

A Preliminary Identification of MDA Components

Jean Bézivin
University of Nantes
Faculty of Sciences
2, rue de la Houssinière
France
+33 - 251 125 813
Jean.Bezivin@sciences.univ-nantes.fr

Sébastien Gérard
CEA
Centre d'Etudes de Saclay
F-91191 Gif sur Yvette Cedex
France
+33 - 169 086 259
Sebastien.Gerard@cea.fr

ABSTRACT

This position paper reports on an initial investigation in the fields of MDA and generative approaches. Our view is that MDA aims at providing a precise framework for generative software production. Unfortunately many notions are still loosely defined (PIM, PSM, etc.). We propose here an initial exploration of some basic artifacts of the MDA space. Because all these artifacts may be considered as assets for the organization where the MDA is being deployed, we are going to talk about MDA abstract components. We show in this paper the central role that the undergoing RFP on model query and transformation may play in the global MDA organization. Our position is that, if this RFP is successful, the resulting "UTL (Unified Transformation Language)" may soon become the cornerstone of the MDA and contribute to bring closer the generative techniques and the MDA communities.

Keywords

UML; MOF; MDA; CIM; PIM; PSM; Transformations.

1. Introduction

The ratio of generated vs. hand-written code has been rapidly increasing in the past years. Sometimes this code is generated from precise definitions and sometimes it is generated from non-explicit sources like in direct manipulation GUI building. At the end of the year 2000 [11], the OMG proposed a rapid change from OMA and interpretative approaches (building more and more complex middleware platforms) to MDA and transformational approaches (generating models from other models). In order for the MDA to become a mainstream technology, some clarification is needed in the various associated concepts. This paper proposes a contribution to define a basic terminology for the MDA and to clarify some situations, for example with respect to model transformation operations.

2. MDA Main Concepts

The main concepts of the MDA are beginning to be identified ([2][1][3][11]). A model represents a particular aspect of a system under construction, under operation or under maintenance. A model is written in the language of one specific meta-model. A meta-model is an explicit specification of abstraction, based on shared agreement. A meta-model acts as a filter to extract some

relevant aspects from a system and to ignore all other details. A meta-meta-model defines a language to write meta-models.

There are several possibilities to define a meta-meta-model. Usually the definition is reflexive, i.e. the meta-meta-model is self defined. A meta-meta-model is based at least on three concepts (entity, association, package) and a set of primitive types. The OMG MDA postulates the use of the MOF as the unique meta-meta-model for all IT-related purposes. The MOF contains all universal features, i.e. all those that are not specific to a particular domain language. Among those features we find all that is necessary to build meta-models and to operate on them.

Maintaining a specific tool for the MOF would be costly, so the MOF is aligned on the CORE part of one of its specific meta-models: UML. UML thus plays a privileged role in the MDA architecture. As a consequence, any tool intended to create UML models can easily be adapted to create MOF meta-models.

A meta-model is composed of three parts: terminological, assertional and pragmatics. The terminological part corresponds to UML class diagrams. The assertional part corresponds to OCL (Object Constraint Language) assertions that may decorate the various elements of the meta-model. The pragmatics corresponds to details that could not fit into the previous parts. Example of a pragmatic item is for example how to draw some particular concepts or relations. In UML 2.0 a tentative has been made in the presentation RFP to separate the content aspects from the presentation aspects in a given meta-model. This shows the tendency to reduce the pragmatic part by integrating the corresponding aspects into separate meta-models. Usually the pragmatics elements are expressed in natural language informal descriptions.

A meta-model defines a specific domain language. It may be compared to the formal grammar of a programming language. In the case of UML the need to define variants of the base language was expressed. The UML meta-model was then equipped with extension mechanisms (stereotypes, tagged values, constraints) and this allows defining specialization of the basic meta-models as so called profiles.

The MOF contains features to serialize models and meta-models in order to provide a standard external representation. The XMI standard defines the way serialization is performed. This is a way

to exchange models between geographical locations, humans, computers or tools. When a tool reads a XMI serialized model (a UML model for example), it needs to check the version of the meta-model used and also the version of the XMI applied scheme.

