DYNAMIC LANGUAGES AS MODELING NOTATIONS IN MODEL DRIVEN ENGINEERING

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

Keywords: Model driven engineering, Dynamic language, Modeling language.

Abstract: There has been a gradual but steady convergence of dynamic programming languages with modeling languages. Modern dynamic languages such as Groovy and Ruby provide for the creation of domain-specific languages that can provide a level of abstraction comparable to that of modeling languages such as UML. This convergence makes dynamic languages suitable as modeling languages but with benefits that traditional modeling languages do not provide. One area that can benefit from this convergence is model driven engineering. By using a dynamic language as an augmentation to MDE's traditional UML notation, it is possible to create models that are executable, exhibit flexible type checking, and which provide a smaller cognitive gap between business users, modelers and developers.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Programming languages and modeling languages are gradually but steadily converging. Languages such as Groovy and Ruby allow for the creation of domain-specific languages (DSLs), which are languages that describe a particular problem domain. DSLs can provide a level of abstraction comparable to modeling languages like UML.

UML is itself a DSL that describes software models. While it has always captured critical information such as the structural and behavioral aspects of software, UML has evolved to include the Object Constraint Language (OCL), the MetaObject Facility (MOF) and XML Metadata Interchange (XMI). This evolution has made UML and its related standards the backbone of another, more ambitious approach to software development: model driven engineering (MDE).

MDE has the potential to deliver great cost savings in software development by automating many time-consuming and error-prone tasks such as detailed design, coding, and testing. These savings are compounded when one application can be deployed across diverse operating systems and hardware platforms, such as is the case with mobile applications.

Despite its potential benefits, MDE faces significant challenges to its adoption including: limitations of UML; inadequate tool support; model transformation complexity; and its apparent incompatibility with popular Agile software development methodologies.

In this paper we describe MDE as it exists today and discuss the possible benefits of adopting a new modeling notation that is based both on a dynamic language and UML. Section 2 gives an overview of MDE and dynamic languages. Section 3 examines the challenges facing MDE in greater detail. Section 4 introduces dynamic languages and describes their use in Agile methodologies. In section 5 we propose an approach, called AXIOM, that brings MDE and dynamic languages together and Section 6 examines the potential benefits of that synergy. Finally, section 7 provides our conclusions and closing thoughts.

2 BACKGROUND

2.1 Model Driven Engineering

Model driven engineering builds software systems by defining platform independent models (PIMs), which capture the compositions and the core functionalities of the system in a way that is independent of implementation concerns. The PIMs are transformed into platform specific models (PSMs) from which executable code is ultimately generated. MDE shifts the development focus away from writing code (Selic, 2003) and toward the development of visual models such as those in UML and its profiles.

Jia X. and Jones C..
DYNAMIC LANGUAGES AS MODELING NOTATIONS IN MODEL DRIVEN ENGINEERING.
DOI: 10.5220/0003607102200225
In *Proceedings of the 6th International Conference on Software and Database Technologies* (ICSOFT-2011), pages 220-225
ISBN: 978-989-8425-77-5
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

Some key characteristics of MDE include:

- An emphasis on visual modeling that allows components and relationships to be viewed in a more comprehensible form. This is critical in dealing with complex, large-scale software systems.
- The use of a high-level abstract modeling and constraint language like UML to define platform independent models. This allows the software architecture, design, and logic to be defined completely and separately from implementation concerns.
- Customizable model transformations that allow for flexibility in defining the PIMs and realizing their corresponding PSMs.

2.2 Dynamic Languages

While MDE is based on UML, most software development still relies on programming languages such as Java or Groovy. Dynamic languages like Groovy often infer data types based on program execution rather than requiring that those type be explicitly declared in the program source code. This is fundamentally different from static languages, where data types must be explicitly declared before the code is even compiled. Dynamic languages give developers the same kind of flexibility that modelers have enjoyed with UML, which is the ability to define only what is needed, when it is needed.

