used to build the finished software. Similarly, Z is not itself executable although some work has been done on making Z schemas executable [143].

While various object-oriented extensions to Z have been proposed [50] such as MooZ [51], Object-Z [52], OOZE [53], Z++ [54], and ZEST [55] in an attempt to bring Z's formalism to object-orientation, they all suffer from the same fundamental challenge, which is that they are intended to formalize requirements rather than implementations. This is of particular concern where agile methodologies are concerned, since their focus is precisely on the primary deliverable of application code rather than intermediate deliverables such as formal proofs that the application requirements are consistent and complete.

Z was never designed for MDD or agile approaches. However, its formalism allows for the analysis of application requirements. That formalism is a key feature within AXIOM and is what enables some of those same model-checking capabilities that Z enjoys.

Work on platform independence has been done by Jia and Liu in their work on ZOOM (Z-based Object-Oriented Modeling), which seeks to combine the formal semantics of Z [47] with the object-oriented notations of UML [144–146]. ZOOM divides an application into three components: structure, behavior and user interface, and provides separate models for each. These models are then integrated via an event-driven framework. Code is generated based on the formal specifications embedded within the model as well as with more traditional model transformation techniques [147–149].

ZOOM provided the foundation on which AXIOM is based and has thus heavily influenced the AXIOM approach. Unlike AXIOM, ZOOM attempted to remain faithful to

the standard MDA approach and was thus hampered by UML's limitations as well as with dealing with the complexities of MOF. AXIOM distinguishes itself from ZOOM by deviating from the OMG MDA standard and using a DSL with a visual representation that is similar to UML, but that is closer to the code that would be written by a developer.

## 10.2 Mobile Domain Approaches

These approaches are inspired by traditional MDA and code generation, but which make little or no attempt to follow OMG's MDA standards and have little, if any, reliance on UML and its profiles. Instead, they focus on trying to solve the same problems as MDA but do so using domain-specific languages, in particular those that target the domain of mobile applications. In general these languages result in one of two types of generated code: code that is native to the target OS or code that is expected to run in a virtual machine, such as JavaScript and HTML.

There are many DSL-based approaches to mobile development including Rhodes [150], DragonRAD[1], Appcelerator, mobl, Canappi, MobDSL, DIMAG, Mobia Modeler, and md[2]. Many, although not all, of these have been summarized by Ribeiro [151], but they each fall into one of the two basic approaches mentioned at the start of this section.

Mobl [152] 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.

Canappi [153] is an application that allows for the rapid creation of mobile applications. According to the Canappi website:

> "The vision of Canappi is to abstract the mobile app such that a well defined solution model exists regardless of the technology and architecture."

[1] DragonRAD appears to now be defunct.

Canappi uses a DSL called "m|dsl" to act as the PIM. This model is uploaded to Canappi's website where it is converted into source code. This code is then downloaded and loaded into an appropriate IDE, where it can be compiled and deployed into the target runtime environment.

Canappi provides many of the basic elements of MDD including the definition of a PIM and the generation of appropriate platform-specific code. However, it has significant limitations. First, the model files only represent the mobile user interface. For more complex applications, those models must be hooked to back-end services. While Canappi provides for that integration, it does not provide for the actual generation of those services. Second, the Canappi models do not provide for complex behavioral logic. There is no concept of a state machine nor any indication that such a model is necessary. Finally, Canappi is currently limited to only mobile applications.

MobDSL [154] is a DSL that uses a VM as the intermediary between the MobDSL application and the underlying platform. This is a similar approach as that taken by tools like Cannapi or Appcelerator save that the VMs are not based on existing cross-platform technologies such as HTML5 and CSS, but rather based on the native platform APIs. This provides less separation between the modeling language and the executable environments, but still suffers from the basic performance degradation resulting from the use of the VM. AXIOM avoids the VM in favor of completely native applications.

DIMAG (Device Independent Mobile Application Generation framework) [155] is a framework that allows for a single, declarative application definition to be used to generate applications across a range of devices. The model is separated into several elements including the DIMAG-root language, the DIMAG-ui user interface language, and an SCXML [156] workflow description to address the actions to be triggered when the user interacts with the UI elements of the application.

DIMAG enforces separation of concerns by splitting up the UI definition from the workflow that underlies that UI. In the case of AXIOM these two elements are combined into a single MADL file. There are similar elements that can be used to provide guard conditions and trigger actions and transitions between different elements of the application. One significant difference between DIMAG and AXIOM is AXIOM's ability to take

advantage of different APIs on the target platform by being able to incorporate them directly into the model as code. DIMAG provides some limited support for such integration, but the developer may need to provide "wrappers" that expose those libraries in a way that can be easily incorporated into the model.

Mobia Modeler [157, 158] attempts to make it possible for non-technical people to easily build their own applications. This is different from AXIOM, which seeks to ease the burden of developing mobile applications from traditional software application developers. Mobia Modeler follows the standard separation of PIM from PSM while still generating native code that can be deployed to the target application. The process of code generation is similar to that of AXIOM although it differs in the particulars.

md$^2$ [159] 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 md$^2$ though with differences in the role of the developer in advising the transformation process. Unlike AXIOM, md$^2$ suffers from certain limitations such as the inability to easily provide scrollable lists of data, a common user experience in data-driven mobile applications.

Vaupel [160] defines an MDD tool for mobile development that consists of several complementary meta-models. Provider models define an implementation of those meta-models. The tool uses multiple modeling notations including EMF [161] for the data and UI models, and BPMN [162] and WS-BPEL [163] for the behavioral aspects. AXIOM uses only a single language and does not suffer from potential inconsistencies between the provider models and the meta-models.

There are a host of other tools that address key elements of AXIOM but which, by themselves, are inadequate to provide complete applications. We briefly discuss two of them: Balsamiq and State Machine Compiler.

Balsmiq Mockups [164] is a wireframing tool that allows developers to rapidly develop linked sketches, or wireframes, of application screens. Once developed, these screens can then be exported to various formats such as PNG or PDF. The Balsamiq wireframes themselves are defined in an XML format.

Balsamiq by itself does not provide a complete MDD implementation. However, its approach of defining user interfaces in an XML format provides a foundation that could support MDD. UML does not provide for user interface modeling very well; there is simply no concept of a UI layout in its representation. [2] By using an XML format and segregating the user interface into its own set of dedicated components, Balsamiq has effectively provided for the separation of UI concerns from the remainder of the application logic and the ability to transform the XML model of the UI into other forms through technologies such as XSL. In these respects Balsamiq is entirely consistent with MDD both in terms of its UI segregation into what is effectively an MDA *domain* as well as in providing for the transformation of a UI PIM into a UI PSM.

Because AXIOM includes models for not only the user interface, but also for the structural and behavioral elements of an application, it provides a more complete solution than Balsamiq. However, Balsamiq's approach to UI representation has inspired AXIOM's user interface models.

The State Machine Compiler [165] is a DSL for state machines that can then be used to generate the code that implements the State design pattern. SMC supports the definition of states, transitions, actions, and guards. In addition, SMC allows for the navigation across different state machines, which allows for the assembly of smaller state models into larger, collaborative models.

SMC does not provide for a complete MDD solution. While it deals with an important part of an overall application model, the behavioral aspect, it does not effectively define the structural characteristics that the application's components needs to possess. Similarly, while the DSL captures the capabilities of a state machine, it does not do so in a way that is consistent with UML and it does not provide for a graphical means of representing those state charts; such graphics can be generated from the state machine source files, but can not be used to generate the state chart itself. SMC does allow for the generation of code targeting a variety of languages such as Java, C++, Perl, VB, Groovy and C# based on the common DSL. From this standpoint it does provide for rich PIM-to-PSM transformation.

[2] Some approaches suggest using profiles and stereotypes as a means of representing user interfaces, but this approach is non-intuitive and has never gained traction within the UML community.

Like SMC, AXIOM captures the behavioral aspects of an application in the form of a state machine. However, it also captures the domain and user interface aspects, making it a more complete approach to MDA than SMC alone.

Research on the encoding and accessing of the native platform APIs within models has been done by Cuadrado [166] in describing a process whereby a meta-model is used to generate an intermediate language that produces Java bytecode that references the API. AXIOM relies instead on pre-defined mappings of objects and their properties.

AXIOM emphasizes one-way transformation from model-to-platform. Research on bi-directional transformations such as that by Anjorin [167] can allow changes to the implementation to be incorporated into the model to support round-trip engineering. In practice the reversing of native code back into a model, particularly a platform-independent model is challenging although research in Adaptive Domain-Specific Languages (Chapter 9.3.2) might make such round-trip engineering more feasible.

# Chapter 11

# Conclusions

It is desirable for mobile applications to run on different platforms. However, writing applications for multiple platforms can require significant development effort, leading to increased development and testing costs. There are two common ways of reducing this effort. First, code can be written using common, industry standard technologies and run within a VM. Second, code generation can be employed such that the application is written once in a platform-neutral way and then converted into a more platform-specific representation for runtime execution.