3. MDA Components

The MDA organization may be viewed as a set of artifacts, some being standard building blocks, some being user developed. We may envision, in the not too far future, an organization starting with a hierarchical library of meta-models and extending it as an adaptation to its own local context (models as assets). Model reusability will subsume code reusability, with much more efficiency. This may be seen as orthogonal to code class libraries (e.g. Java, Swing, EJB, etc.). Inside a company, the various business and service models will be developed and maintained to reflect the current situation. When buying a platform (Unix, EJB, DotNet, etc.), the price tag will cover not only the executable version of the platform, but also a model of this platform (external specification) and/or a set of tools to generate for this particular platform. Such tools may be delivered as a specific transformation model that will drive a generic transformation tool. If this schema may look like futurology, it is already operating in some limited and specific contexts. The purpose of the MDA consists in making it more general and more universal.

The UML meta-model plays different roles in the MDA. First it defines the language used for describing object-oriented software artifacts. Second, its kernel is synchronized with the MOF for practical reasons as previously mentioned. There is much less meta-modelers (people building meta-models) than modelers (people building models). As a consequence it is not realistic to build specific workbenches for the first category of people. By making the MOF correspond to a subset of UML, it is possible with some care to use the same tool for both usages. As a consequence the MDA is not only populated by first class MOF meta-models, but also with UML dialects defined by UML profiles for specific purposes languages. This is mainly done for practicality (widening the market of UML tools vendors) and there is some redundancy between UML profiles and MOF meta-models (It is even possible to find conversion tools).

There are many examples of profiles. Some are standardized by OMG working groups and other are independently defined by user groups or even by individuals. The situation is similar to the XML galaxy where independent users may decide to develop specific DTDs or XML schemas. In the case such a development becomes popular or widely available, it may constitute a de facto standard. Another important source for profiles are UML CASE (Computer Assisted Software Engineering) Tools vendors. These organizations may provide libraries of profiles, usually adapted to the specificities of their tools and with limited portability.

Examples of profiles are "UML for Corba", "UML for C++", "UML for Scheduling Performance and Time" (real-time applications), "UML for EJB", "UML testing", "UML for EAI", "UML for QoS and fault tolerance", "UML for EDOC" (Enterprise Distributed Object Computing), etc. As we can see, all; of these are related to some specific usage of UML and may be referred to in the explicit software process.

The SPEM (Software Process Engineering Meta-model) provides generic guidelines to organize a software project, i.e. to answer the question: who is doing what, when, how and why? The SPEM

has this particularity of having officially been defined as a MOF meta-model and as a UML profile. The MOF meta-model could be used by non-UML tools vendors for example CAPE (Computer Assisted Process Engineering) or CARE (Computer Assisted Requirement Engineering) tools vendors.

The CWM (Common Warehouse Metadata) provides the means to partially deal with legacy systems (relational databases and data warehouses). This is an interesting example of a complex meta-model structured as a set of modules dealing with various concerns like object modeling, data modeling, data transformations, business information, type mapping, data warehousing, data mining, OLAP, etc. This example shows how the notion of composite model (see below) is becoming important.

One well-known distinction on meta-models is between products (structured set of artifacts produced or consumed) and processes (the way this is done, the characterization of tasks, actors, roles, goals, etc.).

Less classical is the distinction between dynamic and static models. A static model is invariant while a dynamic model changes over time. This should not be confused with the static and dynamic aspects of systems. The most common situation is a static model of a dynamic system.

The fact that a model is executable or non-executable has nothing to do with the previous property. A Java program may be naturally considered as an executable model. Unless extended, a UML model is non-executable. The way to give some specific executability semantic to a UML model is by using the "Action Semantics" meta-model.

Like any software component, a model may be atomic or composite. An atomic model contains only basic elements. A composite model contains at least another model. The possibility of composition is defined at the meta-model level. The containment hierarchy for models is distinct from the specialization hierarchy.

A model may be primitive or derived. A derived model may be obtained from other models (primitive or derived) by a derivation operation or a sequence of such operations. A derivation operation is a simple and deterministic model transformation.

A model may contain functional or non-functional elements. Usually it does not contain both. Typical non-functional elements are related to QoS properties like performance, reliability, security, confidentiality, etc. The elements of a non-functional model are usually related to specific elements in a base functional model. A more general solution consists in using a correspondence model (see below) to state the explicit associations between non-functional and functional elements.

An essential model is a model that is intended to stay permanently in the model repository system. A transient model is disposable, i.e. it has been used for some temporary purpose and has not to be saved. Compilers use for example some intermediate files to carry on their transformation from high-level languages to low level machine language.

