Tooling the MDA framework: a new software maintenance and evolution scheme proposal

Jean Bézivin & Nicolas Ploquin

Pôle Modélisation, CRGNA, Université de Nantes,
2, rue de la Houssinière, BP 92208
44322 Nantes cedex 3, France
http://www.sciences.univ-nantes.fr/info/lrsg/

Abstract:

The Object Management Group is rapidly moving from its previous Object Management Architecture vision (OMA) to the newest Model-Driven Architecture (MDA). At the center of this framework, the so-called four-level meta-modeling architecture provides the foundations for building a variety of automatic and semi-automatic model-transformation and code-generation tools. The most important will be platform targeted code-generation tools but there are plenty of other possibilities that we can expect to become available in the near future. As an example, this paper shows how modern meta-modeling and meta-programming techniques may be combined to implement a new generation of software maintenance tools. The proposal deals with the future maintenance of software written in the C# programming language, supported by the DotNet platform and uses all support available in the MDA framework (UML, MOF, XMI). In addition to this, we take advantage of the introspection properties of C#. A similar approach could also be used with other modern programming languages like Smalltalk or, to a lesser extent, Java.

Keywords:

Software evolution and maintenance; Meta-modeling; Meta-Programming; Reflection; MOF; MDA; Smalltalk; Java; C#

1 Introduction

The Object Management Group (OMG) is proposing a completely new vision called Model Driven Architecture (MDA) [14], [7] for constructing and maintaining information systems. The Unified Modeling Language (UML) has already gained wide acceptance for describing all kinds of object-oriented software artifacts. To allow other similar languages to be defined as well, the OMG uses a general framework based on the Meta-Object Facility (MOF [11]).

This paper presents an original software evolution scheme, proposed to become part of the MDA framework. The described prototype implements a first version of the scheme as a simple C# program [15], using the basic reflective mechanisms of the language. The main idea is to allow the C# program to automatically generate a model of itself, based on a given specific MOF-compliant meta-model. In the first prototype, the correspondence between the meta-model and the generation code is hand-synchronized. Building on this preliminary work, a more automatic way of proceeding is suggested.

The paper is organized as follows. In section 2 we present the OMG/MDA framework. We first provide in section 3 a compared definition of meta-programming and meta-modeling and proceed to describing our proposed scheme and its initial implementation. Some related work is presented in section 4. The conclusion summarizes the original contribution of this work and discusses several extension paths.

2 Presentation of the OMG-MDA framework

2.1. Multiple meta-models

The MOF has emerged from the recognition that UML was one possible meta-model in the software development landscape, but that it was not the only one. Facing the danger of having a variety of different non-compatible meta-models being defined and independently evolving (data warehouse, workflow, software process, components, etc.), there was an urgent need for a global integration framework for all meta-models in the software development scene. The answer was thus to provide a language for defining meta-models, i.e. a meta-meta-model. This is the purpose of the MOF. As a consequence, a layered architecture has been defined, with the following levels (see Figure 1):

- M3: the meta-meta-model level (contains only the MOF)
- M2: the meta-model level (contains any kind of meta-model, including the UML meta-model)
- M1: the model level (any model with a corresponding meta-model from M2).
- M0: the concrete level (any real situation, unique in space and time, described by a given model from M1).

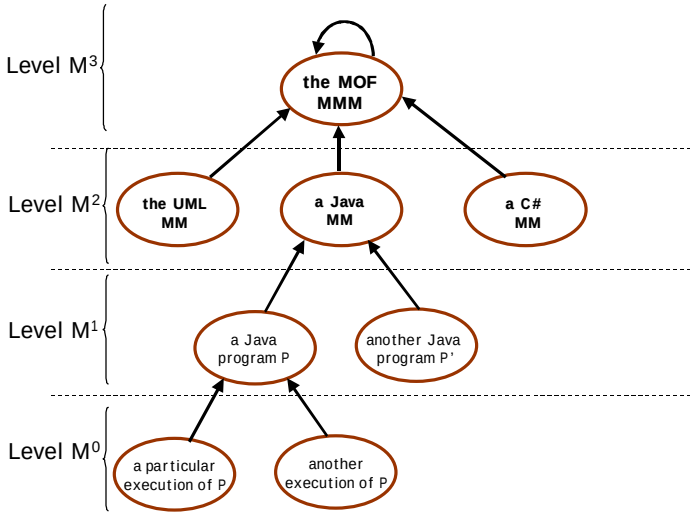

Figure 1: Several spaces, pertaining to different levels

The importance of model engineering in the information system and software development process is rapidly growing. Models are defined (constrained) by meta-models. A meta-model is used for defining a set of concepts and relations between these concepts, i.e. for abstraction filtering and consensual agreement in a particular modeling activity. From a given system, we can extract a particular model with the help of a specific meta-model. A parallel may be drawn with the domain of formal programming languages. Level M2 corresponds to the grammar level. Level M1 corresponds to the program level. Level M3 corresponds to the meta-grammar (like the EBNF notation for example). Level M0 corresponds to one given dynamic execution of a program. A given execution of a program at level M0 is not a model by itself; it is depicted by a model (the source code of the program that describes the infinity of possible different executions of this program). Exactly the same situation holds for the four OMG meta-modeling layers.

