

Generating MOF M1-level XMI Document Type Definitions

Gene Mutschler
Unisys Corp

1. Introduction

This paper addresses a question that enterprise developers are increasingly likely to ask: Once a model for some enterprise system has been developed, how can the power of XML be harnessed for the purpose of interchanging data with this system? This is expected to be especially true in the case of the recently released Common Warehouse Metamodel, wherein large quantities of data are to be extracted from multiple databases for processing.

The use of XML requires the definition a Document Type Definition (DTD). The DTD serves as the “schema” for the XML document that is produced when model data is created. The problems of creating a DTD are similar to those of creating a database schema. As the model grows, the complexity of the problem of DTD creation grows faster than linearly, due to the increasing number of interrelationships among the entities in the model.

Developers are increasingly developing their models using tools that implement metamodels, such as the Unified Modeling Language (UML). Metamodels provide the formal framework and discipline to assist them in managing the complexity of developing complex, enterprise-level systems. A mechanism is needed that can work with such tools to perform the task of DTD creation. Otherwise, the fast-cycle development made possible by modeling tools will come to a halt each time a DTD for the model must be created or modified manually.

In the context of the UML, the DTD generation problem is conceptually manageable, given the large number of developers that are familiar with it. Furthermore, UML modeling tools are becoming numerous enough to foresee their inclusion of DTD generation modules in the future if, indeed, they do not already exist. However, as new metamodels, such as CWM, are developed, familiarity with the paradigms of UML might not prove useful, nor automated tools readily available.

Fortunately, it is possible to take advantage of the fact that metamodels can themselves be formalized in a model specification framework. This framework is called the Meta Object Facility (MOF). Using the formalisms MOF provides, it is possible to analyze a model in the context of its metamodel in such a manner as to be able to automate the process of enterprise model DTD development.

The next section of this paper describes the MOF in more detail. It then discusses the XML data transport mechanism developed for use with the MOF, known as XMI. The main body of the paper describes the approaches that can be taken to the problem of generating model-level XMI DTDs. An ad hoc approach is discussed briefly, followed by a discussion of an approach based on the fact that some metamodels have close parallels to the MOF. This is followed by a more general approach that can be used for the models of an arbitrary MOF metamodel. An algorithm for implementing the general MOF-based approach to DTD development is informally described in the final section of the paper.

2. Meta Object Facility (MOF)

The Unified Modeling Language (UML)^{UML} is widely accepted as a standard for expressing models used for object-oriented analysis and design. As a “model” that is used to define other models, UML can be called a *metamodel*. It provides modeling constructs corresponding to those in common use in object-oriented programming languages, such as classes, attributes and operations. UML also provides constructs that allow it to be extended—the Stereotype and TaggedValue. These allow UML to be used for purposes other than those for which it was originally intended. It is possible, for example, to use them to extend

UML to provide support for data modeling tools, such as those that implement the entity-relationship (ER) model.

The extension mechanisms of UML eventually break down when attempting to capture subtleties of design paradigms for which it was not designed. A more general approach is needed, for example, in describing the complicated relationships among data in a data warehouse.

While UML, even with its extensions, cannot express all possible design paradigms, the concepts embodied in UML can be used for the purpose of guiding the creation of other metamodels for various paradigms. In other words, if UML is a metamodel for modeling object-oriented programming systems, one could, by analogy, create the (fictional) “Entity-Relationship Language” (“ERL”) as a metamodel for modeling ER schemas.

The Meta Object Facility (MOF)^{MOF} was created by the Object Management Group (OMG) as a *meta*-metamodel, i.e. a “model” whose instances are metamodels. The MOF is intended to be a framework for developing new metamodels, such as the fictional “ERL”.

Developing metamodels in the context of the MOF has two obvious advantages:

- Metamodelers can re-use constructs from one MOF-compliant metamodel as part of the new metamodel, and
- Developers and tool builders familiar with one metamodel can apply their understanding of this metamodel when encountering a new metamodel to achieve productivity sooner.

