An Empirical Evaluation of AXIOM as an Approach to Cross-platform Mobile Application Development

Christopher Jones and Xiaoping Jia
DePaul University, College of Computing and Digital Media, 243 S. Wabash Ave, 60604, Chicago, IL, U.S.A.

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

Abstract: AXIOM is a domain-specific modeling language for cross-platform mobile applications. AXIOM is based on more general techniques such as Model-Driven Architecture and generates native code for the iOS and Android platforms. Previous small-scale quantitative experiments suggested that AXIOM had the potential to provide significant productivity benefits. We have since conducted a limited set of more complex, mid-scale experiments and analyzed AXIOM's capabilities using both quantitative and qualitative metrics to further define AXIOM's ability to improve developer productivity when building cross-platform mobile applications. In this paper we describe the methodology of our mid-scale experiments and present the findings from source code and SonarQube analyses. We evaluate these findings and discuss what they mean to AXIOM in general. Finally, we look at possible changes to AXIOM's syntax and capabilities.

1 INTRODUCTION

As of July, 2015 there were, by some estimates (sta, 2015), over 1.6 million apps in the Google Play Store and another 1.4 million apps available in Apple's App Store. The popularity of these platforms, and the emergence of new platforms such as the Windows Phone, makes it desirable for mobile application providers to write cross-platform applications.

AXIOM (Jones and Jia, 2015) is an approach to model-driven development (MDD) (Vaupel et al., 2014) that uses the Groovy language as its modeling notation. Rather than depending on approaches that use cross-platform programming languages and virtual machines such as HTML5 and JavaScript (Appcelerator, Inc., 2011; The Apache Group, 2015) (e.g. Appcelerator, Apache Cordova, etc.), AXIOM uses a domain-specific modeling language (DSML) as its representation. Its models undergo a series of transformations and translations during their lifecycle. By using a dynamic DSML we hope to avoid some of the shortcomings of MDD approaches such as the Object Management Group's MDA (Staron, 2006; Uhl, 2008; Whittle et al., 2014; Mussbacher et al., 2014), such as the lack of first-class support for user interface design, while still keeping the focus on writing models instead of code (Frankel, 2003; Selic, 2003). By completely generating native code, we hope to avoid the potential performance impacts exhibited by many cross-platform, virtual machine-based approaches (Charland and Leroux, 2011; Corral et al., 2012).

Our research attempts to answer questions about model-driven development for mobile platforms using the AXIOM approach. Specifically, when compared to native, handwritten code, we want to understand AXIOM's impact on developer productivity and code quality.

2 THE AXIOM LIFECYCLE

The AXIOM lifecycle (Jia and Jones, 2013) has three stages: *Construction*, *Transformation*, and *Translation*. As shown in Figure 1, each stage emphasizes a different model that is gradually transformed into native source code.

During the Construction stage, business requirements, user interface, and application logic are captured in a platform-independent *Requirements model* using AXIOM's DSML. Applications are defined as a set of related views. Views support both platform-independent and platform-specific widgets. Transitions are defined as attributes on the UI controls that trigger them and may have optional guard conditions and actions. AXIOM provides an event model to help deal with common mobile interactions such as screen taps and orientation changes.

Jones, C. and Jia, X.
An Empirical Evaluation of AXIOM as an Approach to Cross-platform Mobile Application Development.
DOI: 10.5220/0005995902640271
In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT 2016) - Volume 1: ICSOFT-EA*, pages 264-271
ISBN: 978-989-758-194-6

