CHARTERIS

A Meaning Definition Language

Draft 2.02

Robert Worden, 22 May, 2001

rpw@charteris.com

Abstract: A Meaning Definition Language (MDL) is proposed, whose purpose is to define what XML documents mean and how they express that meaning. MDL defines what a document can mean in terms of a UML class model or RDF Schema, and defines how to extract the meaning, in terms of XPath. MDL is a simple language. It has many applications such as: (1) validating that an XML language can convey its intended meaning, (2) automated translation of documents between XML languages, (3) automated retrieval of information on the Semantic Web, (4) supporting meaning-level XML query languages, and (5) programming APIs to XML at the level of meaning, independent of document structure. MDL will enable tools and users to interface to XML at the level of meaning rather than structure. MDL-based automated XML translation and a meaning-level query language are already supported.

Keywords: UML, XPath, XSLT, XML Schema, Semantic Web, RDF Schema, DAML+OIL, Schema Adjunct Framework

CONTENTS

1. INTRODUCTION

This paper proposes a Meaning Definition Language (MDL) whose purpose is to define what the documents in an XML language can mean, and how they express that meaning.

MDL addresses the data-centric end, as opposed to the text-centric end, of the XML spectrum – XML whose primary audience is not a human reader, but a computer program which will process the information, respond to it, store data items in a database, and so on. MDL is relevant to the W3C Semantic Web Programme.

A key component of the Semantic Web is RDF Schema, which can be used (on its own or with ontologies such as DAML+OIL) to define what any resource – including an XML document – can mean. While RDF Schema gives the means to define what an XML document means, it does not define how it means it. It does not define which structures within an XML document express particular kinds of meaning, or how you access them. That is the problem which MDL addresses – defining what meaning is conveyed, and how it is conveyed.

RDF Schema is about semantics; XML schema languages are about syntax; MDL is about the bridge between syntax and semantics.

To define **what** an XML document means, you need an ontology. MDL uses a minimal ontology framework built on the concepts of UML class diagrams - of objects in classes, simple properties, and associations. This is closely equivalent to RDF Schema, or a subset of DAML+OIL. MDL could be extended to handle more expressive ontologies if required.

MDL defines **how** an XML document conveys meaning by using XPath. To define how an XML document conveys some meaning, you need to define not just what information is on each node, but also how to navigate around the document, to pick up the different kinds of meaning correctly. MDL does this systematically for all meanings about objects, simple properties, and associations, using a common XPath-based notation. That is the core of MDL.

The key benefit of MDL is this: users and application developers can express their needs by saying what meaning they need to know. Using MDL, automated tools can then convert the ‘what’ into the ‘how’, to navigate XML documents to get that meaning. The user then does not need to know the ‘how’ of document structure and navigation. MDL-based tools can provide structure-independent, meaning-level interfaces to XML documents.

MDL has a wide range of potential uses, including:

- **XML Queries and Presentation**: Using MDL in conjunction with future XML query and presentation tools, it is possible to express both query conditions and query results in domain-model semantic terms, rather than in terms of XML structure. This meaning-level query language shields users from XML structure, and enables users to use the same query across multiple XML languages which express the same meanings

in different structures. A working demonstrator of a meaning-level XML query language now exists.

- **Insulating Applications from XML Structure Differences**: We envisage future application development tools which, by using MDL, allow developers to view XML documents in terms of the meaning they convey, independent of their structure. For instance, we have developed a Java-to-XML API, derived from the MDL, which delivers Java objects from an XML document in the classes of a semantic model of the domain (UML or RDF Schema), rather than (as in interfaces such as DOM) classes based on XML structure.

- **Automated XML Transformation**: The purpose of transforming an XML document from one language to another is to preserve its meaning. MDL gives a structured catalogue of the meanings of any two XML languages, which defines their meaning overlap and so defines what can be translated from one to the other. The meaning overlap between languages can be found automatically from their MDL definitions. MDL can act as a specification of the translation, and can also (by using the 'how' information) generate the XSLT to do the translation. An MDL-based tool exists which can do this now (XMuLator from Charteris).

- **The Semantic Web**: The vision behind the W3C Semantic Web initiative is that the meaning and content of resources on the web should be accessible to machine processing, so that automated agents can do useful tasks. To do this, those agents will need to know not only what information is in a resource, but also how to extract it. MDL provides the means to do this, for all data-centric XML documents – not just for RDF documents.

- **Documentation**: Authors of XML languages are often quite lax in defining what those languages mean - perhaps hoping that the tag names make clear what is meant. If there were a simple XML-based way to define XML language meanings (i.e. MDL), language authors might be encouraged to define meanings explicitly in MDL, just as they now define XML syntax in DTDs or schemas. Language users and application designers would benefit from the extra clarity.

- **Language Validation**: From the MDL and the schema of a language, you can check automatically that the language structure is capable of conveying the intended meaning. Discrepancies between the language structure and its intended meaning (of which there are many) can be flagged automatically.

- **Support of XML Application Design**: In MDL, each XML language meaning is defined in terms of a UML semantic model, which should be the same model as is used to design applications using the XML – or should relate very closely to that model. The MDL, XML schema and UML model can then underpin the links between the XML and the applications which will use it, more precisely than an XML syntax definition could do on its own. For instance, if a relational database is mapped onto the same UML model, the validity of any XML-database links can be checked automatically from the MDL; you can check that loading XML into a relational database does not change its meaning.

In summary:

MDL will enable both applications and users to interface to XML at the level of its meaning, rather than its structure.

MDL provides a simple and practical bridge from the structure level to the meaning level. The history of programming is a progression from implementation-centred tools to user-centred, meaning-centred tools – from assembler code to objects, from Codasyl to Relational, from 3270 to GUI. Our interfaces to XML must inevitably rise to the meaning level, and MDL can help them to do so.

2. HOW XML EXPRESSES MEANINGS

2.1 A Minimal Model of XML Meanings

Before describing the 'how' of XML meanings, we must first define the 'what'. A minimal model of XML meanings assumes that any XML document can express meanings of three kinds:

- **About Objects in Classes**: information of the form "there is a student" or "there are three cars"
- **About the Simple Properties of the Objects**: "the student's name is John" or "the make of the car is 'Volvo'".
- **About Associations between the Entities**: "this student owns that car" or "this manufacturer made that product".

We use the term 'simple property' to refer to a property whose value can be naturally expressed in a single character string, or XML simple type. Any more complex property is taken to be an association to one or more complex objects. Simple properties have values which are typically stored as XML attribute values, or as the text values of elements with no child elements.

These three concepts – of objects, simple properties and associations – are the building blocks of UML class diagrams. They have a successful track record of application in modelling of information and knowledge – for instance, in Entity-Relation Diagrams and AI frames. More recently, RDF Schema, which is proposed as a foundation for defining the meanings of web resources in RDF, embodies the same three concepts (in RDF and RDF Schema, the term 'property' encompasses both what we here call 'simple properties' and 'associations' – see below).

More expressive ontology formalisms, such as DAML+OIL and KIF, are also in development and in use. We envisage that in due course, MDL might be extended to use more of these formalisms. For the moment, however, a lot can be done with the minimal model of meaning above; we propose that MDL should initially support just the three core types of meaning above. We propose to use a subset of DAML+OIL interchangeably with UML (XMI) to describe those elements of meaning that MDL supports. These languages say what an XML document can mean; MDL also says how it means it.

It is hard to see how much meaning can be expressed at all without using all three of the core meaning types. Inspection of any data-centric XML document will show that it expresses meanings of all three types: about objects, simple properties and associations. That is why we have adopted the three types as our minimal model of XML meaning.

At this point, we could strive to define more precisely just what is an object, what is a simple property and what is an association. This would get us into deep semantic waters, where actually there is a lot of user choice available. For instance, simple properties can often be turned into associations (e.g. the simple property 'birth date' can be regarded as

an association between a person and a date) and associations can be turned into objects with other associations (e.g. the association 'person owns car' can be turned into an object 'ownership' with associations to 'owner' and 'owned' roles, and with simple properties such as 'start date'). For the moment, we prefer to leave these choices open to users (i.e. to analysts who build class models of domains), with only two stipulations:

- **Simple properties are defined to be atomic** – simple properties should be the sort of thing whose value you can express by a simple text string, and so would typically put into an XML attribute, or element with simple type. If you are tempted to make a more complex property, don't; make it a related object of a different class.

- **Simple properties are defined to be single-valued** (whereas UML allows 'attributes' to be multiple-valued). What might look like a multiple-valued property can be modelled as a 1:M association to objects which carry the multiple values.

With these definitions, we can now discuss how XML documents typically express the three main elements of meaning. What is said in English in the next three sub-sections, MDL is designed to say formally.

MDL is designed to address the typical ways people use XML to express meanings, not necessarily to address all conceivable ways people might use XML to express meanings. However, it may turn out that the general XPath-based underpinning of MDL can be extended to accommodate other ways of expressing meaning in XML, beyond those we discuss below.