The UML, though not originally developed specifically as a MOF metamodel, was developed using the principles of MOF and has subsequently been formally redefined as a MOF-compliant metamodel. The Common Warehouse Metamodel, recently approved by the OMG, is a MOF-compliant metamodel that re-uses the UML while implementing an entirely new metamodel.

The MOF specification categorizes models, metamodels, etc, into 4 layers of abstraction:

- The M3 (meta-metamodel) layer is the MOF itself.
- The M2 (metamodel) layer is the set of metamodels that are instances of the MOF.
- The M1 (model) layer is the set of models that are instances of the various metamodels in the M2 layer.
- The M0 (instance) layer is the set of instances of the various models in the M1 layer.

(As a mnemonic aid, note that the “x” in M_x corresponds to the number of “m”s in “meta-metamodel”, “metamodel”, “model” and “instance”, respectively.)

3. MOF Data Transport

During and after the process of model or metamodel creation, there is often a need to share it. This is especially true in the current era of cooperative development and enterprise computing. The CWM, for example, was developed by as many as a dozen metamodelers in various locations^{Tolbert}. This process necessitated that each use the same version of the same modeling tool in order that they be able to exchange the work in progress. The OMG developed the XML Metadata Interchange (XMI)^{XMI} format as a tool-independent medium for exchanging MOF data.

Though it was originally intended as a means of exchanging MOF metamodels, the XMI format is independent of the level of MOF abstraction at which it is used. In XMI, a Document Type Definition (DTD) is defined at the M_x layer of the MOF abstraction hierarchy, using rules appropriate for that layer. This DTD can then be used to transport data at the M_(x-1) layer. A DTD is defined by the OMG at the M3 layer for the MOF itself, and it is used to transport metamodels. Similarly, a DTD is defined at the M2 layer and included in the specification for UML, beginning with UML 1.3. This DTD is used to transport models for use by such UML-based modeling tools as Rational Rose and Softeam Objectteering. Similarly, there is a DTD at the M2 layer included in the CWM 1.0^{CWM} specification.

Note: Since XML Schema has not yet been incorporated into the XMI specification; this paper confines itself to XMI DTDs only.

The XMI specification defines the rules for producing a DTD at the M2 level. (Since the MOF is defined in terms of itself, these rules also cover the DTD at the M3 level for the MOF.) The specification is, of necessity, silent on rules for producing DTDs at the M1 level, since the production of an M1-level DTD is tied to a metamodel rather than to the MOF.

An enterprise developer, having used a modeling tool to develop a model at the M1 level of the MOF hierarchy, might wish to transport instance data among the various programs running the model. For example, a developer might use CWM to define a schema for the extraction of data from the sales database for use in a sales analysis tool. An obvious choice for such data transport would be XMI. However, since instance data is at the M0 level and therefore requires an M1-level DTD, the XMI specification does not provide the necessary generation rules or other guidance in developing an XMI DTD for such transport of instance data.

The balance of this paper discusses various approaches to creating M1-level XMI DTDs.

4. M1-level DTD Generation Approaches

There are three general approaches to generation of an M1-level DTD, depending on characteristics of the metamodel used to create the M1-level models.

There is an ad hoc approach that ignores the metamodel and MOF entirely. In this approach, the developer simply creates a DTD based on knowledge of the model itself and XML.

For *MOF-analogue* metamodels, i.e. metamodels composed of instances of those MOF artifacts that cause them to resemble the MOF itself, it is possible simply to rewrite the M2-level DTD generation rules defined in the XMI specification as a rule set for the creation of M1-level DTDs. The UML is an example of such a metamodel. The requirements for a MOF-analogue metamodel are given in a later subsection. The MOF-analogue approach is taken by Carlson, for example, in his forthcoming book^{Carlson}.

In the general case, a set of rules derived from the MOF itself can be developed for direct M1-level DTD generation. These rules make few assumptions about the metamodel, other than the relatively obvious requirement that it contain instances of MOF Classes.

The following three sections describe each of these approaches in more detail. The ad hoc approach will be discussed in general terms, since it does not rely on the formalism of the MOF. The MOF-analogue case will be described, also briefly, since it primarily involves rewriting existing work. The general case is described in some detail.