A meta-model defines a language for describing a specific domain of interest. For example UML describes the artifacts of an object-oriented software system. Some other meta-models may address domains like process, organization, tests, quality of service, etc. Their number may be very important as identified domains are highly specialized.

On the same level as the UML meta-model, we may also consider programming languages meta-models such as Java, C#, Eiffel or Smalltalk (Figure 1). Such meta-models usually have a scope that goes much beyond the basic grammar of the language. Environment features may for example be considered like *include* files in C++, class categories and method protocols in Smalltalk, etc. Similarly to grammars meta-models contains terminal and

non-terminal entities to help organize the information. To the contrary of grammars, meta-models do not respect a strict syntax/semantic boundary. They are not limited to syntactic categories but may also consider more abstract entities like classes representing *events*, or *interfaces* (GUI) or *business entities*, etc. In a Smalltalk program for example, classes corresponding to models, views and controllers in so-called MVC triads may be distinguished in the meta-model. How the recognition of these different entities may be performed is a different subject that will be discussed later in this paper. The important fact for now is to state that these categories may be distinguished and identified in a meta-model.

Enterprise JavaBeans™ (EJB) technology defines a scheme for the development and deployment of reusable Java server components. Components are pre-developed pieces of application code that can be assembled into working application systems. Java technology currently has a component model called JavaBeans, which supports reusable development components. The EJB architecture logically extends the JavaBeans component model to support server components. As a consequence, an EJB meta-model will be more useful than a simple JavaBeans or Java meta-model. These different meta-models may be separated and related by a generalization relation. Many efforts are currently going on in using basic MDA technology to help specify and deploy Java-based technology like the UML/EJB mapping specification undertaken in the scope of the Java Community Process (JSR 26).

A common mistake consists in stating that all object language schemes are equivalent. It is now becoming clear that the differences between them often overtake their similarities. An Eiffel meta-model will consider several categories of assertions (invariants, pre-conditions, post-conditions) and the relationships between these and the classes and methods. A Java meta-model will make the distinction between classes and interfaces, and will consider different extension mechanisms between them (single for classes, multiple for interfaces). Furthermore the implementation relationship between interfaces and classes will be defined. In a Smalltalk meta-model, the concepts of instance variable, class variable, pool variable and block variable will be distinguished. The concept of explicit anonymous meta-class may also have to be considered as in Figure 2. Obviously all these programming language meta-models are different from the UML meta-model.

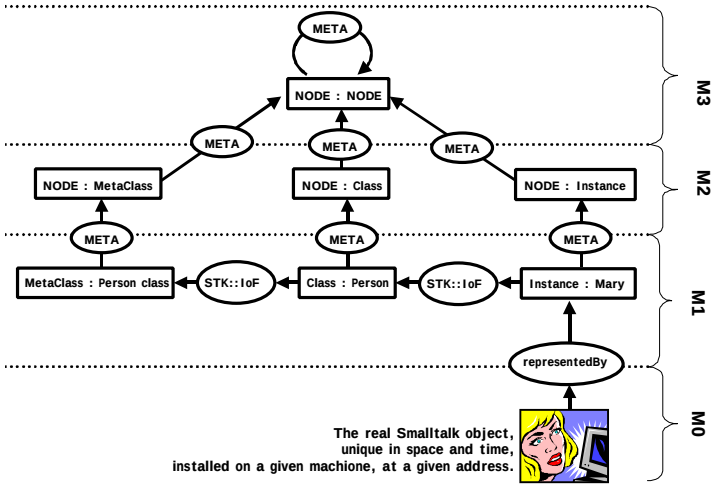

Figure 2: A Smalltalk program (M1), between meta-model (M2) and execution space (M0)