Dynamic languages are increasingly popular, especially in software development projects using Agile methodologies. This is in part because of their expressive syntax, but also because they often support a high level of abstraction either through the use of abstract data types or through the definition of DSLs. Such DSLs lend themselves to writing code that is expressive and understandable to developers and modelers alike.

3 CHALLENGES OF MDE

MDE has proven effective and successful in many industrial enterprise applications targeting mature middleware platforms with widely adopted common standards such as JEE, .NET, and SOA (Object Management Group, 2010). Nevertheless there remain critical challenges to its widespread industry adoption (Uhl, 2008; Staron, 2006) including:

- Limitations of UML and its related standards.
- Inadequate tool support.
- Model transformation complexity.
- Incompatibility with agile methodologies.

3.1 Limitations of UML

UML and its related standards contain some inherent weaknesses (France et al., 2006; Henderson-Sellers, 2005). First, UML models can be incomplete and/or inconsistent (Lange et al., 2003) and the OCL, which provides a degree of formalism to those models, has an awkward, non-intuitive syntax that limits its practical usefulness. Second, UML is not executable, which means that model verification is limited to inspections and model checkers. While there are executable extensions to UML such as xUML (Mellor and Balcer, 2002), support for such extensions varies widely. Third, UML lacks powerful modeling frameworks and libraries. While specific frameworks and libraries are always language and platform specific and are therefore ill-suited for platform independent notations such as UML, comparable frameworks and libraries are available for most platforms and languages. For example, Java supports the Java Persistence API (JPA) for object-relational mapping, whereas Ruby supports Active Record. So defining platform independent models without the benefits of such frameworks and libraries is a huge and unnecessary obstacle.

3.2 Inadequate Tool Support

Another obstacle to the adoption of MDE is the limited availability and effectiveness of supporting tools. For example, issues that are readily addressed in source code control systems can be problems for models. The merging of changes, conflict resolution and the ability to revert to earlier versions of the model are all examples of problems that require tool-specific solutions.

A related problem is that UML enjoys only limited support for its XMI standard, which provides for the interchange of UML models across different UML modeling tools. One recent study found that the success rate for such interchange among leading UML tools was less than 5% (Lundell et al., 2006).

3.3 Model Transformation Complexity

The process of creating the final executable from MDE models is non-trivial and typically requires a custom code generator. If the code is to be generated completely from the model, then the transformation often requires significant development (Heijstek and Chaudron, 2010). On the other hand, if the generated code is only a skeleton to be fleshed out by developers, then the model and the implementation will rapidly diverge. Round-trip engineering of mod-

els from code is often inadequate in practice, so this divergence must be avoided if the model isn't to be a throwaway artifact. Since a key tenet of MDE is the ability to generate multiple PSMs from a single PIM, this challenge is significant.

3.4 Incompatibility with Agile Software Development Methodologies

At first glance, MDE appears to be incompatibile with Agile software development approaches like *eXtreme Programming (XP)* and *Scrum*. Agile software development methodologies adopt the principles articulated in the *Agile Manifesto* (Beedle et al., 2001) including: embracing changes; refactoring throughout the development process; using working software as the primary measure of progress; and developing iteratively and incrementally. Agile methodologies reject most software artifacts if they do not directly contribute to the final, working application code, an approach that is at odds with the basic premise of MDE, which is that models are everything.

There have been several attempts to combine agile and model driven development. Two such examples are Agile Model Driven Development (AMDD) (Ambler, 2009) and Continuous Model Driven Engineering (CMDE) (Margaria and Steffen, 2009).

Agile Model Driven Development (AMDD) is an iterative software development process that shares the notations and tools commonly used in MDE, but deviates from traditional MDE in one notable way: AMDD keeps the code as the focus of the development effort. While AMDD supports sophisticated code-generating models, there is an explicit assumption that only 5-10% of models can benefit from such treatment. Most AMDD models are strictly passive, serving only as a communication medium, and are not used to generate code.