4.1. Ad hoc DTD generation

In the ad-hoc XMI M1-level DTD generation process, the DTD is developed on a per-model basis. This is a process that can be automated, for example as part of a modeling tool. However, it is currently generally necessary to perform this task by hand. This process requires a considerable amount of knowledge of XML and the model, as well as perseverance, since the development and testing process can ultimately consume hours per class in complicated models.

The primary task in the creation of an XMI DTD for a model (or any XML DTD) is defining its XML Element definitions. In the case of a model, it involves finding the containers in the model and mapping them to XML Elements. Once the XML Elements have been denoted, the specification of their content model is a matter of adding the relevant contained objects and xmi.idref values for associated objects. This might or might not be a simple matter, depending on the model, its metamodel and any modeling tools used.

For models that are instances of familiar metamodels, such as UML, ad hoc DTD generation is straightforward, and it can produce a compact and efficient DTD. It is clear which model artifacts are containers, which artifacts are contained and which relate them to each other. For more exotic metamodels, the process is likely to be considerably less straightforward.

However it is created, an ad hoc DTD, being model-specific, has no traceability to MOF, or even to the metamodel of which the model is an instance. Furthermore, when the DTD is hand-generated, there is a high degree of risk that further changes to the model will not be carried forward into the DTD.

4.2. MOF-analogue metamodels

The MOF specification provides entities and combining rules for creating (M2-level) metamodels from the (M3-level) MOF. Similarly, a metamodel specification provides the entities and combining rules for creating (M1-level) models from the (M2-level) metamodel.

The XMI specification includes rules for generating an M2-level DTD from a metamodel specification. There is a need for a set of rules for generating an M1-level DTD from a model specification.

The M2-level DTD generation rules are stated in terms of certain MOF constructs. Clearly, in order to rewrite the DTD generation rules at the next lower (M1) level, analogues for the MOF constructs used in the M2 rules must be present in the metamodel. For this reason, a MOF-analogue metamodel is distinguished by the following characteristics:

- It contains at least one construct that is analogous to the MOF Package.
- It contains at least one construct that is analogous to the MOF Class and its contained Attributes.
- It contains constructs that are analogous to the MOF Association and its contained AssociationEnd.
- It contains constructs that are analogous to the MOF Reference, or its AssociationEnd analogue has Reference-like characteristics.
- It contains a MOF Association instance analogous to the MOF “IsOfType” Association.

Note that an analogous construct in a metamodel is not, in general, an instance of the MOF construct to which is analogous; rather, it simply has the same characteristics that its MOF analogue has.

Once the analogous metamodel elements have been identified, the DTD generation rules can be created by substituting the names of these analogous constructs into the M2-level rules defined in the XMI specification. Once this set of rules has been defined, they can be applied to create M1-level DTDs for any model defined in the context of the metamodel in question.

Since there is only one MOF, the XMI specification need contain only a single algorithm for M2-level DTD generation. On the other hand, there is an indeterminate number of MOF-analogue metamodels that can be defined, meaning that a single algorithm cannot be defined that creates all M1-level DTDs. The rules must, in general, be developed on a per-metamodel basis. This is due to the fact that it is metamodel constructs, not MOF constructs that are being used in the rule specification.

While it is not possible to create a single algorithm for creating all possible M1-level DTDs, it seems possible to create a single template that can be applied to any MOF-analogue metamodel, with the result being an M1-level XMI DTD. In the template approach, the rules are stated in terms of the original M2-level rules, with blanks into which constructs from the target metamodel are substituted to produce the DTD. Such a template algorithm is a good candidate for standardization as part of XMI. Once this has been done, it is possible that a tool can be developed to automate this process.

4.3. General case metamodels

The general case approach to M1-level DTD generation is metamodel independent, though it does require that the metamodel definition be available for reference. It relies on the characteristics of model constructs as determined by the fact that they are instances of metamodel constructs, which are, in turn, instances of MOF constructs.

These relationships are made somewhat clearer by the following definitions.

- A Class Instance (CI) is an instance of a MOF Class. It is an M2-level entity and is part of a metamodel definition. In the UML specification, the Attribute object is a CI.