When we have a conversion to be done between an important numbers of different meta-models, it may be interesting to define a pivot meta-model to facilitate the transformation process.

UML is a product meta-model for object-oriented software artifacts. We may also consider models of legacy systems. Part of a Cobol meta-model may be found in the CWM. Complete Cobol meta-models have been defined and are being used by companies. The migration from a legacy system (e.g. Cobol) to an object-oriented system (like Java) may be described by a migration process. There is not yet a specific migration process meta-model standardized by the OMG, although the SPEM could be adapted to this task.

One important kind of model that is being considered now is the correspondence model. A correspondence model explicitly defines various correspondences that may hold between several models. In the usual case, there are only two models: the source and the target. There may be several correspondences between a couple of elements from source and target. The correspondences are not always between couples of elements and they are strongly typed. There is not yet a global consistent view on correspondence models since this problem is appearing from different perspectives. The traceability

A commonly found model that we should not ignore is a source program, written in a given programming language. The meta-model corresponds to the formal grammar of the language and the model is executable. The same source program could be in turn considered as a system and other models can be extracted from it (static analysis). One particular execution of this program may also be considered as a system and other models can be extracted from this execution. A typical example is an execution trace that could be considered as a model, based on a simple specific meta-model. The concept of "just-in time" model production is presently taking shape. All models generated by a given execution program should not be based on the same meta-model. Exploitation on these dynamically generated models may be done concurrently with the execution of the program.

There is currently a lot of thinking about platform description models (PDM). Very often these are kept implicit in the MDA presentation. This means that the corresponding information should be hidden in software production tools (CASE tools or IDEs). This is in contradiction with the global goal of the MDA that is to make explicit the notion of platform independence and the notion of platform binding. The concept that comes closer to the idea of a platform is the notion of virtual machine. A virtual machine may be built on top of other virtual machines and the relations of abstraction between different virtual machines may be made explicit. There is a lot of insight in the explicit definition of virtual machines that may be reaped out of the research in these topics that occurred in the 80's. Also we need a classification of various platforms according to the technology they are using or to the underlying paradigm on which they are based: objects, components, services, rules, services, transactions, etc. Practically we need models of most popular platforms like CORBA, EJBs, Unix, Windows, etc. The CCM may be considered to contain such a material while proprietary or collective efforts are producing other models (like the Sun Java Community Process). Here also a platform model may be composite in the sense that it may contain several different components (e.g. an OS like Linux and a DBMS like Oracle). Many efforts are presently aiming at capturing a model of the WEB as a specific MDA platform, in order to be able to automatically generate systems for this important target.

When the notion of PDM and virtual machine is clarified we may then tackle the definition of a PIM, a model containing no elements associated to a given platform. In other times this was simply called a business model, but as for platform models we need to progress now towards a less naive and a more explicit view. The first idea is that the PIM is not equivalent to a model of the problem. Many elements of the solution may be incorporated in a PIM as long as they don't refer to a specific deployment platform. For example we may take algorithms hints into account. There may be several PIM to PIM transformation before arriving to the state where a PIM may be transformed into a PSM. To express this idea, some use the notion of a CIM (Computation Independent Model), which is a PIM where the problem has not yet been worked out as a solution.

The previous list is by no means complete. We could mention for example test models, the way they are produced and the way they may be used and many more categories of models and meta-models. This list is however sufficient to show that there is at least a similar diversity in the various operations intended to operate on these items. We may then attempt now a classification of operations on models and meta-models.

Some operations apply on a single model and are called monadic by opposition to dyadic operations applying to two models. Operations applying on more than two models are more rare.

A model may be built, updated, displayed, queried, stored, retrieved, serialized, etc. When most of these operations are applied, there is an implicit or explicit meta-model. A UML CASE tool has very often an integrated built-in UML meta-model. Sometimes it is possible for such a case tool to dynamically adapt to new versions of the UML meta-model. In a similar way, a meta-model may be built, updated, displayed, queried, stored, retrieved, serialized, etc.

Efficiently storing and using a model or a meta-model to/from persistent storage is not always easy to implement, especially when configuration and version management are involved. In many cases using simple file systems after XMI serialization lacks efficiency and don't scale up.

Filtering a model means extracting a specific view on this model. The important question here is how the view is specified and if this operation may be considered as a dyadic operation producing another model. There are many other apparently monadic operations that turn out to be dyadic, if we are able to define the argument as a meta-model or a model. Some examples are measure, verification, normalization, optimization, etc.

