Considerations on modeling and model transformation in MDE

KARIM EL GUEMHIoui
Department of Computer Science and Engineering
University of Quebec in Outaouais
101, Saint-Jean-Bosco, C.P. 1250 succ. Hull
Gatineau (Québec) J8X 3X7
CANADA

Abstract: - MDA, MDE, and MDD are some of the latest buzzwords in the software development area which unveil the still emerging and evolving nature of the field. Though promising, the approach still lacks theoretical foundations. In this paper, we quickly review and explain the approach. We reflect on foundational concepts, question some key notions, and bring additional insight to the process. Finally, we elaborate on the open issue of model transformation and semantics preservation and propose some elements of solution.

Key-Words: - MDA, MDE, Modeling, Metamodel, MOF, PIM, PSM, Modeling language semantics

1 Introduction
It is a truism to state that, on the one hand, software is expensive and, on the other hand, the demand for software is ever increasing. Modern software systems are characterized by their complexity and distributed nature. Furthermore, the technological middleware platforms they are deployed on (e.g.; CORBA, J2EE, .NET) seem to evolve endlessly raising two somewhat conflicting issues:
a) How to continue keeping up with the latest middleware technology that industry and customers are pushing us to embrace;
b) How to recoup our investments in past and present costly middleware platforms.

The Object Management Group (OMG), a not-for-profit consortium that creates and promotes object-based software interoperability specifications, has come up with the Model Driven Architecture (MDA) initiative [9]. The basic idea of MDA is to decouple the specification of system functionality from the specification of that functionality on a specific middleware platform; both specifications being expressed as models.

In MDA, the Unified Modeling Language (UML) is a privileged means for expressing user models. UML itself is defined in terms of a higher expression language called the Meta Object Facility (MOF).

The key concepts in MDA are Platform Independent Models (PIMs) and Platform Specific Models (PSMs). The PIM is used to describe the business or application logic. The goal is to describe the functionality completely using a vocabulary of the problem domain. The PSM is a model that expresses the system in terms of an implementation technology. Classical examples of implementation technologies are J2EE/EJB, .Net, CORBA, XML/SOAP, etc. The PSM can then easily be translated into code in a traditional programming language.

2 A Closer Look at Modeling
In MDA, models are first class entities, and not only used to sketch design ideas and probably be thrown away once the code has been written. Therefore, it is important to have a good understanding of what a model means and how to express it.

A model is a representation of a system under study. We build models to increase productivity, under the assumption that it is cheaper to manipulate the model than the real thing. Models then enable cheaper exploration and reasoning about the real system.

It is important to differentiate between the role that a model plays as an interpretation of some real system under study; and the role it plays as a specification of some software system to build. In the former case, the model represents a set of statements about the system under study; and it will be considered correct if all the statements are true for the real system. In other words, the interpretation of the model gives the model meaning relative to the system under study [10].

In the latter case, the model serves as a design specification to which the software system (under construction) must comply. Any deviation from the specification invalidates the system, not the model. In software engineering, models take on both meanings: in the analysis phase, they are used to interpret the system (under study); then in the design phase, they are evolved into a specification of the system (to develop).

In any case, a model must exhibit some qualities. To be useful and effective, it must be: abstract (emphasizing
important aspects, hiding irrelevant ones), understandable (expressed in a form readily understood by users), accurate (faithfully representing the modeled system), predictive (can be used to derive correct conclusions about the system), inexpensive (cheaper to construct and study than the real system) [11].

Finally, a model should be expressed (i.e.; written) using some formalism. Note that the expression language used for modeling exists at some abstraction level. For example, if a model is expressed in UML, the realization of associations will be ignored by the modeling language (since it readily supports this concept) and left to a model compiler or a human programmer [8].

3 Metamodeling

A metamodel is a model of a well defined language used to express a model. Thus, metamodeling consists in defining the grammar and vocabulary, called metamodel, allowing the realization of a family of models. In other words, a metamodel is a specification model of a class of systems, where each system is itself a valid model expressed in a certain modeling language [10].

![A Simple Java Metamodel (Fragment)](image)

Fig. 1. A Simple Java Metamodel (Fragment)

For example, in Java (a modeling language for specifying source code), we can use the concepts of “Class”, “Interface”, “Method”, and so on, when defining a Java program, because in the metamodel of Java there are elements that define what is a class, an interface, a method, etc. Figure 1 shows a fragment of a Java metamodel.

Similarly, the UML metamodel specification defined by OMG is indeed a metamodel for UML that comprises all the abstract syntax, concrete graphical notation, and semantics for UML. A specific user-defined UML model can then be viewed as an instance of the UML metamodel. If the model representation is consistent with its metamodel (i.e.; no element of the UML model contradicts the metamodel), it is considered valid; otherwise it is invalid. Note that a single modeling language (e.g.; Java or UML), might have more that one metamodel.