It is now also becoming clear that there is not a unique standard canonical meta-model for a given programming language. According to the task that has to be performed, a simple or extended version may be considered. The Smalltalk meta-model defined at level M2 in Figure 2 contains the concept of a meta-class, but it is doubtful this may be practically used since meta-classes are anonymous and in direct one-to one correspondence with classes. On the contrary, many other conceptual categories of the Smalltalk environment (Categories of classes, Protocols of methods, etc.) could be incorporated in the meta-model. The notation used in Figure 2 is inspired of Sowa's conceptual graphs, representing concepts as rectangles and relations between concepts as ovals. The relation *STK::Iof* used at level M1 corresponds to the "instance of" relation of Smalltalk. It is defined at level M2 (in the Smalltalk meta-model) as linking a Smalltalk instance to a Smalltalk class or a Smalltalk class to a Smalltalk meta-class, but this is not shown on the illustration by lack of place. Following the conceptual graphs graphical notation, a rectangle contains the global type and the entity, e.g. *Instance : Mary*. The upper level concept noted NODE here corresponds to the *MOF::Class* entity in the MDA. The *META* relationship corresponds to the global type-instance relationship also implicitly defined by the MOF. Notice that layers M1 to M3 are modeling layers while layer M0 is the real world. As a consequence we have emphasized here that

relations between entities of M0 and M1 are different from relations between entities of the three modeling layers M1, M2 and M3. A more complete description of the MDA framework with Sowa's conceptual graphs is being presented in [2].

2.2. Exchanging MOF-compliant models with XMI

The real change in model engineering happened when it became clear that models could be used directly in software production chains. This possibility had been used since long but for the first time we may envision its large-scale industrial deployment. Until now object analysis and design models have mainly been used to document software system. Analysts and Designers were building models that were provided to programmers only as inspiration material to facilitate the production of concrete software. The move from this "contemplative" period to a new situation where production tools will be model-driven has been facilitated by the introduction of the XMI recommendation (XML Model Interchange, [12]).

XMI defines a standard way of serializing models (level M1) and meta-models (level M2). A typical application of XMI would be to exchange UML models as a standard encoded text files, with XML syntax. There are however many other possible usages of XMI like exchanging non-UML models (SPEM, CWM, etc.) or exchanging MOF-compliant meta-models. The XMI recommendation contains a set of rules that are used toi translate MOF models to XML documents, DTDs and soon XML schemas.

It is interesting to study the real nature and meaning of this XMI recommendation, because it builds upon many other standards like UML, MOF, OCL and XML. We may recognize its importance from the fact that many new proposals at OMG are no more provided as simple paper descriptions, but as XMI DTDs as well, corresponding to the MOF-compatible meta-model of the proposal. This helps to reduce the gap between human readable and computer interpretable standards. As soon as the UML 1.4 or the SPEM proposals are released, they can be injected into real tools by the way of these DTDs.

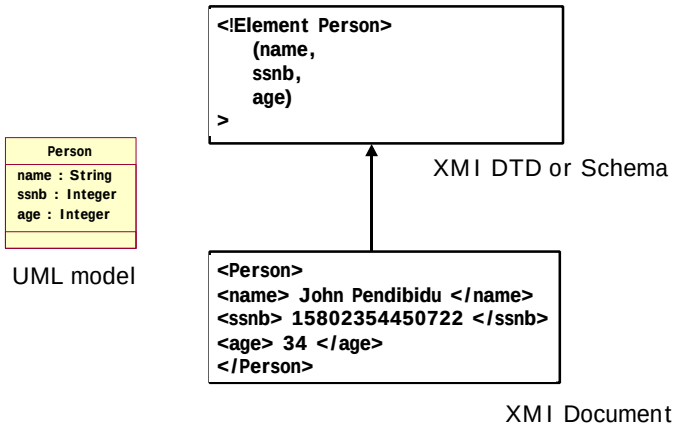

Figure 3: XMI representation of MOF-compatible models

The W3C XML standard provides the transfer syntax but also a complete technological space with widely available and well-engineered tools on which to map the MOF-compatible models. This will allow for example to apply transformation systems like XSLT to any kind of high level models. As Figure 3 suggests, there is a similarity between the relation of a XMI document to a XMI DTD or schema on one side and the relation of a MOF-based model to a MOF-based meta-model on the other side. The XML, MOF, UML and OCL standards are well integrated in XMI and play together to provide a powerful model serialization tool. The move from DTDs to XML schemas is being integrated into this process and will strengthen the resulting possibilities.

2.3. Model transformation based on explicit meta-models

The question of model transformation lies at the center of the MDA approach. The general organization of meta-model driven transformation is illustrated in Figure 4. We may consider that we have here two meta-models: the source one could be UML for example and the target one could be Java ore more realistically the EJB meta-model. The transformation of the UML model to EJB code may be specified by a set of rules defined in terms of the corresponding meta-models. The expression of these rules may be facilitated if a basic generic framework is present in the MOF. Suggestions for building this may be found in the CWM meta-model (Common Warehouse

Meta-data, one of the important components of the MDA used in particular for describing non object-oriented artifacts like relational databases tables or other legacy system artifacts). The transformation engine itself may be built on any technology like the XSLT tools.

