Model Driven Architecture for Software Application Generation

Ryan Clark a, Petraq Papajorgji b

a SOSY, Inc., San Francisco, CA, USA, rclark@sosyinc.com
b IFAS Information Technologies Office, University of Florida, Gainesville, FL, USA, petraq@ifas.ufl.edu

Abstract

Modeling software system is a common aid to the software development process. There are many advantages to modeling, however, these are limited by the fact that modeling is typically only descriptive, and can not be used to generate code. Model Driven Architecture (MDA), an OMG initiative, allows for the creation of declarative models that can be used as the specification for the generation of entire software systems. Using such an approach has significant advantages, especially with regard to agricultural and environmental simulations which are often complex and evolve over time.

This paper presents an overview of Model Driven Architecture and how it can be used to drive application generation. Platform Independent Model, Platform Specific Model, and transformation definition concepts are discussed, the MDA process is reviewed, and key advantages of the MDA approach are presented, with special emphasis on the generation of applications that simulate agricultural systems.

Key words: MDA, UML, code generation.

1 Software Modeling

1.1 Why Model?

Descriptive models of software and the real systems they represent have long been used to aid in the software development process. Even before formal methods were developed, software professionals routinely conceptualized different ways to condense the details of writing code into a set of core abstractions that could be concisely expressed and communicated (Rosenberg and Scott, 1999). In a very real sense, these abstractions created specialized vocabularies, and hence languages, that could be used to describe software systems.

Software builders had good reason to develop modeling vocabularies. Software, in general, is complex (Carnegie Mellon, 1995). Management of this complexity requires an effective way to capture and communicate software requirements. Software code is precise, but it is hard to create, unwieldy and not many people understand it. Natural languages, on the other hand, are broadly understood but lack precision. Modeling languages filled this void, enabling unambiguous expressions that were nonetheless concise and could be easily understood. Modeling languages are (or at least have the potential to be) unambiguous because the vocabularies of the languages are generally small, targeted, and can be agreed upon by all parties to the discussion (Booch et al., 1999). Modeling languages are concise because they abstract away details; not only do the languages deal with high level concepts, but models built using these vocabularies often focus on a key aspect or view of the system, rather than trying to define the entirety of a system’s complexity.
While many visual representations have been created for communicating about software design (including entity/relationship models by Peter Chen and notations for information engineering by Clive Finkelstein), the industry rallied around the Unified Modeling Language (UML) in 1997 as a more formal approach (Hay, 2003, pp. xxix-xxxi). The increased precision and comprehensiveness of current modeling languages like the UML has expanded their usefulness, yet the fundamental reasons for modeling have remained largely the same:

- **Manage complexity.** Modern software systems are large, sophisticated, and interact with external and legacy systems. Simply put, they have too many details and interactions to understand in a holistic sense. Models provide software engineers the ability to focus on core aspects of system without getting lost in details that are not relevant from a particular viewpoint.

- **Communicate.** Most modern systems can not be built without effective communication between project participants. Modeling languages allow for a concise yet unambiguous expression of project technical requirements.

- **Increase productivity.** Writing the code of a system is perhaps the most challenging, error prone, and time consuming part of a project. Anything that can be done to ease code development is likely to dramatically speed up a project and the quality of its software (Jones, 1996). As a special form of communication, using a model, system architects can often create a blueprint of a system which can greatly simplify or make straightforward the process of writing code.

- **Gather requirements.** System users typically express their needs in a non-technical and informal way. Unfortunately, this can result in incomplete and imprecise requirements. Using a modeling language during requirements gathering can greatly enhance the quality of a requirements document (Sommerville and Sawyer, 1997).

- **Document systems.** Good software systems exist and often grow over a long period of time. Proper maintenance and extension of a software system requires an understanding of its parts, yet by the time either is necessary the original project team is often gone. A good model can provide a roadmap to a system, effectively communicating through time, where and how changes can be best implemented (Jazayeri et al., 2000).