Now, a metamodel being itself a model, it can be expressed using some modeling language. This modeling language can, in turn, be specified using a metamodel, which itself can be expressed using some higher level modeling language, and so on. To put an end to this otherwise endless escalation in recursive definitions, we need to define the concept of reflexive metamodel. In a reflexive metamodel, the same modeling language whose specification is the metamodel is also used to describe the metamodel. Such metamodel is said to be self-describing.

OMG has defined the Meta Object Facility (MOF) as a reflexive metamodeling language, in terms of which, ultimately, all the other models can be defined. The MDA architecture consists of four layers (Figure 2) with MOF being at the top level, therefore, not requiring any higher layer because of its reflexive nature.

At the M0 level, lies what is to be modeled. At the M1 level, reside the models (e.g.; a UML model). The M2 level holds the metamodels (e.g.; a UML metamodel). Finally, the M3 level holds the MOF metametamodel.

Note finally that when we say that a model is an instance of a metamodel, we are abusively characterizing the exact nature of the relationship between a model and the metamodel to which it must conform. In fact, the relation between these two entities is more akin to the relation between a programming language and its grammar.

4 MDA: Special Case of MDE

Model Driven Engineering (MDE) encompasses and goes beyond MDA by breaking free from the central position of the couple formed by MOF and UML. MDE still relies on modeling as a central activity and models as first class entities. However, it recognizes that UML/MOF may not fit every problem, and that there are technological spaces more appropriate for some domains. A technological space (TS) is a working context with a set of associated concepts, body of knowledge, tools, required skills, and possibilities. It is often associated to a given user community with shared know-how, educational support, common literature, etc.
Example: the XML TS, or the DBMS TS. Note that MDE still borrows the four layer modeling architecture, therefore a TS is characterized by a distinct and unique meta-metamodel at the M3 level.

Fig. 2. MDA Layered Architecture

The buzzword MDD, which stands for Model Driven Development, is also often used to refer to model-based design and implementation approaches, whether they are MDA or MDE.

5 Model Transformation and Semantics

The key challenge of the model-driven approach is to automatically transform an input model (usually, at an abstraction level close to the problem domain) into some output model whose format is acceptable by some tool that can generate the corresponding code. By automating the transformation, accidental design complexities are avoided and productivity is improved; similarly to what happened when we moved from assembly to third generation programming languages, and relied on the compiler to automatically produce the source code.

Though model transformations manipulate models at the M1 level, the transformation itself is specified at the level of the metamodels (M2). This ensures that the transformation works for any model conforming to its metamodel, and that it can be executed time and again.

A model transformation can be horizontal or vertical. A vertical transformation occurs when a source model is transformed into a target model at a different level of abstraction (e.g.; from PIM to PSM, or from PSM to code). A horizontal transformation occurs when the target model remains at the same level of abstraction as the source model (e.g.: at the M1 level, from EJB to .NET [2]). An additional degree of difficulty arises if the input and/or output models consist of more than a single model. In the vertical context, an example would be the projections of a simple abstract model into several specific models each one embodying a distinct implementation technology. In the horizontal case, an example would be the merging of several input models, each one depicting some aspect of the system (e.g.; security, fault tolerance), into a single consolidated output model at the same level of abstraction.

The source and target of the model transformation may or may not belong to the same technological space (TS). In the former case, both models are expressed in modeling languages that share the same meta-metamodel (at the M3 level). In the latter case, the modeling languages used to express the input and output models belong to separate TSs with distinct M3 metamodels, making model transformations much more complex.

Finally, an essential property we are seeking in a transformation is to insure that the output model still represents the same system as the one depicted by the input model. Therefore, a transformation must not limit itself to syntax conversion, but also take into account semantics. We recall that syntax and semantics, of a modeling language used to express a model of a real system, are defined at the metamodel level.

If the semantics of the input modeling language is a subset of the semantics of the output modeling language (Figure 3), fully specifying the transformation should not be a problem. Otherwise, several situations may arise.

Fig. 3. Domains with Shared Semantics

Depending on whether the modeling languages of the input and output models share some common semantics, it may be impossible to define a transformation that preserves the semantics of the original system. The complexity of the issue is amplified by the fact that the semantics of input models are not always clearly defined.
Figure 4 illustrates the case of input and output modeling languages not fully sharing the same semantics. We may not be able to fully describe a real system using only the shared portion of the semantics. For a more detailed discussion of all these issues (beyond the scope of our paper), we refer the reader to [5].

From the previous discussion, it follows that we cannot generally ensure that both models represent indeed the same system; and that the best we may hope for is that at least they represent equivalent systems with respect to behavioral and non behavioral properties.

6 Validation of Transformations

Ensuring that a transformation produces an output model that complies with the input model can be a very complex task. How can we assess the quality of a transformation? Going all the way down to code generation would only show, on a case by case basis, that a running instance of the output model behaves as expected in comparison to an instance of the input model. This is far from constituting a satisfactory proof. In our context, validation based on testing suffers from the well known combinatorial explosion problem.