Model-driven development is ideally suited to this kind of development since it emphasizes the use of models to represent applications rather than relying on native code. Traditional model-driven development, such as the OMG's Model-Driven Architecture, rely on UML to represent their models. However, such approaches often suffer from UML limitations, a lack of adequate tool support and limited access to useful frameworks and libraries. These approaches take the initial platform-independent models and transforms them into platform-specific models, which are then transformed into executable code. This transformation can result in executable code that adopts the same two approaches mentioned earlier such as native code or code that runs within virtual machines that rely on standardized technologies such as HTML5 and CSS.

An alternative approach to using a general-purpose modeling language such as UML to represent the models is to use a domain-specific language instead. Such languages can be constructed to more easily represent concepts from the mobile domain and to

make it simpler to develop applications for that domain. Domain-specific languages can be internal or external. Modern dynamic languages such as Ruby and Groovy provide built-in support for the design of domain-specific languages, making it simpler to design and develop such languages as well as given them full access to the libraries and APIs of the host language. Through the use of dynamic languages, model-driven development becomes more feasible than if it remains bound solely to UML.

AXIOM is an approach that attempts to retain the model-centricity of classic model-driven approaches, but which uses an internal DSL to represent those models. AXIOM's goal is to evolve MDD by augmenting UML with a dynamic language. While there are challenges to overcome, the benefits provided by this synthesis are significant and include a high degree of abstraction in the form of a DSML, support for existing libraries and frameworks, and freely available tool support in the form of existing text editors and source code control systems. Dynamic languages yield executable models, which can be rapidly validated and verified.

AXIOM's DSML retains key elements of UML state charts to represent application behavior. The use of Groovy as the actual modeling language facilitates the transformation of AXIOM platform-independent models into platform-specific models. These models are fast to develop and easy to verify, making them compatible with mainstream agile methodologies.

AXIOM's requirements model captures the user interface and behavioral aspects of the application. The requirements model is passed through AXIOM's multi-phase transformation process. The transformation process incorporates macro-level architectural elements as well as micro-level elements to produce a final implementation model, which is then translated into native source code for the target platform. The code generation process is template driven and those templates can be defined at both macro- and micro-levels to better control the generated code.

AXIOM is completely generative. Thus, developers need not edit the generated code to incorporate additional logic because all such logic is specified during model construction. 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 just 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.

Tests have been conducted on a sample of mobile applications that reflect common requirements such as cross-screen navigation and 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. Our results may be due in part to natural variability in developer skill, although AXIOM embeds much of that domain knowledge in its DSL, reducing its overall impact. There are doubtless more efficient implementations than those submitted during our tests. Based on these tests, this research attempts to answer some key questions about model-driven development for mobile platforms using the AXIOM approach:

1. Is developer productivity with AXIOM significantly different when compared to writing native mobile applications?
2. Does the code created using AXIOM exhibit similar quality to equivalent hand-written code?

Conceptually, AXIOM can improve developer productivity because it emphasizes up-front modeling and because the transformation rules and templates can be changed and reused. 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 be provided by the development team. 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 and without being required to start *ex nihilo*.

The preliminary results of AXIOM's impact on developer productivity are promising. It has the potential to deliver significant cost savings, particularly for cross-platform application development, while improving overall application quality. Based on the analysis using Jiang's productivity metric, AXIOM can deliver significant productivity benefits to developers. This is because of its use of a single platform-independent DSML that can define both the interaction and behavoral aspects of the application. This allows productivity that is about 400% greater than producing the equivalent code by hand.

A similar analysis of the SLOC required for the AXIOM models compared to hand-written code produced by the native tools for each platform suggests a wider range in terms of productivity. In the worst cases AXIOM required only slightly less code than

the equivalent native applications while in the best case AXIOM required only about 8% of the lines of code of an equivalent iOS application and only 5% of the lines of code of an equivalent native Android application.

Finally, our preliminary results with respect to AXIOM's representational power and conciseness are encouraging, showing that AXIOM can represent concepts in its DSL more concisely than can be done in the native programming languages.

AXIOM's impact on code quality is more ambiguous. With respect to the AXIOM prototype the answer is "no" since it generally produces more code and thus more issues than the equivalent hand-written code. However, with further optimizations and better templates, there is every reason to believe that the AXIOM approach can indeed be comparable, and perhaps even better, than hand-written code precisely because any changes made to the template will be rendered in the generated code everywhere that template is used rather than requiring individual fixes for each observation of a particular issue. The analysis of the SonarQube data suggests that while the native tools for each platform doubtless help prevent some of the issues that can be introduced during AXIOM's Translation stage, they are not prevented in their entirety. Those issues may be simple to eliminate, but each issue requires at least some developer time, which can prove to be a drain on productivity.

The AXIOM DSML is comparatively young and could benefit from several optimizations. In particular it would be useful to incorporate more patterns and idioms of the mobile platform directly into the language itself. This would further reduce the size of the AXIOM models while also making it easier to design and construct code generation templates to produce efficient and optimized native code. Furthermore, the approach at present only recognizes a comparatively small subset of the iOS and Android platforms. Further research suggests that extending AXIOM to use an Adaptive DSML could further improve developer productivity and code quality by automatically ingesting changes to the underlying platforms and exposing them via the AXIOM DSML.

Thus far we have not found any inherent limitations in AXIOM's approach, although we have found several in our prototype and in the current DSL. For example, the DSL does not currently have a formalized entity model, relying instead on available Groovy types

such as maps. This limits the automatic discovery and layout of data fields since there is not enough type information available to enable AXIOM to accurately determine which graphical widget is appropriate for each data element. Similarly, as we have seen, the prototype does not always generate the most optimal code. Finally, the prototype currently uses only a subset of the iOS and Android APIs although AXIOM's DSL can be extended by importing additional libraries. All of these are limitations of the prototype and not of the approach in general. The goal of incorporating additional libraries into the language has led to research in the area of "adaptive DSMLs" as described in Section 9.3.2.

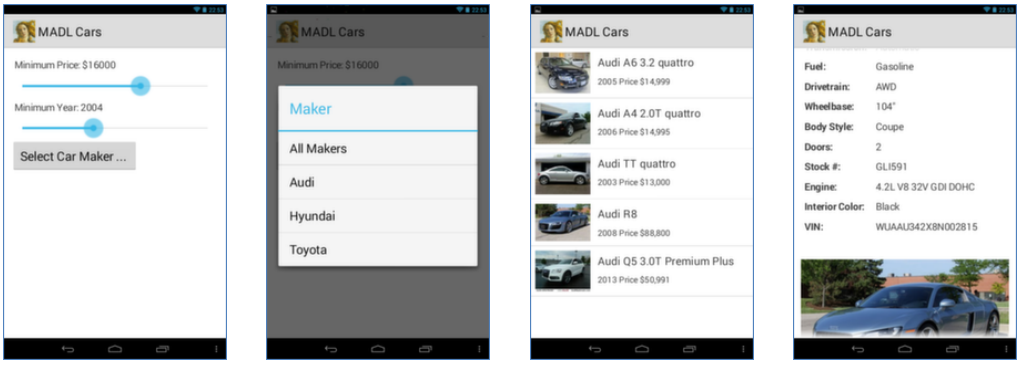
Other researches have been exploring the use of DSLs for mobile application development as well. Appcelerator, Mobl, MobDSL, Cannapi, and md$^2$ are all examples of such DSLs. However, each of these approaches takes a least-common denominator approach to their languages that make it difficult, if not impossible, to adequately integrate the native API and related libraries into the model itself. This can lead to developers using those tools to generate the initial native code, which is then augmented with hand-written code. This approach improves the time to deliver the initial application skeleton, but then forces the developer to leave the model behind in favor of hand-written code, which thus negates many of the benefits that MDD provides.

AXIOM is an approach to model-driven development in the mobile domain that seeks to improve developer productivity while maintaining a high degree of code quality. AXIOM does this using a DSML and a multi-pass transformation and template-driven code generation. The AXIOM approach has demonstrated significant productivity benefits and appears well suited to modern agile development approaches that favor only artifacts that directly contribute to the understanding of the application and its final, executable form. The use of a dynamic language further reduces the overhead of using the DSML to model mobile applications. The results of the analysis of AXIOM's impact on productivity and code quality are promising and further potential improvements to the language have been identified, including the use of adaptive domain-specific modeling languages that may further enhance AXIOM's ability to quickly adapt to the rapidly changing capabilities of mobile platforms.

