BIOGRAPHAPHICAL INFORMATION

Charles F. I. Savage
Technical Lead
Ubisense

Specific Responsibilities

Currently technical lead in North America for Ubisense. Responsible for technical sales and customer implementations. Mr. Savage's areas of technical expertise include enterprise application integration, modeling and designing distributed, multi-tier, web-based applications.

Past Experience

Previously worked for GE Network and Reliability Services / Smallworld as Chief Architect. Responsible for overseeing architecture of all NRS products and enabling interoperability between them. Prior to becoming Chief Architect, Mr. Savage was the architect of the Smallworld Internet Application Server product suite. Prior to working for GE Network Reliability Services, Mr. Savage worked in the environmental consulting field.

Education Information

M.S. – Environmental Engineering, Stanford University
B.S. – Mechanical Engineering and Geology, Stanford University

Professional Memberships

GITA
Institute of Electrical and Electronics Engineers
American Society of Civil Engineering
American Society of Mechanical Engineering

USING MODEL DRIVEN ARCHITECTURE FOR INTEGRATION

Charles F. I. Savage
Ubisense
8400 E Crescent Pkwy Ste 600
Greenwood Village, CO 80111
720-528-4130
cfis@interserv.com

ABSTRACT

Organizations use many diverse systems to manage their information and data. These systems range from mainframes to legacy systems to relational databases. Due to increasing competitive pressures, it is important for these systems to interoperate. However, this task is hard. One cause of difficulty is that most systems use proprietary techniques for describing their data. These descriptions of data are known as metadata. Consequently, systems must include significant amounts of additional code to manage, query and store their metadata. It is also necessary to write large amounts of code to transform a system's internal data formats to external data formats, and vice versa. Not only is writing this code time consuming and expensive, it generally is not reusable.

Model Driven Architecture (MDA) reduces the complexity of integration by providing a standard way of describing a system's metadata. Not only does this allow systems to replace large amounts of code with off-the-shelf components, it also means that metadata becomes portable between different systems and technologies. This standardized metadata can then be used to declare how data from one system should be transformed to another system. These declarations, which can be defined in a configuration file or using a graphical editor, make it possible to largely automate data transformations via the use of generic transformation engines. These engines eliminate the need to write costly and time-consuming transformation code, making integration projects faster, less complex and in the end more successful.

INTRODUCTION

Organizations use many diverse systems to manage their information and data. These systems range from mainframes to legacy systems to relational databases. Due to increasing competitive pressures, it is important for these systems to interoperate. However, this task is hard. One cause of difficulty is that most systems use proprietary techniques for describing their data. These descriptions of data are known as metadata. Consequently, systems must include significant amounts of additional code to manage, query and store their metadata. It is also necessary to write large amounts of code to transform a system's internal data formats to external data formats, and vice versa. Not only is writing this code time consuming and expensive, it generally is not reusable.

Model Driven Architecture (MDA) reduces the complexity of integration by providing a standard way of describing a system’s metadata via the use of models. Models make it much easier to understand systems and data. As stated by the Object Management Group (OMG), an industry consortium that develops MDA standards:

> Developing a model for an industrial-strength software system prior to its construction or renovation is as essential as having a blueprint for large building. Good models are essential for communication among project teams and to assure architectural soundness. We build models of complex systems because we cannot comprehend any such system in its entirety…In the face of increasingly complex systems, visualization and modeling become essential.

Models are particularly useful for describing data since they can fully capture the structure of the data in a platform independent manner. Models also make it possible to declare how data from one system should be transformed to another system. These declarations, which can be defined in a configuration file or via a graphical editor, make it possible to largely automate data transformations via the use of generic transformation engines. These engines eliminate the need to write costly and time-consuming transformation code, thereby making integration projects faster, less complex and in the end more successful. Thus MDA provides a significantly more scaleable, and easier to understand, method of integrating systems than the traditional approach of hard-coding data transformations in programming languages such as Java or XSL.

DIFFICULTIES WITH INTEGRATION

It is very common for systems to need to exchange data with other systems. In most instances this requires converting data, which may be stored in a variety of formats such as text files, relational databases, XML files or custom binary formats, into some native format. This process of converting data from its stored format to a native format, and vice versa, is known as data binding.

