On the Applicability Scope of Model Driven Engineering

Jean Bézivin Mikaël Barbero
Frédéric Jouault
ATLAS Group, INRIA and LINA
University of Nantes, France
{jean.bezivin, mikael.barbero, frederic.jouault}@univ-nantes.fr

Abstract

Model Driven Engineering (MDE) is frequently presented as an important change in software development practices. However behind this new trend, one may recognize a lot of different objectives and solutions. This paper studies the multiple facets of MDE and its evolution in the recent period. It concludes by emphasizing the broad potential application scope of MDE and suggests some new areas of usage and a classification scheme for related applications.

1. Introduction

In this paper we refer to the OMG MDA™(Model Driven Architecture) proposal as the initial initiative that triggered a broader area of research in the automation of model processing. We consider MDE (Model Driven Engineering), as being the more general approach, including but not limited to MDA. MDE encompasses other non OMG projects like Microsoft DSL Tools, Eclipse Modeling Project (EMP), Vanderbilt University Generic Modeling environment (GME), and many other initiatives making explicit references to software and system modeling.

It is not fair to date the origins of MDE only to the announcement date of the MDA proposal in November 2000 [10]. Many previous contributions were already going in this direction well before this period. Acronyms like AD/Cycle, CDIF, GraphTalk, OIS, sNets, MetaCASE among others testify of a previous very strong activity in this domain. The story of all influences that have helped to shape the modern history of MDE has still to be written and is not the subject of this paper. We consider here the recent history of MDE, starting from the initial OMG MDA proposal [10]. We are interested in the evolution of its application scope.

Since this initial OMG proposal, MDA has come to mean a lot of different things to different people. This situation of diversity is sometimes creating confusion. The goal of the present paper is to study some of the different meanings of MDE and to propose some definitions and clarifications that may help the mutual recognition of the variety of goals in the model engineering community.

In order to clarify as much as possible the discussion, we will distinguish between the principles, the standards, and the tools. One assumption on which this work is based is that there is a common and minimal set of basic MDE principles. The common principles are becoming widely accepted. As presented in figure 1, we can summarize them by stating that MDE relies on two basic assertions (1) the *representation* of any kind of system by models and (2) the syntactic *conformance* of any model to a metamodel. Within this context, models are intended to be automatically processed by a set of operators. One of the major automatic operations for a model is a transformation [1].

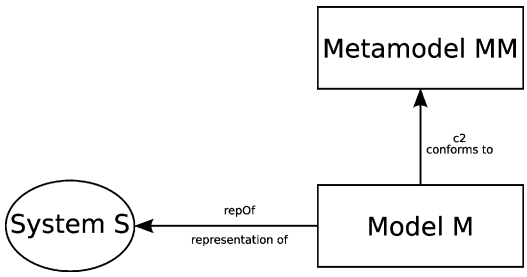

Figure 1. Basic relations of *representation* and *conformance* in MDE

MDE promotes the usage of models as first class entities [2]. Any model used to represent a given system conforms to a precise metamodel [5]. Some models may be obtained from other models by chains of transformations or other automated operations on models (composition, difference, etc.). We call them *derived models*. However some other models have to come from somewhere in the first place: we call them *initial models*. In classical *forward engineering*,

many initial models are created by human operations with the help of CASE tools like Rational Rose. In *reverse engineering*, initial models may sometimes be produced from systems by automatic or semi-automatic means. This is for example the case when a model is created from legacy code by a text to model extraction operation.

In addition to forward engineering where systems are created from models and to reverse engineering where models are extracted from systems, there is a new area of application of MDE where the system and the model coexist in time and are kept synchronized. We refer to this application area by "models at run time" in figure 2. This expression was coined in to state that a dynamic system may have, at execution time, a knowledge, and operational access to various models representing it, models which are constantly kept in synchronization with its current state.

One important idea in this work is based on the dichotomy between what we call systems and what we call models. A system lies in the real world while a model lies in the modeling world. This distinction is absolutely essential to any MDE conceptual framework. This allows distinguishing the three cases presented in figure 2:

- "Forward engineering" where a system is created from a model,
- "Reverse engineering" where a model is created from a system,
- "Models at run-time" where a model coexists with the system it represents.

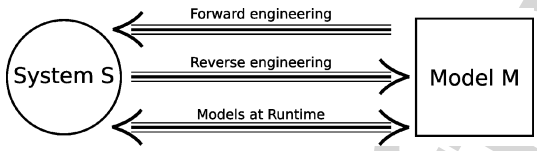

Figure 2. The three main families of MDE applications

Note that the conformance relation between the model and the system holds in the three cases but, in the first two (forward and reverse engineering) it should be considered more as a post condition for the operation.