- **Preserve business knowledge.** The effort required to produce a working system is about much more than simply writing good, bug-free code. In order for the code to meet the needs of the business, software builders must have a deep understanding of the domain (business) for which the software is targeted. If this knowledge is expressed only in the functional code, it is extremely difficult to communicate and understand. As a result, it becomes laborious or even impossible to take advantages of new technology, and the knowledge can even be effectively lost when the particular code base is no longer supported. Expressing domain knowledge in a model ensures that it will not be tied to the fate of a particular code implementation. The model can even be reused to guide implementation of the system again for a different target platform at a later date.

### 1.2 Limitations of Modeling

Despite the many potential advantages of formal modeling, it has yet to obtain a central role in the software development process. Though widely recognized as a best practise by many in the industry, formal modeling is practised in a relatively small percentage of software development processes (Zeichick, 2002). Even fewer actually keep models of systems up to date as systems are extended and modified over time. The central problem of a model, of course, is the same problem shared by all document artefacts of software development: it is not code. Rightly or wrongly, many in the industry feel that if the goal of the development process is to build software, then the only real productive task is writing code. Everything else is just overhead on the way to producing code. Indeed, the idea of Extreme Programming (XP) has become popular because it asserts that code is indeed the driving force of all software development (Kleppe et al., p.2).

The “code first” mentality results in a number of limitations that reduce or otherwise marginalize the benefits of modeling.

- **Incompleteness.** Models do not dictate actual functionality; therefore, there is no real need for a model to be complete. Typically, only enough time is spent modeling to solve or understand a particular problem. Because comprehensive models usually are not available, in practise, models...
often provide a poor vehicle for understanding, documenting, or preserving knowledge about a system or domain.

- **Inaccuracy.** Models are not tied to code, and developers often feel that models are simply suggestions about how to structure code. Moreover, many companies discover that models are rarely kept up to date as changes are made to extend or maintain a system. As a result, even the best models quickly become inaccurate once the code is implemented.

- **Imprecision.** Modeling languages have the ability to be precise, but in practise, because code is not directly tied to a model, analysts and developers often do not feel the need to be rigorous in their model expressions. Indeed, there is often no point to model rigor, as the developer will need to “work out the details” in the target language anyway. As a result, in practise many models are ambiguous, have unresolved contradictions, or leverage model elements with poorly defined semantics.

- **Insufficient expression.** Modeling languages have long made it relatively straightforward to express both the structural features and even the dynamic interactions in a domain. Indeed, visual representations are commonly available for both static and dynamic models. In the recent past, however, most modeling languages had no real ability to precisely express behaviour; behaviour was either expressed in an informal (and hence imprecise) manner or simply omitted from models altogether. Both the Object Constraint Language (OCL) and Action Semantic (AS) extensions to UML argue to provide some or all of the necessary behavioural semantics. Yet despite their availability for several years, there appears to be no great demand for either AS or OCL support in most modeling tools. It is not hard to speculate on why this is true. Without any compelling gain in code productivity, why would either a developer or analyst desire to learn a behavioural semantic language? The developer has no incentive, as he already knows several code languages that are more than sufficient, and will eventually have to express the behaviour in one of those languages anyways. To the analyst that is already comfortable with the relative ease of being imprecise or leaving behaviour up to the developer, the idea of learning and using a rigorous behavioural language is simply an invitation to more work without reward. As a result, models usually lack an expression of the business logic that is essential to the system.

Despite the potential advantages of modeling, limitations imposed by an early focus on coding dramatically decreases the realized value of modeling. As long as the only proven way to produce code is for programmers to write code, modeling – no matter how promising – will never reach its full potential.

### 2 Model Driven Architecture

#### 2.1 The Evolution to Model Driven Architecture

Model Driven Architecture (MDA) is an Object Management Group (OMG) initiative for building software (OMG, http://omg.org/). The central idea of MDA is to separate the business logic of a system from its technological implementation details. Do this and both business knowledge and technology can continue to develop without necessitating a complete rework of existing systems (Miller, 2003). This idea is nothing new, but the way MDA provides for it is revolutionary, despite being based upon established ideas about modeling and transformation. Core to MDA are the ideas of system models, and transformations between models. Business information and logic is captured in a Platform Independent Model (PIM). Technological implementation details are added by applying a set of rules that depict the specifics of a target platform to arrive at a Platform Specific Model (PSM). Information from the PSM along with additional transformation rules are applied to generate source code – a functioning and executable version of the business system defined in the PIM.