Continuous Model Driven Engineering (CMDE) combines model driven design, extreme programming, and process modeling into *eXtreme model-driven design (XMDD)* (Margaria and Steffen, 2008). XMDD deviates from traditional MDE by using process models as the means of eliciting requirements and behavior, which allows for simulation and process execution via BPM engines. The process models can be decorated as needed with temporal constraints, symbolic type information and cross-cutting concerns and thus serve the same purpose as UML and OCL.

Both AMDD and CMDE alter the core MDE approach. In AMDD, the use of models is relegated to a supporting role rather than being the primary focus. In CMDE, the model retains a central role, but the core UML foundation has been replaced with a process model. Ideally, we would like an approach that retains the benefits of UML while compensating for its deficiencies.

4 THE PROMISE OF DYNAMIC LANGUAGES

4.1 Properties of Dynamic Languages

Many properties that are essential to modeling languages are present in modern dynamic languages. While the specifics of each dynamic language are different, most of them share some common traits such as a very succinct syntax, being interpreted, and being dynamically or optionally typed. They often provide abstract data types such as sets, lists, maps, iterators, generators, and closures, and thus support high levels of abstraction. The ability to use highly abstract data types and to provide little or no type information is extremely useful to developers because it permits them to define what they know, when they know it, and to fill in the gaps later as more information becomes available.

Modern dynamic languages are highly extensible by virtue of their support for meta-programming and domain-specific languages. This property allows them to provide even greater levels of abstraction that can help reduce the cognitive gap between software designs and their implementations.

Another property of dynamic languages is their ability to use existing frameworks and libraries. Some dynamic languages such as Groovy are based on Java and can use not only their own libraries and frameworks, but all of the libraries and frameworks available to Java in general. This makes these languages extremely powerful and reduces the amount of handwritten code to be provided. Indeed, these libraries and frameworks allow developers to focus on the business-specific aspects of an application and to defer the lower-level details until necessary.

4.2 Dynamic Languages in Agile Software Development

While Agile methodologies can be applied to software development using any language, of particular interest is one branch of Agile methods using dynamic languages and their associated frameworks. This style of Agile development, pioneered by Ruby on Rails, has gained a great deal of popularity. However, Agile development methods using dynamic languages have not been widely adopted in critical en-

terprise applications. One reason is that the runtime environments are not yet considered sufficiently mature, robust, and trustworthy as compared to JEE or .NET. Furthermore, there are inherent risks with dynamic typing and meta-programming, which allow new kinds of application defects to be introduced and which may significantly impact application performance.

5 THE AXIOM APPROACH

Our approach, called AXIOM, uses a dynamic language to augment UML as the core MDE modeling notation. The prototype of AXIOM uses Groovy, but in principle any modern dynamic language could be used.

5.1 Core Modeling Notation

Groovy is executable and supports many of the features found in traditional specification and constraint languages such as Z and OCL including: 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. Groovy already supports the important features necessary for adequately defining constraints in object-oriented models. Groovy is fully compatible with the Java language and JVM, a mature runtime environment with extensive libraries and frameworks and a broad install base. Groovy is highly extensible because of its meta-programming capabilities, which allows additional features to be added to the base language in the form of DSLs.

AXIOM uses only the subset of Groovy's features that are suited to modeling. A few simple extensions are added to Groovy to provide useful modeling concepts that are not natively supported, such as associations and design contracts. Groovy's annotation mechanism is used to support MDE stereotypes and profiles. In addition, we augment the Groovy language in two notable ways. First, we provide dual visual and textual representations that are consistent with a subset of UML. Second, we add a state machine mechanism based on the UML state diagram to act as a coordinator for the various application components.

5.1.1 Visual Representation

Since visual modeling is one of the cornerstones of MDE, our first major augmentation to the Groovy language is to have complete and interchangeable visual and textual representations of all models. The concrete syntax of both visual and textual representations will be formally defined. The textual representation is human readable Groovy code. The visual representations are based on, and are consistent with, UML class and state diagrams.