The way this paper is organized reflects this classification. After the presentation of these different views, a concluding section will propose some perspectives on the future of MDE.

2. Forward engineering

Model Driven Engineering was initially intended to provide some high level "compilation" process between platform-independent business situations expressed in the UML language into platform-dependent executable programs. The basic idea proposed in the MDA approach is to translate from an abstract PIM expressed in UML into a more concrete Platform Specific Model (PSM):

> "Whether your ultimate target is CCM, EJB, MTS, or some other component or transaction-based platform, the first step when constructing an MDA-base application will be to create a platform-independent application expressed via UML in terms of the appropriate core model [...]. [...] Platform specialists will convert this general application model into one targeted to a specific platform such as CCM, EJB, or MTS. Standard mappings will allow tools to automate some of the conversion." [10]

When the PSM is obtained, then we still need to generate code:

> "The next step is to generate code itself. For component environments, the system will have to produce many types of code and configuration files including interface files, component definition files, program code files, component configuration files, and assembly configuration files. The more completely the platform specific UML dialect reflects the actual platform environment, the more completely the application semantics and run-time behavior can be included in the platform-specific application model and the more complete the generated code can be. In a mature MDA environment, code generation will be substantial or, perhaps in some cases, even complete." [10]

So the initial goal could be summarized as follows:

$$(PIM, PDM) \rightarrow PSM \rightarrow Code$$

This means that from a PIM expressed in UML and from a Platform Description Model (PDM) probably also expressed in UML, it should be possible to generate a synthesized PSM, also expressed in UML. The initial PIM was supposed to be developed by hand, from the observation of the contextual business and application requirements.

Since the publication of the initial whitepaper, the OMG has been constantly updating, improving, and extending its very ambitious vision. There are many ideas in this proposal and this explains its high popularity. Some of these ideas have been implemented while others still stay at the level of open research problems:

- The PIM to PSM translation problem, when both are expressed in UML (or profiles dialects of UML) has prompted the study of model transformation techniques. The natural way was to follow a declarative expression of rule-based declarative model-to-model (M2M) transformation programs. This gave birth to the QVT recommendation.

- The integration of platform knowledge (coded as a PDM) into a PIM in order to get a PSM is still out of reach for several reasons. The first one is that we do not have yet a sufficient understanding of what is platform knowledge. The second reason is that even if we had, we do not understand clearly how to compose a PIM with some platform knowledge encoded as a PDM. It seems that this is more complex than just M2M transformation.

- The more mature part is the PSM to code that is achieved by so called M2T (model-to-text) transformations. This achieves for example the generation of Java code from a PSM. Presently this represents a major part of practical applications claiming to follow an MDA approach.

Another important contribution to MDE was the MDA manifesto [3] written in 2004 by a group of authors from IBM. One central message in this contribution is about domain modeling. The strong idea that comes out of this manifesto is the need for DSLs (Domain Specific Languages).

> "Model driven Architecture gets power by building models that directly represents domain concepts. Domain-specific languages may either be built on top of general languages such as UML or through meta-modeling frameworks such as MOF. By using frameworks that explicitly model assumptions about the application domain and the implementation environment, automated tools can analyze models for flaws and transform them into implementations, avoiding the need to write large amount of code and eliminating the errors caused by programming [...].
> [...] Of course, there are many different application domains, often requiring very different concepts. Furthermore, as experience and knowledge grow, these domains become increasingly more complex and sophisticated. It does not require much insight to realize that different domains will need different domain specific languages (DSLs)." [3]

It is worth noting this important evolution since this was not very apparent in first generation MDA projects. Separating platform dependent from platform independent aspects was the first goal. Rapidly it appeared that these aspects were not the only ones to be dealt with in a software system. Using UML as a general purpose modeling language to describe at the same time business systems, functional requirements, quality of services, platforms, processes, etc. started to become problematic, even with the profiling mechanisms. The idea of using different languages to describe different aspects of the system made its way quit rapidly, through DSLs. The abstract syntax of these DSLs could be easily implemented by MOF metamodels [8]. This gave much more impact to MDE approaches, since they could be much useful, for example within the context of a single stable platform.

The M2M transformation was easy to adapt to exogenous transformations, i.e. the case when the source and the target models conform to different metamodels. Furthermore, the transformation language itself could be associated to another DSL, with a specific metamodel. As a consequence, transformation programs could be considered as models which gave easy ways to deal with higher order transformations, producing or consuming other transformations.

