MDA Journal

October 2004

David S. Frankel
David Frankel Consulting

df@DavidFrankelConsulting.com

www.bptrends.com

Introduction

I was working for a computer service bureau in the late 1970s that offered payroll and accounting programs to customers who connected to the company's computer via telephone modems and dumb terminals. One day a customer interested in the payroll system walked into the office. He immediately asked whether our company used our payroll system to do its own payroll. The answer was no. That wasn't the answer he was looking for. He said that if we don't use our system to do our own payroll, then he wouldn't either.

I tell this story because the notion of practicing what you preach is important with MDA. MDA proponents preach formal design and auto-generation from models. But how many MDA tools are actually built that way themselves?

Andy Evans, the author of this month's MDA Journal article, is one of the people on the frontiers of MDA. He has built tooling by bootstrapping up via model-driven methods from a kernel that itself is built in a model-driven fashion.

Bootstrapping is not new, of course. Many C compilers, for example, were written in C, usually using a subset of the full C language. One of the advantages of bootstrapping a transformation environment—which is what a C compiler is—is that using the transformation capabilities at a low level makes it possible to auto-generate many parts of the system that would otherwise have to be coded at a lower level of abstraction (in assembly language when building a C compiler). A corresponding advantage to bootstrapping an MDA tool is that the transformation engine and other parts of the system are reused at many levels, resulting in an environment that has a relatively small footprint and that can be built without a lot of relatively low-level coding.

I met Andy when he was part of a team that produced a proposal to the OMG for the UML 2 and MOF 2 standards. His team's submission was not ultimately adopted, although some of its ideas were. It had some particularly interesting proposals for how to define the semantics of languages as part of the metamodeling process. These proposals may yet be codified under the banner of an Executable UML standard that will, in essence, define a UML profile that the Executable UML community can rally around.

This article gives us a glimpse of what a totally model-driven environment based on precise semantics looks like, and should raise the bar of expectations as to what MDA can achieve.

See you in November…

—David Frankel

MDA Journal

October 2004

Tony Clark, Andy Evans, Paul Sammut, and James Willans

Xactium Limited

Language Driven Development and MDA

Introduction

Model Driven Architecture (MDA) [1] aims to be a major step forward in the way that systems will be developed in the future. MDA is a framework for unifying technologies based around OMG standards. It is founded on the Meta Object Facility (MOF™), which is used to define other languages such as UML® and CWM™, and is the basis for XMI®. Primarily, MDA concerns models and mappings between models. The most widely recognized application of MDA is the mapping or transformation between Platform Independent Models (PIMs) and Platform Specific Models (PSMs).

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 modeling requirements. The aim is to provide developers with rich modeling 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 and UML) and looks at a particular approach to extending the standards to support it based on the construction of executable metamodels. This approach is the basis for a commercial tool called XMF that facilitates the rapid design and deployment of languages and tools. Language driven development realizes substantial productivity gains by constructing rich development environments that automate significant parts of the development process. This includes domain specific modeling, model validation, model simulation, and transformation to code. We also relate language driven development to recent discussion about domain-specific languages (DSLs) 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 formalized form than natural languages). Developers use a surprisingly varied collection of languages. This includes high-level modeling 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 specialized 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*: Generating test cases from models and validating them against programs.
- *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 interconnected, whether informally or automatically, through code generation or compilation.
- *Integration*: Integrating features from one model or program into another, e.g. by merging different viewpoints or aspects of a problem domain.

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. These include high level modeling languages like UML, programming languages like Java, scripting languages like Perl, database languages such as SQL, and so on. Furthermore, developers have to make a choice about the specific design abstractions that they should use, which may include everything from objects, components, and patterns to process or event based abstractions. This leads to a problem of diversity: not only diversity of languages, but diversity of tools, programming styles, development environments, and so on. As a result, significant development time is wasted in having to manually tailor and integrate languages, tools, and development environments to meet the specific needs of projects. A good example is the manual encoding of design patterns in a programming language, or the configuration of J2EE servers, or the manual transformation of designs into code. 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 only 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.

Modeling languages on the other hand aim to provide richer modeling abstractions that are aligned more closely with the domain concepts used by the developer. Modeling 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 modeling languages is that they are generally weakly defined. Many offer little more than notations. This informality severely limits their use during development in two principal ways:

- 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 modeling language.