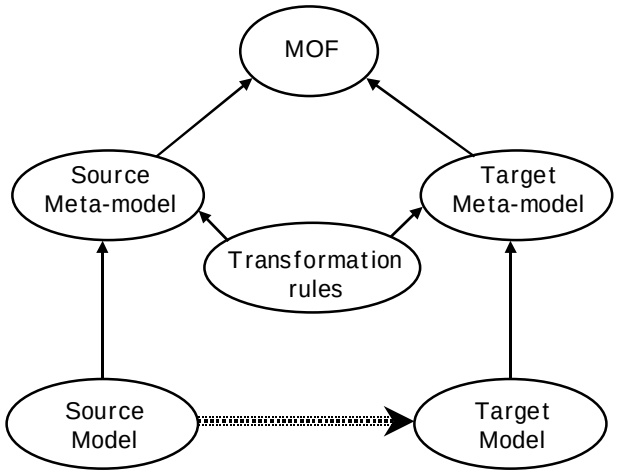

Figure 4: Meta-model based model transformation

3 Meta-Programming vs. Meta-Modelling

Meta-programming encompasses such concepts as reflection, introspection, intercession and reification. It is difficult to give a better definition of these notions that the one provided in [8]: *"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability of a program to observe and therefore reason about its own state. Intercession is the ability of a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification"*. Another distinction that is often made is between *structural reflection* and *behavioral reflection*. Unfortunately the definitions proposed in the literature are not always very clear and consistent. We do believe that a regular organization of models and meta-models may help to shed some light in this area. Program (source code) may be viewed as a model for one or all of its possible executions. There are also other models of the execution that may be dynamically extracted. At the same time the (source) program may be considered like any system and a model of this program (static system) may be extracted by code analyser tools. The various models extracted from the execution may themselves be transformed or combined together to form new models. First experiments seem to indicate that this “external reflection” may prove pratically more powerful than the classical “Internal reflection”. (unique interpretation system)

As a matter of fact, meta-programming and meta-modelling can be considered as two orthogonal dimensions. This can be assessed when considering Figure 5, which elaborates on Figure 2. The vertical axis clearly deals with the meta-modeling dimension. The ubiquitous *meta* relation is basic to this aspect. The horizontal axis corresponds here to a particular context, namely the Smalltalk meta-model. Other meta-models could be defined in this dimension. It is clear that there are obvious similarities between both dimensions, but there are also important differences. The "instance of" relation named here *STK::IoF* relates Smalltalk instances, classes and meta-classes similarly to the "meta" relation that relates *Cat*, *STK::Class* and *MOF::Class*.

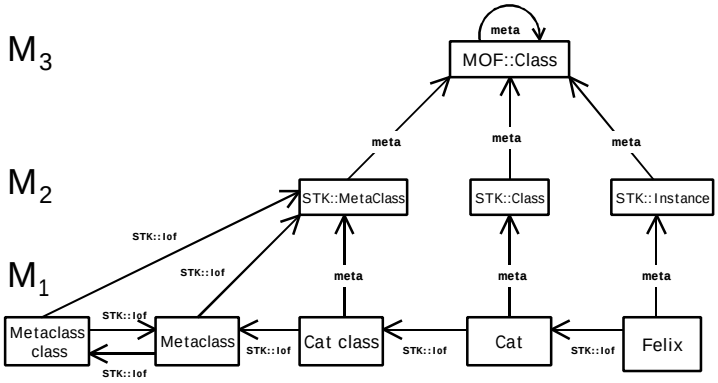

Figure 5: Two orthogonal dimensions

The categories defined at level M2 in Figure 2 and Figure 5 represent only a very small part of the entire Smalltalk meta-model. Furthermore relations holding between these categories and logical assertions expressed in the OCL language for example should complement them.

Although very simplified, Figure 6 illustrates a slightly more complete meta-model for the C# language. This example will serve to illustrate our approach. One central entity in a C# program is called an *Assembly* (This is a file containing executable code similar to a *.class* file in a Java program). Each assembly contains a number of *Types* (classes) and many other usual object-oriented artifacts like *Methods*, *Constructors*, *Parameters*, *Attributes*, *Properties*, etc. A more detailed description of this meta-model may be found in [13].