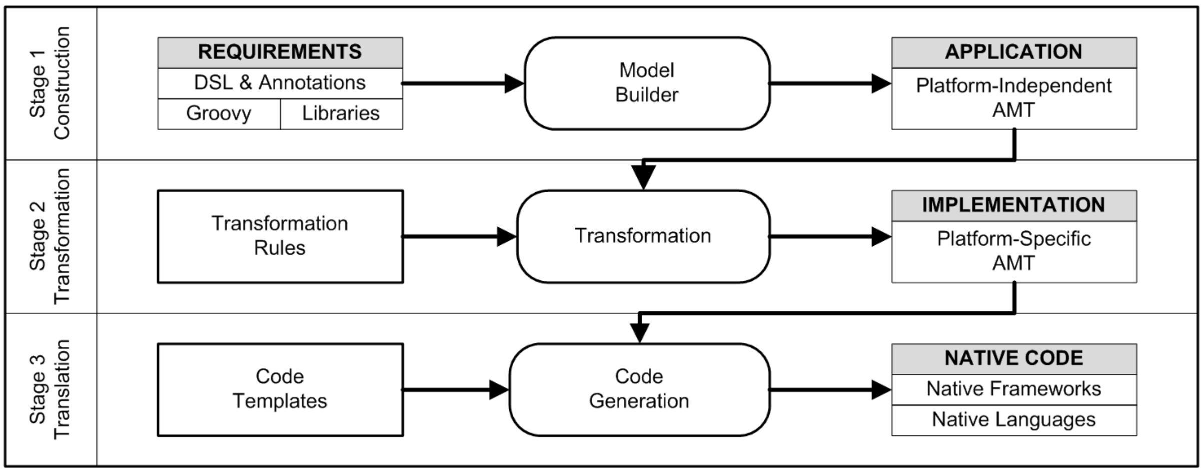

Figure 1: Evolution of AXIOM models.

Once the Requirements model has been defined, it is transformed into an *Application model* using a model builder. The model builder uses a preprocessed representation of the iOS and Android APIs to expose a platform-independent version of the most common widgets between the two platforms. Platform-specific widgets can still be used if needed.

A series of transformation rules converts the Application model into an *Implementation model*. The rules preserve the original APIs so that platform-specificity may be used when appropriate, while abstracting common features into the core DSML. The Implmentation model contains information needed to generate three key aspects of the application's organizational hierarchy: modules, the macro-organizational aspects of the application; resources, the source files for the modules; and fragments, snippets of content used to construct the resources.

The Translation stage converts the Implementation model into native code for the target mobile platforms. AXIOM's code templates capture knowledge and information about the target platform's native language and SDK. The code generation process uses these templates, combined with information within the Implementation model, to completely generate all artifacts such as project, class, and resource files.

3 METHODOLOGY

A proof-of-concept prototype tool was 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.

Using the prototype tool, we conducted two kinds of analyses: small-scale and mid-scale. The results of the small-scale tests have been reported on before (Jones and Jia, 2014). The mid-scale tests compared the code generated from the AXIOM model to hand-written code provided by experienced software developers. In these experiments, five mid-sized applications were developed featuring a variety of navigation and user interactions. Table 1 describes the applications and some aspects of their structure and complexity. These applications were designed to include UI components common to their native platforms, some platform-neutral, others platform-specific. We did not test AXIOM against the more sophisticated components such as GPS. Each mid-scale application was analyzed quantitatively, to gauge AXIOM's relative power and impact on developer productivity, and qualitatively, to gauge its source code organization, issues and issue density, and overall complexity. These metrics were then compared to equivalent hand-written code.

Table 1: Description of Mid-Scale Applications.

App.	Description	Screens
CAR	Shop cars by make and model.	6
CVT	Various unit conversions.	8
EUC	Data on EU member countries.	3
MAT	A matching memory game.	1
POS	A simple point-of-sale system.	8

4 QUANTITATIVE ANALYSIS

4.1 Relative Power

Relative 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 models to the generated SLOC for both iOS and Android. For the comparative evaluation of the SLOC, we used CLOC (Danial, 2013) with Groovy as the source language for AXIOM. The Android and iOS platforms were accounted for using Java and Objective-C respectively. The SLOC counts do not include "non-essential" code such as comments or block delimiters such as braces.

While SLOC is not ideal for 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 unlikely to be a significant factor. Second, we had only a single developer perform the actual coding, which controlled for the variation in ability. Third, had we analyzed story or function points, we would likely have seen clustering of the data because of the comparative simplicity of the applications.

Kennedy's relative power metric, ρ_L, is based on SLOC and measures the relative expressiveness of one language to another (Kennedy et al., 2004):

$$\rho_{L/L_0} = \frac{I(M_{L_0})}{I(M_L)} \tag{1}$$

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.

Table 2 shows the SLOC and relative power for each of the mid-scale experiments. The results show that the AXIOM-generated code can be comparable to, if not smaller than, handwritten Android and iOS code. In the cases of the CVT and POS applications, there was much less handwritten code than generated code.