Finally, modeling 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 modeling languages that are tailored to support a specific business 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. These language definitions should 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 modeling 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 modeling languages like UML can be made precise and semantically useful: i.e., UML models can be executed and analyzed by extending the language with a definition of an action language and the rules for executing actions.
- Rich language abstractions, such as components, aspects, patterns, constraints can be defined, including their semantics. For example, an aspect can be modelled as a new language concept that encapsulates a model-element, along with the rules for merging the aspect with a model.
- Language abstractions can be readily combined in multiple ways to build specific flavors of languages. For instance, a component modeling language could be merged with an aspect language or a pattern language, thus supporting a richer design environment.
- Domain-specific languages can be rapidly created and (if required) integrated with general-purpose languages. For instance, a domain specific language for describing inventories (see below) can be constructed that makes use of inventory-specific concepts and a general-purpose constraint language to capture and execute rules relating to inventories.
- Transformations between languages can be validated, as the behavior of models or programs written in one language can be checked against the behavior of the language it is translated to. For example, if the behavior of a state machine can be simulated at the design level, it can be validated against its implementation at the code level by comparing state changes.
- Languages can be specified precisely, thus ensuring that their definition is unambiguous to stakeholders including standards bodies, tool vendors, and application developers.

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 integrate languages and automate the transformation from one language to another. They can manipulate their models and programs in significantly more sophisticated ways than they can code. In this sense, development environments become more akin to the CAD/CAM tools used in the engineering domain in which highly sophisticated analysis of the properties and behavior of a product are conducted well before it is physically constructed. Moreover, provided the language definitions are flexible, they can adapt their languages to meet their development needs with relative ease.

Application Development and Language Definition

It is worth highlighting that application developers are more involved in language driven development than they may think. Consider financial applications. Many developers will create or utilize frameworks or patterns that capture the financial concepts across multiple projects. We would propose that they are, in fact, defining a language that is specific to their domain. How they interact with this language (e.g., via web-services) is, in fact, a product of their concrete syntax and semantics. Recognizing this fact leads to a radically different way of understanding (and realizing more efficient approaches to) application development.

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 modeling standards (i.e., languages) used by MDA. Languages that are defined 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 a MOF definition of each language is available.

MOF describes different languages through *metamodels*. Typically, a metamodel is a model of the concepts that the language supports. In the case of UML, this includes 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, MOF cannot express the execution semantics of a state machine or a business process. The tool designer must resort to specifying these semantics 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. While MOF models can be exported in terms of XML, this is of limited use to modellers, who require a more human usable form.

• MOF does not provide abstractions for describing 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 to define uni-directional and synchronized mappings between MOF models. These are necessary to describe language transformations and the synchronization of language elements. Synchronization is essential for capturing the relationship between different components of tools, for instance, between a diagram editor and model browser. Currently, work is ongoing to complete the QVT (Query, Views, Transformations) standard, which will provide a means of expressing transformations as an extension to MOF. However, it is not yet clear whether this will support synchronisation.

Rich Metamodeling: Raising the Bar

While MOF's approach to defining language is insufficient, what can be done to address the issue? How can we rapidly create the rich modeling languages that are required for MDA?

In order to support rapid language definition in MOF, MDA needs to be applied to itself, in the sense that the MOF should provide a domain-specific language appropriate to defining all aspects of a language in a platform-independent way. The domain for such a domain-specific language is metamodeling. Moreover, this metamodeling 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 this approach 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 metamodeling language itself. Without it, language definitions become reliant on external implementation technologies in order to define language semantics.

XMF (eXecutable Metamodeling Facility) is a commercial implementation that supports full language definition using rich, executable metamodels. It has been constructed specifically to support language-driven development. XMF is fully bootstrapped, in the sense that the tools it supports (diagram editors, compiler, parser, interpreter, etc., are all modelled in its own language). This facilitates a completely uniform language definition architecture. XMF is composed of the following components. (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 metamodeling language that is bootstrapped independently of any implementation technology*. This supports the minimum language definition capabilities necessary to define languages. It includes a MOF-like OO modeling language called XCore, a generic grammar definition language called XBNF, a diagramming language and an imperative extension of OCL called XOCL[1]. A language is thus "modelled" by describing its language concepts, its grammar, and its diagrammatical syntax. Its operational behaviour

[1] XOCL extends OCL with simple imperative constructs such as new(), and assignment, which turn it into a powerful meta-programming language. Although UML already supports an action language, it is more appropriate for application development. Moreover, extending OCL avoids the need to support both languages at the same time.

(its semantics) is described by constructing an interpreter or compiler for the language in XOCL.