Rapid prototyping is a special kind of operation that associate some limited executability to a model in order to interactively evaluate its properties before transforming it into an executable systems. This may not be done with any meta-model. A typical usage is to find design errors in an initial UML model for example.

Obviously one of the most popular operations on models is code generation. One may convert a UML model into a Java or a C++ programs. Many aspects of this kind of operation should be considered here. In some cases we may consider bidirectionality, i.e. backward as well as forward transformation, with for example transformations of C# code into UML code. However one should not be misled by the apparent simplicity of these transformations.

Usually the underlying algorithms are quite simple and much progress will be made in the coming years in this area. If we consider the target language (C++, C#, Java, etc.) as defined by a meta-model (corresponding to its grammar), then we may envision generic transformations and really parameterized tools.

Many dyadic operations on models should also be discussed. A confrontation of two models may produce the similarities and differences between two models, possibly as a new model. The relations between the involved meta-models are obviously of higher importance. An alignment of two models may allow to define a new model if some equivalence rules would have been defined. A fusion/merge of two models is more complex operation since it supposed the the weaving rules have been specified in a third model.

There are many other notions to discuss, for example the life cycle of models and meta-models. When describing the life cycle of a model, we can make the distinction between incremental verification and batch verification. When considering the life cycle of a meta-model for example we should also take into account its evolution after being put in service. The need to cope with various version numbers of the UML meta-model is an example of how serious this problem may be.

Obviously the most apparent components in an MDA workbench are the precise tools composing this workbench. Fortunately in this context we should be able to propose a rather precise definition of a tool: it is an operational implementation of a set of operations applicable on specific models. The meta-models supported by a tool should be exhaustively and explicitly defined. As a consequence of this, the tool may also be referenced in a given process model for automatic (stand-alone) or semi-automatic (in association with a human agent) execution.

An MDA tool implements a set of operations on models and meta-models. For example Rational Rose, Objecteering or Argo UML implements model creation and browsing, serialization (XMI), code generation (limited set of languages like C++ and Java), etc. Any MDA tool will have a unique operation profile. To compare the tools, we need to base the comparison on the precise list of implemented operations.

The main characteristics of MDA tools is that they should be interoperable, i.e. they should be able to operate on a virtual MDA platform, collectively implementing complete chains of operations on models. This is one of the main advantages that should come from the usage of standards like MOF, XMI, SPEM, UML, CWM and many others.

4. Transformations

Obviously the last mentioned operation (transformation) in the previous section is essential. This is why we are going to discuss it here in more detail.

A transformation t transforms a model Ma into another model Mb

$t: Ma \rightarrow Mb$

Model Ma is supposed based on meta-model MMa and model Mb is supposed based on meta-model MMb . We note this situation as:

$sem(Ma, MMa)$

$sem(Mb, MMb)$

As a matter of fact, a transformation is like any other model. So we'll talk about the transformation model Mt .

$Mt: Ma \rightarrow Mb$

Obviously since Mt is a model, we postulate the existence of a generic transformation meta-model MMt , which would be similar to any other MOF based MDA meta-model:

$sem(Mt, MMt)$

$sem(MMt, MOF)$

$sem(MOF, MOF)$

The nice property here is that we may envision the possibility of transformations producing transformations (higher order functions):

$Mx: Mu \rightarrow Mt$

Obviously the existence postulate for MMt is based on the forthcoming result of the OMG RFP on the transformation language.

The scheme described above should be completed in several aspects. First one should provide the process model associated to a set of transformations, i.e. a definition of when the transformations are applied, who is applying them, what are the pre-conditions and post-conditions of these transformations and so on.

In some cases the transformation takes some particular form if the source and target meta-models are in the relation of refinement like a CORBA and a CCM meta-model.

We may also understand the need to check before applying a transformation the consistency of this transformation by studying the relation between both meta-models.

Further more there is a subtle and important aspect that is related to transformation, which is traceability. A transformation operation should produce a traceability model Mtr between the elements of the target model Mb and the elements of the source model Ma . Obviously this traceability model is based on a given meta-model $MMtr$. There are obvious relations between MMa , MMb , MMt and $MMtr$. The study of these relations is at the hearth of the definition of the MDA.

