

# Language Driven Development and MDA

Tony Clark, Andy Evans, Paul Sammut, James Willans

Xactium Limited

[andy.evans@xactium.com](mailto:andy.evans@xactium.com)

## Introduction

The Model-Driven Architecture (MDA) [1] aims to be a major step forward in the way that systems will be developed in the future. The goal of MDA is to provide an enterprise modelling architecture that enables developers to capture their organisations business and software assets.

In a typical MDA scenario, different aspects of an organisation's data will be represented in terms of models. These models will capture aspects at different levels of abstraction. Implementation independent models of the system will drive the development of the system, whilst platform specific models will provide the deployment of the system. Translation between PIM and PSM models describe the way in which PIMs are converted into a deployed system.

Although PIM to PSM transformations have an important role to play in system development, there is increasing interest in the wider application of MDA to what we term *language driven development*. Language driven development involves the application of MDA technologies to rapidly generate and integrate semantically rich languages and tools that target specific modelling requirements. The aim is to provide developers with rich modelling abstractions appropriate to their development needs, thus enabling them to clearly focus on the problem domain in isolation from implementation details.

This article discusses language driven development and the benefits it offers. It identifies limitations with the current standards that support MDA (MOF, QVT, UML) and looks at a particular approach to extending the standards to support it. We also relate this to recent discussion relating to domain specific languages (DSL's) by Steve Cook [2].

## The Role of Languages

One of the distinguishing features of being human is our use of language. Languages are fundamental to the way we communicate with others and understand the meaning of the world around us.

Languages are also an essential part of systems development (albeit in a more formalised form than natural languages). Developers use a surprisingly varied collection of languages. This includes high-level modelling languages that abstract away from implementation specific details, to languages that are based on specific implementation technologies. Many of these are general-purpose languages, which provide abstractions that are applicable across a wide variety of domains. In other situations, they will be domain specific languages that provide a highly specialised set of domain concepts.

In addition to using languages to design and implement systems, languages typically support many different capabilities that are an essential part of the development process. These include:

- *Execution*: allows the model or program to be tested, run and deployed.
- *Analysis*: provides information of the properties of models and programs.
- *Testing*: support for both generating test cases and validating them must be provided.
- *Visualisation*: many languages have a graphical syntax, and support must be provided for this via the user interface to the language.
- *Parsing*: if a language has a textual syntax, a means must be provided for reading in expressions written in the language.
- *Translation*: languages don't exist in isolation. They are typically connected together whether it is done informally or automatically through code generation or compilation.
- *Integration*: it is often useful to be able to integrate features from one model or program into another, e.g. through the use of configuration management.

## Features of Languages

Although there are many different types of languages, there are some common features that they all share. These must be understood if we are to develop a generic approach to language definition. The primary features are:

### Concrete Syntax

All languages provide a notation that facilitates the presentation and construction of models and programs in the language. This notation is known as its concrete syntax. There are two main types of concrete syntax: textual and visual. A textual syntax enables models and programs to be described in a structured textual form. A visual syntax presents a model or program in a diagrammatical form. The advantage of a textual syntax is that it is good at representing detail, while a visual syntax is good at communicating structure.

### Abstract Syntax

The abstract syntax of a language describes the vocabulary of concepts provided by the language and how they may be combined to create models or programs. It consists of a definition of the concepts, the relationships that exist between concepts and may also include rules that say how the concepts may be legally combined. It is important to emphasize that a language's abstract syntax is independent of its concrete syntax and semantics. Abstract syntax deals solely with the form and structure of concepts in a language without any consideration given to their presentation or meaning.

### Semantics

The semantics of a language describes what models or programs in the language actually mean and do. In the context of programming languages, an execution semantics is essential in order to run programs written in the language. Semantics are also important in the context of modeling languages. Without semantics, modeling languages like UML offer little more than a collection of notations and their usefulness to developers is reduced.