2.2 How XML Represents Objects

In principle, there might be many different ways for an XML document to denote the existence of an object in some class, but in practice only one way seems to be commonly used.

Objects are almost always denoted by XML elements. There is typically a 1:1 correspondence between element instances and objects in a class. There are some exceptions to this, described below.

The simplest way that an element can represent an object is by saying in effect 'every element with tag name T represents one object in class C'. There are two useful elaborations of this scheme:

- You may say 'an element with tag name T represents an object of class C only if it can be reached by a path /R/S/…/T from the root of the document'. Elements with the same tag name reached by different paths may have different meanings

- You may also say 'an element with tag name T represents an object of class C only if it has an attribute A with value V (or a nested element with some special value)'. This enables one element type to represent objects of several different classes (typically all subclasses of a fairly narrow superclass) conditionally, depending on the value of some attribute.

More complex elaborations are possible. The class of the denoted object might depend on the values of several attributes taken together, or on both the path and attribute values. These forms can all be expressed in MDL.

There is sometimes a choice about which element tag names denote objects of a particular class. For instance, if an element with tag T always has just one child element with tag name V (along with other children), then objects of some class C might be denoted either by the T elements or the V elements; it makes no difference, because the Ts and Vs are in 1:1 correspondence to one another. In practice there is usually a more 'natural' choice out of the two.

There are always paths in the document (defined by XPaths) from the element which represents an object to the other nodes (see below) which represent the simple properties and associations of that object. If you have a choice of which element type denotes an object (like the T/V choice above), choose the one which has the simplest XPaths to its properties and associations. That should be regarded as the more 'natural' choice.

If an XML document denotes objects of a certain class, you need to know just which objects in the class it denotes. For instance, if element T denotes objects of class 'person', we need to know which people are included in a particular document – surely not all people living or dead. Part of the meaning definition is 'which objects of a certain class are represented in a given document?'

This is defined by the **Inclusion set conditions** for the objects in the document – the conditions an object must satisfy to be included in the document. Some inclusion sets are (from information visible in the document itself) arbitrary – such as 'this document represents just one purchase order'. Other inclusion sets depend on simple properties of the objects ('all employees of the company with salary less than $50,000') while others depend on associations to other objects represented in the document ('all employees who work in the departments represented in this document'), or a combination of the two.

While typically there is a 1:1 association between XML elements and the objects they represent, there are some exceptions:

- It is quite common for an XML document to use the style of a de-normalised relational database – for instance, to have a set of 'product' elements, each of which contains (nested within it) information about the manufacturer of the product. Then information about one manufacturer (who makes several products) may be stored redundantly once for each product, much as in a de-normalised relational table. One 'manufacturer' object is represented by several elements. As we shall see below, MDL is built around XPath, and XPath 1.0 does not give us the means to remove the redundancy from this redundant representation in MDL. However, the XQuery proposal contains a 'distinct' function which would enable MDL to define it, effectively removing the duplicates in the results of an XPath expression. For simplicity, we will not discuss this case further in this draft.

- Objects can on occasion be represented by XML Attributes. Although an XML attribute can have little internal structure (which therefore gives little chance to represent the properties and associations of the object internally to the attribute) nevertheless, properties and associations can be represented by nodes which can be found from the attribute node – e.g. 'peer' attributes of the same element. MDL accommodates these cases.

- An object is occasionally represented by multiple elements in a way which (unlike the ‘denormalisation’ technique above) does not represent the properties and associations of the object redundantly. In stead, different properties of the same object are represented inside different elements, each of which ‘represent’ the object just as much as any other. In effect, the object is represented as pieces of a jig-saw puzzle which can be put together. One of the recommended XML encodings of RDF allows such a representation of objects.

2.3 How XML Represents Simple properties

Knowing an object exists, without knowing any of its properties, is typically not much use. An XML document needs to convey the values of some simple properties the objects which it represents – that is, to represent those properties. As described above, because of our choice that simple properties should have simple atomic values, simple property values are easily represented as the values of attributes, or as the values of elements with simple types. In practice, simple properties are nearly always represented in one of these two ways.

Either a simple property is represented by an attribute (i.e. the value of the attribute represents the value of the simple property)

Or the value of a simple property is represented by the text value of an element.

Again, it is possible to think of exceptions (e.g. where a numeric simple property is represented by position of an element in a list of elements), but these are not very common and we will not discuss them further in this draft. We believe they can be accommodated in the XPath-based approach we propose.

As when XML represents objects, the fact that ‘an element with tag name X represents a simple property P’ need not be unconditional; it may depend on other conditions being true. For instance, there is a generic (uncommitted-schema) style of using XML in which a person might be represented as:

```
<person>
<persProp propName=’age’ propValue = ‘20’ />
<persProp propName=’sex’ propValue = ‘male’ />
...
</person>
```

This uncommitted style allows you to represent all sorts of unanticipated properties of people, without having to change your DTD or schema. However, the attribute ‘propValue’ only represents the simple property ‘age’ conditionally - when another attribute ‘propName’ of the same persProp element has the value ‘age’.

If the value of an element represents the value of a simple property, you need to know precisely which object that property belongs to. We assume for simplicity that this must be one of the objects represented in the same document. As these objects are in 1:1 correspondence with elements of some type within the document, one possible question is then: which element represents the object which owns the property represented by this element/attribute?

It turns out that this is not the important question. The important question is: given an element which represents an object, how do you navigate in the document, to get to an

element or attribute which represents one of the simple properties of the object? Often this navigation is trivial; the element representing a property is one of the immediate children of the element representing its object, or an attribute representing a property belongs to the element representing the object. In some cases it is not so trivial. For instance, the simple property may be represented by a node branching from some node higher in the tree, above the element representing the object. Other remote nodes can be used, and are sometimes used in practice.

In all of these cases, to specify how the XML represents simple properties, you need to define the path from the node representing an object, to a node representing one of its properties. XPath is an appropriate notation to define these paths. When a node represents a simple property only conditionally, the condition can either be represented in the XPath expression, or could be represented separately.

Only certain node types and paths are appropriate to represent a simple property of an object. There are three main tests of appropriateness:

- The XPath must be capable of reaching a node of the correct name.
- Since each simple property must be single-valued, the XPath from the element representing the object to the element representing its attribute should be **unique** – guaranteed to give a node-set with at most one node in it.
- When the simple property is an obligatory (non-optional) property of the object in the UML model (for instance, when that property is a part of a required unique identifier for objects), then the XPath from the element representing the object to the element representing the property must be **guaranteed** to give one and only one node in its node-set.

These tests of appropriateness can often be made automatically – for instance, using the XML schema for the language.

2.4 How XML Represents Associations

For representing objects and their simple properties, the main choices offered by XML (and used in practice) are quite limited – although, as we have seen, there are variations and elaborations within those main choices.

For representing associations, however, XML offers three markedly different methods, each of which is widely used, and which have profound consequences for document structure. Within the three main types of representation, variants can be found. The three main ways to represent an association are:

- **By Nesting of Elements**: To represent an association between objects of classes A and B, the elements representing B objects may be nested inside the elements representing A objects – either as immediate children, or more remote descendants. (typically this will be a 1:1 or 1:M association; but it need not be – see below).
- **By Identical Elements**: For example, if each product in a purchase order is made by just one manufacturer, then the simple properties of the manufacturer may be grouped inside the same element as the simple properties of the product. Then that one element

effectively represents the product, the manufacturer and the association between them.

- **By Shared Values**: For example, elements representing 'student' objects and elements representing 'course' objects are not nested inside one another; but somewhere inside each 'course' element can be found the names of all students attending that course. Then the N:M association 'student attends course' is represented by values (student names) shared between the student element and the course element.

All three of these representations are commonly found in XML languages. The first two types occur frequently in 'hand-crafted' languages to represent the key (hierarchical) relations of the application domain, while the third is more common in languages which must represent a wide range of associations – particularly N:M associations – in a systematic way.

Within the third type of representation, many variants are possible. The shared values may be element values, or attribute values, or IDs and IDREFs. An N:M association may be factored out into two different 1:M associations with a new class of object in the middle (like a UML 'association object'). Objects of this new class may be represented by other elements remote from the elements representing objects at the 'ends' of the original association.

While these representations of associations differ profoundly from one another, we can nevertheless use a common framework to describe all of them.

For all representations, we take the view that each instance of an association is represented by one node of the XML tree. Regardless of the cardinality of an association, any one instance of the association binds together two individual objects – the two objects at either end of the association. So to define how an XML document represents an association, we first define which node type (= element name or attribute name) represents an instance of the association. Call these nodes the **Association Nodes**. We next define how each association node is bound to the two elements representing the objects at either end of the association. Thus we define how the association node 'finds' its two object instances.

For instance, when an association is represented by element nesting, the association node is the inner of the two nested elements, which also represents the object at one end of the association. Finding that object node from the association node is done via a trivial 'stay here' path. The path to find the element representing the other object (at the other end of the association) consists of going up the tree to an ancestor node.