- An Instance of a Class Instance (ICI) is an instance of a CI. It is an M1-level entity and is part of a model definition in the context of the metamodel containing the CI of which it is an instance. In the example in Figure 1, below, the Person object is an ICI.
- A Composite Reference is a MOF Reference in the metamodel definition that refers, via the exposedEnd link, to a MOF AssociationEnd whose aggregation is composite. In UML, the “ownedElement” Reference held by the Namespace is a Composite Reference.
- A Non-composite Reference is a MOF Reference in the metamodel definition that is not a Composite Reference. In UML, the “type” Reference held by the StructuralFeature is a Non-composite Reference.

A general algorithm for creating M1-level DTDs must or should confine itself to the MOF itself; any assumptions made about a specific metamodel or class of metamodels excludes those metamodels that do not meet the assumptions. If the algorithm is stated only in terms of the above entities, then it is sufficient to show that any of their characteristics that are used in the algorithm are traceable only to the MOF.

A DTD generated by a general algorithm would, by virtue of its MOF compliance, be consistent with other DTDs generated by the same means, offering the possibilities for reuse that often accompany MOF compliance.

The following section informally describes one general purpose M1-level DTD generation algorithm. This algorithm has been implemented in prototype form by the author. In order to keep the algorithm simple enough to describe in a compact fashion, some metamodel-specific assumptions have been made. As a result, some metamodels are necessarily excluded. However, since these assumptions are designed to reflect the thinking that would go into the creation of a realistic metamodel, the class of excluded metamodels is likely to be small.

4.3.1. An M1-level DTD Generation algorithm

The MOF-based M1-level DTD algorithm described in this section makes certain assumptions about the metamodel that defines the model. The first two of these are metamodel-specific.

- It is assumed that there is a “topmost CI” in the hierarchy of MOF classes in the metamodel definition. The topmost CI is a parent class for all CIs. In the UML specification, this is the ModelElement class.
- In order to be able to produce the XML Element tags for the model constructs, it is assumed that first MOF Attribute contained in the topmost CI can be treated as though it were a “naming attribute” of the topmost CI and by extension any CI derived from it. This assumption can be relaxed somewhat if a MOF Tag were to be defined that would designate the naming attribute for a metamodel. In the UML specification, the “name” MOF Attribute of the ModelElement class can be treated as the naming attribute.
- It is assumed that the content model for an ICI whose CI holds no Composite or Non-composite References, the type of whose ExposedEnd is navigable, can be represented as #PCDATA. The reasoning is that, since this ICI has no associated or contained ICIs, it is actual data. (In UML, many of the members of the Data Types package are such CIs.)

A simple example model is shown in Fig 1., below. For convenience, the model was created in Rational Rose and is, of course, a UML model. However, the only UML-specific assumptions made about it are those stated above.