## Development Challenges

One of the biggest problems faced by developers is dealing with the plethora of languages that are necessary to support the development of modern day systems. This leads to a problem of diversity: not only diversity of languages, but diversity of tools, programming styles, development environments, and so on. Moreover, as languages are constantly being developed, older languages become obsolete leading to problems of legacy software.

Another problem for developers is abstraction. Most development is carried out using programming languages. The reason for this is that programming languages are executable and precise, and offer developers powerful development environments such as IDEs. Yet, many programming languages provide relatively low-level design abstractions. Thus, the effort required to express the problem domain in a programming language is much greater than if it were expressed in a more abstract language.

Modelling languages on the other hand aim to provide richer modelling abstractions that are aligned more closely with the domain concepts used by the developer. Modelling languages try to avoid commitment to specific implementation technologies. This is particularly valuable as it enables developers to concentrate on what a system is required to do, as opposed to the detail of how it is to be implemented. A problem with modelling languages is that they are generally weakly defined. Many offer little more than notations. This informality severely limits their use during development:

- Models constructed in the languages cannot be validated, as they are too informal to support semantically rich capabilities such as execution, analysis and testing.
- The correctness of their translation to other languages cannot be verified, as it cannot be shown that the semantics of the target language implements the semantics of the modelling language.

Finally, modelling languages need to be able to capture abstractions of relevance to developers. In many situations, developers will want to make use of general-purpose abstractions, such as components, objects, patterns and frameworks. In other situations, they will benefit most from modelling languages that are tailored to support a specific modelling domain. An example of this might be an inventory-based system, where developers consistently have to express their models in terms of inventory type concepts such as resources, services and products. While languages like UML address the requirement of providing general-purpose abstractions, the ability to design semantically rich domain specific languages (see [2]) is missing.

## Language Driven Development

Language driven development involves adopting a unified and semantically rich approach to describing languages. A key feature of the approach is its ability to describe all aspects of languages in a platform independent way, including their concrete syntax and semantics. However, the language definitions should also be rich enough to generate tools that can provide all the necessary support for use of the languages such as syntax-aware editors, GUI's, compilers and interpreters.

Language driven development strikes at the heart of the diversity and abstraction issues, by making it possible to rapidly integrate multiple languages and to define semantically rich modelling capabilities at a high level of abstraction. By supporting a unified approach to language definition, language driven development provides developers with a rich array of languages that support their development needs:

- Traditionally informal modelling languages like UML can be made precise and semantically useful: i.e. UML models can be executed and analysed.
- Rich language abstractions, such as components, aspects, patterns, constraints can be defined, including their semantics.
- Language abstractions can be readily combined in multiple ways to build specific flavours of languages. For instance, a component modelling language that supports aspects or a pattern language that supports OCL.
- Domain specific languages can be rapidly created and (if required) integrated with general-purpose languages.
- Transformations between languages can be validated, as any behaviour offered by the metamodelled language can be checked against the behaviour of the language it is translated to.
- The languages used in standards can be specified precisely, thus ensuring that their definition is unambiguous to stakeholders.

The right languages enable developers to be significantly more productive than using traditional development technologies. Rather than dealing with low level coding issues, developers can use powerful language abstractions and development environments that support their development processes. They can create models that are rich enough to permit analysis and simulation of system properties before completely generating the code for the system. They can manipulate their models and programs in significantly more sophisticated ways than they can code. Moreover, provided the language definitions are flexible, they can adapt their languages to meet their development needs with relative ease.

## **MDA and Language Definition**

The need to capture languages independently in a platform independent format is not new to MDA. At the heart of MDA is a standard for describing meta-data, called the MOF (the Meta Object Facility) [3]. The purpose of the MOF is to define a common way of capturing all the different modelling standards (i.e. languages) used by MDA. Every standard that is expressed in terms of MOF can be related to each other simply because they are defined in the same way. For example, if one wants to move from a model written in UML to a model that describes a Java program, the process is greatly eased because they are represented in the same way.