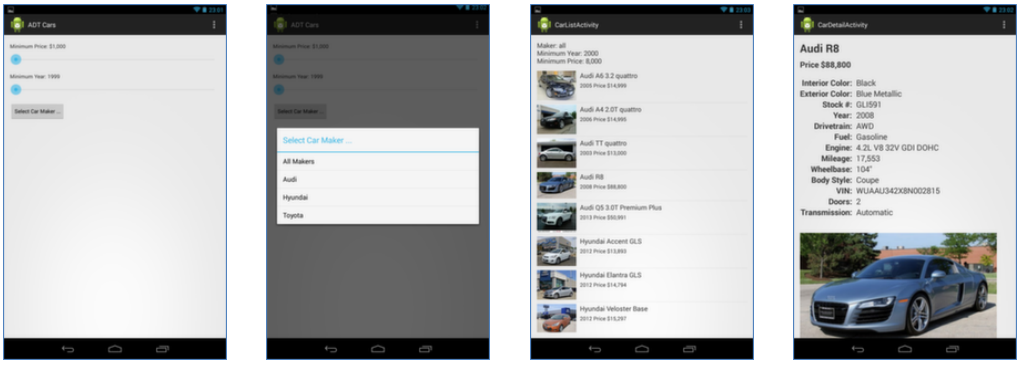
# Appendix A

# Mid-Scale Application Screenshots

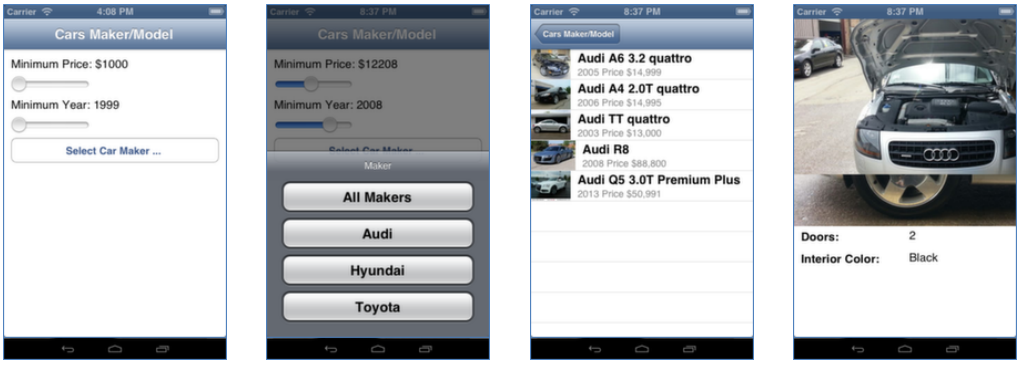
We provide side-by-side screen captures of the executing version of the AXIOM and natively developed mid-scale applications described in Chapter 7.



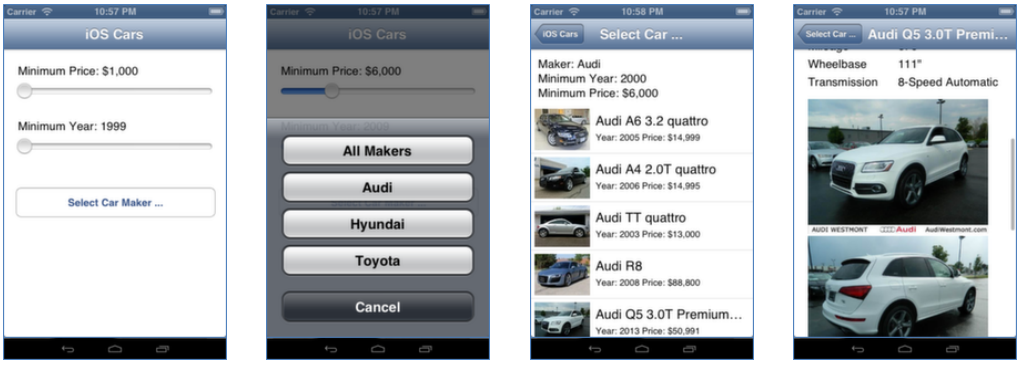

(A) AXIOM-generated Android screens. Author's images.




(B) Native Android screens. Author's images.

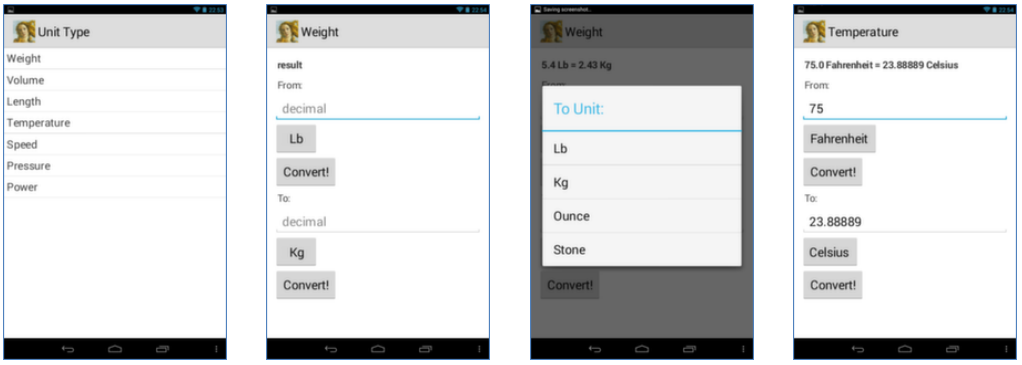



(C) AXIOM-generated iOS screens. Author's images.




(D) Native iOS screens. Author's images.

FIGURE A.1: AXIOM, Android, and iOS Screen Captures for CAR Application.



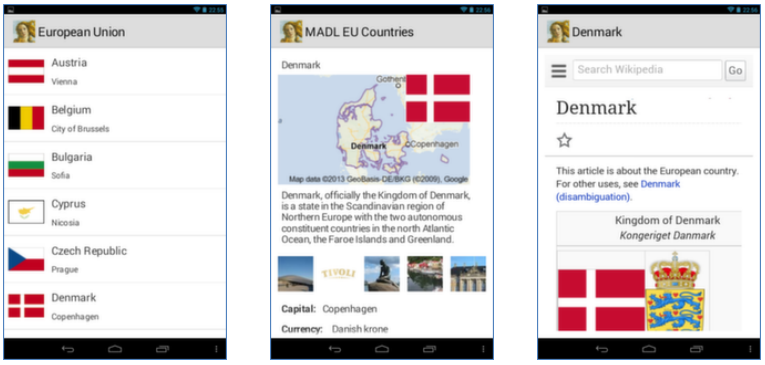

(A) AXIOM-generated Android screens. Author's images.


(B) Native Android screens. Author's images.

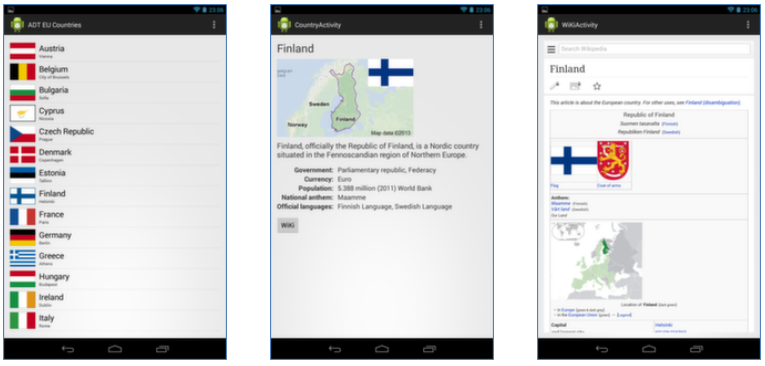

(C) AXIOM-generated iOS screens. Author's images.


(D) Native iOS screens. Author's images.

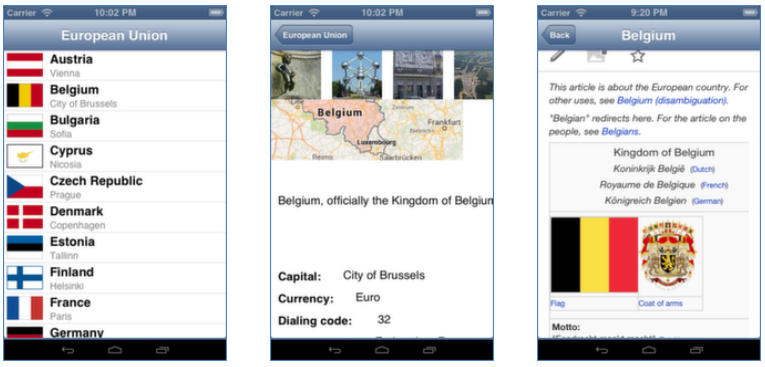
FIGURE A.2: AXIOM, Android, and iOS Screen Captures for CVT Application.