As briefly mentioned, the definition of a transformation that takes as input a model M1 expressed in some formalism whose metamodel is MM1, and produces as output a model M2 whose metamodel is MM2, is generically defined by specifying a mapping between the two metamodels MM1 and MM2, as illustrated in Figure 5. Guaranteeing that the transformation produces a model that still represents the same system (or an equivalent one) boils down to proving that the mapping preserves semantics.

Currently, most of the model transformations publicized rely on some manual inspection to assert that each and every element of the target metamodel is required to embody and transmit the properties and semantics of the source metamodel [1]. The process is tedious, error-prone, and non reproducible, since it relies on the expertise of the developer and her familiarity with the formalisms in which both models have been expressed. In annex, we show an example of published transformation [4] that is nicely and rigorously defined, but whose validity cannot be fully appreciated by a reader without a deep understanding of both metamodels.

[6] addresses testing and validation of transformed model, but only with respect to the preservation of syntactical properties. Few endeavors have dealt with the preservation of semantics. Most of the research work on this issue fall within one of these three categories:

- Use formal languages to express metamodels before submitting them to a formal validation. In addition to the complexity of the process, a large spectrum of applications and domains don’t lend themselves to formal specification.
- Use probabilistic computation results from repetitive executions of instances of the two models to evaluate the compliance of the output model to the input one based on execution feedback. We have already expressed our reservation regarding model validation based on testing.
- Use a semi-formal process that takes into account the domains of the models to compare, and the intention behind the transformation. Our approach falls into this last category.

As a starting point, we need to classify input and output metamodels according to the degree of overlapping of their respective semantics. We identify the following three situations:

- All the semantics of the input metamodel can be represented in the output metamodel (Fig. 3).
- Only a part of the semantics of the input metamodel is directly represented in the output metamodel (partial overlap; Fig. 4).
- The two metamodels don’t share any common semantics.

Note that whether all the semantics of the target metamodel can be represented in the source metamodel is irrelevant, unless a bidirectional transformation (i.e., reversible) is sought. Indeed, a unidirectional model transformation only aims at expressing the semantics of
the source metamodel in terms of the semantics of the target metamodel.

In our first situation, the transformation is rather straightforward and is mainly a matter of identifying the corresponding elements whose semantics are equivalent. Note however, that a single element from the input metamodel may be represented by more than one element of the output metamodel; or vice-versa.

In the second case, we will have partial equivalence between the models. Depending on the domain and the application, the transformation may or may not be acceptable. In the last case, the semantic gap is irreconcilable.

To facilitate the transformation validation, we need to narrow the semantic gap between the two metamodels MM1 and MM2. We assume that both metamodels belong to the same TS. The main idea of our approach is to replace MM1 and MM2 with an intermediate model MM0 with the same expressive power. We are sure of finding an MM0, since we can always take as a starting point for MM0 a metamodel consisting of MM1 and MM2. However this resulting metamodel has just worsened the problem because MM0 lacks cohesion (duplications) and is much heavier to handle due to a larger number of elements. But if we can refine and purge MM0 so that it only contains modeling elements expressing shared semantics between the two initial metamodels MM1 and MM2, and no semantics is represented twice, then by expressing the models M1 and M2 in terms of MM0 we can immediately verify whether they are equivalent or not (see Figure 6).

Whenever a shared semantics \( S_j \) is entirely and solely represented by a single element \( E_1 \) of MM1 and a single element \( E_2 \) of MM2, only one of the two elements is maintained in MM0.

In one of the metamodels MM1 or MM2, the \( S_j \) semantics is not represented by a single element but by the association of several elements. We will say that each element in the association represents a part of the \( S_j \) semantics. (3) A combination of (1) and (2). The processing consists mainly in the extraction of a specific shared semantics from larger semantics by either splitting existing metamodeling elements or by creating new ones (to add to MM0). The purpose behind the extraction operation is to make a semantics manifest exclusively in a single \( E_j \) element (allowing MM0 to be purged). In practical terms, the extraction operation entails the rewriting of metamodel fragments. We have developed algorithms for the refinement of MM0 as well as a set of metrics to gauge the quality of the transformation. The algorithms use an iterative approach and can be partially automated. Detailed explanations beyond the scope of this paper can be found in [3].

### 7 Conclusion

There seems to be some misconceptions about the core technology behind the model driven approach and reluctance in embracing it. In this paper, we have helped demystify some issues by providing a detailed discussion of some key concepts of the model driven development approach. We have elaborated on the activity of modelling and metamodeling; and we have clarified the issues at stake when transforming models.

We have presented a semi-formal approach to verify whether a model transformation preserves the semantics of the original model. Though our approach is grounded in the MOF technological space, we believe that it can easily be ported to other TSs.

We are currently developing a framework for the automation of our approach with minor input from the developer. We are also refining a set of metrics for assessing the quality of model transformations.

### References:


Appendix

Metamodel mapping from UML to WSDL:

Bézivin, 2004