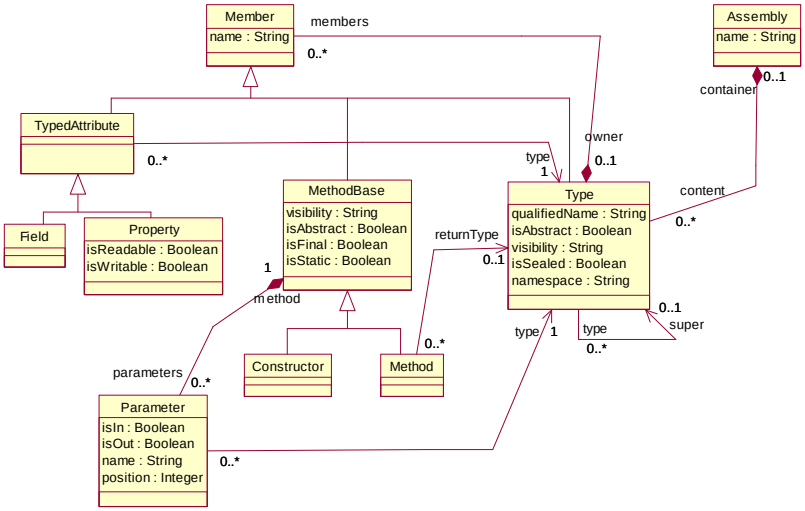

Figure 6: A simple C# meta-model

The general organization of our proposed scheme is described in Figure 7. As a *Main* method launches a program execution, a *Rep* method launches a model generation (branch 2). To achieve the generation, we have to attach some semantics to the source code content. This is being done manually in the first prototype, i.e. the reflective code in the C# program is hand-written from the observation of the meta-model.

Our intent later is to pass the reference of the meta-model as a parameter to the *Rep* method, which will read and interpret it (branch 1). The meta-model will describe the semantics of the programming language, at an abstraction level that may be parameterized (see the previous discussion about the example of Java, Java Beans and EJB meta-models).

In our initial experiment, a C# source program could for example use the meta-model described in Figure 6.

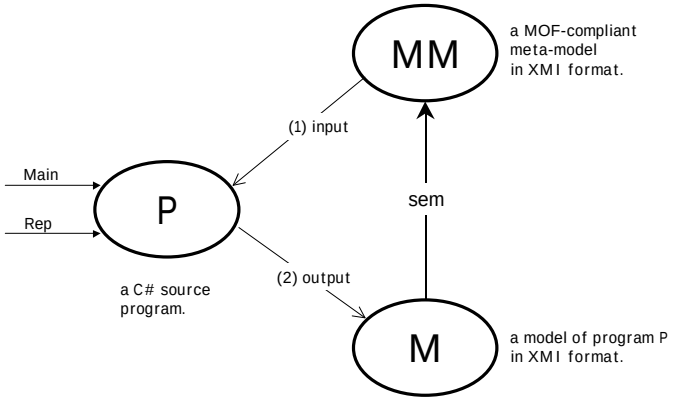

Figure 7: General organization

Most object-oriented programming languages such as Java, C# or Smalltalk provide class libraries to explore a source program. The *Rep* method has to link input meta-model to such class libraries. Actually, the *Rep* method implements a hand-made generation. To get a flavour of this hand-made generation of the XMI model of program P, we may examine at the following code excerpt:

```
Assembly ass = Assembly.GetExecutingAssembly();
    … ;
    Type[] typs = ass.GetTypes();
    foreach (Type typ in typs)
    {
        … ;
        MemberInfo[] mems = typ.GetMembers();
        foreach(MemberInfo mem in mems)
        {…}
        MethodInfo[] mets = typ.GetMethods();
        foreach(MethodInfo met in mets)
        {…
            ParameterInfo[] pars = met.GetParameters();
            foreach (ParameterInfo par in pars)
            {…}
            …
        }
    }
    …
```

To explore a source program from the C# meta-model (Figure 6), we first need to get the executing *Assembly*. Next, we use the Assembly class of the C# class library to get the description and content of this assembly. According to C# meta-model, an assembly contains types. Just as assembly, we use the *Type* class of the C# class library to get the description and content of each contained type, and so on. On each step, we generate the related XMI code.

The generated XMI file looks like the following code:

```
    ...
    <Assembly.content>
        <Type xmi.id="_2_">
            <Member.name>aClass</Member.name>
            <Type.namespace>aNamespace</Type.namespace>
            ...
            <Type.members>
                <Field xmi.id="_3_">
                    <Member.name>aField</Member.name>
                    ...
```

The C# generation code is composed of two kind of instructions. Some are meta-models dependent and others are meta-models independent. To generalize the generation to any types of meta-model, we have to clearly

separate from the C# generation code all meta-model specific instructions used to generate XMI data. Such instructions must be added to the meta-model.

In our proposal, constraints, attached to each class of a meta-model, specify which methods of the C# reflection namespace must be used to define attributes or references value. A constraint ({XMIWriter}) contains two parts. The *Element* section is a boolean expression which must be true to allow the generation on an C# element. This condition can be as simple as testing an entity type. However, it can be more complex and test (e.g.) whether a class inherits of a particular class ([this is Type && this.BaseType is Object]). Conditions are expressed using C# programming language. More generally, text in square brackets is C# code.