For example, assume we need to share the location of transformers stored in a GIS with another application such as an ERP (Enterprise Resource Planning) system. To exchange data we need to convert a coordinate in the GIS to some wire-format for transportation and then to a coordinate in the ERP system. Let’s assume that the GIS represents the coordinate as an instance of a Java object, the wire-format is XML and the ERP system represents the coordinate as a record in a database. **Table 1** shows representations of the same coordinate in these different domains.

Table 1: Representations of a coordinate in three different domains

Java Object	XML	Record
Coordinate coord1 = Coordinate.new(10,10);	<coordinate id=”coord1”> <x>10</x> <y>10</y> </coordinate>	ID X Y coord1 10 10

In this simple case, mapping between these different representations is straightforward. However, it requires writing code in both the GIS and ERP system to convert the coordinates from one format to another. The GIS needs to convert its Java object to XML while the ERP system needs to convert XML to a record in table.

Of course, integration is never this simple. The first problem is that there are number of different representations of a coordinate in XML. Additional representations might include:

```
<coordinate x="10" y="10"/>

<coordinate>10,10</coordinate>

<gml:coordinate xmlns:gml="http://www.opengis.net/gml">
    <x>10</x>
    <y>10</y>
</gml:coordinate>
```

Although these are subtle changes that do not change the meaning of the XML, they nevertheless invalidate the code we wrote previously. Consequently it is necessary to rewrite the code in both the GIS and the ERP System to take into account these three new representations of a coordinate. Note that we have not even considered possible mappings and transformations, which almost always occur when exchanging more complex data.

Now multiply this problem by the number of different types of objects you wish to exchange by the number of different integration projects you work on. For example, let's say we have 50 integration projects a year and on average each project requires transferring 100 different types of objects. All of a sudden we have to write 10,000 different conversion routines (50 projects * 100 objects * 2 conversions - input and output).

Writing all this code is tedious, error-prone, hard to test and difficult to maintain – and worst of all it is not reusable.

METADATA

Although it is not a "silver bullet," Model Driven Architecture (MDA) makes it easier to exchange data between different systems. MDA makes this possible by using models to provide a common way to of describing the data that needs to be exchanged.

The first problem in exchanging data is to understand the data that will be exchanged. For example, if someone hands you a piece of paper that has the number "100" written on it you have no way of knowing what the "100" represents. To understand the "100" you must have information about it. This data about data is called metadata. Metadata is what tools, databases, applications and other information services use to define the structure and meaning of their objects, services, and other computing artifacts.

Unfortunately, almost all data sources have custom ways of representing their metadata, resulting in proprietary semantics, structures, and syntax. This impedes the flow of information across system boundaries and leads to great difficulty in sharing even the simplest data. Some common examples of metadata include:

- Database schemas, which describes how entries are laid out in a relational database and what they mean.
- XML Schemas, which describes the structure and types of an XML document.

- Classes in programming languages that describe objects, methods and attributes.
- UML diagrams, which describe objects, methods and attributes.
- COM and CORBA IDL files, which are used to describe interfaces in a language independent manner.
- Descriptions of business processes, such as business rules, business nomenclature and business terminology.

To make this more concrete, let's return to the coordinate example from above. **Table 2** shows we have three different types of metadata – a Java class, a XML Schema and a database schema.

Table 2: Metadata of a coordinate in three different domains		
Java Class	**XML Schema**	**Database Schema**

Class Coordinate {
public int x;
public int y;
}
	<xs:element name="CoordinateType">
	<xs:complexType>
	<xs:sequence>
	<xs:element name="x" type="xs:integer"/>
	<xs:element name="y" type="integer"/>
	</xs:sequence>
	</xs:complexType>
	</xs:element>
		Field Name — **Type**
		x — integer
		y — integer

Shared understanding of this metadata is the primary means by which interoperability can be achieved. The idea is that all applications share metadata via MDA standards as shown in below in Figure 1.