The way that MOF describes different standards is through *metamodels*. Typically, a metamodel is a model of the concepts that are provided by the standard. In this case of UML, this might include such concepts as Class, Attribute, Operation, and so on.

Although the MOF is a good starting point for defining languages, it has a number of limitations:

- It is not rich enough to capture semantic concepts in a platform independent way. For instance, the execution of a state machine or a business process cannot be expressed in MOF. The tool designer must resort to implement these aspects in an external implementation technology such as Java.
- MOF does not provide a means of expressing the concrete syntax of a language, whether it is a textual or diagrammatical syntax. Whilst MOF models can be exported in terms of XML, this is of limited use to modellers, who require a more human readable form.
- MOF does not provide abstractions for capturing user interfaces and tools in a generic fashion. This means that either the language designer has little control over the user interface of a tool that supports the language, or these aspects must be encoded in a platform specific way.
- There is currently no way of defining both uni-directional and synchronised mappings between MOF models. These are essential to describe language transformations and the synchronisation of language elements. Whilst the QVT standard is aiming to address the issue of a uni-directional language, the need for a synchronised mapping language is paramount in order to describe synchronisation between language elements and diagram editors, user interfaces, and so on.

In order to support rapid language definition in MOF, MDA needs to be applied to itself, in the sense that the MOF should provide domain specific language concepts appropriate to defining all aspects of a language in a platform independent way.

### **Rich Metamodelling: Raising the Bar**

While a simple meta-data approach to defining language is insufficient, what can be done to address the issue? How can we rapidly create the rich modelling languages that are required for MDA?

What is required is a metamodelling environment that is sufficiently rich to capture all aspects of a language in a platform independent way. Moreover, this language should be self describing and self-supporting, thus making it possible for any language to be defined completely independently of external technology.

A central feature of such architecture is executability – this must be built in from the start in order to be able to capture the operational semantics of languages, and to be able to define the semantics of the metamodelling language itself. Without it, language definitions become reliant on external implementation technologies in order to define language semantics.

To support rich metamodelling, we propose the following components of a rich metamodelling architecture (an overview is shown in figure 1):

- A virtual machine for executing metamodels. Its purpose is to act as a platform independent execution environment for language definitions.
- A small, precise, executable metamodelling language that is bootstrapped independently of any implementation technology. It should support the minimum language definition capabilities necessary to be self-supporting. Thus includes a generic parsing language and diagramming language, a compiler and interpreter, and a collection of core executable MOF modelling

action primitives. Such a language can be built on top of MOF with the addition of appropriate executable primitives.

- A layered language definition architecture, in which increasingly richer languages and development technologies are defined in terms of more primitive languages via operational definitions of their semantics or via compilation to more primitive concepts or extension of existing concepts: These will include:
  - Definitions of richer metamodeling languages, such as mapping languages (QVT), UI languages, constraint languages, testing languages and pattern languages.
  - Definitions of general-purpose language primitives such as components, aspects, patterns, etc, that can be combined into general purpose modelling languages such as UML.
  - Definitions of bespoke, domain specific languages that are built from the above definitions.
- Support for the rapid deployment of metamodels into working tools. This involves linking the UI metamodels with appropriate user-interface technology. An open source toolset like Eclipse and GEF would be a good choice here.

The aim is to build support from the ground up for language definition in a layered fashion. Using the architecture it is possible to rapidly implement many different modelling languages and associated tools in a platform independent way. In other words, the metamodel becomes a complete definition of the language and tool that supports it.