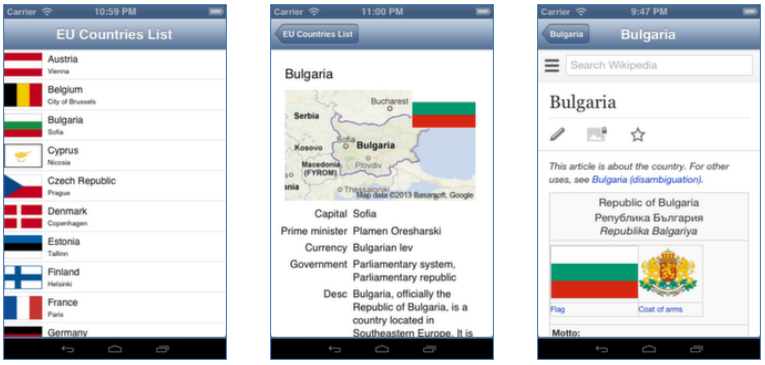

(A) AXIOM-generated Android screens. Author's images.




(B) Native Android screens. Author's images.

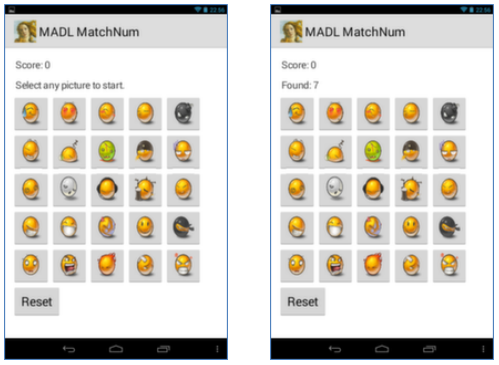



(C) AXIOM-generated iOS screens. Author's images.

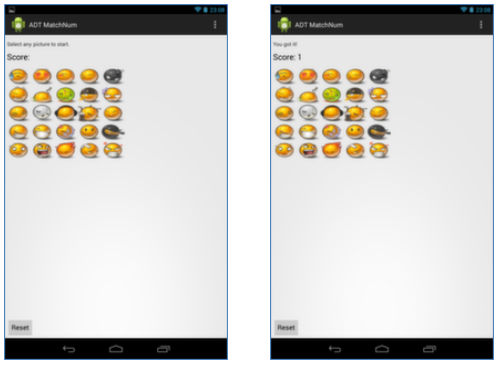



(D) Native iOS screens. Author's images.

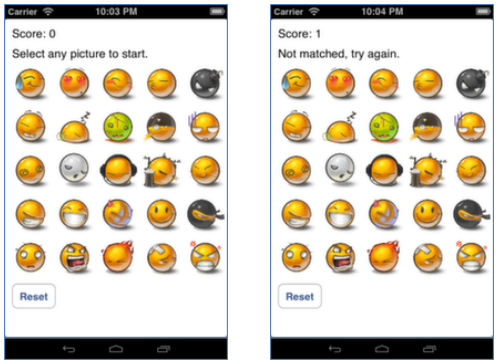
FIGURE A.3: AXIOM, Android, and iOS Screen Captures for EUC Application.



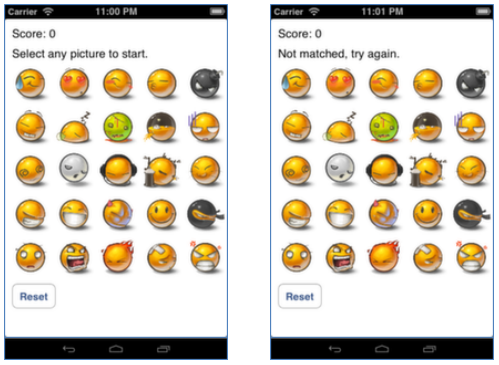

(A) AXIOM-Generated Android screens. Author's images.




(B) Native Android screens. Author's images.

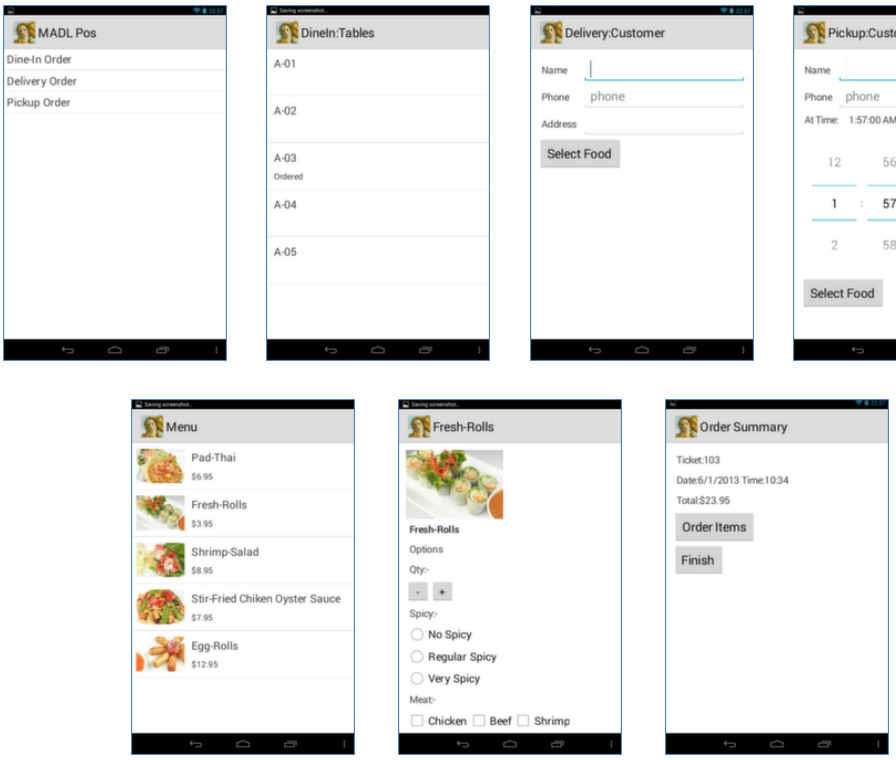



(C) AXIOM-Generated iOS screens. Author's images.

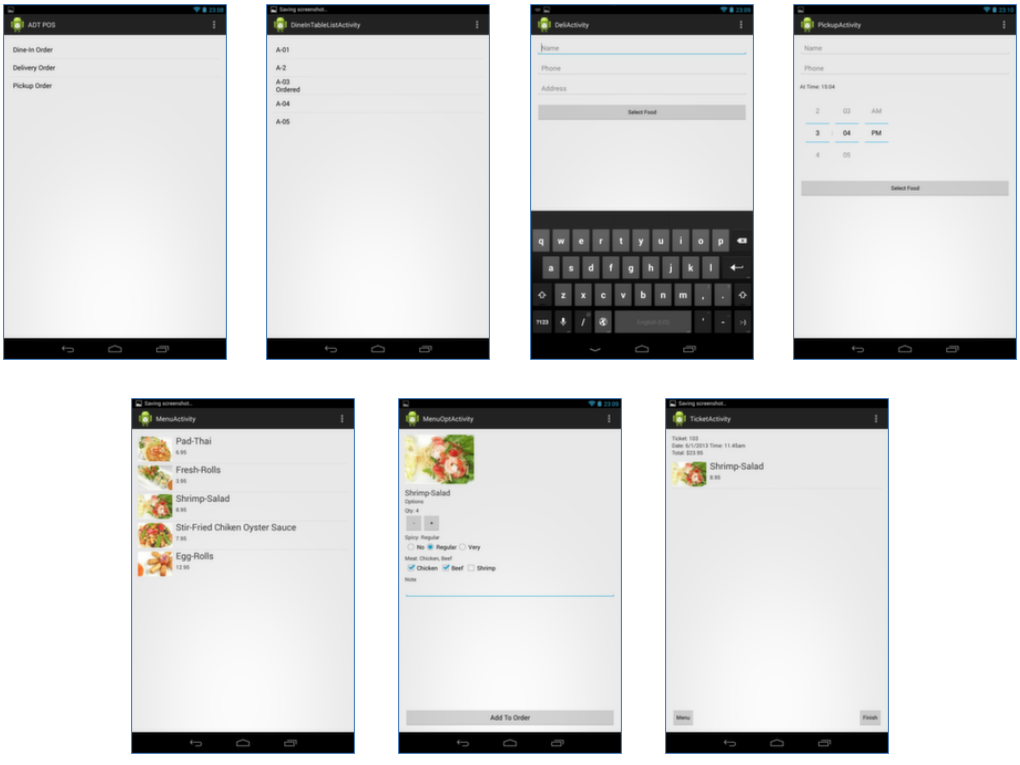



(D) Native iOS screens. Author's images.