For all three types of representation we can define how the association node finds the two object-representing nodes by XPath expressions. Conditions in the XPath expression can represent sharing of values, and can also denote conditional representation – when a node only represents an association subject to other conditions. Thus XPath gives us the tools we need to define how an XML document represents an association, by any of the three main methods above.

We may need to start at either end of the association, and navigate to the other end. This means we need to know both the XPaths to navigate from an association node to each of its 'end' objects, and the XPaths to navigate from an object-representing element to the

association nodes for one of its associations. This makes four different Xpaths. To define how any given association is represented, all four should be provided.

As when representing simple properties, it is not *a priori* obvious that a given association node type and XPaths are appropriate to represent some association. The conditions to decide 'appropriateness' are quite complex, depending both on the cardinalities of the association, and the inclusion set conditions of the objects at either end. These appropriateness conditions can be checked automatically from the MDL and the schema/DTD.

For instance, if an association (A Rel B) is represented by nesting of the A elements inside the B elements, this requires that the inclusion sets of object A are of the form 'include only those A which have an association (A Rel B) to some B represented in this document'; for an element representing any A which does not have this association, there would be no place to put it.

2.5 Discussion

We believe, from examination of a modest number of XML languages, that the three core types of meaning – about objects, their simple properties their associations - do a good job of underpinning all the meanings which are commonly expressed in data-centric XML documents. This might have been expected, because the same core types of meaning have done a good job of modelling the application domains of computer systems, for many years. UML class diagrams have evolved to their present form through many years' experience.

It may seem to be a deterrent that, before you can express the meaning content of an XML document in this way, you first have to build a UML class model of the domain. This might seem a prohibitive effort. We believe it will not be, for three reasons:

- For many domains, UML class models already exist.
- UML Class models are closely equivalent to RDF Schemas, which underpin the semantics of the Resource Definition Framework. Under the auspices of the Semantic Web Initiative, there are emerging ontologies using RDF Schema, or based on it – notably in DAML+OIL. We expect these will be the starting point for specialisation to cover the semantics of many specialised domains, without (we hope) too much proliferation and duplication.
- We have found that in practice a UML class model does not need to be honed to the utmost elegance to act as a foundation for describing XML meanings. It is almost sufficient that the model be a catalogue of all the classes of object, their simple properties and associations that you will need in the domain. Arranging this catalogue into the most elegant class hierarchy (to maximise the leverage of inheritance) is actually not very important; XML meanings can be mapped onto the catalogue even if its inheritance hierarchy is not the 'best', whatever that means. The XML mappings can be preserved as the inheritance hierarchy is refined later.

While having the 'best' class hierarchy is probably only of secondary importance, what is important (and sometimes neglected in the XML literature) is to pay attention to all three facets of meaning – to classes of objects, to simple properties and to associations. In the

catalogues of meanings which are constructed to support XML languages, objects and simple properties are typically prominent – because they are clearly visible in XML or other message formats – but associations are sometimes neglected, perhaps because they are often implicit or taken for granted. Associations are a fundamental component of meaning, and there are hardly any complex meanings you can express without them.

We believe that trying to get by without a proper treatment of associations in XML meanings is like sitting on a two-legged stool; you won't stay upright for very long.

3. PROPOSED MEANING DEFINITION LANGUAGE

3.1 Requirements and Design Choices

We require that MDL should be

- Simple and easy to learn
- Expressive enough to capture most of the ways in which XML is used to convey meaning – that is, to capture all the meaning devices described in the previous section
- Clear and self-explanatory
- Precise and amenable to machine processing
- Compatible with XML standards and recommendations
- Usable with different XML-based means of expressing XML structure, such as XML Schema, Relax or TREX.
- Expressible as a Schema Adjunct
- Expressible in RDF form

From the discussion above, XPath does a lot of the work required of MDL, so the remaining design choices are not very complex. However, there is one design choice worth discussing.

MDL describes a set of mappings between an XML language and a UML class model. Should the 'point of view' of MDL be UML-centric or XML-centric? Should it be semantics-centred or syntax-centred? In other words, when reading an MDL document, should it be easier to answer the question "How is this meaning construct represented in the XML?" or to answer the question "What meaning constructs does this piece of XML represent?" ?

A top-down approach would perhaps start from the meaning (UML or RDF Schema) and work down to the implementation (XML) – which would favour a meaning-centric organisation. However, other design aims conflict with this:

- It would be useful if MDL could be embedded within XML Schema documents as 'appinfo' annotations, so that the syntactic and semantic aspects of an XML language can be viewed together.
- It would also be useful if MDL could take the form of a Schema Adjunct document, so that the Schema Adjunct framework can support run-time use of MDL information.

Both XML Schema and Schema Adjuncts are XML-centric in structure, and not at all UML-centric. Therefore we propose to make the primary structure of MDL to be XML-

centric, (i.e. to easily answer questions of the form "what does this XML construct mean?"), mainly to make its structure compatible with the structures of XML Schema (or other schema languages) and the Schema Adjunct proposal.

This decision does not imply a far-reaching lock-in to XML-centric structures. There will be a simple XSLT stylesheet to convert from the XML-centric primary form of MDL to a UML-centric secondary form for MDL documents, and similarly in the reverse direction. So you can start or end with whichever form of MDL you prefer.

Each MDL document will be written with reference to a UML class model (or RDF Schema), and it will frequently be useful to process an MDL document in conjunction with its UML/RDF Schema model. The standard XML interchange form for a UML model is XMI, so it might seem natural to represent UML models as XMI. However, the XML representation of RDF Schema is more concise, and almost meets our needs. The notation of DAML+OIL is a modest extension of RDF Schema which even more closely meets our needs, and we propose that a subset of DAML+OIL be the primary representation of the semantic model. For brevity, we will generally refer to this as a DAML model.

We believe this choice will make better contact with a number of RDF-based semantics initiatives. Again, the choice does not represent any heavy lock-in to the RDF world rather than XMI – we envisage that stylesheets can easily be generated to transform between the two; in fact such stylesheets already exist.

This DAML model representation can be used, for instance, in a tool to check that an MDL meaning specification actually matches its DAML model – before checking that the XML structures and their MDL mappings are appropriate for the kinds of meaning they are trying to express.

3.2 The Core Principle of MDL

The core principle of MDL is a simple one:

To extract any piece of meaning from an XML document (e.g. to extract the value of a simple property or the target objects of an association) you need to navigate around the document. MDL defines how you need to navigate to get any piece of information, using XPath as the language to define the required navigation.

That is it. Although this may seem very simple, it is in sharp contrast to the view of meaning embodied in certain XML tools (e.g BizTalk Mapper and other XML mapping tools) and E-commerce standards initiatives. These take the view that 'Information is embodied in nodes of the document. Therefore you need to define what piece of information is embodied in each node.' The node-centric view is almost sufficient for meaning about objects and simple properties, but it falls down completely for meaning about associations. To define how XML represents associations, it is essential to talk about XPaths.

RDF is a metadata notation for talking about information resources. Amongst other things, it can talk about their meanings. So RDF can be used, in conjunction with RDF Schema or DAML, to define what some set of XML resources may mean. You might think that RDF can also define how these documents express their meaning, by treating

nodes in the documents as resources and defining what meaning is carried by each node. However, this is not sufficient on its own – or at least, not in the ways in which (to our knowledge) RDF has been used so far. Information is not just carried by nodes; it is carried by nodes and paths. That is why MDL is required - to define the required XPaths, which are the bridge between meaning and XML structure.

3.3 A Simplification of the Language

MDL requires you to specify XPaths for both simple properties and associations – to define how you get from a node representing an object to the nodes representing its properties and associations.

Specifying all of these paths might be a lot of work, unless you had an automatic tool to help you do it. Fortunately, in the vast majority of cases, the required path – for instance the path from a node representing an object to a node representing one of its simple properties - obeys a 'shortest path' heuristic; it is the shortest possible path from the one node to the other. Similarly, nearly all paths from object-representing nodes to their association nodes are shortest paths.

We can therefore simplify the language by defining that the default XPath is always the simplest path; you only need to define the XPath explicitly when it is some different path. This means that the vast majority of XPaths need not be provided explicitly, but can be simply computed by MDL-based tools.

In the examples which follow, we will usually give the full form of MDL, with all paths specified. However, on occasion we will illustrate the abbreviated form which can be used when default 'shortest paths' occur – i.e. in the vast majority of cases.

3.4 Structure and Meaning of MDL

Here we give an informal description, with examples, of the structure and meaning of MDL. In the Appendix we will (in later drafts) define the structure of MDL as an XML Schema, and define its meaning with respect to a DAML model (of XML, DAML and the links between them) in MDL. (This stress-test of the formalism may lead to design changes.)