In many cases the SLOC of the AXIOM-generated and handwritten code are almost the same. However, 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. 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. Some efforts were made to optimize the model further, but this exposed some architectural limitations in the AXIOM 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 handwritten code, the developer took an approach that allowed the application to take better advantage of its data-driven nature.

Table 2: Comparison of relative powers.

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

4.2 Developer Productivity

Developer productivity is influenced by many factors but, as shown by Jiang (Jiang et al., 2007) and others (Fried, 1991; Maxwell et al., 1996; Pendharkar and Rodger, 2007), the two most significant factors in overall productivity are the average team size and development language, accounting for approximately 25% of the variability in Normalized Productivity Delivery Rate (PDR). PDR is 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. In evaluating AXIOM's ability to deliver on its productivity goals, we focus on these two factors.

Jiang's model was constructed by analyzing the project database of the International Software Benchmarking Standards Group (ISBSG) (ISBSG, 2015). This database contains metrics and descriptive information about each of its over 6,700 development and enhancement projects[1]. These projects include 100 types of applications across 30 industry verticals

[1]Jiang's work was based on release 10 of the ISBSG database, which held data on 4,100 projects.

spanning 26 countries. This project database is the basis for other analyses of software productivity such as those done by Liu (Liu and Mintram, 2005), Jeffery (Jeffery et al., 2000), and Lokan (Lokan, 2000). This breadth of projects makes Jiang's model well-suited for our analysis.

Jiang's complete model captures several 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). Some multi-technique factors were included as well, that is, the factor applied only if both of a pair of techniques were used. Except for team size, each factor, $I(X)$, takes a boolean value: 1 if X was used and 0 if it was not.

We held many factors constant in our experiments. The team size was always 1 and we treated AXIOM as a 4GL rather than an application generator since it requires up-front development in the DSML first. Furthermore, we did not use event or business modeling or formal regression testing. Finally, we consider both Android and iOS to be for the "PC" platform, a decision we justify in greater detail below. This reduces Jiang's equation to that shown in equation 2:

$$\begin{aligned} ln(PDR) \quad = \quad & -0.463 * I(3GL) \\ & -1.049 * I(4GL) \\ & -0.269 * I(PC) \\ & -0.403 * I(OO) \\ & +2.651 \end{aligned} \tag{2}$$

We emphasize that there is only one dominant and variable factor remaining: the type of language being used to develop the application. For native development we consider both Objective-C and Java as 3GL languages. When we apply the model for the AXIOM and handwritten styles of development we find that PDR_{Native} is 4.554 and PDR_{AXIOM} is 1.097. We thus find that PDR_{AXIOM} is slightly more than 4-times greater than PDR_{Native}. However, 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. 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, accounts for almost all 100% of the variability in $ln(PDR)$ in our analyses. We thus find that

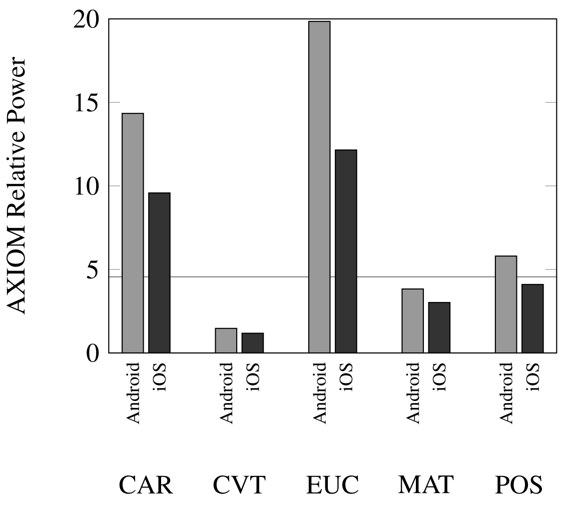

Figure 2: Comparison of relative power.

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 for the handwritten code from Table 2, shown in Figure 2, we find significant variance. The median representational power is 4.10 for iOS and 5.80 for Android, which is consistent with Jiang's model's prediction of 4.554, shown as the horizontal line. One 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. That particular AXIOM model was naive, resulting in code bloat. A more efficient, but semantically more complex model was devised using arrays of closures, but the AXIOM prototype was unable to interpret it, a limitation of tool rather than of technique.