The ideas behind MDA represent a natural progression focused on the value of modeling, which is to say MDA did not start off with the intent to create a new way to generate code. Rather, MDA began with the idea that the valuable business and systems knowledge that could be captured in models could be leveraged, reused, and shared if a set of methodologies existing for common storage and communication of the information in the model (Lema, 2004). Common storage requires transforming the information in a model into a common format. Not long after transformation entered the picture, it became clear that if
we could transform to and from a common storage format, we could also transform into another format, perhaps with slightly different semantics that provided additional value. For example, given a well defined use case model, it is not too much effort to define a set of rules that will transform the use cases into a candidate class diagram.

2.2 Components of MDA

Let’s take a brief tour of MDA, reviewing its building blocks to see how they all fit together to create software.

Model

In its most general sense, a model is simply a description or specification of a system or part of a system, usually for a specific purpose or point of view. “System” may be a software system, but in this context it also equally applies to a business system, environmental system, or anything that exists in reality. “Purpose” and “point of view” both imply that a model usually has a certain semantic space. For example, we might have a domain model of an environmental system with real world objects like “plant” and “soil” and we also might have a software model of the same system with objects like “user”, “simulation controller” in addition to objects named “plant” and “soil”. Both models are views of the same real world system, and hence have similarities, but in the first case, the objects of the model represent real biological actors, while in the second they represent software code objects. Strictly speaking, a model can be written in any language, but, in order to support automated transformations, the language needs to be computer readable. In terms of MDA, model languages must have well defined form (syntax) and meaning (semantics) (Kleppe et al., 2003).

MDA recognizes several different important types of models. The most important of these are the PIM, and the PSM but the other relevant types are discussed below as well.

• Computation Independent Model (CIM) – A CIM is a formal specification that defines how a real world system works without any reference to software. A CIM focuses on the environment and the requirements of a system; structural details and any processing logic remain hidden. In the case that it defines a specific system to be implemented, it defines how the company will use the system and what it needs to accomplish but never defines any aspects of the system’s internal structure.

• Platform Independent Model (PIM) – A PIM is an abstract model that resolves functional requirements through purely problem-space terms. Functional behaviour and system logic are described in the PIM, but because its goal is not to solve non-functional requirements, no platform-specific details are included. A PIM cannot be generated directly from a CIM, as decisions about the conceptual structure of a system, and the pieces of a CIM that will be supported by a PIM are often human. Because of this, while a CIM is an important reference, the PIM is the starting point of MDA transformations.

• Platform Specific Model (PSM) – A PSM is a solution model that resolves both functional and non-functional requirements. Because its goal is to solve non-functional requirements as well as functional requirements, it requires and is dependent upon platform technologies (libraries, operating systems, server resources, etc.). The target platform of a PSM is crucial as it defines the actual viewpoint of the model. A PSM targeted at Java, for example, will be quite different from a database entity-relationship PSM, even though they may both derive from and describe different aspects of a software system described by the same PIM. Lastly, we should recognize that PIM and PSM are somewhat relative terms. While some models are certainly more platform independent that others, in reality there is a continuum extending from a PIM all the way to source code. In practise, this could involve multiple intermediate models that are progressively more platform (and even architecturally) specific.

• Code model – While we tend to think of source code as the end goal, it is equally valid to say that source code is also a model of a system, albeit written in a more detailed and flexible language. If we think about environmental simulations, the analogy is clear (they are even explicitly called “models”). But even with other software, we recognize that there is an important distinction between
Transformation

Transformation is the process of converting one model (view) of a system into another model (view) of the same system. Automated transformation requires a source model, a transformation tool, and a description of how elements of the source model map to elements in the destination model (Kleppe et al., 2003). This description is called a transformation definition or a mapping. The transformation definition itself consists of rules that describe the semantic mapping of individual elements from source to destination model. For example, a transformation definition from a PIM domain model to a database entity-relationship diagram would include rules similar to the following:

1. For each class in the PIM, create a table with the same name as the PIM class.
2. For each attribute in the PIM, create a column in the corresponding table where the name of the column is the same as the name of the attribute.