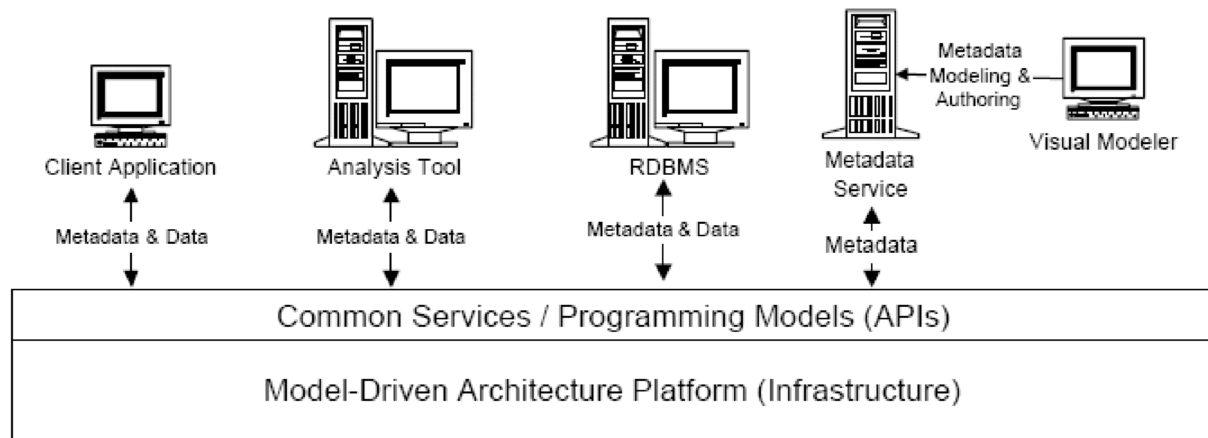

Figure 1: Shared metadata in an organization. Image courtesy (Poole 2001).

Meta Object Facility (MOF)

The primary MDA standard for sharing metadata is the Meta Object Facility (MOF) standard. The premise of MOF is that different systems will always have different ways of storing data. A single modeling language, even one as powerful as UML, cannot model all problem domains. Therefore there will always be the need for many types of models, and as a result, modeling languages. MOF is used to define these modeling languages. MOF provides a small set of

concepts, such as packages, classes, methods, attributes and associations that allow the creation of modeling languages.

For example, the three types of metadata shown above in **Table 2** can each be described by a model. The Java class can be represented via a UML model as shown below in Figure 2:

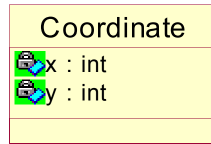

Figure 2: UML model of a coordinate class

The XML Schema could be represented by an XML model while the database schema by a relational model.

Reviewing our coordinate example, we see that we end up with four layers. The bottom layer is the data layer and contains the data we want to exchange, such as a Java object. The second layer is the model layer and contains a model of the data we want to exchange, such as a UML diagram. The third layer is the modeling language layer and contains standards such as UML. The fourth, and final, layer is the MOF layer which is used to define modeling languages. Figure 3 shows this four-layer metadata architecture:

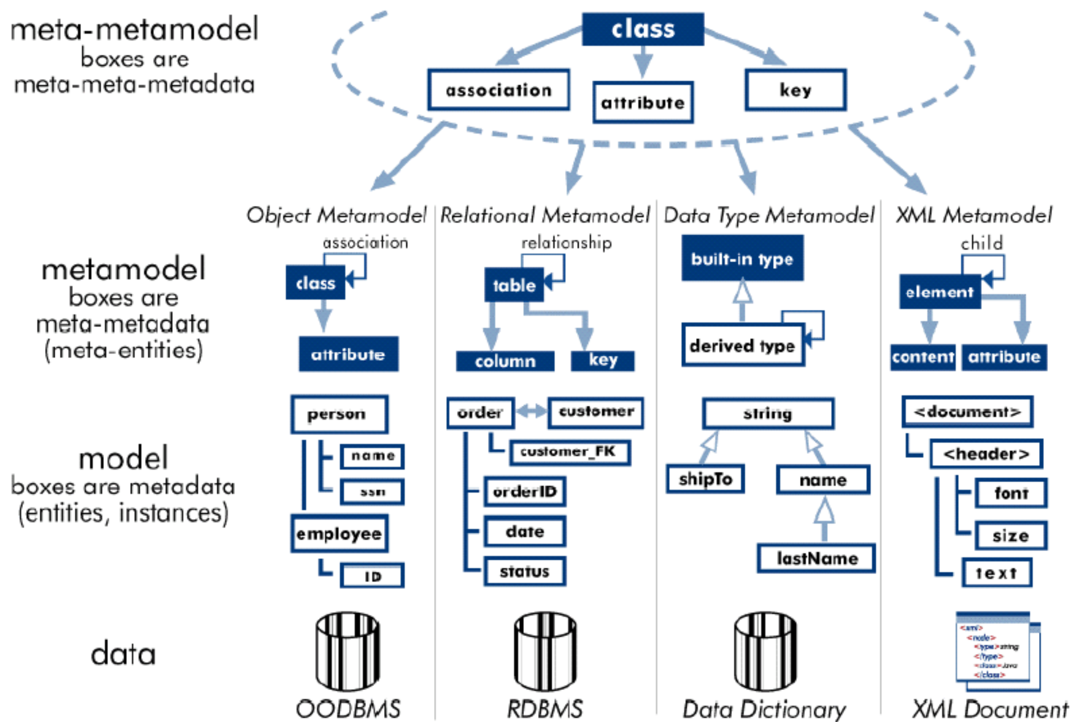

Figure 3: MDA can be used to model multiple types of data sources using a four layer architecture.