5 QUALITATIVE ANALYSIS

For the qualitative analysis we used SonarQube[2], an open-source, static code quality analysis tool that is popular with software developers. SonarQube performs static source code analysis using common plugins such as FindBugs (Hovemeyer et al., 2015). 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 does the standard Java analysis, which was a reason-

[2]SonarQube was formerly called Sonar.

able second choice. For iOS we used an open-source Objective-C analyzer called *OCLint*.

We eliminated from consideration ancillary files such as XML files that are required to execute the application, but which are often generated by IDEs and supporting tools. We instead focus only on the Java or Objective-C code, which is where most qualitative issues will be found.

5.1 Issues and SQALE

SonarQube performs its analysis by applying individual rules to the source code based on the language being analyzed. For projects with multiple languages, SonarQube can analyze each language according to its own set of rules and then aggregate the results. The rules are based on the SQALE Quality Model (Letouzey, 2012; Letouzey and Ilkiewicz, 2012), which organizes the non-functional requirements that relate to code quality and technical debt into eight SQALE characteristics: Testability, Reliability, Changeability, Efficiency, Security, Maintainability, Portability, and Reusability. Each rule is given a severity indicating how much of an impact its violation may have on the finished application. 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. For example The most common issue identified during our analysis, "Hardcoded Text", is considered a minor issue; it is undesirable and does not represent a latent defect, but it does impact our ability to update and maintain that code.

For our analysis we used only two categories of issue severity: Major and Minor. Table 3 shows the SonarQube and OCLint mappings to these categories.

Table 3: Mapping of issue severities.

Analysis	SonarQube	OCLint
Major	Blocker, Critical, Major	Major
Minor	Minor, Info	Minor, Info

Given the number of issues identified for an application, and given the count of an application's SLOC, we calculate the number of issues per SLOC, or *issue density*. Figure 3 shows the issue densities for the handwritten (AND-H, IOS-H) and generated code (AND-G, IOS-G).

In most cases, the AXIOM-generated code does not fare as well as its handwritten counterpart. When comparing the generated and handwritten 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 handwritten code. This is consistent with AXIOM's template-based nature since any issues present in the template, or in the translation algorithm, will be injected into the generated code.

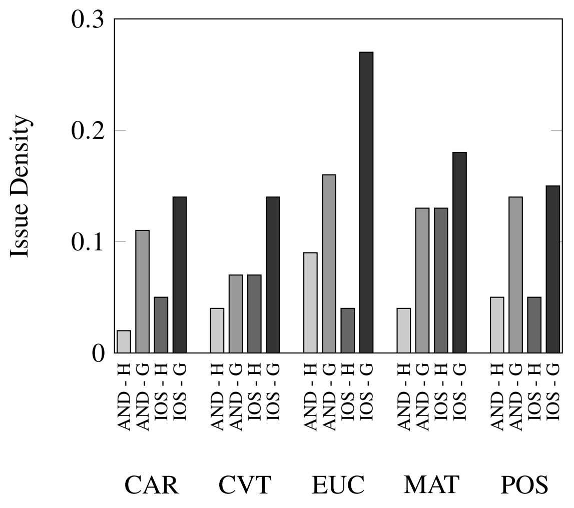

Figure 3: Comparison of issue densities.

Issue density by itself is only one aspect of the analysis. We must also consider the estimated time, cost, and risk to remediate the issues once they have been found. For the handwritten code we must address each instance of the issue, and there is no guarantee that new instances of the issue will not be introduced during subsequent development. Furthermore, the fixes do not propagate to other applications, so they must be updated separately. With AXIOM, the remediation of a translation template will likely take longer than with the handwritten code. However, once the update to the template is complete, the fix will be applied and the issue will not be re-introduced. Because the fixes are cumulative, AXIOM represents a better long-term investment for the remediation effort. As industry best practices evolve, those practices can be incorporated into the code templates more readily than if we must refactor each application by hand.

5.2 Complexity

The final set of qualitative metrics involve code complexity. SonarQube provides an analysis of cyclomatic complexity for the overall application. While this complexity is arguably not 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.