The primary form of an MDL document is a schema adjunct. It is an adjunct to a schema (e.g. XML Schema) which defines the structure of a class of documents. The MDL defines the meanings of the same that class of documents. Thus it takes a form such as:

```
<schema-adjunct  target=http://www.myco.com/myschema.xsd
xmlns:me=”http://www.myCo/dmodel.daml”  >

<document>
...
</document>

<element context = ‘product’>
...
</element>

<element context = ‘product/manufacturer’>
...
</element>


<attribute context = ‘product/@price’>
...
</attribute>

</schema-adjunct>
```

The attribute ‘target’ of the top schema-adjunct element is URL of the schema of the XML language which this MDL describes, if there is a unique schema. For languages which use elements from several namespaces, the elements in different namespaces may be constrained by different XML schemas, and it is important to define which prefix MDL uses to refer to each such namespace. This is done in the ‘document’ element below.

The namespace in the schema-adjunct element (in this example with prefix ‘me’) has a namespace URI for the semantic model (UML or DAML) which this meaning description is referenced to. This could be an RDDL URI, enabling access to some form of the model – e.g. XMI or DAML+OIL For simplicity in this draft we consider only MDL which defines the meaning of an XML language relative to one semantic model at a time. It is possible in principle for a language to convey meaning relative to two or more semantic models. In this case, there could either be two model namespaces declared here, or two or more MDL files for the language. We have not yet considered this in detail.

Thus the schema-adjunct element gives the means for an MDL processor to access both the schema and the semantic model, and to check the MDL against each of them individually or together.

The structure and meaning of the next-level elements – document, element and attribute – are described below.

3.4.1 The Document Element

MDL may be used to define the meaning of an XML language in which the documents have elements from several namespaces. In this case, we need to define the prefixes whereby MDL will refer to the various namespaces in the values of MDL attributes. This is done by namespace declarations in the document element such as:

```
<document xmlns:po= “http://www/myco/pospace”
          xmlns:ce = “http://www/myco/cespace” />
```

These prefixes must match the prefixes used in XPath expressions in the MDL which follows. If the MDL is to be used to generate XSLT, it is best to prefix all namespaces

and not to have any default namespace without a prefix. This is because XSLT treats un-prefixed element as having no namespace, rather than being in any default namespace.

While it should always be possible to access the semantic model when processing the MDL, you can often do useful things without accessing the whole model. MDL goes some way to describe just those parts of the semantic model which it requires. For instance, a tool can easily calculate the meaning overlap of two MDL descriptions without recourse to their shared semantic model, as long as they have such a shared model.

In order to make the MDL more self-contained, it may be useful to describe parts of the semantic model – which are not naturally described elsewhere in the MDL - inside the 'document' element. The language used to do this is the same as the language proposed for describing a complete semantic model, defined in section 6 below – for which we propose to use DAML+OIL.

A processor may then use these parts of the semantic model to support whatever it does. It also has the option to check that the partial semantic description in the 'document' element is a true subset of the full semantic model.

3.4.2 The 'Element' Element

As in any application of schema adjuncts, the value of the 'context' attribute of the 'element' element is an XPath expression which describes the path from the document root to a set of elements. The meaning of the 'element' element is that "each element in the set of elements with this name, reached from the root by this XPath, has these meanings".

One element (instance) in an XML document can have several meanings. It can denote several different objects and associations at the same time, or it can denote several different properties of different objects. Each one of these meanings is defined by a nested element <me:object> , or <me:property> or <me:association>. These are described below.

3.4.2.1 Elements Representing Objects

Consider an XML language which describes schools and has 'pupil' elements representing students. A typical document might look like:

```
<school name = "St Custard's" >
      <pupil name = "N. Molesworth" />
      <pupil name = "B. Grabber" />
</school>
```

The MDL for the element 'pupil' will then be:

```
<element context = '/school/pupil'>

      <me:object class= 'student'>
            <me:inclusion>
                <me:condition assoc="attends"
                    obj1="student" obj2="school" />
            </me:inclusion>
      </me:object>

</element>
```

This says that every pupil element which can be reached by the defined 'context' path represents one object of class 'student'. The 'inclusion' element states that the only students represented in this document are those students who have an 'attends' relation to a 'school' object which is also represented in the same document.

(The MDL for the 'pupil' element will also say that it represents the association [student] attends [school], by nesting. We have not put this yet, in because we have not yet explained how associations are represented in MDL – see below.)

There can be any number of conditions inside the 'me:inclusion' element; the only objects represented in the XML are those which satisfy all the conditions simultaneously.

Inclusion set conditions for objects in a class (the values of the 'set' attribute) are purely about meaning, not XML structure, and so refer to the semantic model – not to the structure of the XML document (while other conditions do refer to XML structure – see below).

All class names in the semantic model must be unique. In RDF Schema or DAML, classes are resources defined by guaranteed-unique URIs. When a class name such as 'student' appears inside an XML element such as me:object, it means "the student class in the DAML model at the namespace URI of prefix 'me'".

A more complex example, where an element may represent objects in two different classes:

```
<element context = '/school/pupil>

      <me:object class= 'mature-student' >
            <me:when objectToLeftValue = '@age'
                  test = '>' rightValue = '30'/>
            <me:inclusion>
                <me:condition assoc="attends"
                    obj1="mature-student" obj2="school" />
            </me:inclusion>
      </me:object>

      <me:object class= 'young-student' >
            <me:when objectToLeftValue = '@age'
                  test = '<=' rightValue = '30'/>
            <me:inclusion>
                <me:condition assoc="attends"
                    obj1="young-student" obj2="school" />
            </me:inclusion>
      </me:object>

</element>
```

In this example, the 'pupil' element may represent an object in one of two classes, depending on the value of one of its attributes. This dependence is described in a

'me:when' element, whose 'objectToLeftValue attribute is an XPath expression for the path from the node which represents the object to a node representing the left-hand value value, which is then compared to the 'rightValue' attribute, which is the value on the right hand side of the comparison. Both classes 'mature-student' and 'young-student' are typically subclasses of a class 'student', inheriting the association 'attends' from it, and the two classes have similar inclusion sets (both include just students at this school).

3.4.2.2 Elements Representing Simple properties

Typically when an element represents a simple property, the value of the property is carried by the value of the element.

In a language which contains 'stud' elements of the form

```
<stud> <longName>Jethro Albert Limburger</longName> </stud>
```

where the element 'stud' represents the object 'student' and the element 'longName' represents his name property, the MDL for 'longName' is:

```
<element context = '/school/stud/longName'>

      <me:property class = 'student' property ='name' type =
                  'fullName' >
            <me:find  objectToProperty = 'longName' />
            <me:convert lang='xslt' inTemplate ='t1'
            outTemplate = 't2' />
      </me:property>

</element>
```

In the nested 'me:property' element:

- The obligatory 'class' and 'property' attributes give the class in the semantic model of object the simple property applies to, and the name of the property. The model is the one whose namespace prefix is 'me:'.

- The 'type' attribute refers to a set of definitions of simple types which should be associated with the semantic model, to support any type conversions of data as necessary.

- The 'find' element has an attribute 'objectToProperty' which gives the XPath to go from object to simple property (that is, from the element representing an object of class 'student' to the element representing this property). Because simple properties must be single-valued, the objectToProperty path should always be unique, giving a node-set of size 1. In short-form MDL, the 'find' element is omitted when the objectToProperty is the shortest path.

- Further support for type and format conversion is provided in the 'me:convert' element. This envisages that the semantic model defines a 'central' format for any simple property (e.g. US-style dates) and that language definers may wish to use different forms (e.g. UK-style dates). When they do so, they may provide or use format conversion software in a variety of languages (e.g java, XSLT), and define in MDL which conversion software can be used to convert data values in and out of the central format. In the example, XSLT templates with names 't1' and 't2' are provided.

An element might represent a simple property only conditionally, when certain conditions apply. Consider a 'generic' language whose schema does not restrict it to representing certain properties:

```
<stud>
      <studProp pName='age'> <pVal>24</pVal> </studProp>
      <studProp pName='surname'> <pVal>Smith</pVal> </studProp>
      ..
</stud>
```

the element 'pVal' may represent many different properties, depending on the attribute 'pName' associated with it. MDL defining the meaning of the element 'pVal' is:

```
<element context = '/school/stud/studProp/pVal' >

      <me:property class='student' property='name' type = 'surName' >
            <me:when propertyToLeftValue = '../@pName'
                  rightValue = 'surname' >
            <me:find  objectToProperty = 'studProp/pVal' />
      </me:property>

      <me:property class='student' property ='age' type =
                  'integer' >
            <me:when propertyToLeftValue = '../@pName'
                  rightValue = 'age' >
            <me:find  objectToProperty = 'studProp/pVal' />
      </me:property>

      ..

</element>
```

The first me:property element says: the element pVal represents the student's name only when the attribute pName of its parent has the value 'surname'. This is captured in the 'me:when' element, which says that the element represents the property only when the left-hand side has the specified value. The attribute 'propertyToLeftValue' is an XPath to get to the left-hand side value from the element representing the property. In this vase the context node (start node) is the pVal element. '..' navigates to its parent 'studProp' element, and '/@pName' finds that parent's attribute 'pName'. There can be several 'when' elements; the node represents a property only when they are all true.

3.4.2.3 Elements Representing Associations

XML can represent associations in three main ways, which at first sight look very different from one another – by nesting of elements, by 'overloading' of elements, and by shared values. However, the three all share some common underlying principles, which means that the same XPath-based form of description can be used to define all of them.