Figure 3 also shows that MOF is able to unite metadata from multiple domains – object oriented systems, relational databases, data type definitions and XML documents. It is also possible to add new domains by creating new modeling languages as needed.

A four-layer metadata architecture has a number of advantages over simpler modeling approaches, including:

- It can support most kinds of models
- It can allow different kinds of metadata to be related
- It can allow metamodels and new kinds of metadata to be added incrementally
- It can support interchange of arbitrary metadata (models) and meta-metadata (metamodels) between parties that use the same meta-metamodel.

To summarize, the power of MOF is that it enables otherwise dissimilar modeling languages to be used in an interoperable manner. Therefore it can be used to model domains such as object oriented programming, relational databases, XML, etc. in a standard way.

Consequently, MOF is becoming more important in Enterprise Application Integration. For example, SAP's next generation NetWeaver platform and IBM's WebSphere platform both include metadata repositories. In addition, since MOF is standardized off the shelf-components are starting to become available that can be leveraged by systems. Exaples include open source projects such as NetBean's MDR (metadata repository) project and Eclipse's modeling framework.

TRANSFORMATIONS

Once an organization has described its data using MDA standards, it can then define transformations to exchange data between disparate systems. As shown in Figure 4, these transformations use shared metadata to define how data in one format should be converted to another format.

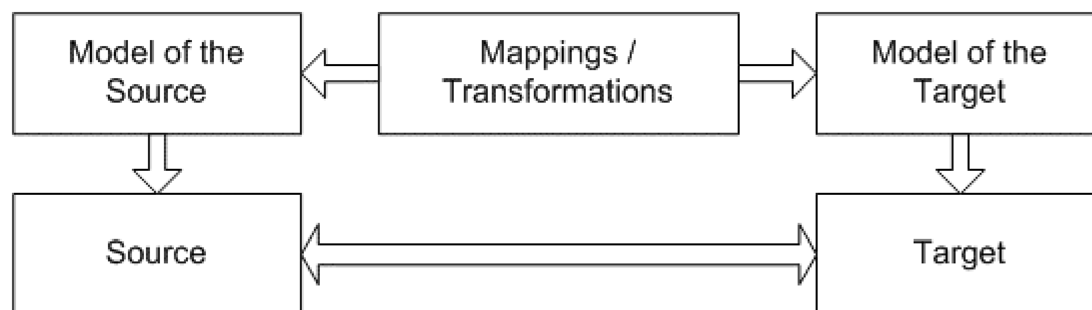

Figure 4: Defining mappings and transformations between sources and targets.

Transformations can be defined declaratively, via the use of configuration files or graphical editors. These declarations make it possible to largely automate data transformations via the use of generic transformation engines as shown in Figure 5 below.

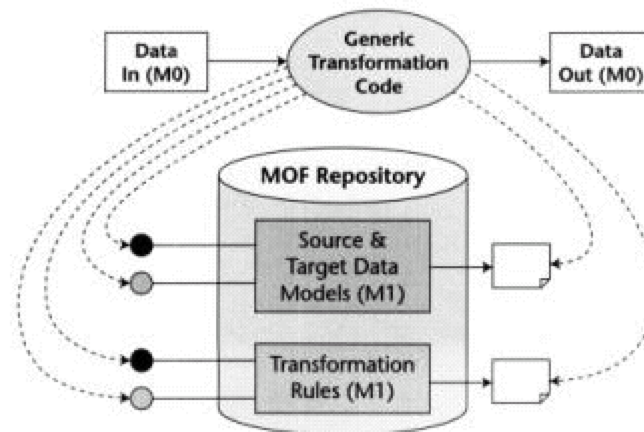

Figure 5: Use of a generic transformation engine. Image courtesy of (Frankel 2003).