Transformation rules do not have to be one to one. For example, a transformation rule for mapping a PIM class attribute to Java might include creation of a private attribute as well as appropriately named public “get” and “set” operations.

In MDA, transformations are applied to the PIM to generate one or more models of the system that are platform specific. For example, from the same PIM we might apply one transformation to generate an Entity-Relationship diagram of the tables and other database structures necessary to store the persistent data for an application, another transform to generate a model of the Java business classes that perform the work of the application, and finally a third transform that models the Java Server Pages (JSP) for a user interface layer that presents the data and services of the application to the user (Fig. 1). If multiple PSM need to work together to fulfil the work of the modeled system, a representation of the communication bridges between the models is also created by the PIM to PSM transforms.

Additional transformations are applied to the PSM to generate the executable code (model) of the application, including any necessary bridging code. Because the PSM is typically very specific to the target platform, the PSM to code transform is usually very straightforward.

Languages and Metamodels

Up to this point, we have talked about models that “express” views of a system and about transformation rules that “define” mapping, but without talking about the language used for either. Clearly we need some sort of language to write these expressions, but what are the characteristics of these languages and how do we create them? Because automated transformation is at the heart of MDA, both models and transformation definitions must be written in an unambiguous, machine readable language. These languages are developed through the use of metamodels.
A metamodel is simply the model of a model. Because a metamodel is a model, we can learn much about it from carefully inspecting a more intuitive example. For example, take the simple model shown in Fig. 2 (left) of an environmental system that contains classes representing Weather, Plant, Soil, and DailyWeather. Weather has an association with DailyWeather that shows Weather can have many instances of DailyWeather. Plant also has an attribute that contains information about its leaf area density. Note that if we want to talk about an instance of this model, that is, a real environmental system, the model completely defines the things we can say. Using this model, we can talk about a specific weather day, and the temperature of that day, but we can not talk about, say, phosphorous loading, (at least not without adding a representation of it to our model). The model defines what elements can exist in our system.

If our model tells us what elements (soil, weather, and plant) can exist in a real instance, what then are the elements that make up our model? In our model, Weather, Plant, Soil, and DailyWeather are all classes, that is, each is an instance of “Class”, just as a real plant is an instance of Plant from our model. We also find associations (the connections between classes), and attributes in our model. In other words, the model of our model – our metamodel – must contain Class, Attribute, and Association, among other elements. Just as our simple model defines what we can say about a real instance of a simple environmental system, so too does our metamodel define the language of what we can say when we are creating a model (Fig. 2, right).

We have now come back to our stated need and can see that to create an unambiguous, machine readable language we define a metamodel. The metamodel defines everything that can be said in a model, and assigns unambiguous semantics to its elements. If we write instructions (for a machine) to handle the elements of the metamodel, it will then be able to properly process all of the models we can write using that metamodel (Kleppe et al., 2003).

In MDA, the instructions written most frequently are transformation definitions. In other words, transformations are defined in terms of metamodels. By doing so, we only need create a single transformation definition for a pair of source and destination metamodels in order to be able to generate an instance of the destination model from any source model that complies with (is an instance of) the source metamodel. To make this clear, look back at our sample weather-soil-plant model and mentally apply the example transformation rules from the “Transformation” section. Note that the rules make no reference to the actual content of the model -- nowhere are weather, soil, or plant referenced. This is good, because otherwise we would need to write a separate set of transformation rules for each different model. Instead, the rules reference the elements of the source metamodel, “class”, “attribute”, “name”, and the elements of the destination metamodel, “table”, “column”, “name”.

3 Benefits of an MDA Process

Now that we understand the core elements of MDA, it is easy to see that the “heavy” work of the process involves writing transformations and in creating the PIM that is the input for a series of automated transformations. Certainly, we also need languages for expressing our various models, transformation rules, and tools for executing the transforms, but these are infrastructure pieces that should be reusable once created. Transformations too, should be quite reusable. A transform is dependent only upon its
source and destination metamodels, so if we assume that the source metamodel is more or less constant, we need only create only a single transform for each destination platform or platform variant.