In any XML representation of an association [E]A[F] between objects of class E and class F, nodes of some type denote instances of the association. Each instance of the association ties together one object of class E and one object of class F. For an association node (which represents an instance of the association) we need to define how it is linked to the element representing the E-class object, and how it is linked to the element representing the F-class object.

We do this using XPath expressions, and we provide the means to define the XPaths both from the object-representing elements to the association nodes, and in the reverse direction. When extracting association information from a document, paths in either

direction may be needed – either to go from E => A => F, or to go in the reverse direction.

Therefore the full MDL definition of an association has a path from the root to define the set of association nodes, and it has relative paths between the association nodes and the elements representing objects at the two ends of the association. This will be illustrated by examples of the three types of representation, and the MDL which defines them.

Representing Associations by Nesting

Consider a typical 'purchase order' XML message, which represents the relation between a purchase order and its order lines by nesting the 'orderline' elements inside its 'order' element. This typically looks like:

```
<po poNumber='123435' poDate='23/12/01'>
      <poLine prod='paperclip' qty='4000' />
      <poLine prod='pencil' qty='200' />
</po>
```

Here, the element 'po' represents an object of class 'purchase order', the 'poLine' elements represent objects of class 'order line' and the nesting of 'poLine' inside 'po' represents the association [purchase order]contains[order line].

The MDL for the element 'poLine' using the full-form representation of the association is:

```
<element context='/po/poLine'>

      <me:object class='orderLine'>
            <me:inclusion>
                <me:condition assoc="contains"
                    obj1="purchaseOrder" obj2="orderLine" />
            </me:inclusion>
      </me:object>

      <me:association assocName='contains'>
         <me:object1 class = "purchaseOrder"
                   objectToAssociation = 'poLine'
                   associationToObject = 'parent:po' />
         <me:object2 class = "orderLine"
                  objectToAssociation = '.'
                   associationToObject = '.' />
      </me:association>

</element>
```

The me:association element gives the name of the association in its 'assocName' attribute. It has two nested elements which are about the two objects linked by the association. 'object1' gives the XPath from the association node to the element(s) representing object 1 (whose class name usually occurs to the left of the full association name when referring to the relation) and the reverse path. 'object2' does the same for object 2 (whose class name occurs to the right of the association name). The values of objectToAssociation and associationToObject work as follows:

- 'objectToAssociation' of 'object1' is 'poLine', saying that: to get from the element 'po' representing the purchase order object (which is class 1 of the association) to the element 'poLine' representing the association itself, you just go to the 'poLine' child of the 'po' element.

- ‘associationToObject’ of ‘object1’ is ‘parent:po’ saying that: to get to a ‘po’ element representing an object of class 1 from a ‘poLine’ element representing the association, you go to the parent of the association node, which should be a ‘po’ element.

- ‘objectToAssociation’ of ‘object2’ is ‘.’, saying that: to get from the element ‘poLine’ representing the order line object (which is class 2 of the association) to the element ‘poLine’ representing the association itself, you just stay where you are.

- ‘associationToObject’ of ‘object2’ is ‘.’ saying that: to get to a ‘poLine’ element representing an object of class 2 from a ‘poLine’ element representing the association, you just stay where you are.

However, as all of these paths are the shortest paths between their respective end nodes, in the short-form MDL all these ‘objectToAssociation’ and ‘associationToObject’ attributes can be omitted.

Representing Associations by Shared Values

As an example of an association denoted by a shared value, consider a document denoting students and courses, where courses are represented as:

```
<crs name = ‘Maths’/>
<crs name = ‘English’ />
```

And students are represented as:

```
<stud name = ‘J Doe’>
      <att>Maths</att>
      <att>English</att>
</stud>
```

The relation [student] attends [course] is represented by the sharing of a value (the course name) between ‘stud’ and ‘crs’ elements. The nodes which represent each instance of the association are the ‘att’ elements. The MDL defining how the ‘att’ elements represent the association is:

```
<element context = ‘/stud/att’ >

      <me:association assocName = ‘attends’ >
         <me:object1  class = ‘student’
                  objectToAssociation = ‘att’
                  associationToObject = ‘..crs’ />
         <me:object2 class = ‘course’
            objectToAssociation = ‘../stud/att’
            associationToObject = ‘ancestor::school/crs’ >
            <me:when associationToLeftValue="."
                        objectToRightValue="@name"/>
         </me:object2>
      </me:association>

</element>
```

The interpretation the ‘object1’ element is as in the previous example. For ‘object 1’ (in the class on the left-hand side of the [1]assoc[2] notation) the paths simply pick out a parent or a named child element. These paths can be omitted in the short form MDL.

For the ‘object2’ element, the paths are again shortest paths and can be omitted in the short form MDL. The me:when element specifies a value that must be shared between the node representing object 2 of the association and the association node. The attributes associationToLeftValue and objectToRightValue denote respectively

- a path from the association node to find a left-hand value

- a path from the object-representing node to find a right-hand value

Then the association node only represents an association when these two values are equal.

Suppose that in stead of having several ‘att’ elements inside each ‘stud’ element, we had used just one ‘att’ element as in:

```
<school>
      <crs name = ‘Maths’/>
      <crs name = ‘English’ />
      <stud name = ‘J Doe’>
            <att>Maths English</att>
      </stud>
</school>
```

Then the MDL defining the association would be :

```
<element context = ‘/stud/att’ >

      <me:association assocName = ‘attends’ >
         <me:object1  class = ‘student’ objectToAssociation = ‘att’
                  associationToObject = ‘..crs’ />
         <me:object2 class = ‘course’
            objectToAssociation = ‘../stud/att’
            associationToObject = ‘ancestor::school/crs’ >
            <me:when associationToLeftValue ="."
                        objectToRightValue="@name"
                        test = “contains” />
         </me:object2>
      </me:association>

</element>
```

Here an me:when element has been used with an attribute test = “contains” (in stead of the default test = “=”) because a course name must be found in a list of course names in the ‘att’ element. This type of representation only works if course names are known to contain no spaces.

In effect an association node can be linked to each of its two object-representing nodes either by a simple path (when e.g the association is a child of the object-representing node) or by a longer path with me:when elements to limit the number of eligible object nodes.By defining separately an association node and two object-representing nodes, we allow for the possibility that the association node is quite remote from both object-representing nodes, and has me:when conditions for both of them.

Representing Associations by Identity

When one element represents two types of entity and an association between them, we call this ‘representing the association by identity’ or ‘overloading’.

As an example of an association represented by overloading of one element, consider an XML language about products, where a typical document is:

```
<stock>
      <prod name= “widget” mfr=”Acme” mfrLoc=”Chicago” />
      <prod name= “trunnion” mfr=”bitCo” mfrLoc=”Denver” />
</stock>
```

Here, information about two different objects (product and manufacturer) has been crammed into one element, ‘prod’. The name and location of the manufacturer are stored as if they were properties of the product. This is a bit like de-normalisation in a relational database, where separate tables are joined together for convenience or performance. We can get away with it if each product has only one manufacturer. It is frequently done in XML languages for E-commerce.

In this case, the element ‘prod’ represents the object ‘product’, the object ‘manufacturer’ and the association between them [manufacturer]makes[product]. The MDL for the element ‘prod’ is then:

```
<element context = ‘/stock/product’ >

   <me:object class = ‘product’>
      <me:inclusion description=”products currently in stock” />
   </me:object>

   <me:object class = ‘manufacturer’>
      <me:inclusion>
          <me:condition assoc="makes"
              obj1="manufacturer" obj2="product" />
      </me:inclusion>
   </me:object>

   <me:association  assocName=”makes”>
      <me:object1 class = ‘manufacturer’
            objectToAssociation = ‘.’
            associationToObject = ‘.’ />
      <me:object2 class = ‘product’
            objectToAssociation = ‘.’
            associationToObject = ‘.’ />
   </me:association>

</element>
```

This says that the XPaths to go between the ‘object 1’ node (in the association [1]assoc[2]), the association node and the ‘object 2’ node are all trivial, because they are all the same node. In the short form MDL, these trivial paths can be omitted.

Representing two objects by the same element implies that the objects are in a 1:1 correspondence. This only works if the association between them is 1:1 or M:1, and if the inclusion conditions for the object at the ‘1’ end of the association require the association to hold. In this case, a manufacturer will only be represented in the document if he makes a product which is represented in the same document.

Note how in the above example, the inclusion conditions for the ‘product’ object have been stated verbally in a ‘description’ attribute, rather than formally in a ‘set’ attribute.

3.4.3 The ‘Attribute’ Element

Anything which an element can represent, an XML attribute can also represent. The MDL defining how an attribute represents some meaning is identical to the corresponding MDL defining how an element represents the same meaning.

Attributes can denote simple properties, in just the same way as elements with simple types can denote simple properties. To define how they do, the ‘attribute’ element can contain <me:property> elements which have identical form and meaning to the <me:property> elements inside the ‘element’ element.