5.1.2 Dynamic Modeling

The second major augmentation to Groovy is to provide visual formalism by defining the UML state diagram as a DSL. State diagrams are used to 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. State diagrams are also amenable to an analysis of certain concurrency-related properties (Yu et al., 2008; Paltor and Lilius, 1999). Furthermore, using state diagrams to define behavioral logic is cognitively more natural than using linear textual code and allows us to retain one of the major benefits of UML, which is improved communication between business users, modelers and developers.

5.2 Prototype

Preliminary work on the AXIOM models and tools is already underway. As shown in Figure 1, simple AXIOM models can currently be transformed into implementations in Java and Objective-C. Our goal is to fully develop the model transformation tool so that it can be 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.

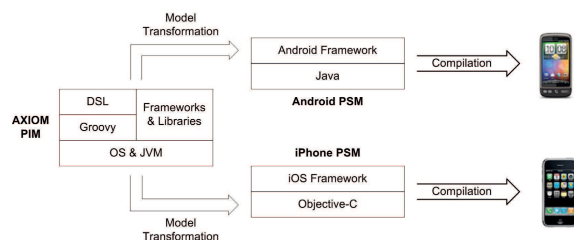

Figure 1: AXIOM development approach.

The AXIOM modeling tool supports two perspectives, one for the textual representation and one for the visual representation. Models can be created and manipulated either textually or visually and alternated between the two. Switching between the two perspec-

tives automatically triggers the conversion from one representation to the other.

AXIOM is based on the prior research of the ZOOM (Z-based Object-Oriented Modeling) (Jia et al., 2007; Jia et al., 2008) and HRMT (Hierarchical Relational Metamodel Transformation) (Liu and Jia, 2010) projects. ZOOM is an extension of OCL that is based on Z. HRMT, based on ZOOM, is a template-based model transformation framework that uses a simplified metamodel.

6 DISCUSSION

6.1 Benefits

The AXIOM approach of augmenting UML with a dynamic language makes MDE accessible to a broader range of applications than MDE with UML alone. This is because AXIOM addresses the challenges of MDE by combining the model-centricity of UML with the flexibility of a dynamic language.

6.1.1 Flexible Typing

Most dynamic languages are optionally typed, so developers can provide as much or as little type information as desired. It is possible to imagine tools using configurable type inferencing that would permit modelers and developers to define a scale of typing strictnesses. Such a scale could be used within the models to ensure that enough type information has been provided to avoid common type-checking errors. Minimal type information could be provided during the early stages of a project when requirements are changing rapidly. More type information could then be added as an application matures and code quality becomes of greater importance.

6.1.2 Model Executability

In addition to being optionally typed, dynamic languages are executable. This means that DSLs built using those languages, and thus the models based on those DSLs, are also executable. This executability can reduce the cycle times between modeling, implementation and testing. Similarly, errors in the models can be discovered and addressed quickly. This in turn facilitates MDE that is targeted to smaller, more Agile software development efforts, where minimal time is spent in the development of classic design artifacts in favor of working, executable code.

6.1.3 Frameworks & Libraries

Because all model code is native dynamic language code, the models themselves can take advantage of the vast assortment of frameworks and libraries available to the underlying programming language and platform. The ability to draw upon such artifacts can significantly reduce the time needed to create and update the models. In addition, the code generated from the PSM will be improved because it can draw upon industry-standard technologies and techniques. For example, Groovy can fully leverage all the frameworks and libraries available for Java, such as EJB, Spring, Android and others. The availability of this extensive collection of runtime frameworks and libraries removes one of the major obstacles to MDE.

6.1.4 Reduced Cognitive Gap

The use of dynamic languages as modeling languages also ensures less of a cognitive gap between business users, modelers and developers, making it more attractive to smaller-scale development efforts or those projects following an Agile methodology. This benefit is derived in two ways. First, a DSL is built that allows for high-level reasoning about the application. Second, because the DSL and all model representations are in a single language, it is comparatively simple for someone fluent in that language to understand the DSLs and their models as well as any lower-level code that is required to transform those models. This approach permits both kinds of MDE: completely generative, where all of the code comes from the model, and partially generative, where only skeleton code is generated that must then be completed by a developer. Because only a single language is used, it may also be easier to provide round-trip engineering if necessary.