The core non-infrastructure work of creating each new system with MDA is building the PIM that defines the system. This, of course, brings us full circle, back to our starting discussion of the potential benefits and the unfortunate practical limitations of modeling. MDA changes the entire nature of modeling. Because a PIM in MDA can be used to facilitate the generation of code, it becomes the most important artefact of the entire development process. No longer is a model “just a picture” that competes with the real work of writing code. Instead, creating the model is the real work, which means that organizations, processes, and individuals can finally start to realize all of the benefits of modeling discussed previously.

Make no mistake that creating an MDA model is serious intellectual work. Unlike pleasant diagrams that simply communicate a few high level concepts, MDA demands that it have a comprehensive, coherent, absolutely unambiguous, and functionally complete specification of the system being implemented. This sort of rigor has always been present, as it is demanded by machine readable code; it was just being created by programmers that knew they had to “fill in the gaps”. MDA, for the first time, moves this level of rigor, and hence, value, out of the code, and into the model.

Up to this point, there really has been no need to discuss the specific needs of creating agricultural and environmental computer simulations, because we have been backing into it by instead discussing the solution. In the context of the many benefits of MDA, it is useful to look at the specific needs of agro-environmental simulations and how MDA addresses each.

- **Domain knowledge preservation and reuse.** This most important benefit of MDA is especially key to agro-environmental simulations. It is well known that many agricultural and environmental simulations exist as software that can only be maintained and reused by their authors (Papajorgji et al., 2004). As a result, reusing or modifying these simulations can be quite difficult. Yet at the same time, there is an even greater need than with other software to reuse, modify, recombine, and extend simulation models in order to build new composite simulations or bring older simulations up to date with new understandings of agricultural and environmental phenomenon. By putting the domain knowledge of the system into a PIM, the PIM or parts of the PIM can be easily shared and incorporated into other models in a straightforward fashion. Moreover, in the case where a system needs to be deployed on a different platform or multiple platforms, the same PIM can be used to generate each application simply by applying different transform(s).

- **Communication.** While it is certainly true that a software system that is better understood is likely to be a better system, this is doubly true for simulations models. Many simulation models represent the implementation of academic knowledge about agricultural and environmental processes. Simulations that are poorly documented or expressed only as code are not accessible to non-programmers, and therefore can not be easily discussed (Papajorgji et al., 2004). MDA allows us to express our ideas in a body of models that can more easily participate in an exchange of ideas critical to improving collective knowledge.

- **Productivity.** Many software systems are never built because the cost of writing the code is simply too great. The same certainly applies to simulation models, where funding must often come from a relatively small pool of academic grants. This is perhaps even more frustrating in that many of the simulation models that could be built in order to further academic knowledge have a low level of traditionally expensive non-functional requirements. In other words, simulation models typically do not need, for example, elaborate user interfaces or communication bridges with legacy systems. Current MDA toolsets are perfectly suited to build such systems. They allow the user to focus on the functional requirements of the system, replacing costly “coding” with a reusable transformation. Using MDA, academic users can quickly develop domain models, generate, and then modify in order to support rapid hypothesis testing all at a fraction of traditional code development costs.
4 Conclusion

MDA elevates the model to its rightful place in the application development process. What was formerly simply a diagram that suggested how a software system might be implemented, in MDA becomes a concrete specification that directly describes exactly how a system functions.

Intellectually, simulations in and of themselves have always been models of environmental phenomena. With MDA, we can now represent these environmental models as software models that have a one to one representation with both our understanding of a physical reality and the implemented code reality. Computer simulations no longer need to be black boxes, the internals of which are only fully understood and accessible by their authors. Rather, MDA allows simulation models to exist as first class artefacts that are open for inspection, criticism, refinement, and sharing by the rest of the modeling community.

Perhaps the most important aspect of MDA is that it makes clear what it knows and does not know. Instead of having all the answers, it is a system that is explicitly open so that the introduction of new information, technologies, approaches, frameworks, or platforms won’t ruin existing investments already made by those adopting MDA.

5 References


Object Management Group (OMG), http://omg.org/