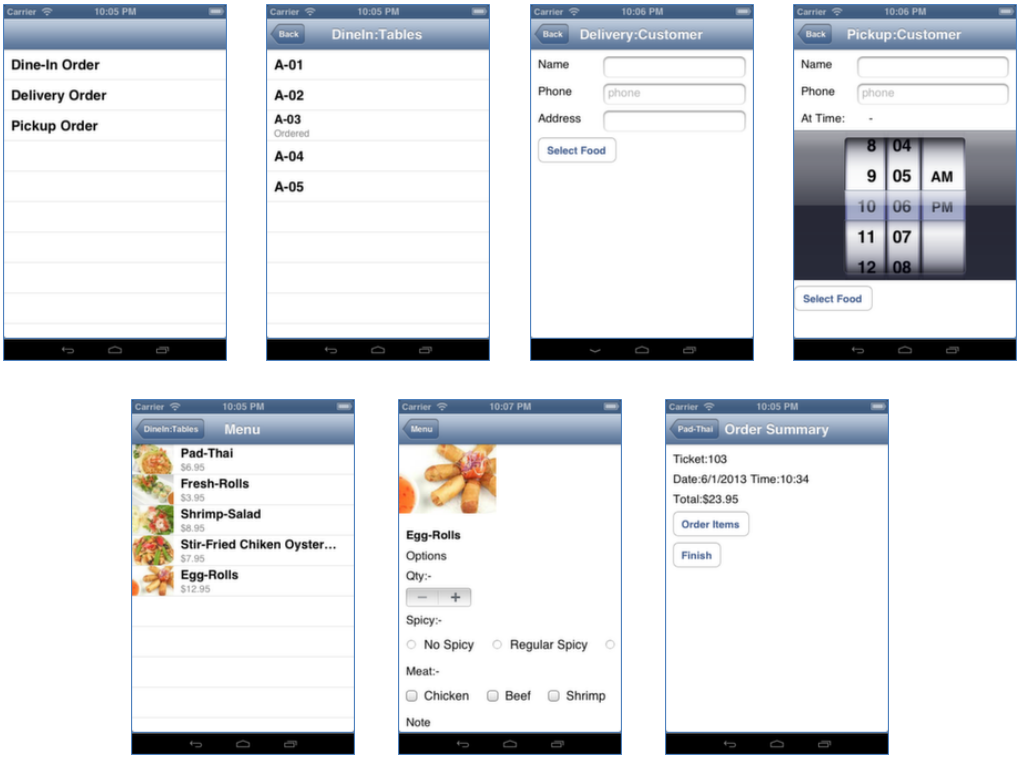
FIGURE A.4: AXIOM, Android, and iOS Screen Captures for MAT Application.



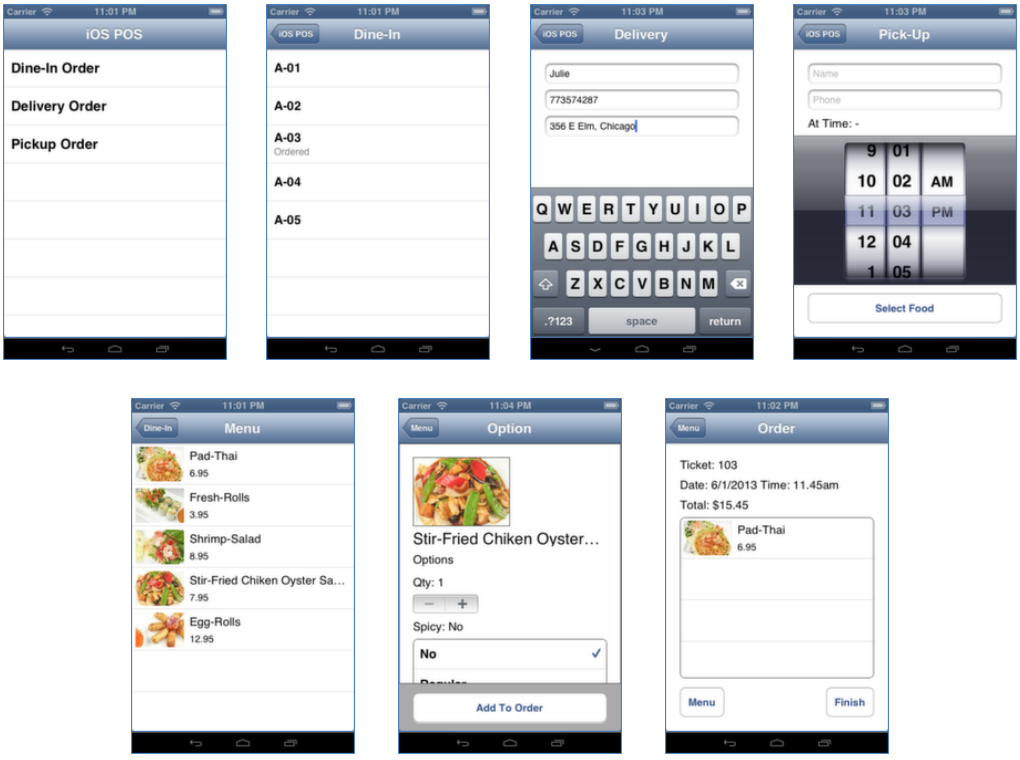

(A) AXIOM-generated Android screens. Author's images.



(B) Native Android screens. Author's images.




(C) AXIOM-generated iOS screens. Author's images.




(D) Native iOS screens. Author's images.

FIGURE A.5: AXIOM, Android, and iOS screen captures for POS application.

# Bibliography


[1] Number of Android Applications, October 2015. URL http://www.appbrain.com/stats/number-of-android-apps.

[2] Number of Apps Availabile in Leading App Stores as of July 2015, October 2015. URL http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[3] Count of Active Applications the App Stores, October 2015. URL http://www.pocketgamer.biz/metrics/app-store/app-count/.

[4] Anthony I. Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0427-6. doi: 10.1145/1882362.1882443. URL http://doi.acm.org/10.1145/1882362.1882443.

[5] Josh Dehlinger and Jeremy Dixon. Mobile application software engineering: Challenges and research directions. In *Workshop on Mobile Software Engineering*, volume 2, pages 2–2, 2011.

[6] J. Whittle, J. Hutchinson, and M. Rouncefield. The state of practice in model-driven engineering. *Software, IEEE*, 31(3):79–85, May 2014. ISSN 0740-7459. doi: 10.1109/MS.2013.65.

[7] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, BettyH.C. Cheng, Philippe Collet, Benoit Combemale, RobertB. France, Rogardt Heldal, James Hill, Jörg Kienzle, Matthias Schöttle, Friedrich Steimann, Dave Stikkolorum, and Jon Whittle. The relevance of model-driven engineering thirty years from now. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 183–200. Springer International Publishing, 2014. ISBN 978-3-319-11652-5. doi: 10.1007/978-3-319-11653-2_12. URL http://dx.doi.org/10.1007/978-3-319-11653-2_12.

[8] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43703-7. URL http://portal.acm.org/citation.cfm?id=647983.743552.

[9] Xiaoping Jia and Christopher Jones. Dynamic languages as modeling notations in model driven engineering. In *6th Annual International Conference on Software and Data Technologies. (ICSOFT 2011)*, pages 220–225, Seville, Spain, July 2011.

[10] Xiaoping Jia and Christopher Jones. AXIOM: A model-driven approach to cross-platform application development. In *7th Annual International Conference on Software and Data Technologies. (ICSOFT 2012)*, pages 24–33, Rome, Italy, July 2012.

[11] Christopher Jones and Xiaoping Jia. The AXIOM model framework: Transforming requirements to native code for cross-platform mobile applications. In *9th Annual International Conference on Evaluation of Novel Approaches to Software Engineering. (ENASE 2014)*, pages 26–37, Lisbon, Portugal, April 2014.

[12] Christopher Jones and Xiaoping Jia. Using a domain specific language for lightweight model-driven development. In *Software and Data Technologies*, volume 551 of *Communications in Computer and Information Science*, page Forthcoming. Springer Berlin Heidelberg, 2015. ISBN Forthcoming. doi: Forthcoming. URL Forthcoming.

[13] Xiaoping Jia and Christopher Jones. Cross-platform application development using AXIOM as an agile model-driven approach. In José Cordeiro, Slimane Hammoudi, and Marten Sinderen, editors, *Software and Data Technologies*, volume 411 of *Communications in Computer and Information Science*, pages 36–51. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-45403-5. doi: 10.1007/978-3-642-45404-2_3. URL http://dx.doi.org/10.1007/978-3-642-45404-2_3.

[14] P. Braun and R. Eckhaus. Experiences on model-driven software development for mobile applications. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the*, pages 490–493, March 2008. doi: 10.1109/ECBS.2008.50.

[15] Stephen J. Mellor, Anthony N. Clark, and Takao Futagami. Model-driven development. *IEEE Software*, 20(5):14–18, 2003. ISSN 0740-7459.

[16] Ed Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, 2003. ISSN 0740-7459.

[17] Krishnakumar Balasubramanian et al. Developing applications using model-driven design environments. *Computer*, 39(2):33, 2006. ISSN 0018-9162. doi: http://dx.doi.org/10.1109/MC.2006.54.

[18] Object Management Group. MDA guide, June 2003. URL http://www.omg.org/mda.