6.1.5 Improved Model Interchange

The use of a dynamic language as a modeling language also helps eliminate one of the most troublesome aspects of MDE, which is the need for model interchange using XMI and the varying degrees to which it is supported. By using a programming language the interchange becomes one of syntax rather than of model representation. Sophisticated modeling tools are not required, even though they will certainly be useful, and issues such as the merging of, reverting of and overall control of the modeling files can easily be handled by existing source code control systems like CVS, Subversion or Git.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

6.2 Limitations & Challenges

The AXIOM approach has several limitations and challenges. First, it deviates slightly from the classic MDE approach. Second, it may prove challenging to achieve true platform independence even though Groovy compiles to JVM bytecode and is thus portable. This may prove particularly challenging when the target platform is not Java-based as is the case with the iPhone OS. Third, there is limited tool support for this approach. While many tools support UML models or Groovy code in isolation, AXIOM requires that they be unified and interchangeable.

7 CONCLUSIONS

Through the use of dynamic languages, model driven engineering becomes more feasible than if it remains bound solely to UML. Existing MDE models suffer from UML limitations, a lack of adequate tool support and limited access to useful frameworks and libraries.

AXIOM's goal is to evolve MDE by augmenting UML with a dynamic language and while there are challenges to overcome, the benefits provided by this synthesis are significant. Many dynamic languages support high degrees of abstraction in the form of DSLs. Models based on these languages have access to all of the language's libraries and frameworks. Tool support for these models is readily available in the form of existing text editors and source code control systems, which provide well understood means of model interchange. Finally, dynamic languages yield executable models, which can be rapidly validated and verified. By retaining the key visual elements of UML, these models remain accessible to modelers and developers alike.

The AXIOM approach will yield models that are faster to develop, easier to verify, and which will be compatible with mainstream Agile methodologies. If AXIOM's promise is realized, it has the potential to deliver significant cost savings, particularly for cross-platform application development, while improving overall application quality.

REFERENCES

Ambler, S. (2003-2009). Agile model driven development (AMDD): The key to scaling agile software development. http://www.agilemodeling.com/essays/amdd.htm/.

Beedle, M. et al. (2001). Manifesto for agile software development. http://agilemanifesto.org/.

France, R. B. et al. (2006). Model-driven development using UML 2.0: Promises and pitfalls. *Computer*, 39(2):59–66.

Heijstek, W. and Chaudron, M. R. (2010). The impact of model driven development on the software architecture process. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:333–341.

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

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

Jia, X. et al. (2008). A model transformation framework for model driven engineering. In *Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, 2008*, Barcelona, Spain.

Lange, C. et al. (2003). An empirical investigation in quantifying inconsistency and incompleteness of uml designs. In *Incompleteness of UML Designs, Proceedings of the IEEE Workshop on Consistency Problems in UML-based Software Development, 6th Intl. Conference on UML*, pages 26–34.

Liu, H. and Jia, X. (2010). Model transformation using a simplified metamodel. *Journal of Software Engineering and Applications*, 3(7):653–660.

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

Margaria, T. and Steffen, B. (2008). Agile it: Thinking in user-centric models. In *ISoLA 2008*, pages 490–502.

Margaria, T. and Steffen, B. (2009). Continuous model-driven engineering. *Computer*, 42(10):106–109.

Mellor, S. J. and Balcer, M. (2002). *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley, Boston, MA, USA.

Object Management Group (2010). Success stories. http://www.omg.org/mda/products_success.htm/.

Paltor, I. and Lilius, J. (1999). Formalising uml state machines for model checking. In *UML*, pages 430–445.

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

Staron, M. (2006). Adopting model driven software development in industry - a case study at two companies. In *MoDELS 2006*, pages 57–72.

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

Yu, L. et al. (2008). Scenario-based static analysis of uml class models. In *MoDELS 2008*, pages 234–248.