The *Content* section defines which C# instructions must be used to value attributes and references. In addition, references contain the type of *XMIWriter* constraints to apply. This type is define into braces.

The constraint *XMIWriter.Init* defines the entry point of the XMI generation. This constraint contains additional information like meta-model name, model name, model version and so on. The *RootNode* element specifies which C# "static" method gives the first source code element to generate.

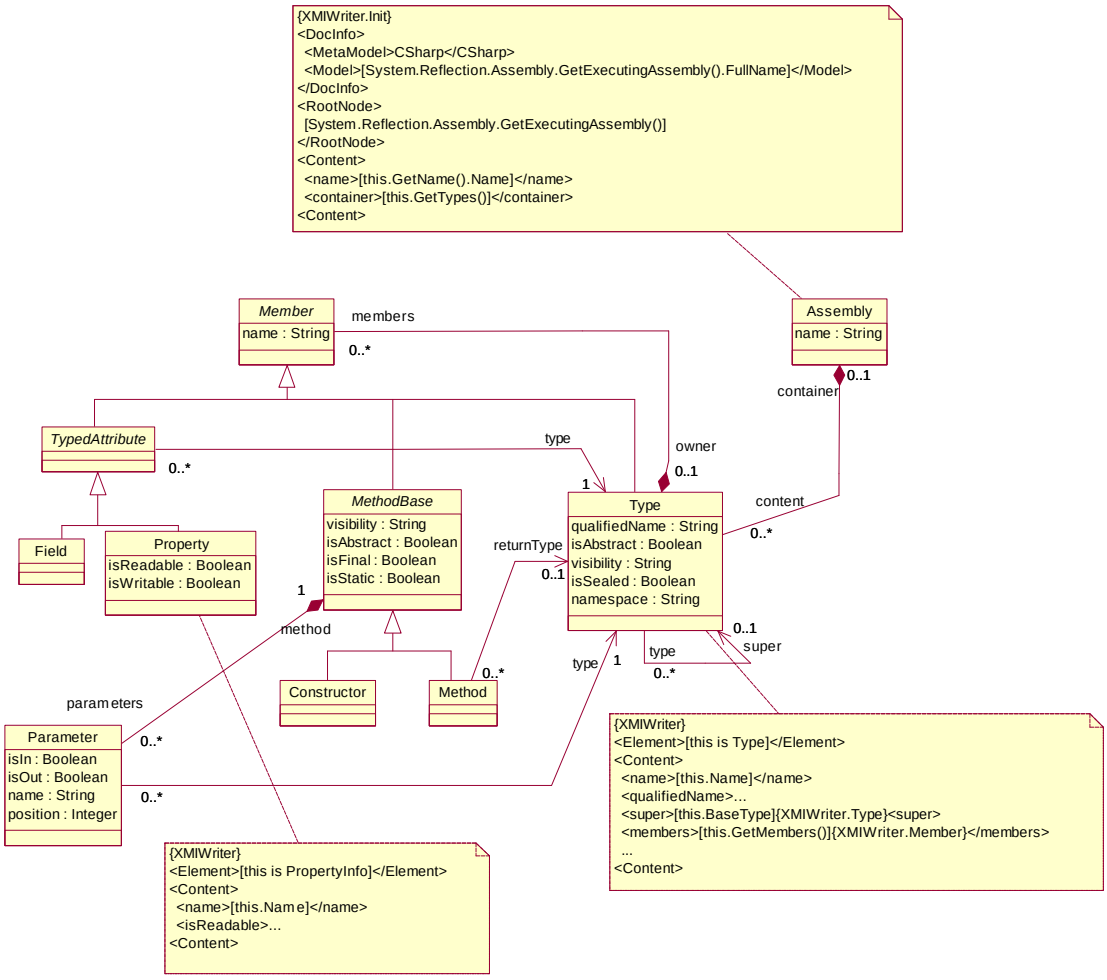

Figure 8: An example of constrained meta-model.

The potential extensibility of the presented scheme has virtually no limits. This comes from the integration into the MDA framework and the strict conformance with well-accepted standards (UML, XML, MOF, XMI, OCL, etc). Among various possibilities, we may mention:

- ?? Reverse engineering from C# to UML
- ?? Conversion to other languages, including XML, IDL, Java, etc.
- ?? Pattern extraction
- ?? Testing and model checking

?? Metrication (measurement made on the extracted model)
?? Business model and Service model extraction
?? etc.

Some of these possibilities are currently being investigated by the authors and will be described in subsequent publications.