- *A metamodel-driven compiler/interpreter*. This enables models written in the core metamodeling language to be compiled into VM code, or to be directly interpreted via the VM.
- *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: It includes:
  - o Definitions of richer metamodeling languages, such as mapping languages (QVT), UI languages, constraint languages, testing languages, and pattern languages.
  - o Definitions of general-purpose language primitives such as components, aspects, patterns, etc., that can be combined into general purpose modeling languages such as UML.
  - o Definitions of customized, domain-specific languages that are built from the above definitions.
- *Support for the rapid deployment of metamodels into working t*ools. This involves linking the UI metamodels with appropriate user-interface technology. It is implemented by modeling a number of generic user interfaces (browsers, diagram editors, text editors) and connecting them to Eclipse and the Graphical Editing Framework (GEF).

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

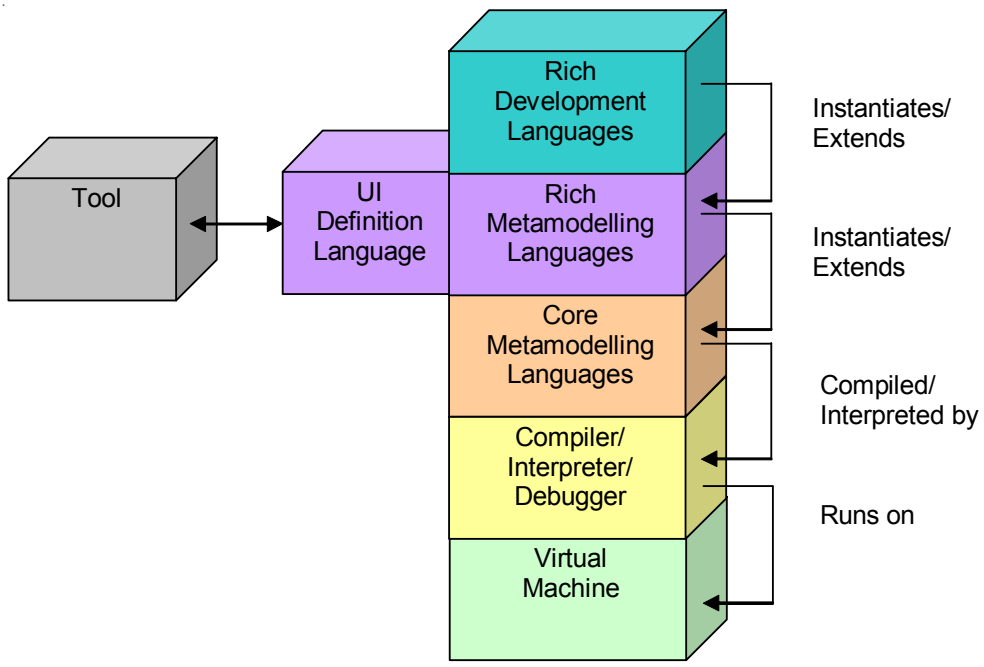

Figure 1: A Metamodel Architecture that Supports Language Driven Development

modeling languages and associated tools in a platform-independent fashion. In other words, the metamodel becomes a complete definition of the language and tool that supports it.

Language Definition Example

Imagine developing an inventory modeling language that supports the specification as well as the instantiation and execution of inventory entities, including products, services, and resources within a telco environment.

A language definition can be constructed by creating a metamodel of all aspects of the inventory language: its abstract syntax, concrete syntax, and semantics. A brief overview of the main components of the definition is given below. More detail about the approach can be found in [4] and [5].

Abstract Syntax
An abstract syntax model for the language is first constructed using the XCore language (see figure 2). The inventory modeling language provides entities that can be products, services, and resources. Each of these entities can be associated with a specification. There is a specification for each type of entity. Entities may be related by entity attributes. Note that the abstract syntax model extends XCore, so entities are modelled as specializations of classes; specifications are specializations of constraints; and entity attributes are specializations of attributes.