Use of a transformation engine eliminates the need to write costly and time-consuming transformation code, thereby making integration projects faster, less complex and in the end more successful. Thus MDA provides a significantly more scaleable, and easier to understand, method of integrating systems than the traditional approach of hard-coding data transformations in programming languages such as Java or XSL.

CWM

MDA, via its Common Warehouse Metamodel (CWM) standard, provides a common way of describing the transformations discussed above. CWM is a broad, MOF-based standard that provides a framework for representing metadata about data sources, data targets, transformations and analysis, and the processes and operations that create and manage data warehouses.

Of particular importance for integration is CWM's transformation support. Transformations are generally "white box" transformations, which means that data sources and targets are related to each other at a fine-grain level. This is done via the use of transformation maps which include classifier maps and features maps as shown in Figure 6.

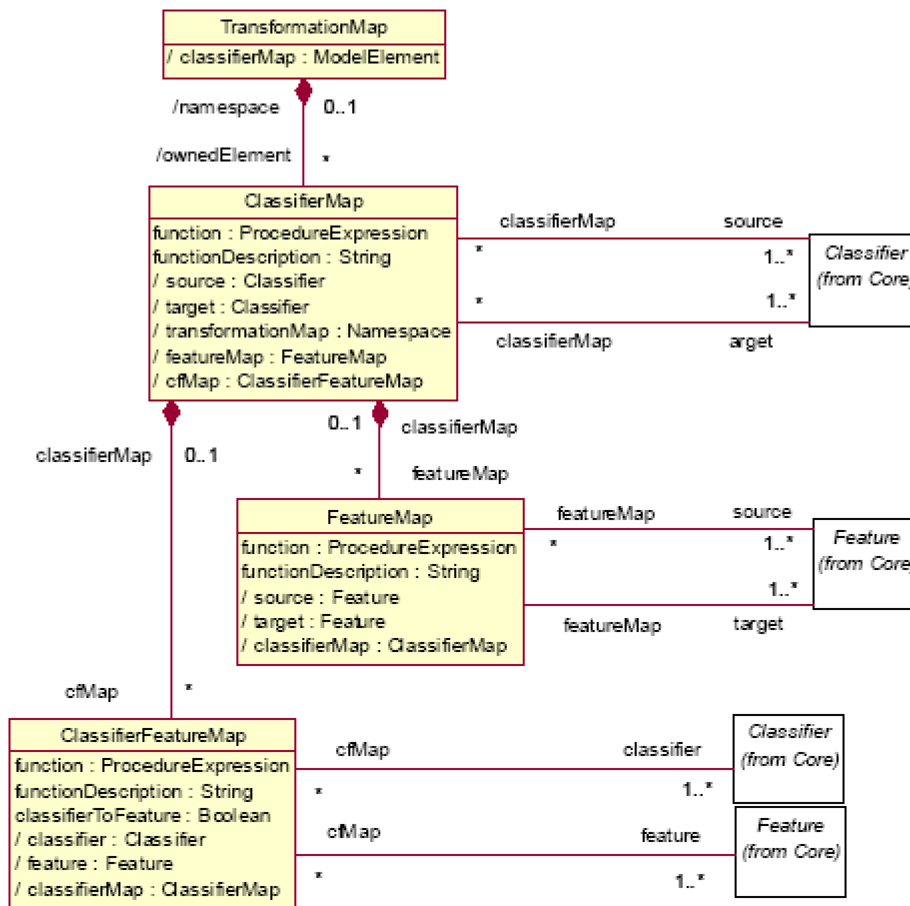

Figure 6: Fragment of the CWM transformation metamodel. Image courtesy (OMG 2001).

Classifier map describes how one type, such as a class, should map to another type. Feature maps, in contrast, specify how a class's attributes map to another class's attributes. Note that classifier maps and feature maps do not contain any type or order information – this metadata is already stored in the main model.