4 Related work

Similar approaches have been proposed in the past, for more limited contexts. In [3], an extraction scheme was proposed for Smalltalk programs. Both the meta-model and the model were expressed in a variety of typed, partitioned and reflective semantic networks called *sNets*. The powerful reflection capabilities of Smalltalk made it very easy to transform a complete application and the Smalltalk-80 class library in a very rich model. Two kinds of meta-models were considered, one taking into account the complete syntax of the methods and the other one focusing only on the class architecture. The main difference between this project and the current one is that, at that time UML, MOF and XMI were inexistent. Also the idea of using the meta-model in-line to direct the generation of the model was not considered. In [5], a similar approach was also proposed for Cobol programs. Obviously, in this case, there was no hope to use introspection. This possibility had to be replaced by an external analysis program (*Tgen*, a kind of compiler generator written in Smalltalk similar to *Lex/Yacc*). This academic experimental work has since been transferred to industry and is presently commercialized under the Trademark *Semantor* by the Sodifrance Group. It has been used for several years for legacy system maintenance and evolution (Y2K, Euro, etc.).
The present work builds on these previous experiments, but adds a completely different dimension because of its close integration in the OMG MDA framework. It helps the MDA achieving independence from the platform. In order to assess this, we may compare our proposal of meta-model driven software maintenance and evolution to the Microsoft JUMP system (Java User Migration Path to Microsoft DotNet). JUMP gives Microsoft customers a number of paths for bringing their Java language investments to the DotNet platform. JUMP to DotNet automatically converts existing Java source code into C#, migrating both language syntax and library calls. Any code that cannot be converted is flagged within the Visual Studio DotNet integrated development environment to help developers easily find and quickly address any remaining conversion issues.

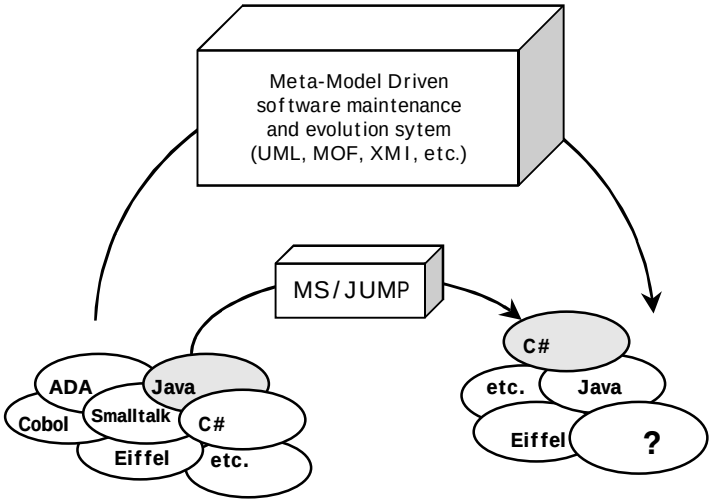

Figure 9: MS/JUMP vs. a general MDA software maintenance and evolution scheme

A comparison of our approach with MS JUMP helps us to understand some important issues of the MDA vision. As we can infer from Figure 9, higher the abstraction level, more powerful and resistant to change on the long term will be the approach. The MS/JUMP only consider the syntactical transition from Java-based software to C#/DotNet based software. Our proposal may take any input or output, on condition that adequate meta-models could be built. The high level of parameterization provided by these meta-models opens many avenues for applications and improvements.

5 Conclusions

As systems become more complex, more evolutive and more open on their environments, the software crisis seems to be worsening. Many end-users are reacting about the annoyance of iteratively moving their information

system from one platform to another one, at a high cost, with limited ROI and with little hope that this conversion process ever halts. In theory, computer systems should have lives measured in decades but unfortunately in practice the obsolescence period of technical support platforms is usually much shorter. The only solution seems to lie in separating the stable parts from the variable parts of the information systems.

In reaction to this new software crisis, the OMG is proposing the MDA framework. In this proposed vision of the way to construct and maintain information systems, the regular evolution and obsolescence of technical platforms is taken as a normal process and no more as an exceptional event. Since there is virtually no hope that an ideal and universally accepted middleware platform will prevail, solutions have to be found to integrate this new parameter in modern software engineering practices. The obvious conclusion that comes out from this observation suggests to rapidly switch from code-centered to model-centered approaches. Fortunately the OMG has a set of readily available standards that could provide the backbone for the new vision: UML, MOF, XMI, CWM, SPEM, etc. These specifications guarantee the basic interoperability for the multiple tools that are needed to transform the abstract MDA vision in commercial workbenches for industrial software development and maintenance.

Tooling the MDA is thus the next important step. This means that existing technology has to be identified and that its compliance with current standards has to be assessed. This also means that new techniques may have to be invented to facilitate the emerging model engineering practices. CASE tools (e.g. Rational Rose) and Application Development tools (e.g. IBM Visual Age) will certainly have to closely interoperate if they keep separate identities. An important number of other more specialized tools will rapidly become available to populate the framework. A classification of the necessary functionalities is needed. Among the more obvious we may quote assisted model transformation, model testing, model metrication, parameterized code generation, reverse engineering, prototypers for UML specification, verifiers for OCL statements, model and meta-model management including version management and repository support, meta-model alignment, merging and comparison, process-centered environments, collaborative systems, and much more.