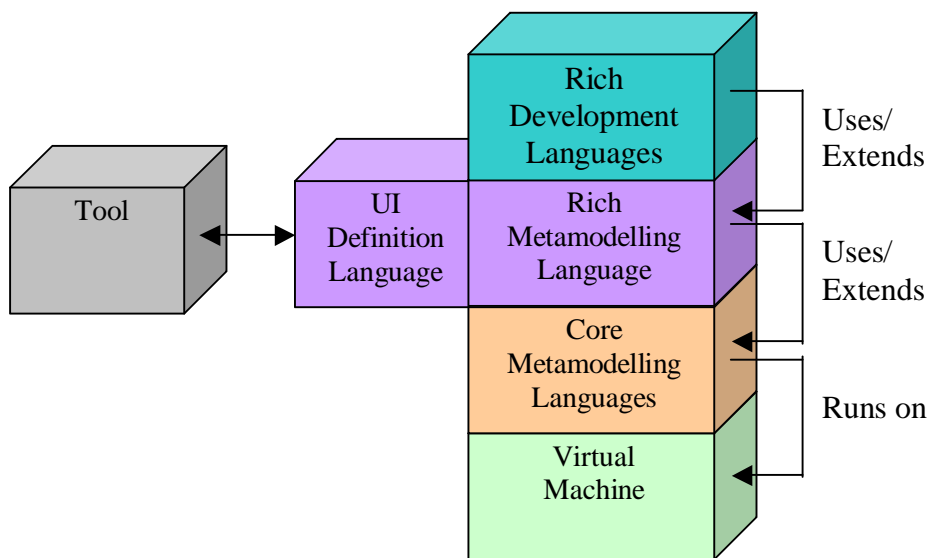


Figure 1: A Metamodel Architecture that supports Language Driven Development

### Example

Imagine an inventory modelling language that supports the specification of inventory entities, including products, services and resources within a teleco environment. Specifications (expressed as constraints) can be associated with entities. Inventory entities can also have operations and attributes, can be associated with other inventory

entities, and can specialise inventory entities of the same type. The language is to be integrated into a component modelling language because of the distributed nature of the telco environment. Furthermore, inventory models should be deployable into Java and an associated user interface that permits creation/deletion of instances of the models and the checking of specifications.

In order to implement a development environment for this scenario, a number of different languages will be required, each with their own metamodel definitions:

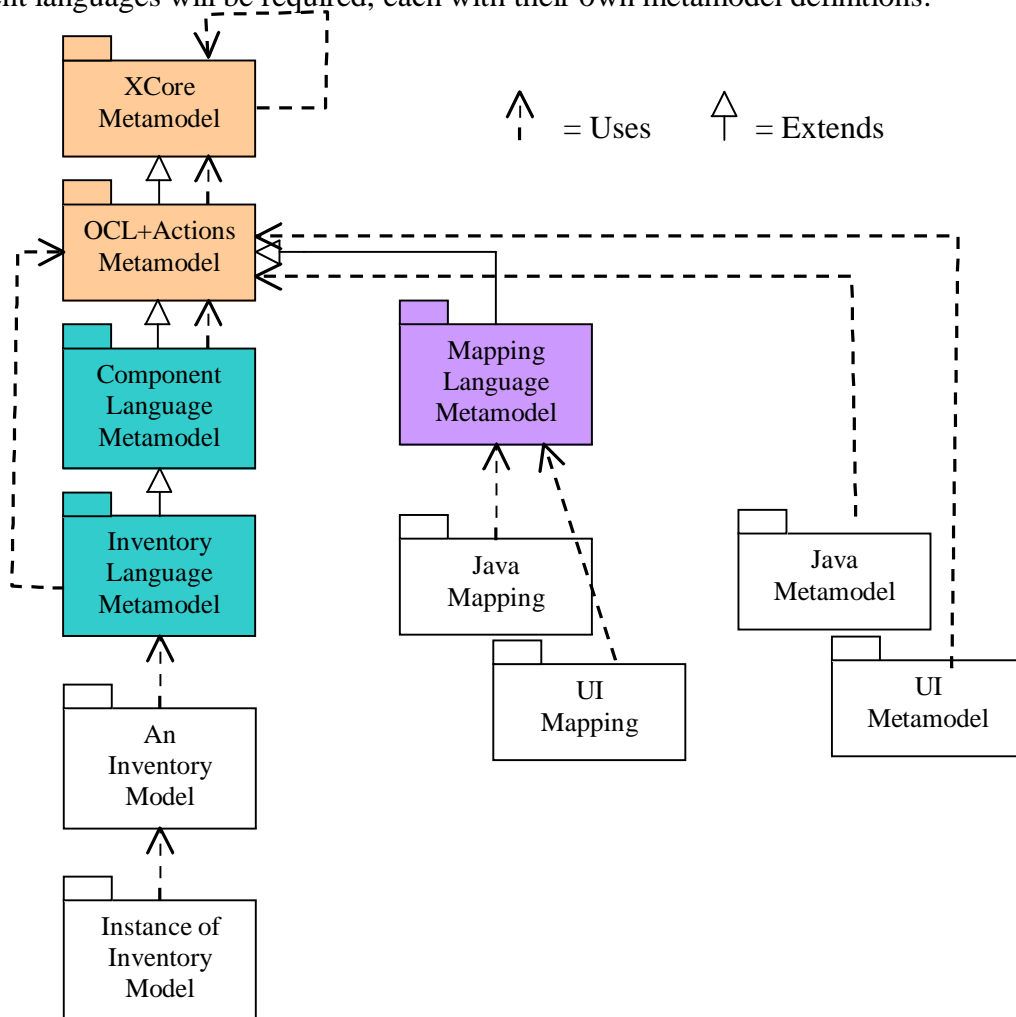


Figure 2: Inventory Example

At the core of language definition are the XCore and OCL/Actions metamodels. These provide a minimal, executable language for building metamodels that runs on the virtual machine. All other metamodels will extend and instantiate these metamodels (including the XCore metamodel, which instantiates itself).

The mapping language metamodel is an example of a rich metamodeling language definition as it adds the functionality to express mappings between metamodels more concisely than using the core metamodeling facilities alone.

The component language metamodel is an example of a rich (general purpose) development language definition, while the inventory language is an example of rich (domain specific) development language definition. Note that the domain specific

language is also dependent on a number of general-purpose languages, such as OCL for capturing specifications.

Finally, mappings are instances of the mapping language metamodel that relate inventory models to Java and UI metamodels.

The key point about the architecture is that they are all built on a common, executable language metamodel. Thus, models written in these languages will also be executable and will be well defined. For instance, the inventory language is a fully executable language: models can be created, and instances can be created from those models. Thus, specifications and standards can be verified for correctness very early in the lifecycle. Moreover, the completeness of the inventory metamodel means that a complete implementation can be generated in Java.

## **Language Driven Development and DSL's**

In [2] it has been proposed that domain specific languages have specific advantages over traditional style MDA PIM to PSM transformations. DSLs aim to provide targeted domain specific modelling concepts, which can be used to accelerate development. However, our experience tells us that developers need to access a wide variety of languages. In particular, there will always be the need for general-purpose languages that cover multiple domains. Languages like UML attempt to fill this gap, but are not well defined. Discarding the general-purpose abstractions provided by UML is our opinion throwing out the baby with the bathwater. Instead, these abstractions need to be semantically well defined so that developers can benefit from them.

In short, DSL's are just one type of language that is encompassed by language driven development, but they are no more special than other type of language.

## **Conclusion**

The MDA vision is one that has the potential to encompass a world in which languages are managed in a unified and semantically rich way. However, to achieve this, we must understand better how generic language driven development techniques can be layered on top of existing MDA standards. As outlined above, this can only occur if we raise the bar with respect to what metamodels can represent, and build a layered metamodeling architecture that can support semantically rich language definition capabilities.

## **References**

- [1] [www.omg.org/mda](http://www.omg.org/mda)
- [2] Cook, S. Domain-Specific Modeling and Model Driven Architecture, MDA Journal, January 2004 (<http://www.bptrends.com/publicationfiles/01-04%20COL%20Dom%20Spec%20Modeling%20Frankel-Cook.pdf>).
- [3] [www.omg.org/mof](http://www.omg.org/mof)