Figure 4 shows that, in most cases, the AXIOM-generated code has a much higher overall cyclomatic complexity. However, the raw SonarQube data showed that at the function, class and file levels, the complexity is often less than that of the handwritten code. This disparity is explained because the AXIOM prototype tends to generate more code than human developers. It thus stands to reason that the overall 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. Thus these complexity values can be improved by enhancing the translation templates and transformation rules to be more parsimonious.

6 DISCUSSION

Based on the mid-scale experiments, AXIOM has generally been observed to reduce the number of SLOC to be written when compared to the equivalent native application even though the final, generated SLOC might ultimately be greater, an admittedly undesirable state. Our 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 as well as the fact that its transformation rules and templates are reusable across applications. This is reflected in the comparative representational power values from Table 2. However, some

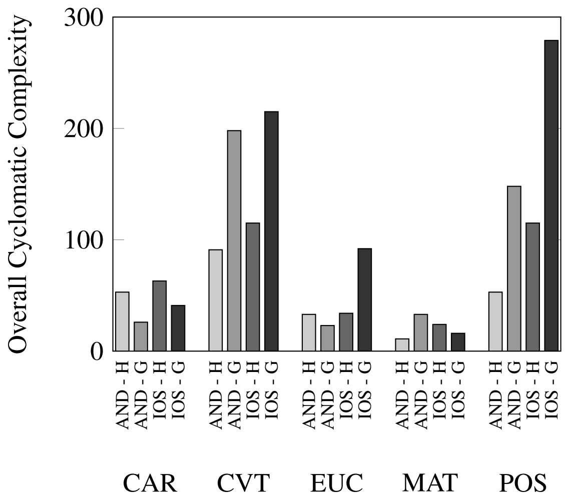

Figure 4: Comparison of cyclomatic complexities.

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 treated each platform as equivalent to general PC development, but it is uncertain if this is the case. Regardless, we expect that the coefficients associated with the platform would contribute equally in both cases and thus not have a material impact on the overall productivity difference described by Jiang's model. 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. Third, this analysis does not include the effort required to build either the templates or transformation rules. Including those factors would likely reduce the overall impact of AXIOM on productivity for our mid-scale tests. However, because the transformation rules and templates are generally intended to be re-used across many applications, we think it likely that, over time, the cost of their development would be amortized and the productivity gains would approach the more ideal case that we have outlined.

There are other reasons why 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. In this evaluation, we had only a single developer, which eliminated that variability. However, the fact that there was only a single set of trials obviously makes our generalizations to the broader mobile developer population preliminary at best.

Second, the trials themselves did not exercise the full suite of capabilities of many moderns mobile devices including the position, direction, or motion sensors. The focus was instead on the

Third, the DSML was in a state of flux when the mid-scale evaluations were being conducted. This meant that the developer who produced the AXIOM models might have encountered problems that caused them to produce a less optimal model. Similarly as new features of the language were introduced, the developer may not have gone back and incorporated them into any already complete models. We found evidence of this in two of the mid-scale experiments. In an earlier version of the CAR application, the AXIOM model was significantly larger because the modeler provided a naive model. Based on SonarQube analysis, the resulting source code was approximately 1,300 lines long. After updating the AXIOM model to use newer features of the DSML, the generated code was reduced to its present level of closer to 500 lines of code. A similar problem was discovered with the EUC application with similar results.

This suggests that one of the challenges with this kind of MDD approach 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 to prevent bloating of the final code. 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.

There are several ways in which AXIOM's DSML could be improved. For example, AXIOM is platform-independent, but not particularly abstract. Raising its level of abstraction is one important way in which AXIOM could be extended. 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. AXIOM exhibits many of its Groovy roots, and much of the code uses native Groovy syntax. AXIOM could also incorporate 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, making it resemble an Architecture Description Language (ADL) (Medvidovic and Taylor, 2000). These techniques would provide more power to the modeling notation and enable modelers to avoid the use of Groovy syntax for common modeling constructs.

7 CONCLUSIONS

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, a multi-pass transformation process, and template-driven code generation.