Attributes can be used to represent associations by shared values, and when they are, the attribute node is the association node. To define how this works, the ‘attribute’ element contains an <me:association> element which is identical in form to the <me:association> element for elements, described in 3.2.2 above.

For instance, if the relation [lecturer]teaches[course] is represented by the attribute ‘taughtBy’, as in the XML sample:

```
<school>
      <lecturer lName = ‘Fred Smith’  />
      <lecturer lName = ‘Peter Jones’  />

      <course cName=’maths’ taughtBy=’Fred Smith’ />
      <course cName = ‘English’ taughtBy=’Peter Jones’ />
</school>
```

the MDL which defines the meaning of the attribute ‘taughtBy’ is:

```
<attribute context = ‘/school/course/@taughtBy’ >

  <me:association assocName = ‘teaches’ >
      <me:object1 class = ‘lecturer’
         objectToAssociation = ‘../course/@taughtBy’
         associationToObject = ‘ancestor::school/lecturer’ >
            <me:equals associationToLeftValue = ‘@lName’
                  objectToRightValue = ‘.’ />
      </me:object1>
      <me:object2 class = ‘course’
         objectToAssociation = ‘@taughtBy’
         associationToObject = ‘..course’ />
    </me:association>

</attribute>
```

As usual, for the short form MDL we can omit all the objectToAssociation and associationToObject attributes.

In this case, there was actually a certain degree of arbitrariness in what node we chose to say ‘represents’ the association. We chose to say that the attribute ‘taughtBy’ denotes the association [lecturer]teaches[course]. We could equally well have said that the element ‘course’ denotes this association – again by shared values, but this time to find the shared value, you need to look at one of the attributes of ‘course’. Then the MDL and the XPath expressions would have been different – leading to and from the ‘course’ element rather than the ‘taughtBy’ attribute - but the result would have been the same.

3.5 Embedding MDL in XML Schemas

To embed MDL in an XML Schema, the identical syntax (including the ‘document’ , ‘element’ and ‘attribute’ elements from the Schema Adjunct proposal) can be used within ‘appinfo’ elements of schema annotations.

The simplest place to define what an element means is in an ‘appinfo’ element in the declaration of that element’s type – which may be either a global element declaration or a local declaration within some other type definition. In either case, it may be possible to

reach the element by several paths from the root of the document, and it is necessary to define (by the ‘context’ attribute) which paths to the element lead it to have particular meanings. Similarly for attributes.

An XML schema with embedded MDL will have two distinct types of ‘element’ elements (i.e. elements with local name ‘element’) in different namespaces. There is the XML schema ‘element’ which defines the type (=structure) of the element and its usage in other types, and there is the MDL ‘element’ which defines its meaning. The latter will have a namespace URI which links to the semantic model. Similarly for ‘attribute’ elements.

3.6 MDL in RDF

A combination of MDL and RDF might be useful, with RDF defining some metadata about a document, and MDL defining what the document means and how to extract that meaning.

There are two possible ways to realise this. Either one could embed MDL directly inside RDF documents, using different namespaces to distinguish them; or one could redefine MDL in terms of RDF triples, which are then expressed in XML along with other RDF. We have not yet investigated either of these in detail.

3.7 Summary of the Language

MDL uses the framework of Schema Adjuncts and the concepts of UML class diagrams, and it makes use of XPath expressions. It only requires the use of a small number of element and attribute types. Using these few constructs, we believe that MDL enables you to define essentially all the ways in which XML is used in common practice to convey meanings.

4. VALIDATION AND DOCUMENTATION

There are important uses of MDL for documentation and validation of XML languages: to enable XML language authors to state precisely what the languages mean, and to validate those statements against a number of constraints – in effect, to validate that the language can mean what its author intends it to mean.

An MDL definition can be statically validated in three ways: It can be validated against the structure definition of the language (e.g. XML Schema), against the semantic model, and jointly against the structure definition and the semantic model.

In addition, individual document instances can be validated against the MDL definition of the language. This would allow semantic checks of document instances which go beyond the syntactic checks against language schemas.

The first two validations are really just hygiene checks to make sure you have made the MDL consistent with some obvious constraints. The third is the most valuable, and enables you to check that the language really can express what it is meant to express. It can make this check more precisely than by looking at language structure alone.

For the purpose of explanation in this section, we shall assume that the structure of an XML language is defined in W3C XML Schema. For other language structure definitions such as TREX or RELAX, appropriate changes can be made.

In all cases we give informal summaries of the checks which can be made, rather than formal statements of conditions the MDL must satisfy. Trial implementations of the MDL validators described below would flush out many issues, and should perhaps be done in tandem with formal specification of the validation checks.

4.1 Validation Against the Language Schema

The following validation checks can be made between an MDL definition and the corresponding XML Schema:

- Every element or attribute declared in the MDL must also be declared in the Schema.
- Every XPath which appears as the value of a ‘context’, ‘associationToObject’ or ‘objectToAssociation’ attribute must be a valid XPath, given the language structure, from its starting node set (which is defined for every path).
- In ‘find1’ and ‘find2’ elements, the XPaths for ‘associationToObject’ and ‘objectToAssociation’ must be inverse paths; traversing the two in succession, in either order, should always deliver the starting node in the final node set.
- The values of ‘when’ attributes should be valid boolean XPath expressions.

4.2 Validation Against the Semantic Model

The following checks can be made between an MDL model and the corresponding semantic model:

- The value of every 'class' attribute must be a class in the model
- The value of every 'propName' attribute must be a simple property in the model, written in a form 'class:property' and belonging to a class which appears elsewhere as a 'class' attribute in the MDL.
- The value of every 'assocName' attribute must be an association in the model, written in a form [class1]association[class2] and belonging to classes class1 and class2 which both appear elsewhere as 'class' attributes in the MDL.
- For every class that is represented, there should be either an informal inclusion description, which describes which objects of the class are included in the document, or a formal inclusion set definition which does the same.
- The conditions in any inclusion 'set' attribute must only mention classes, simple properties and associations in the model, and constants (precise syntax of inclusion set attributes to be defined)
- If objects in some class are represented, there should also be represented sufficient simple properties and/or associations of this class to uniquely identify the objects within their defined inclusion set.

4.3 Validation Jointly Against the Schema and the Semantic Model

The following checks can be made of the MDL against the XML Schema and the semantic model jointly:

- If an element or attribute represents a simple property, then the type of the element or attribute (in the sense of XML Schema datatypes) must be capable of holding the type of the property.
- The XPath from a node representing an object to a node representing one of its simple properties must have cardinality matching the property. That is, if the property is an obligatory property of the object, the XPath must deliver a node set of cardinality 1. In any case, the XPath must not deliver a node set of cardinality >1.
- If an element representing an object of class C1 is nested inside an element representing an object of class C2, there must also be an association between C1 and C2 represented by the nesting. The inclusion set of C1 must depend on this association.
- If the same element represents two or more objects, then it must also represent associations between those objects, and those associations must be part of the inclusion set conditions for all of the objects except one.

- There are consistency conditions between the cardinality of an association, the cardinality of the XPaths in its representation, and the inclusion set conditions of the objects involved in the association. For each class of object, the possible number of instances as governed by the inclusion conditions and the cardinality of the association must match with the possible number of instances as governed by the XPaths.

- (Completeness) every element or attribute node in the XML structure must either represent some meaning (i.e have an me:object, me:property or me:association declaration) or be on a path from the root node to such a node.

4.4 Validation of XML Document Instances

While we can make many static checks that the schema of an XML language is consistent with the meaning it is intended to convey, there are constraints that cannot be conveyed statically in the schema language, or which the schema author did not write down.

If a document instance is linked to its MDL, then there can be further run-time checks that this instance is consistent with its intended meaning. The nature of these checks is hard to anticipate – but most semantic models have some domain-specific constraints which are not easily expressed in generic terms (such as cardinalities). Such constraints are best expressed in terms of the semantic model, rather than in terms of the XML structure. Then MDL can be the bridge to check those constraints.

For instance, the XML definition of the semantic model – whether in DAML, or XMI, or some other XML language – should obey the following constraints:

- No class should be a subclass of itself, either directly or indirectly

- All classes should be subclasses of a single root class

- No simple property name declared for a class should clash with the name of any simple property it inherits from any of its ancestor classes

- No association name for any pair of classes should clash with an attribute name they inherit.

MDL could support tools for checking that XML instance documents (in this case, semantic models) obey constraints such as these. Checking a constraint is similar to running a query. Each constraint is expressed in meaning-level language. MDL then defines what XPaths must be navigated to extract the information used in those meaning-level constructs. The XPath expressions are run against the instance document, and the results evaluated against the constraint expression, to report any violations.

We can then envisage a two-step validation of any instance document – first a structure/type validation against its schema, followed by validation against semantic constraints using MDL.

4.5 Analysis and Design of XML Applications

Complex applications need to be designed before they are built. The analysis and design uses tools such as UML, which operate at the level of meanings in the application domain, rather than the level of implementation structures. However, the present generation of tools for describing XML (e.g XML Schema) operate largely at the level of XML structures rather than meaning.