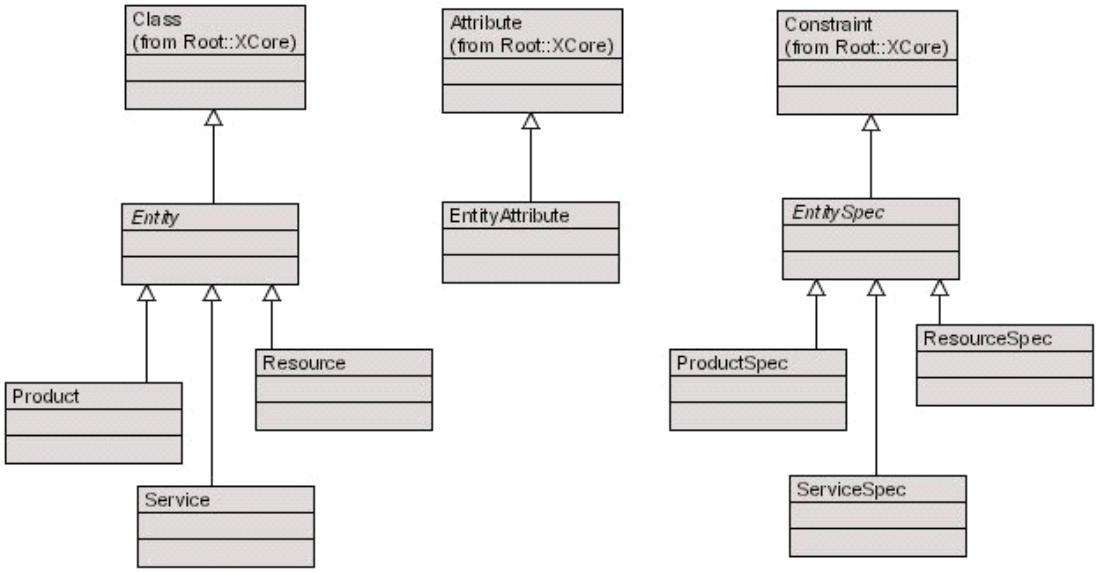

Figure 2: Abstract Syntax Model for Inventory Language

Finally, OCL constraints can be written on the metamodel to rule out invalid combinations of model elements. For example, an entity cannot inherit from an entity of a different type, such as a Product specializing a Service. This can be expressed in XOCL as follows, where of() is an XOCL operation returns the

class that an object is an instance of:

```
context Entity
@Constraint SameParentType
parents->reject(p |
   p = XCore::Object)->forAll(p |
   p.of() = self.of ())
```

Semantics

Because the core metamodeling language has a well-defined executable semantics, its semantics will be inherited by the language concepts. For instance, because entities specialize the class Class, they will inherit the ability to create new objects. Class has an operation new() that defines what it means to create a new instance of a class. It is also possible to evaluate objects against their entity's constraints and for their operations to be invoked. Note that semantics by specialization is just one approach to defining semantics: others, such as compilation down onto the virtual machine and writing interpreters for the language, are all supported by XMF.

Concrete Syntax

The concrete syntax of the inventory modeling language can be modelled in a number of different ways. One approach is to define a textual syntax for the language. To do this, XBNF can be used. This extends BNF with the ability to state how each parsed element is to be translated into an instance of the language's abstract syntax model. An example of its use would be as follows:

```
Service ::= name = Name body = Body {Entities::Service(name,body)}
Body ::= BodyExp*
...
```

The grammar for a Service is defined to be a Name (an inbuilt expression that represents any valid name string) followed by a Body, where a Body is defined to be a sequence of BodyExp (a sequence is denoted by a *). Note the definition of the BodyExp is not given here, but would include definitions of valid body expressions such as Attribute and Constraint. Whenever the grammar matches with a valid textual input (i.e., a Name followed by a Body) the values in the expression are matched with the variables name and body. These are then passed to the expression in the {}s, which creates an instance of the class Service[2].

The following example shows an example of the textual syntax that can be written based on this grammar:

```
@Service IPVPN
 @Attribute numberOfServices : Integer end
 @Constraint ServiceLimit
  containedServices->size() <= numberOfService
 end
end
```

[2] The power of XBNF is its ability to flexibly define mappings from textual syntaxes to models. In this example, the mapping is from same textual syntax element to same model element. However, the mapping could equally be used to construct mapping to more primitive model concepts. This supports a simple means of creating richer language abstractions via "desugaring" (see [5]).

The result of parsing this syntax would be an instance of the class Service, with the name IPVPN and containing a single attribute and constraint.

Alternatively, we might consider constructing a model of the language's diagrammatical syntax and describing how this model is related to its abstract syntax model. This approach is fully supported by XMF, but it is not described here for reasons of space.