So we see how the search for a specific solution to platform independence gave birth to a more general solution, capable to solve much wider problems.

And this was only the first period in the development of modern MDE practices.

3. Reverse engineering

Many of the OMG MDA recommendations like UML, MOF, QVT, etc. have been elaborated in the ADTF internal group (Analysis and Design Task Force). A new group named ADM (for Architecture Driven Modernization) soon started working on reverse engineering solutions. Very naturally it started using modeling techniques that had been found useful in forward engineering.

Legacy systems are complex systems because they are voluminous (they have been developed over periods of 10, 20 years or sometimes more) and heterogeneous (they are composed of a lot of different technologies from pre-COBOL to post-Java). They need to be understood, modernized, and migrated. MDE provides powerful techniques to represent different facets of these legacy systems that will allow understanding, modernizing, and migrating them to new technologies.

The first solution is to use UML to extract models from legacy systems. For example UML use case diagrams, activity diagrams, state diagrams, class diagrams could be extracted from legacy systems as suggested in [9].

But sometimes one would like to extract more specific models from a legacy system. This is why the ADM group

has defined several metamodels to this purpose, the most known being KDM and ASTM.

The Knowledge Discovery Metamodel (KDM) standard, provides a comprehensive high-level view of the application behavior, structure, and data, but does not represent application models below the procedural level. It corresponds to a coarse grain representation of a given legacy portfolio. The Abstract Syntax Tree Metamodel (ASTM) deals with constructs within programming languages (fine grain, syntactical), while the KDM models structural, behavioral and data information about application portfolios. The ASTM is expected to complement the KDM as an element of the ADM Roadmap to express fully detailed models of application artifacts.

The approach taken in the MoDisco Eclipse project [7] is even more general because it assumes that you may use any kind of DSL (associated to a metamodel) to describe the facet you want to extract from a given legacy. For example if you want to extract models from a Java or COBOL program, from the Web, from a database, from a UNIX system, etc. you will use adapted metamodels. Furthermore MoDisco provides a toolbox that facilitates the building of "model discovers" from the definition of the target metamodels as depicted in figure 3.

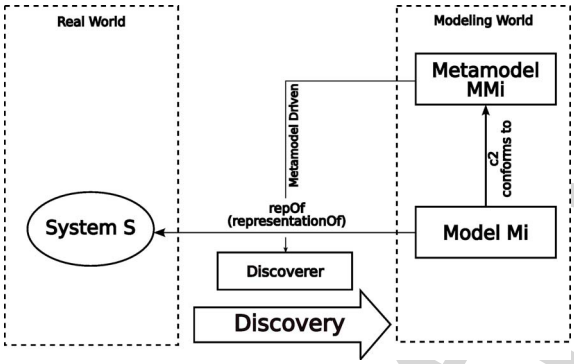

Figure 3. Metamodel-driven model discovery in MoDisco

It is interesting to note that the systems from which we are extracting models may be of different kinds. They may be static or dynamic. An example of a static legacy system is the code of a Java program. An example of a dynamic system is one execution of a Java program. A trace of the dynamic behavior of this program may produce a model, conforming to a given metamodel.

4. Models at Run time

One emerging area of application of MDE is related to complex systems. This is broader than software engineering and addresses the goals of systems engineering. Systems engineering may be defined as an emerging discipline whose responsibility is creating and executing an interdisciplinary process to ensure that the customer and stakeholder's needs are satisfied in a high quality, trustworthy, cost efficient and schedule compliant manner throughout a system's entire life cycle.

One of the main challenges of computer science is how to build, to maintain, to manage, and to monitor highly complex world-wide distributed systems made of many parts, continuously running, and constantly being maintained. These systems are sometimes called systems of systems.

It seems that MDE, with its capability to handle several aspects of the same system, based on different DSLs, is one of the few possible ways to capture this complexity.

In this view, a system will coexist with a set of models of it with the following properties:

- Each model is constantly kept updated of the latest changes of the system
- Each model represents a particular aspect related to the dynamic behavior of the system, to its static architecture, to its evolution, to its maintenance data, to its health measured by some precise constraints, etc. The cost of direct access to the system with the diversity of the technologies used for its various components is much too high to allow direct monitoring and control operations. The idea is to provide some kind of uniform representation knowledge capturing the various facets of the system, from design data to maintenance and run time information.