If you know that a new application will need to interact with some existing XML language, then the analysis and design for that application will benefit from having a description of the language at the meaning level. This description will naturally make contact with other analysis and design tools which operate at the meaning level. MDL is such a tool.

The MDL description of an existing XML language will fit in naturally with other UML-based analysis and design tools, to facilitate good design of those applications and shorten the development cycle.

4.6 Design of XML Languages

We can envisage a design tool for new XML languages which operates as follows: starting from a UML class model or DAML ontology, you select those elements of meaning in the model which you intend to convey in the language. You add further information such as the inclusion set conditions which are part of the ‘meaning model’ side of MDL. In this way you define which classes and which objects in those classes will be conveyed in an instance document of the language, and what information about those objects will be conveyed.

The tool then automatically generates both the XML Schema and the MDL for the language – ensuring that all the static validation checks on the language, as described in section 4, are satisfied. This guarantees that the new language can convey all the meanings which you intend it to, and documents that fact for its users.

5. OTHER APPLICATIONS OF MDL

Besides validation and documentation, there are other potential applications of MDL, described in this section.

The most powerful uses of MDL involve interfacing with XML documents at the level of meaning rather than structure. To do so, we need to solve two problems – the input problem and the output problem.

The Input Problem is to extract the information from an 'incoming' XML document and view that information directly in terms of the classes, simple properties and associations of the semantic model. From the nature of MDL, this problem is fairly simple to solve. MDL defines the XPaths you need to follow in order to extract from a document a given object, or any of its simple properties, or any of its associations. So to find the value of any simple property or association of some object, you simply need to follow the relevant XPaths in the document, as defined in the MDL. This is easily done if you have an implementation of XPath.

The Output Problem is to 'package' the information in an instance of the semantic model into an 'outgoing' XML document which conveys that information. It is not quite so obvious how to do this from the definition of MDL; but in fact it is fairly straightforward. You need to construct the document from its root 'downwards'. Generally you will come to nodes representing objects before you come to nodes representing their properties and associations. As you come to each node type, you check in the MDL what type of information the node type represents (e.g. what class of object, or what property), and you check what instances of that type of information exist in the semantic model instance. You then construct node instances to reflect these information model instances.

These brief descriptions show in principle how both the input problem and the output problem can be solved. We also have a practical demonstration that both problems can be solved, in the application of MDL to XML translation.

Charteris have developed a tool (XMuLator) which takes as input the MDL definitions of any two XML languages against the same semantic model, and their structure definitions (e.g. XML Schemas). It generates as output the XSLT to translate from one language to the other, preserving all their shared meanings. These XSLT transformations have been tested quite extensively, showing (for instance) that a round-trip translation between two or more languages will return an accurate subset of the input document (it is generally only a subset, because the information overlap between the languages is typically not perfect).

To translate between two languages, you need to solve the input problem for one language and the output problem for the other. By generating viable XSLT translations, XMuLator shows that both the input problem and the output problem can be solved in XSLT.

Knowing that both the input problem and the output problem can be solved, we now examine the potential applications which follow.

5.1 Meaning-Level Query of XML Documents

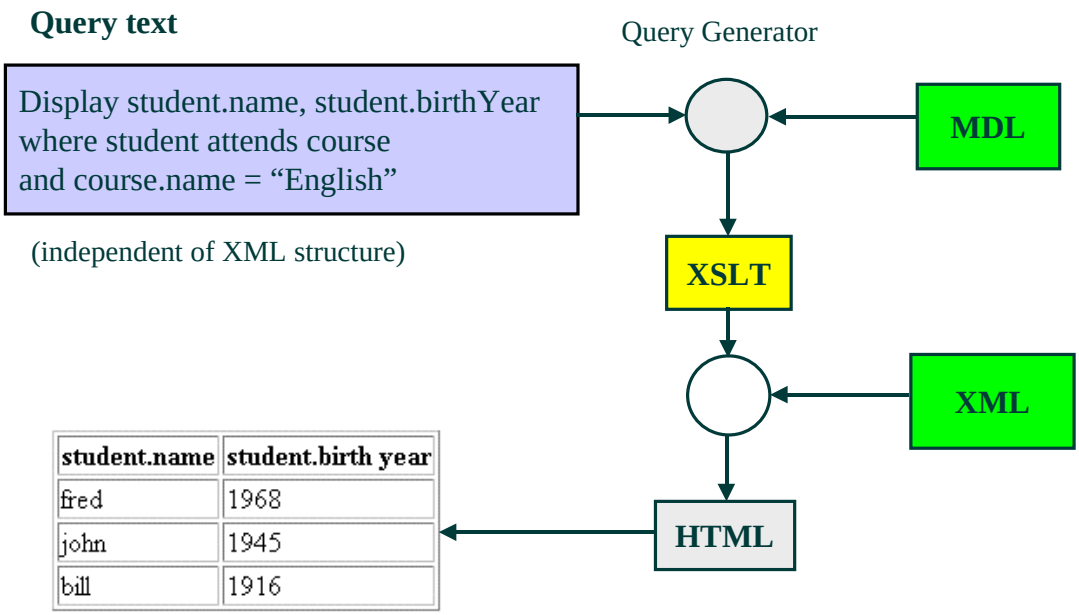
We would like to be able to interrogate XML documents in terms of their meanings, rather than their element structure. We would like to ask questions in meaning-based terms (e.g. 'how many objects of class X have association Y to a given object?) rather than in terms of document navigation. MDL supplies the information 'how do you navigate the document to extract a given piece of meaning?' (needed to solve the input problem) so MDL can support a meaning-level query language which gives meaning-based answers.

For example: a query tool could accept as input a query of the form 'Select all order items which are part of purchase order 12345 and which have quantity > 6'. It could then use the MDL to convert the conditions about properties and associations into XML navigation instructions (XPaths), run the XPaths over a document to retrieve the information, and format it appropriately.

This would have three benefits:

- Queries and answers would be expressed in more meaningful, user-oriented terms
- Users would not need to know about the structures of XML documents; they would only need to know what information they contain.
- A single query could retrieve similar information from many different XML documents in different languages which express the same information by different structures

We have built a small demonstrator of this type of meaning-level query languages, which accepts a query in an XML-independent language (of a form such as 'display orderLine.price, orderLine.quantity where orderLine part of purchaseOrder') and then converts this into XSLT to answer the query, producing HTML as output to display the results in a browser. The information flow for this demonstrator is shown below:

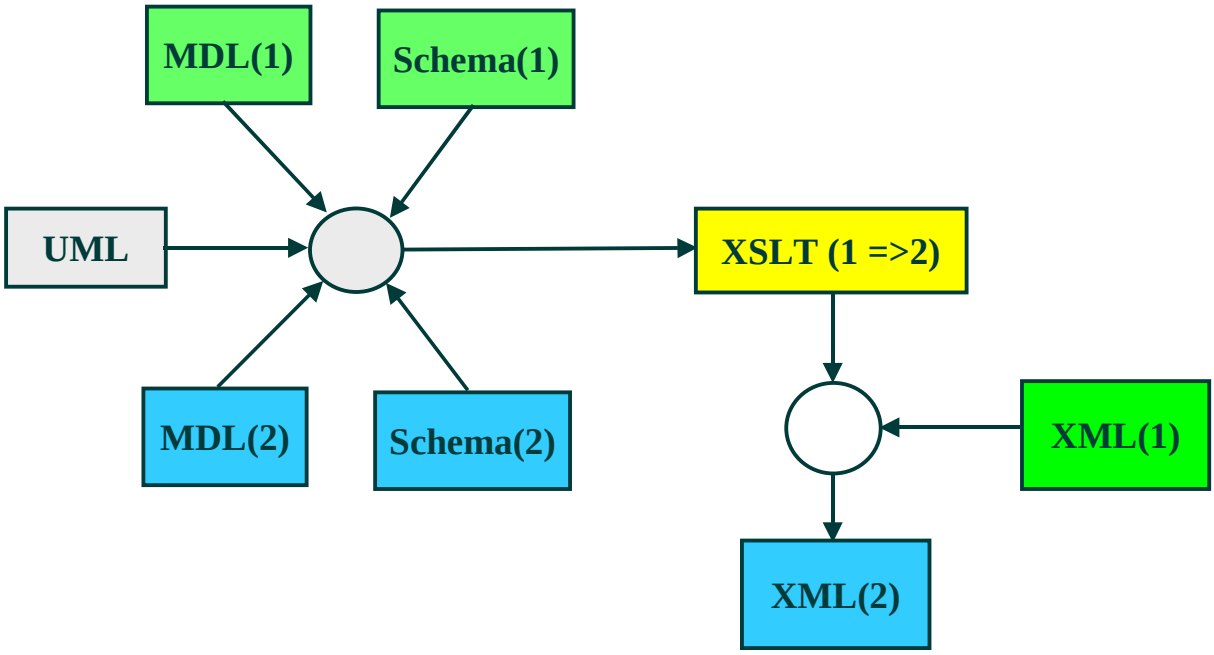


5.2 Specification and Generation of XML Transformations

The whole purpose of transforming an XML document from one language to another is to preserve the meaning – or as much of it as possible. Therefore the meaning contents of the two XML languages, as defined in their MDL, are a vital input to knowing what can be translated from one language to another.