[19] A. Kleppe and S. Warmer. *MDA Explained. The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[20] S.J. Mellor et al. *MDA Distilled: Principles of Model-Driven Architecture*. Addison-Wesley, 2004.

[21] David S. Frankel. *Model-Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, New York, NY, 2003.

[22] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20 (5):19–25, 2003. ISSN 0740-7459.

[23] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language reference manual*, second edition, 2004.

[24] Object Management Group. Unified Modeling Language, May 2010. URL http://www.omg.org/spec/UML/2.3/.

[25] Object Management Group. UML 2.0 OCL, January 2003. URL http://www.omg.org/docs/ad/03-01-07.pdf.

[26] Object Management Group. OMG's MetaObject Facility, January 2006. URL http://www.omg.org/spec/MOF/2.0/PDF/.

[27] Object Management Group. Meta object facility MOF 2.0 query/view/transformation specification, version 1.1. Specification, January 2011. URL http://www.omg.org/spec/QVT/1.1/PDF/.

[28] Ivan Paltor and Johan Lilius. Formalising uml state machines for model checking. In *UML*, pages 430–445, 1999.

[29] Lijun Yu, Robert B. France, and Indrakshi Ray. Scenario-based static analysis of uml class models. In *MoDELS*, pages 234–248, 2008.

[30] Object Management Group. Success stories, 2011. URL http://www.omg.org/mda/products_success.htm/.

[31] Miroslaw Staron. Adopting model driven software development in industry - a case study at two companies. In *Proceedings of Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006, Genova, Italy*, pages 57–72, 2006.

[32] Axel Uhl. Model-driven development in the enterprise. *IEEE Software*, 25(1): 46–49, 2008. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2008.12.

[33] Robert B. France, Sudipto Ghosh, Trung Dinh-Trong, and Arnor Solberg. Model-driven development using UML 2.0: Promises and pitfalls. *Computer*, 39(2): 59–66, 2006. ISSN 0018-9162. doi: http://doi.ieeecomputersociety.org/10.1109/MC.2006.65.

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

[35] C. Lange, M.R.V. Chaudron, J. Muskens, L.J. Somers, and H.M. Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of uml designs. In *Sixth International Conference on the Unified Modelling Language - the Language and its applications (UML'03) Workshop on "Consistency Problems in UML-based Software Development II"*, pages 26–34, San Francisco, California, USA., October 2003.

[36] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, second edition, 2003.

[37] Object Management Group. XML metadata interchange (XMI), version 1.1, 1999. URL http://www.omg.org/docs/ad/99-10-02.pdf.

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

[39] Björn Lundell, Brian Lings, et al. UML model interchange in heterogeneous tool environments: An analysis of adoptions of XMI 2. In *MoDELS 2006, Genova, Italy*, pages 619–630, 2006.

[40] Werner Heijstek and Michel R.V. Chaudron. The impact of model driven development on the software architecture process. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:333–341, 2010. doi: http://doi.ieeecomputersociety.org/10.1109/SEAA.2010.63.

[41] Brian Kernighan and Dennis Ritchie. *The C Programming Language, 2nd Edition*. Prentice Hall, 1988. ISBN 0131103628.

[42] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1990. ISBN 0201514591.

[43] James Gosling, Bill Joy, et al. *The Java Language Specification, 2nd Edition*. Addison Wesley, 2000. ISBN 0201310082.

[44] UIML. OASIS user interface markup language (UIML), version 4.0, 2008. URL https://www.oasis-open.org/committees/download.php/28457/uiml-4.0-cd01.pdf.

[45] XIML. extensible interface markup language: A universal language for user interfaces, 2004. URL http://www.ximl.org/.

[46] Giulio Mori, Fabio Paternò, and Carmen Santoro. Design and development of multidevice user interfaces through multiple logical descriptions. *IEEE Transactions on Software Engineering*, 30(8):507–520, 2004.

[47] J. M. Spivey. *The Z Notation: A Reference Manual, 2nd Ed.*, 1992.

[48] J. Woodcock and J. Davies. *Using Z Specification, Refinement, and Proof*. Prentice Hall Europe, 1996.

[49] J. B. Wordsworth. *Software Development with Z*. Addison Wesley, Boston, MA, 1992.

[50] Susan Stepney, Rosalind Barden, and David Cooper, editors. *Object Orientation in Z*, Workshops in Computing, 1992. Springer.

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

[52] Graeme Smith. Object-Z, 2000. URL http://staff.itee.uq.edu.au/smith/objectz/objectz.html.

[53] Antonio J. Alencar and Joseph A. Goguen. OOZE: An object oriented Z environment. In *ECOOP'91*, pages 180–199, 1991.

[54] Kevin Lano. 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, 1991. Springer-Verlag. ISBN 3-540-19672-2. URL http://dl.acm.org/citation.cfm?id=647278.722479.

[55] Elspeth Cusack and G.-H. Bagherzadeh Rafsanjani. Zest. In *Object Orientation in Z*, Workshops in Computing, pages 113–126. Springer, 1992. ISBN 3-540-19778-8.

[56] Robert Cartwright and Mike Fagan. Soft typing. *SIGPLAN Not.*, 39(4):412–428, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/989393.989435.

[57] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.

[58] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35:26–36, June 2000. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/352029.352035. URL http://doi.acm.org/10.1145/352029.352035.

[59] Jon Bentley. Programming pearls: little languages. *Commun. ACM*, 29:711–721, August 1986. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/6424.315691. URL http://doi.acm.org/10.1145/6424.315691.

[60] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Trans. Program. Lang. Syst.*, 30:31:1–31:40, October 2008. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/1391956.1391958. URL http://doi.acm.org/10.1145/1391956.1391958.

[61] Hans Dockter, Adam Murdoch, Szczepan Faber, Daz DeBoer, Peter Niederwieser, Luke Daley, Steve Appling, and Russel Winder. Gradle - a better way to build, 2011. URL http://www.gradle.org/.

[62] Andrew Oram and Steve Talbott. *Managing Projects with make*. O'Reilly & Associates, Inc., 1993. ISBN 0937175900.

[63] Alistair Cockburn. *Agile software development*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[64] Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001. URL http://agilemanifesto.org/.

[65] Kent Beck. *Extreme Programming Explained*. Addison-Wesley, New York, NY, 2000.

[66] K. Schwaber and M. Beedle. *Agile Software Development with SCRUM*. Pearson Technology Group, 2002.

[67] Tore Dyba and Torgeir Dingsayr. What do we know about agile software development? *IEEE Software*, September/October:6–9, 2009.

[68] L. Layman, L. Williams, and L. Cunningham. Exploring extreme programming in context: An industrial case study. In *Agile Development Conference*, pages 32–41, 2004.

[69] M. Lindvall, V. Basili, B. Boehm, P. Costa, K. Dangle, F. Shull, R. Tesoriero, L. Williams, and M. Zelkowitz. Empirical findings in agile methods. In *XP/Agile Universe 2002*, Chicago, IL, 2002.

[70] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly, 2008. ISBN 9780596516178.

[71] Wesley J. Chun. *Core Python Programming (2nd Edition)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006. ISBN 0132269937.

[72] Dierk Koenig et al. *Groovy in Action*. Manning Publications, Greenwich, CT, USA, 2007. ISBN 978-1932394849.

[73] Dave Thomas, David Hansson, et al. *Agile Web Development with Rails*. Pragmatic Bookshelf, July 2005.

[74] Adrian Holovaty and Jacob Kaplan-Moss. *The Definitive Guide to Django: Web Development Done Right, Second Edition*. Apress, New York, NY, USA, 2009. ISBN 978-1430219361.

[75] Glen Smith and Peter Ledbrook. *Grails in Action*. Manning Publications, Greenwich, CT, USA, 2009. ISBN 978-1933988931.

[76] Wikipedia. Duck typing, 2015. URL https://en.wikipedia.org/wiki/Duck_typing.

[77] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Alef Arendsen, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, and Andy Clement. http://www.springframework.net/doc-latest/reference/html/index.html, 3.0.RC3, 2009. URL http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf.

[78] Robert Hanson and Adam Tracy. *GWT In Action: Easy Ajax with the Google Web Toolkit*. Manning Publications, Greenwich, CT, USA, 2007. ISBN 1933988231.

[79] Jay Goldman. *Facebook Cookbook: Building Applications to Grow Your Facebook Empire*. O'Reilly Media, Sebastopol, CA, USA, 2008. ISBN 978-0596518172.

[80] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, 2nd Ed.* SEI Series in Software Engineering. Addison Wesley, Boston, MA, 2003.

[81] Nelson S. Rosa, George R. R. Justo, and Paulo R. F. Cunha. A framework for building non-functional software architectures. In *Proceedings of the 2001 ACM symposium on Applied computing*, pages 141–147. ACM Press, 2001. ISBN 1-58113-287-5. doi: http://doi.acm.org/10.1145/372202.372299.