The five mid-scale tests used to test AXIOM's impact on developer productivity and code quality were designed with a combination of platform-neutral and platform-specific widgets, but did not attempt to include all possible mobile widgets and capabilities. Although only a single developer performed these tests, the results of both our quantitative and qualitative analyses are suggestive.

Based on the analysis using Jiang's productivity metric, AXIOM's single, platform-independent DSML can deliver productivity that is about four times greater than producing the equivalent code by hand. An analysis of the SLOC required for the AXIOM models compared to hand-written code produced by 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.

AXIOM's impact on code quality is ambiguous. The AXIOM prototype 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 case-by-case fixes. 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 be a drain on productivity.

The AXIOM DSML is young and can benefit from optimization. In particular it would be useful to incorporate mobile application patterns and idioms 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.

REFERENCES

(2015). Number of Apps Availabile in Leading App Stores as of July 2015.

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

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

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

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

Frankel, D. S. (2003). *Model-Driven Architecture: Applying MDA to Enterprise Computing*. John Wiley & Sons, New York, NY.

Fried, L. (1991). Team size and productivity in systems development bigger does not always mean better. *Journal of Inf. Systems Manage.*, 8(3):27–35.

Hovemeyer, D., Pugh, B., Loskutov, A., and Lea, K. (2015). Findbugs. http://findbugs.sourceforge.net/.

ISBSG (2015). Int. software benchmarking standards group. http://www.isbsg.org/.

Jeffery, R., Ruhe, M., and Wieczorek, I. (2000). A comparative study of two software development cost modeling techniques using multi-organizational and company-specific data. *Inf. and Softw. Technology*, 42(14):1009 – 1016.

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

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

Jones, C. and Jia, X. (2014). The AXIOM model framework: Transforming requirements to native code for cross-platform mobile applications. In *9th Annual Int. Conf. on Evaluation of Novel Approaches to Software Engineering, (ENASE 2014)*, pages 26–37, Lisbon, Portugal.

Jones, C. and Jia, X. (2015). Using a domain specific language for lightweight model-driven development. In *Evaluation of Novel Approaches to Software Engineering*, volume 551 of *Communications in Computer and Information Science*, pages 46–62. Springer Intl. Publishing.

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

Letouzey, J.-L. (2012). The sqale method definiton document. http://sqale.org/wp-content/uploads/2010/08/SQALE-Method-EN-V1-0.pdf.

Letouzey, J.-L. and Ilkiewicz, M. (2012). Managing technical debt with the sqale method. *IEEE Software*, 29(6):44–51.

Liu, Q. and Mintram, R. (2005). Preliminary data analysis methods in software estimation. *Software Quality Journal*, 13(1):91–115.

Lokan, C. (2000). An empirical analysis of function point adjustment factors. *Information and Software Technology*, 42(9):649 – 659.

Maxwell, K. D., Van Wassenhove, L., and Dutta, S. (1996). Software development productivity of european space, military, and industrial applications. *IEEE Trans. Softw. Eng.*, 22(10):706–718.

Medvidovic, N. and Taylor, R. N. (2000). A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93.

Mussbacher, G., Amyot, D., Breu, R., Bruel, J.-M., Cheng, B., Collet, P., Combemale, B., France, R., Heldal, R., Hill, J., Kienzle, J., Schttle, M., Steimann, F., Stikkolorum, D., and Whittle, J. (2014). The relevance of model-driven engineering thirty years from now. In *Model-Driven Engineering Languages and Systems*, volume 8767 of *Lecture Notes in Computer Science*, pages 183–200. Springer International Publishing.

Pendharkar, P. C. and Rodger, J. A. (2007). An empirical study of the impact of team size on software development effort. *Inf. Technol. and Manag.*, 8(4):253–262.

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

Staron, M. (2006). Adopting model driven software development in industry - a case study at two companies. In *Proc. of Model Driven Engineering Languages and Systems, 9th Int. Conf., MoDELS 2006, Genova, Italy*, pages 57–72.

The Apache Group (2015). Apache cordova. https://cordova.apache.org/.

Uhl, A. (2008). Model-driven development in the enterprise. *IEEE Softw.*, 25(1):46–49.

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

Whittle, J., Hutchinson, J., and Rouncefield, M. (2014). The state of practice in model-driven engineering. *IEEE Softw.*, 31(3):79–85.