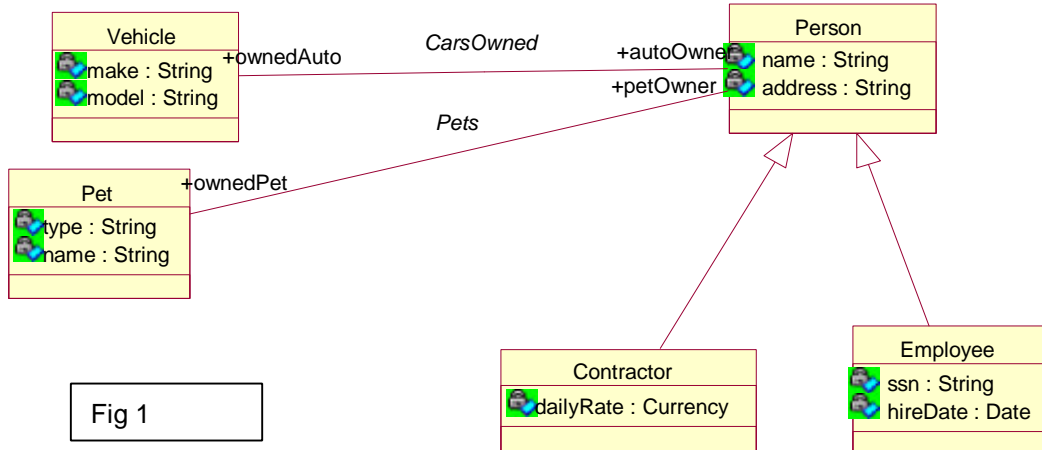


Fig 1

The boxes in the figure are the familiar UML classes. These are obviously ICI elements. Not so obvious, however, is the fact that their attributes and the associations that connect them are also ICI elements, being instances of the MOF CIs “Attribute” and “Association”, respectively, in the definition of the UML metamodel. Further, even the types of the attributes and the association ends (roles) of the associations are ICI elements.

It should be clear that the standard class diagram is not very illustrative of the concepts that need to be grasped to understand the DTD generation algorithm. The algorithm deals with ICI elements, which are instances of CI elements. Therefore, an instance diagram would be more appropriate, since it would show the ICI elements as instances of CI elements. The UML Collaboration diagram can be used for this purpose. Figure 2 is a Collaboration diagram for the example in Figure 1 that shows all of the ICI elements. For readability, a subset of these ICI elements, relating to the Person class, is shown in Figure 2A.