[82] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions Software Engineering*, 18(6):483–497, Jun 1992.

[83] Lawrence Chung and Brian A. Nixon. Dealing with non-functional requirements: three experimental studies of a process-oriented approach. In *Proceedings of the 17th international conference on Software engineering*, pages 25–37. ACM Press, 1995. ISBN 0-89791-708-1. doi: http://doi.acm.org/10.1145/225014.225017.

[84] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Using UML to reflect non-functional requirements. In *Proceedings of the 2001 conference of the Centre for Advanced Studies on Collaborative Research*, page 2. IBM Press, 2001.

[85] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Non-functional requirements: from elicitation to modelling languages. In *Proceedings of the 24th international conference on Software engineering*, pages 699–700. ACM Press, 2002. ISBN 1-58113-472-X. doi: http://doi.acm.org/10.1145/581339.581452.

[86] Luis Corral, Alberto Sillitti, and Giancarlo Succi. Mobile multiplatform development: An experiment for performance analysis. *Procedia Computer Science*, 10:736 – 743, 2012. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2012.06.094. URL http://www.sciencedirect.com/science/article/pii/S1877050912004516. MobiWIS 2012.

[87] Andre Charland and Brian Leroux. Mobile application development: Web vs. native. *Communications of the ACM*, 54(5):49–53, May 2011. ISSN 0001-0782. doi: 10.1145/1941487.1941504. URL http://doi.acm.org/10.1145/1941487.1941504.

[88] Dhananjay Nene. Performance comparison - C++ / java / python / ruby / jython / jruby / groovy, 2008. URL http://blog.dhananjaynene.com/2008/07/performance-comparison-c-java-python-ruby-jython-jruby-groovy/.

[89] Sun Microsystems. JSR 220: Enterprise javabeans, version 3.0, 2006. URL http://cds-esd.sun.com/ESD24/JSCDL/ejb/3.0-fr/ejb-3_0-fr-spec-ejbcore.pdf.

[90] Google Inc. The developer's guide, 2010. URL http://developer.android.com/guide/index.html.

[91] Apple Inc. ios 8 for developers, 2014. URL https://developer.apple.com/ios8/.

[92] Mark Pollack, Rick Evans, Aleksandar Seovic, Bruno Baia, Erich Eichinger, Federico Spinazzi, Rob Harrop, Griffin Caprio, Ruben Bartelink, Choy Rim, Erez Mazor, Stephen Bohlen, and The Spring Java Team. Spring reference documentation, 1.3.2, 2011. URL http://www.springframework.net/docs/1.3.2/reference/pdf/spring-net-reference.pdf.

[93] Tom Barrett, Mike Doerfler, and Peter Smulovics. Nhibernate, 2014. URL http://nhibernate.info/.

[94] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. URL http://dblp.uni-trier.de/db/journals/mst/mst2.html#Knuth68.

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

[96] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *2nd OOPSLA'03 Workshop on Generative Techniques in the Context of MDA*, pages 1–17, Anaheim, CA, USA, 2003. URL http://www.softmetaware.com/oopsla2003/czarnecki.pdf.

[97] Al Danial. CLOC, 2013. URL http://cloc.sourceforge.net/.

[98] Zhizhong Jiang, Peter Naudé, and Craig Comstock. An investigation on the variation of software development productivity. *International Journal of Computer and Information Science and Engineering*, pages 461–470, 2007.

[99] Ken Kennedy, Charles Koelbel, Robert Schreiber, Ken Kennedy, Charles Koelbel, and Robert Schreiber. Defining and measuring the productivity of programming languages. *The International Journal of High Performance Computing Applications, (18)4, Winter*, 2004:441–448, 2004.

[100] SonarSource, Inc. Sonarqube, 2015. URL http://www.sonarqube.org/.

[101] David Hovemeyer and William Pugh. Finding bugs is easy. In *Companion to the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '04, pages 132–136, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4. doi: 10.1145/1028664.1028717. URL http://doi.acm.org/10.1145/1028664.1028717.

[102] David Hovemeyer, Bill Pugh, Andrey Loskutov, and Keith Lea. Findbugs, 2015. URL http://findbugs.sourceforge.net/.

[103] Cyril Picat, Gilles Grousset, Denis Bregeon, François Helg, Romain Felden, and Mete Balci. Sonarqube plugin for objective-c, 2015. URL http://www.sonarqube.org/.

[104] Jean-Louis Letouzey. The sqale method definiton document, 2012. URL http://www.sqale.org/wp-content/uploads/2010/08/SQALE-Method-EN-V1-0.pdf.

[105] Jean-Louis Letouzey and Michel Ilkiewicz. Managing technical debt with the sqale method. *IEEE Software*, 29(6):44–51, 2012. ISSN 0740-7459. doi: http://doi.ieeecomputersociety.org/10.1109/MS.2012.129.

[106] Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34, January 1987. ISSN 0362-1340. doi: 10.1145/62139.62141. URL http://doi.acm.org/10.1145/62139.62141.

[107] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, Boston, MA, 1988.

[108] Robert Martin. Object oriented design quality metrics: An analysis of dependencies. *ROAD*, 2(3), September 1995.

[109] Tom Copeland. *PMD Applied*. Centennial Books, Alexandria, VA, USA, 2005. ISBN 0976221411.

[110] Paul E. Black. Karp-rabin, in dictionary of algorithms and data structures [online], 2009. URL http://xlinux.nist.gov/dads//HTML/karpRabin.html.

[111] Longyi Qi and Ryuichi Saito. Oclint, 2015. URL http://oclint.org/.

[112] Parag C. Pendharkar and James A. Rodger. An empirical study of the impact of team size on software development effort. *Inf. Technol. and Management*, 8(4): 253–262, December 2007. ISSN 1385-951X. doi: 10.1007/s10799-006-0005-3. URL http://dx.doi.org/10.1007/s10799-006-0005-3.

[113] Katrina D. Maxwell, Luk Van Wassenhove, and Soumitra Dutta. Software development productivity of european space, military, and industrial applications. *IEEE Trans. Softw. Eng.*, 22(10):706–718, October 1996. ISSN 0098-5589. doi: 10.1109/32.544349. URL http://dx.doi.org/10.1109/32.544349.

[114] Louis Fried. Team size and producitvity in systems development bigger does not always mean better. *Journal of Information Systems Management*, 8(3):27–35, 1991. doi: 10.1080/07399019108964994. URL http://dx.doi.org/10.1080/07399019108964994.

[115] ISBSG. International software benchmarking standards group, 2015. URL http://www.isbsg.org/.

[116] Qin Liu and RobertC. Mintram. Preliminary data analysis methods in software estimation. *Software Quality Journal*, 13(1):91–115, 2005. ISSN 0963-9314. doi: 10.1007/s11219-004-5262-y. URL http://dx.doi.org/10.1007/s11219-004-5262-y.

[117] R Jeffery, M Ruhe, and I Wieczorek. A comparative study of two software development cost modeling techniques using multi-organizational and company-specific data. *Information and Software Technology*, 42(14):1009 – 1016, 2000. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/S0950-5849(00)00153-1. URL http://www.sciencedirect.com/science/article/pii/S0950584900001531.

[118] C.J. Lokan. An empirical analysis of function point adjustment factors. *Information and Software Technology*, 42(9):649 – 659, 2000. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/S0950-5849(00)00108-7. URL http://www.sciencedirect.com/science/article/pii/S0950584900001087.

[119] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, January 2000. ISSN 0098-5589. doi: 10.1109/32.825767. URL http://dx.doi.org/10.1109/32.825767.

[120] Xiaoping Jia and Christopher Jones. Design of adaptive domain-specific modeling languages for model-driven mobile application development. In *10th Annual International Conference on Software Engineering and Applications. (ICSOFT-EA 2015)*, pages 413–418, Colmar, Alsace, France, July 2015. doi: 10.5220/0005557404130418.

[121] Christopher Jones and Xiaoping Jia. An approach for the automatic adaptation of domain-specific modeling languages for model-driven mobile application development. In *Software Technologies*, volume Forthcoming of *Communications in Computer and Information Science*, page Forthcoming. Springer Berlin Heidelberg, 2015. ISBN Forthcoming. doi: Forthcoming. URL Forthcoming.

[122] Matthias Bohlen, Chad Brandon, et al. AndroMDA, 2003. URL http://www.andromda.org/docs/index.html.

[123] The Eclipse Foundation. GMT Project, 2005. URL http://www.eclipse.org/gmt/.

[124] The Eclipse Foundation. ATL Transformation Language, 2005. URL http://eclipse.org/atl/.

[125] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201748045.

[126] xUML Compiler. xUML Compiler- Java Model compiler Based on "Executable UML" profile, 2009. URL http://code.google.com/p/xuml-compiler/.