Presently the main issue in the MDA initiative is to succeed in defining "transformations as models". This means that there should be a generic MOF-compliant transformation meta-model defining some kind of UTL (Unified Transformation Language). The answers to the ongoing MOF 2.0 QVT [10] (Queries/Views/Transformations) should be shortly know and this will have on the MDA technology an impact as least as important as the impact of XSLT got on the XML community.

5. Conclusions

MDA is about an important paradigm change, from objects to models, from interpretative to generative approaches. Like Monsieur Jourdain, many people are today discovering or claiming that they have been applying MDA principles for decades, and in some cases this may be true. However, with a lot of hype accompanying the model engineering movement, it seems

important to clearly state the essence of this new way of designing information systems and conducting software engineering projects. We have expressed the view, in this paper, that the main advantage of MDA is its proposed regular organization for various kinds of information assets related to software systems under construction, under operation or under maintenance.

Considering models as first class entities is a bigger change that it may appear at first. It obliges us to make explicit many working habits with very often the consequence of improving the associated process. The idea that any model is associated to one specific ontology (its meta-model) is of paramount importance. It allows us to state that each model exactly represents one aspect of a system and to build on solid grounds when discussing aspect separation, aspect weaving and other operations.

One particular advantage of proceeding in this way is that we are allowed to discuss classification schemes for models and meta-models. We have presented in this paper two such schemes, the former considering product, process and transformation models and the latter dealing with PIMs, PDs and PSMs. Both classifications are central to the MDA, yet they are not completely defined. Work is still in progress in defining what a transformation model is and making explicit the notion of platform independence and the operation of platform binding. When these ongoing efforts will converge, we will see more clearly the important impact of the MDA.

Two important operations in model engineering are elicitation and transformation. We use the word elicitation to mean capturing the details of a given external system into a precise MDA model. Once a model is built (in the MDA space) it may be operated upon by automatic and semi-automatic tools, for example undergoing successive transformations.

Transformation operations are common to the MDA and to generative techniques. There is a lot of insight and know-how that has been progressively set up in the last fifty years on automatic transformations. Since a program is a model (its meta-model corresponding to the grammar), program transformation may be seen as a special case of model transformation. There are however many other sources of inspiration when dealing with transformation in the MDA sense (i.e. graph transformation). The DBMS community may provide a lot of algorithms for schema transformation that could be easily adapted to the MDA context.

We have discussed in this paper a lot of concepts related to the MDA. In some cases we are rather satisfied by the provided definitions. In other cases the notions are not yet satisfactorily

defined and much more work is needed. We hope this initial investigation is helpful in identifying the basic notion of an MDA artifact and may help defining a general MDA platform.

6. References

- [1] Bézivin, J. & Gerbé, O. Towards a Precise Definition of the OMG/MDA Framework ASE'01, San Diego, USA, November 26-29, 2001
- [2] Bézivin, J. From Object Composition to Model Transformation with the MDA TOOLS'USA 2001, Santa Barbara, August 2001, Volume IEEE publications TOOLS'39.
- [3] D'souza, D. Model-Driven Architecture and Integration: Opportunities and Challenges Version 1.1. February 2001, Available from www.kinetiuy.com
- [4] Gérard, S., F. Terrier, et al. , Using the Model Paradigm for Real-Time Systems Development: ACCORD/UML, OOIS'02-MDSD, Montpellier, Springer LNCS 2426.
- [5] Kobryn, C. The Road to UML 2.0: Fast track or Detour. SD Magazine, April 2001.
- [6] Kobryn, C. UML 2001: A Standardization Odyssey. Communications of the ACM, V.42, N.10, 1999
- [7] Lemesle, R. Transformation Rules Based on Meta-Modeling EDOC,'98, La Jolla, California, 3-5 November 1998, pp.113-122.
- [8] OMG/MOF Meta Object Facility (MOF) Specification. OMG Document AD/97-08-14, September 1997. Available from www.omg.org
- [9] OMG/XMI XML Model Interchange (XMI) OMG Document AD/98-10-05, October 1998. Available from www.omg.org
- [10] OMG/RFP/QVT MOF 2.0 Query/Views/Transformations RFP, OMG document ad/2002-04-10. Available from www.omg.org
- [11] Soley, R. and the OMG staff Model-Driven Architecture. OMG document Available from www.omg.org November 2000.
- [12] Sun Java Community Process JMI Java MetaData Interface Specification Available from ftp://ftp.java.sun.com/pub/spec/jmi/asdjhfgghg44/jmi-1_0-fr-spec.pdf