We have been mainly interested here in software maintenance and reverse engineering. The contribution has presented an original software evolution scheme, proposed to become part of the MDA framework. An initial prototype has been built. It implements a first version of the proposal as a simple C# program, using the basic reflective mechanisms of the language. The main idea is to allow the C# program to automatically generate a XMI model of itself, based on a given specific MOF-compliant meta-model. In the first prototype, the correspondence between the MOF meta-model and the generation code is hand-synchronized. Building on this preliminary work, a more automatic way of proceeding has been suggested. If we are able to provide facilities for meta-model-driven reverse engineering, the resulting model may be later used in a number of other operations available within the MDA framework (model transformation, verification and validation, installation on other platforms, separation of the business entities from the technical entities, etc.)

We have shown in this paper how modern meta-modeling and meta-programming techniques may play together to implement a new generation of software maintenance and evolution tools. The proposal deals with the future maintenance of software written in the C# programming language, supported by the DotNet platform and uses all support available in the MDA framework (UML, MOF, XMI). In addition to this, we take advantage of the introspection properties of C#. Our proposal is however not linked to a particular language, even if it has nice properties. C# is the latest in the series of object-oriented programming languages, but it will probably not be the last one. A similar approach could also be used with other modern programming languages like Smalltalk or Java. Complementarily of these with more conventional approaches of reverse engineering legacy systems (e.g. Cobol programs) may also be considered. As a consequence, we believe the presented solution to be general enough to be applied not only to C# software, but to a lot of other programming languages as well, of the past and of the future. The result will be an easier way to separate the evolution of the business and service models from the evolution of the implementation platforms.

6 Acknowledgements

Many issues presented here have benefited from discussions and common work with Erwan Breton.

7 References

[1] Atkinson, C, **Supporting and Applying the UML Conceptual Framework**, UML'98, Beyond the notation, J. Bézivin & P.A. Muller (eds.), Mulhouse, 1998, Springer Verlag LNCS #1618.

[2] Bézivin, J. & Gerbé, O. **New Trends in Applied Model Engineering** Submitted for Publication, (June 2001)

[3] Bézivin, J. & Lemesle, R. **sNets: the Core Formalism for an Object-Oriented CASE tool** COODBSE'94, Proceedings of the Colloquium on Object Orientation in Databases and Software Engineering, World Scientific Publishers, ISBN 981-02-2170-3, pp. 224-239

[4] Bézivin, J. **From Object Composition to Model Transformation with the MDA** TOOLS'USA 2001, Santa Barbara, August 2001, Volume IEEE publications TOOLS'39.

[5] Bézivin, J., Lennon, Y. & Nguyen Huu Nhon, C. **From Cobol to OMT- A Reengineering Workbench Based on Semantic Networks** TOOLS USA'95, Santa Barbara, (august 1995), Prentice Hall, pp. 137-152.

[6] Dirckze, R. & Iyengar, S. **A Metamodel-Based Approach to Model Engineering Using the MOF** In Bézivin, J. and Ernst, J., eds, First International Workshop on Model Engineering, Nice, France, (June 13, 2000), available at www.metamodel.com

[7] Dsouza, D. **Model-Driven Architecture and Integration: Opportunities and Challenges** Version 1.1. February 2001, www.kinetiuym.com

[8] Gabriel, R.G., Bobrow, D.G., White, J.L. **CLOS in Context – The Shape of the Design Space**. in Object-Oriented Programming – the CLOS perspective, Chapter 2, MIT Press, 1993, pp. 29-61.

[9] Guarino N., Welty, C. **Towards a methodology for ontology-based model engineering.** In Bézivin, J. and Ernst, J., eds, First International Workshop on Model engineering, Nice, France, (June 13, 2000), available at www.metamodel.com.

[10] Kobryn, C. **UML 2001: A Standardization Odyssey**. Communications of the ACM, V.42, N.10, 1999

[11] OMG/MOF **Meta Object Facility (MOF) Specification**. OMG Document AD/97-08-14, September 1997.

[12] OMG/XMI **XML Model Interchange (XMI)** OMG Document AD/98-10-05, October 1998.

[13] Ploquin, N. **Etude des différents modes d'interopérabilité en génie logiciel**, Master Thesis, University of Nantes, (2001)

[14] Soley, R. and the OMG staff **Model-Driven Architecture**. OMG document available at www.omg.org November 2000.

[15] Surveyer, J. **C# and the .NET Framework: A Better Java Platform?** Internet World, March 15, 2001.