Going back our example, let's say we want to map a Java coordinate to a XML coordinate. To do this we need to map the Java class Coordinate to the XML Schema ComplexType CoordinateType. We then need to map the x and y attributes to each other. This mapping is shown below in Figure 7

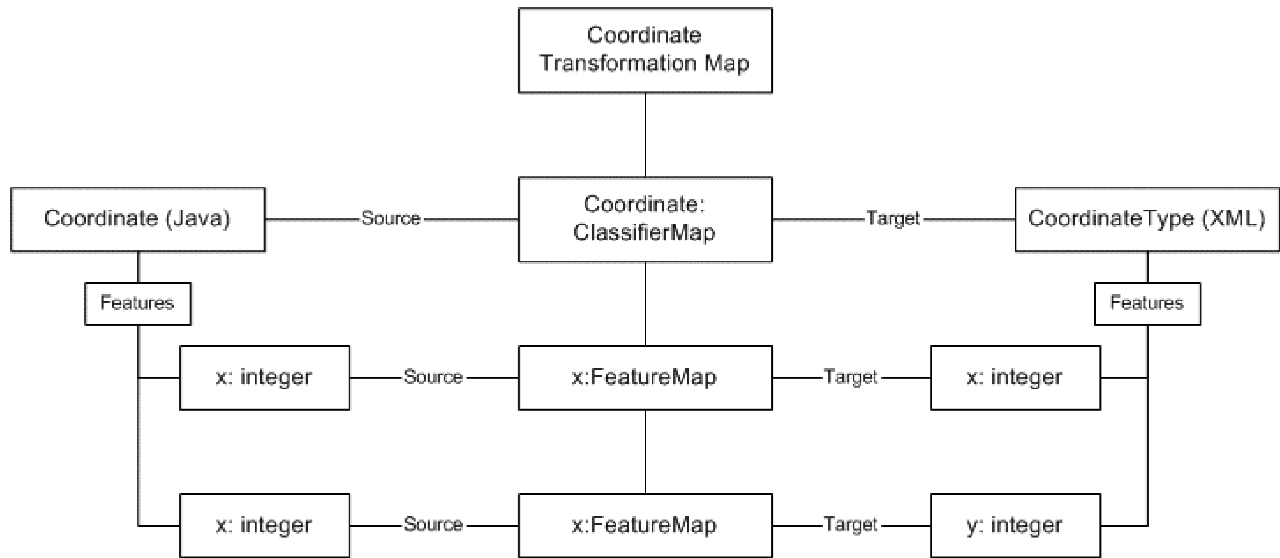

Figure 7: Using CWM to define a transformation map between a Java Coordinate and XML CoordinateType.

A transformation engine can use these transformation rules, in combination with shared metadata, to automate the conversion of a Java Coordinate instance to and from an XML document. As long as the appropriate metadata is loaded into the system, the transformation engine can easily support the different representations of coordinates we discussed above.

This approach is also quite flexible. For example, assume another XML type is added to the system called PointType. To convert a Java coordinate to and from a XML PointType requires loading a description of the PointType into the system and defining the appropriate transformations. Thus an MDA approach scales much better than hand-coding individual transformations.

CONCLUSION

This paper has tried to show how Model Driven Architecture (MDA) reduces the complexity of integration by providing a standard way of describing a system's metadata via the use of models. These models make it possible to declare how data from one system should be transformed to another system. These declarations, which can be defined in a configuration file or via a graphical editor, make it possible to largely automate data transformations via the use of generic transformation engines. These engines eliminate the need to write costly and time-consuming transformation code, thereby making integration projects faster, less complex and in the end more successful. Thus MDA provides a significantly more scaleable, and easier to understand, method of integrating systems than the traditional approach of hard-coding data transformations in programming languages such as Java or XSL.

REFERENCES

Frankel, D. 2003. Model Driven Architecture. Wiley Publishing.

Java Community Process. Jun3, 2002. Java™ Metadata Interface (JMI) Specification - JSR 040. Version 1.0. http://jcp.org/aboutJava/communityprocess/final/jsr040/index.html.

Manes, A. November 1999. Paving the Way for Transparent Application and Data Interchange. A Java API for Metadata. http://java.sun.com/products/jmi/pdf/JavaMetadataAPI.pdf.

Object Management Group (OMG). October 2001. Common Warehouse Metamodel (CWM) Specification, Volume 1. Version 1.0. http://www.omg.org/cgi-bin/doc?formal/01-10-01.pdf.

Object Management Group (OMG). April 2002. MetaObjectFacility (MOF). Specification 1.4. April 2002. http://www.omg.org/cgi-bin/doc?formal/02-04-03.pdf.

Poole, J. 2001. Model-Driven Architecture: Vision, Standards And Emerging Technologies. Position Paper Submitted to ECOOP 2001. http://www.cwmforum.org/Model-Driven%20Architecture.pdf.

Schmelzer, R. November, 2003. Solving Information Integration Challenges in a Service-Oriented Enterprise. ZapThink. http://www.metamatrix.com/whitepapers/Zapthink.pdf