Given the two MDL files, it is fairly straightforward to combine them to find the extent of meaning overlap between the two languages. This can then serve as a specification of what is to be translated, however the translation is to be done – for instance, by hand-coding of XSLT.

If the meaning overlap between two XML languages serves as a specification of all that can be translated from one language to the other, then we should be able to go one step further and actually generate the transformation from this specification. Charteris have developed a tool, XMuLator, which does this (currently in the case of single-inheritance UML models; but the extension to multiple inheritance is not hard). The information flow for this MDL application is shown below:



This approach to translation has significant benefits over hand-coding of XSLT:

- Even with only two languages to translate between, defining their meanings in MDL is much less labour-intensive that writing the full XSLT to translate from one to the other

- When there are N > 2 languages mapped to the same UML model, the effort of defining all their meanings in MDL grows only as N, whereas the cost of hand-writing all possible translations between them would grow as N^2. All possible translations can be generated from the MDL at no extra cost. This is particularly significant if each language goes through several versions, causing N to grow large.

- Because the MDL definitions are much more concise than XSLT and are independently checkable, the resulting translations are generally much less error-prone than hand-written XSLT. For instance, the translations generated in this way through 2, 3 or more intermediate languages have automatic round-trip consistency.

So MDL is the means to much more cost-effective XML translation and interoperability of XML languages – solving the N^2 problem.

5.3 Meaning-Level APIs to XML Documents

Developers need APIs to interface programming languages such as Java or C++ to XML documents. APIs such as DOM and {Sun Java API} are defined at a document structure level; to use them, a developer needs to be familiar with the document structure, its meaning, and how it conveys its meaning.

From the schema and the MDL definition of a language, we could generate (for instance) a set of Java classes which reflect not the document structures, but the meanings they convey. These classes would match classes of the semantic model, not the XML elements and types. This would be quite easily done on top of an XPath API, because MDL supplies the XPaths to get specific information needed by the class implementations. This would have important benefits:

- It would enable developers to work directly at the level of XML meanings, without having to develop XML structure-specific code

- If there are several XML languages in the same application domain, it would enable developers to use the same API to all of them – insulating developers from proliferation and version changes of languages

5.4 Interfaces to Relational Databases

Suppose that there existed an equivalent to MDL for relational databases – an XML language which defines how a relational database conveys information about the objects, simple properties and associations of a UML class model. Such a language would probably be simpler than MDL, because it would not need to talk about paths through the database; it only needs to talk about the values of columns in tables. It could also be output fairly easily from a CASE tool such as Rational Rose which can capture information about object-relational mappings. Call this language RMDL.

From the MDL definition of an XML language, and the RMDL definition of the information content of a relational database (both against the same UML class model), it is a simple and largely automatic task to calculate the overlap in meaning between the XML language and the database (again, multiple inheritance makes this process not quite

fully automatic). This could act as a specification for any XML-database mapping software, to use the XML to populate the database or vice versa.

More powerfully, the MDL and RMDL could either generate code, or support a run-time interpreter, to do the XML-Relational mapping automatically. This would have benefits analogous to the benefits of automated XML-to-XML translation:

- Defining the MDL and RMDL would be less labour-intensive than any manual process of building the interfaces, even with only one XML language and one database

- It would provide some measure of confidence that the meaning of the XML was faithfully preserved in and out of the database

- If a company has to work with M different XML languages and N relational databases, the cost of defining all their meanings against a common semantic model grows only as (M+N), whereas the cost of hand-building all the interfaces would grow as M*N. All required interfaces can be created automatically from the definitions.

5.5 Raising the Level of XML

We can summarise the potential impact of MDL as follows:

MDL will enable both applications and users to interface to XML at the level of its meaning, rather than its structure.

Users and application designers need not be concerned with the details of XML structure – with elements, attributes, nesting structure and paths through a document. They can think purely in terms of the meaning of the document (the objects, properties and associations it represents) and leave it to MDL-based tools to deal with document structure. These tools will automatically navigate the XPaths necessary to extract meaning from structure.

Databases used to be based on a Codasyl navigational model, which exposed a pointer-based database structure to users and application developers. To get at information you had to grapple with database structure, following the pointers. Relational Databases and SQL removed this tight structure dependence of data, enabling us to view data in (modestly) structure-independent ways. This was such an advance that it swept the Codasyl database model into history.

In the next few years, we will make similar advances in how we regard XML documents, seeing them in terms of their information content rather than structure. Structure-centred views of XML may become history, just as Codasyl is now history. MDL can be the key tool to develop this meaning-level view of XML.

6. MDL AND THE SEMANTIC WEB

The vision of the Semantic Web is that the information content of web resources should be described in machine-usable terms, so that automatic agents can do useful tasks of finding information, logical inference and negotiating transactions. Therefore work on the Semantic Web has emphasised tools for describing meanings such as RDF Schema and DAML +OIL.

The Resource Description Framework (RDF) was designed to be semantically transparent – so that an automated agent can extract and use information from any RDF document, provided the agent has knowledge of the RDF Schemas used by the RDF. For RDF documents, therefore, access by automated agents is a realisable goal.

However, RDF was designed primarily to represent metadata – information about information resources on the web. This is how RDF tends to be used, so the semantic transparency and automated processing extends only to metadata in RDF. It is widely recognised (e.g Berners-Lee 1999) that XML itself does not have this semantic transparency – precisely because XML can represent meaning in many different ways.

Therefore as it stands, automated agents cannot access the information in (non-RDF) XML documents. They cannot step outside the RDF world to access the information in the bulk of XML documents on the web. This severely limits the ability of automated agents to access the information they need.

MDL can remove the restriction. If the authors of an XML language define its meaning in MDL, then (as described in the previous section) an automated software agent can access the information in any document in the language – greatly extending the power of automated agents.

We can illustrate this by a typical usage scenario for the Semantic Web. I hear from a friend about some Norwegian ski boots, but do not know the name of the manufacturer. I want to buy them over the web. My software agent finds the leading ontologies (RDF Schema based) used to describe WWW retail sites. From these ontologies it learns that Ski boots are a subclass of footwear and of sports gear; that to buy footwear you need to specify a foot size. It then inspects the RDF descriptions (metadata) of several online catalogues. The catalogues themselves are accessible in XML, whose MDL definitions are all referenced to the same RDF Schema. From the RDF, my agent identifies those catalogues which contain information about the kind of goods I want.

The agent then needs to retrieve information of the form ‘footwear from manufacturer based in Norway who makes sports gear’ – applying the same retrieval criteria to several XML-based catalogues, which use different XML languages, and very different representations of the associations [manufacturer]makes[product], [manufacturer]based in[country] and so on. The only automated way to make these retrievals is to know the XPaths needed to retrieve the associations from the different XML languages. The MDL definitions of the languages provide just this information, enabling my software agent to retrieve and compare what it needs from the different catalogues.

Thus the agent uses a two-stage process of (1) access RDF metadata to find out which catalogues are relevant, and (1) using MDL, access the XML catalogues themselves and extract the required information. This two-stage process is much more powerful that the first enabled by RDF on its own.

In summary, realising the Semantic Web will require not only semantics, but also a bridge between semantics and XML structure. MDL provides that bridge.

7. DESCRIBING SEMANTIC MODELS

MDL is used in conjunction with a description of a semantic model of objects, classes, simple properties and associations – which is approximately the information in a UML class diagram, or equivalently in an RDF Schema.

Such a model is conveniently represented in some XML language. While MDL itself does not depend on the choice of this language – except inasmuch as fragments of the semantic model may be optionally embedded in the 'document' element of an MDL document – to build MDL tools, you need to make some choice of language for the semantic model.

There are three main contenders:

- XMI is a standard encoding of UML, supported by UML tool vendors
- RDF Schema
- DAML+OIL is an ontology extension of RDF Schema

The XMI encoding of UML is driven by a meta-model and is highly generic and adaptable. The XML encoding of RDF schema is a much simpler language, and is a close match to the requirements for use with MDL. However, in developing the automated XML translation application of MDL, we have found specific requirements which RDF Schema does not meet.

Languages for defining ontologies have been built as extenstions of RDF Schema – particularly the DARPA Agent Markup Language (DAML) and the Ontology Interchange Language (OIL). These have recently merged into a language DAML+OIL which has many attractive features. Syntactically it is not a major departure from RDF Schema, so it is still a simple language; several of the extensions beyond RDF Schema are well suited for use with MDL; and DAML+OIL has a well-defined model-theoretic semantics. Therefore we propose to use DAML+OIL to describe semantic models for use with MDL.

8. APPENDIX – SCHEMA AND MDL FOR MDL

To be provided in a later draft.

REFERENCES

To be supplied.