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

[128] André Ribeiro and Alberto Rodrigues da Silva. Xis-mobile: A dsl for mobile applications. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1316–1323, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2469-4. doi: 10.1145/2554850.2554926. URL http://doi.acm.org/10.1145/2554850.2554926.

[129] Alberto Rodrigues da Silva, Joao Saraiva, Rui Silva, and Carlos Martins. Xis-uml profile for extreme modeling interactive systems. In *Proceedings of the Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, MOMPES '07, pages 55–66, Washington, DC, USA, 2007. IEEE

Computer Society. ISBN 0-7695-2769-8. doi: 10.1109/MOMPES.2007.19. URL http://dx.doi.org/10.1109/MOMPES.2007.19.

[130] A.R. da Silva. The xis approach and principles. In *Euromicro Conference, 2003. Proceedings. 29th*, pages 33–40, Sept 2003. doi: 10.1109/EURMIC.2003.1231564.

[131] Tanja Mayerhofer, Philip Langer, Manuel Wimmer, and Gerti Kappel. xMOF: Executable DSMLs Based on fUML. In *Proceedings of the 6th International Conference on Software Language Engineering*, volume 8225, pages 56–75, 2013.

[132] Viet Cuong Nguyen and X. Qafmolla. Agile development of platform independent model in model driven architecture. In *Information and Computing (ICIC), 2010 Third International Conference on*, volume 2, pages 344–347, June 2010. doi: 10.1109/ICIC.2010.180.

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

[134] Paloma Cáceres, Francisco Díaz, and Esperanza Marcos. Integrating an agile process in a model driven architecture. In Peter Dadam and Manfred Reichert, editors, *Lecture Notes in Informatics*, volume 20, pages 265–270, September 2004. URL http://subs.emis.de/LNI/Proceedings/Proceedings50/GI-Proceedings.50-57.pdf.

[135] Paloma Cáceres, Esperanza Marcos, et al. A mda-based approach for web information system development. In *Proceedings of Workshop in Software Model Engineering*, 2003.

[136] Tiziana Margaria and Bernhard Steffen. Agile it: Thinking in user-centric models. In Tiziana Margaria and Bernhard Steffen, editors, *ISoLA*, volume 17 of *Communications in Computer and Information Science*, pages 490–502. Springer, 2008. ISBN 978-3-540-88478-1.

[137] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. ISSN 1049-331X. doi: http://doi.acm.org/10.1145/505145.505149.

[138] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg, and Indrakshi Ray. UML2Alloy: A challenging model transformation. In *Proceedings of Model Driven Engineering Languages and Systems, 10th International Conference, MoDELS 2007, Nashville, USA*, pages 436–450, 2007.

[139] Martin Gogolla, Fabian Büttner, et al. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69: 27–34, 2007.

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

[141] Edmund M. Clarke, Jeannette M. Wing, et al. Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28:626–643, December 1996. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/242223.242257. URL http://doi.acm.org/10.1145/242223.242257.

[142] D. Hamilton, R. Covington, et al. Experiences in applying formal methods to the analysis of software and system requirements. *Industrial-Strength Formal Specification Techniques, Workshop on*, 0:30, 1995. doi: http://doi.ieeecomputersociety.org/10.1109/WIFT.1995.515477.

[143] Xiaoping Jia. An approach to animating Z specifications. In *Proc. 19th Annual IEEE Int'l Computer Software and Applications Conf. (COMPSAC 1995)*, pages 108–113, Dallas, Texas, USA, August 1995.

[144] Xiaoping Jia, Hongming Liu, et al. A model transformation framework for model driven engineering. In *MSVVEIS-2008*, Barcelona, Spain, June 2008.

[145] Xiaoping Jia, Adam Steele, Hongming Liu, Lizhang Qin, and Christopher Jones. Using ZOOM approach to support MDD. In *Proc. of the 2005 International Conference on Software Engineering Research and Practice (SERP'05)*, Las Vegas, Nevada, USA., June 2005.

[146] Xiaoping Jia, Adam Steele, Hongming Liu, Lizhang Qin, and Christopher Jones. Executable visual software modeling:the ZOOM approach. *Software Quality Journal*, 15(1), 2007.

[147] Hongming Liu, Lizhang Qin, Xiaoping Jia, and Adam Steele. Model transformation based on meta templates. In *Proc. of the 2006 International Conference on Software Engineering Research and Practice (SERP'06)*, Las Vegas, Nevada, USA., June 2006.

[148] Hongming Liu, Lizhang Qin, Xiaoping Jia, and Adam Steele. Model transformation framework supported by ZOOM. In *Proc. of the 2007 International Conference on Software Engineering Research and Practice (SERP'07)*, Las Vegas, Nevada, USA., June 2007.

[149] Hongming Liu and Xiaoping Jia. Model transformation using a simplified metamodel. In *Journal of Software Engineering and Applications*, volume 3, pages 653–660, July 2010.

[150] Rhodes, 2015. URL http://docs.rhomobile.com/en/5.1.1/home.

[151] A. Ribeiro and A.R. da Silva. Survey on cross-platforms and languages for mobile apps. In *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*, pages 255–260, Sept 2012. doi: 10.1109/QUATIC.2012.56.

[152] Zef Hammel, Eelco Visser, et al. mobl: the new language of the mobile web, 2010. URL http://www.mobl-lang.org/.

[153] Convergence Modelling LLC. Canappi, 2011. URL http://www.canappi.com/.

[154] D. Kramer, T. Clark, and S. Oussena. Mobdsl: A domain specific language for multiple mobile platform deployment. In *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, pages 1–7, Nov 2010. doi: 10.1109/NESEA.2010.5678062.

[155] Patricia Miravet, Ignacio Marín, Francisco Ortín, and Abel Rionda. Dimag: A framework for automatic generation of mobile applications for multiple platforms. In *Proceedings of the 6th International Conference on Mobile Technology, Application & Systems*, Mobility '09, pages 23:1–23:8, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-536-9. doi: 10.1145/1710035.1710058. URL http://doi.acm.org/10.1145/1710035.1710058.

[156] Jim Barnett and. State Chart XML (SCXML): State Machine Notation for Control Abstraction, April 2015. URL http://www.w3.org/TR/scxml/.

[157] Florence Balagtas-Fernandez, Max Tafelmayer, and Heinrich Hussmann. Mobia modeler: Easing the creation process of mobile applications for non-technical users. In *Proceedings of the 15th International Conference on Intelligent User Interfaces*, IUI '10, pages 269–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-515-4. doi: 10.1145/1719970.1720008. URL http://doi.acm.org/10.1145/1719970.1720008.

[158] F.T. Balagtas-Fernandez and H. Hussmann. Applying domain-specific modeling to mobile health monitoring applications. In *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, pages 1682–1683, April 2009. doi: 10.1109/ITNG.2009.14.

[159] Henning Heitkötter, Tim A. Majchrzak, and Herbert Kuchen. 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, 2013. ACM. ISBN 978-1-4503-1656-9. doi: 10.1145/2480362.2480464. URL http://doi.acm.org/10.1145/2480362.2480464.

[160] Steffen Vaupel, Gabriele Taentzer, JanPeer Harries, Raphael Stroh, René Gerlach, and Michael Guckert. Model-driven development of mobile applications allowing role-driven variants. In Juergen Dingel, Wolfram Schulte, Isidro Ramos, Silvia Abrahão, and Emilio Insfran, editors, *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 1–17. Springer International Publishing, 2014. ISBN 978-3-319-11652-5. doi: 10.1007/978-3-319-11653-2_1. URL http://dx.doi.org/10.1007/978-3-319-11653-2_1.

[161] The Eclipse Foundation. Eclipse Modeling Framework, 2015. URL https://eclipse.org/modeling/emf/.

[162] Object Management Group. Business process model and notation, version 2.0 beta 1, 2009. URL http://www.omg.org/spec/BPMN/2.0/Beta1/PDF.

[163] Organization for the Advancement of Structured Information Standards (OASIS). Web services business process execution language (ws-bpel) version 2.0, 2007. URL http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[164] The Balsamiq Team. Balsamiq mockups, 2011. URL http://balsamiq.com/products/mockups.

[165] Charles W. Rapp. The state machine compiler, 2000–2009. URL http://smc.sourceforge.net/.

[166] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. *The Program Is the Model*: Enabling transformations@run.time. In Krzysztof Czarnecki and Görel Hedin, editors, *SLE*, volume 7745 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2012. ISBN 978-3-642-36088-6, 978-3-642-36089-3.

[167] Anthony Anjorin, Karsten Saller, Sebastian Rose, and Andy Schürr. A framework for bidirectional model-to-platform transformations. In Krzysztof Czarnecki and Görel Hedin, editors, *Software Language Engineering*, volume 7745 of *Lecture Notes in Computer Science*, pages 124–143. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-36088-6. doi: 10.1007/978-3-642-36089-3_8. URL http://dx.doi.org/10.1007/978-3-642-36089-3_8.