There are several important projects currently investigating these possibilities. To name two of them, Microsoft DSI (Dynamic System Initiative) with its latter SML proposal [6] addresses self managing dynamic systems while the IBM Rainforest project aims to facilitate the deployment and configuration of complex J2EE applications in heterogeneous and constrained environments [4]. DSI is about building software that enables knowledge of an IT system to be created, modified, transferred, and operated on throughout the life cycle of that system. Knowledge in this sense includes knowledge of the designers' intent for those systems, knowledge of the environment in which the systems operate, knowledge of IT policies that govern those systems, and knowledge of the user experience associated with those systems. This knowledge is captured in software models using a variety of authoring and development tools. The Rainforest project employs model transformations, search, and planning techniques to handle configuration management of distributed applications and services in large, heterogeneous, and constrained environments.

"Systems at run time" solutions aim to provide external MDE capabilities reminiscent of reflection in classical object-oriented programming languages with a completely different technology. The possibility of introspection is replaced by the capacity of systems to concretely access their models during their execution. For example, during the execution of a given distributed system, in an exception situation, this system could be allowed to access the design data corresponding to state diagrams. These capabilities obviously strongly require a complete and permanent synchronization and update of the various models to represent the latest version of the systems' components. If introspection is one of the important applications of these techniques, it is not clear if intercession based techniques could also be of interest here.

With these new possibilities, any stakeholder could check the relevant information not by directly accessing the system itself, but by accessing the various updated models of the system. Furthermore classical model transformation techniques easily allow providing different views on the system. This nicely complements the fact that the system also accesses similar information. These techniques may have a high potential impact on the way new complex distributed systems are designed.

5. Conclusions

What we have seen is that MDE has evolved in applicability scope in less than ten years. We have proposed to consider three different situations corresponding to "forward engineering", "backward engineering" and what we named "models at run-time" by lack of a better term. Inside each category, there are other possible classifications of MDE techniques.

Techniques used in these three categories may be complementary. For example if a reverse engineering process may produce a pivot model in UML from a COBOL program or another language, this model may in turn be taken as the starting point of a consecutive forward engineering process that will generate for a different target context like a Web service environment.

These three mentioned areas of applicability of MDE are of increasing complexity and maturity. Forward engineering is the most applied, especially when it comes to generating code from PSM with M2T techniques. One reason why it is easier to handle than reverse engineering is that it is directed by source models in a generative process while in the case of reverse engineering, the source of information is the legacy system itself.

One consideration that comes out from this presentation is that understanding the "models at run time" technique means that we have a prior understanding of the two other techniques of forward engineering and backward engineering. But the most important point here is the need to master completely the techniques to extract models from dynamic systems.

There are two ways of characterizing the scope of Model Driven Engineering: by its functional objectives or by its support techniques. What has been suggested in this paper is that both types of definitions have been rapidly evolving in the last years and continue to change. The initial goal of separation/composition of platform dependent/independent knowledge is now only one among a set of different goals. The usage of a visual general purpose modeling language like UML to express various situations in software production has been progressively replaced by the joint usage of a set of DSLs. How these DSLs should be coordinated is still an area of research.

In this very rapidly evolving context, the distinction between systems and models seems necessary to define both the goals and the techniques of MDE.

6. Acknowledgements

This work has been supported by the ModelPlex European integrated project FP6-IP 034081 (Modeling Solutions for Complex Systems).

References

[1] J. Bézivin. From object composition to model transformation with the MDA. In *TOOLS' USA*. IEEE Publications, August 2001.

[2] J. Bézivin. On the unification power of models. In *Software and System Modeling (SoSym)*, volume 4, pages 171–188, 2005.

[3] G. Booch, A. Brown, S. Iyengar, J. Rumbaugh, and B. Selic. The IBM MDA Manifesto. *The MDA Journal*, May 2004.

[4] IBM. The Rainforest project. http://www.research.ibm.com/rainforest/index.html.

[5] F. Jouault and J. Bézivin. KM3: a DSL for metamodel specification. In *Proceedings of 8th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, LNCS 4037*, pages 171–185, Bologna, Italy, 2006.

[6] Microsoft. Dynamic system initiative. http://www.microsoft.com/windowsserversystem/dsi/-default.mspx.

[7] MoDisco. The GMT MoDisco web site. http://www.eclipse.org/gmt/modisco/.

[8] OMG. OMG/MOF Meta Object Facility (MOF) Specification, September 1997. http://www.omg.org.

[9] R. Soley. Extracting UML from legacy applications. *SOA Web Services Journal*, October 2006. http://webservices.sys-con.com.

[10] R. Soley and the OMG staff. The Model Driven Architecture Whitepaper. *OMG document*, 2000. http://www.omg.org/mda.