Another approach is to adopt a profile style approach. Provided that a tool understands when a package is an instance of the inventory metamodel, it can provide appropriate stereotypes in its diagram editor. An example of how this might look is shown below:

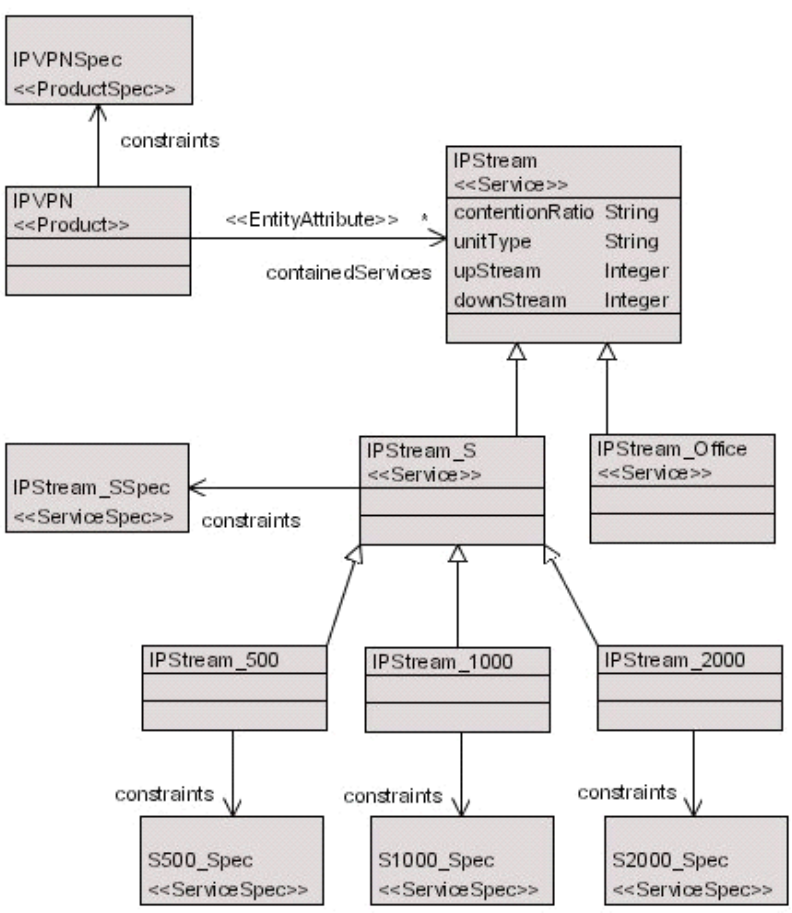

Figure 3: Example Inventory Model

Here, an IP Virtual Network product may be associated with a hierarchy of services, which have attributes and specifications. Note, that the *constraints* attribute shown between individual services and specifications actually represents an instance of the meta attribute *constraints* inherited from the class Class. Constraints can be written in the body of specifications because they are instances of the class Constraint.

Support for Semantically Rich Operations

Once the definition is complete, a large number of semantically rich operations can be performed on models written in the language. Models can be created and evaluated against the constraints on the abstract syntax model. Furthermore, the model itself can also be instantiated, thus enabling instances of the model to be created and checked in the same way. This capability is only possible because the language reuses concepts that are already semantically well-defined. For example, specifications can be checked because a grammar has been constructed for OCL in XBNF, and a semantics has been implemented by defining the rules for compiling OCL down onto the VM. Full access to to the metamodel of the language, along with an executable metamodeling language, also means that developers can rapidly construct new facilities for manipulating models, such as model analysis or transformations to other languages. Indeed, languages have been defined (in exactly the same way) in XMF for constructing transformations between languages, again increasing productivity by automating a specific development process.

Language Driven Development and DSLs

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 modeling 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, in 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, DSLs are just one type of language that is encompassed by language driven development, but they are no more special than other types of languages.

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. The approach we have described has been implemented in a commercial executable metamodeling facility called XMF, details of which can be found at www.xactium.com in [5] and in a freely downloadable book [6]. XMF is being successfully used in the design of and deployment of general-purpose and domain-specific languages in the aerospace and defence industry.

References

[1] 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
[4] Georgalas, N., at al. MDA-Driven Development of standard-compliant OSS components: the OSS/J Inventory Case-Study. Second European Workshop on Model Driven Architecture with an emphasis on Methodologies and Transformations, 7th – 8th September 2004. Canterbury, Kent, United Kingdom.
[5] Clark, A., et al. Applied Metamodeling: a Foundation for Language-Driven Development. Available for download from www.xactium.com, 2004.