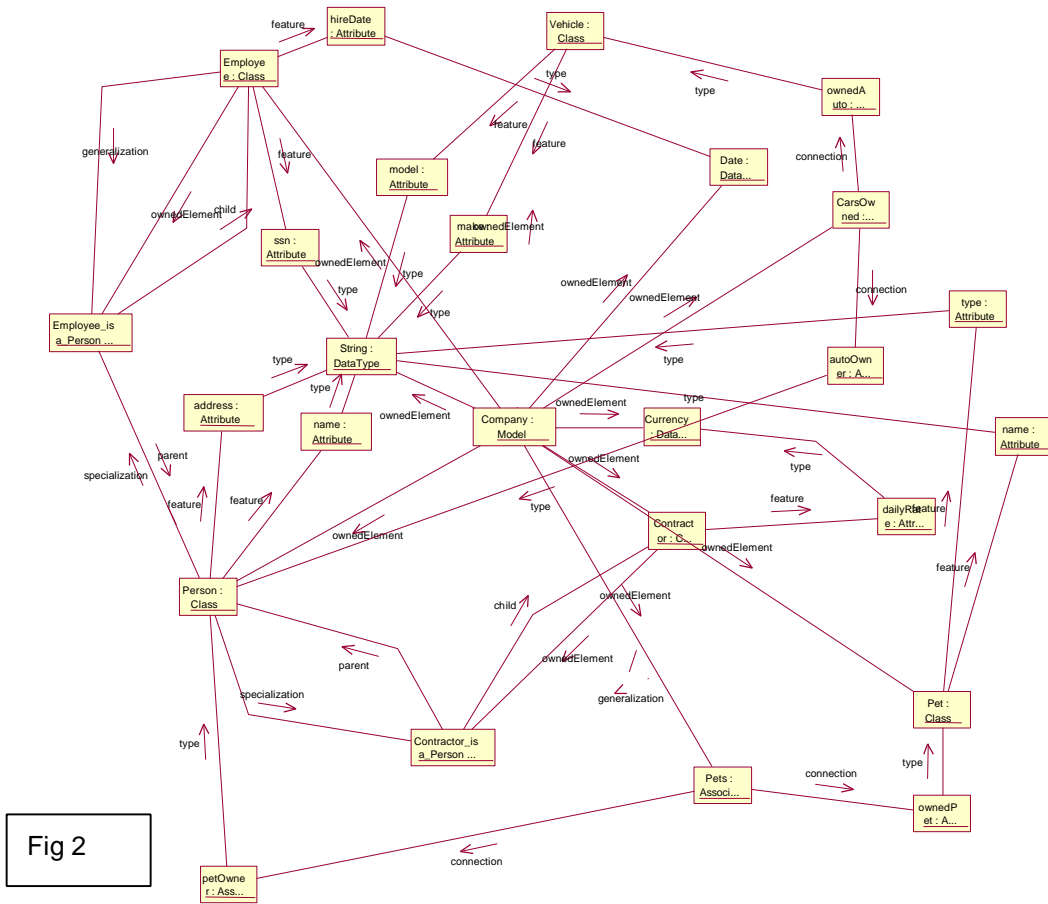


Fig 2

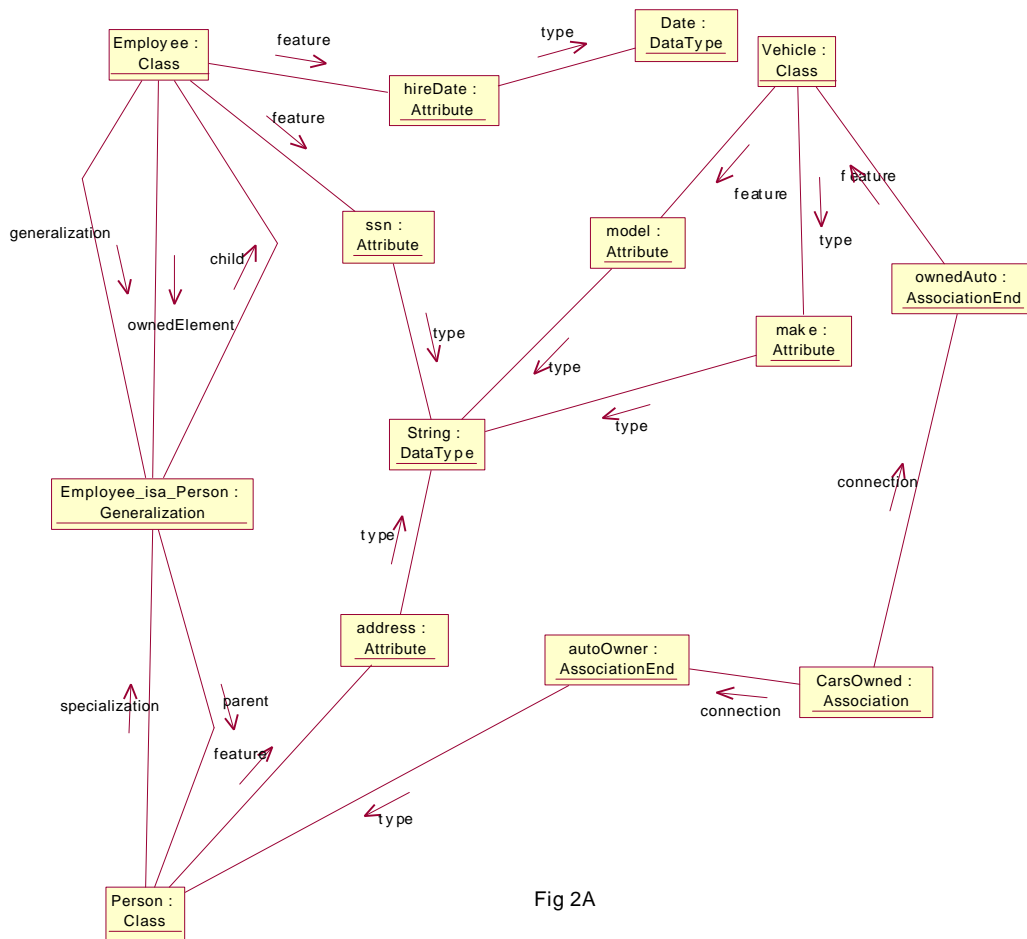


Fig 2A

Figures 2 and 2A give a much better idea of the rules for generating an M1-level DTD. The various ICI items in the definition of the M1-level model are now shown as instances of M2-level CI entities. The links of this instance diagram are instances of the Composite and Non-Composite References that relate the M2-level CI entities in the metamodel definition. Unfortunately, the diagram cannot display the differences between the two types of references. For this reason, the name of the reference is placed on each link as a “message”, with the message direction arrow indicating the direction of traversal of the MOF Association as indicated by that reference.

The algorithm for creating the elements of the M1-level DTD should now be fairly clear. It is essentially the traversal of the links (which are Composite and Non-composite References) between the ICI:CI instances. Note again that, though a UML Collaboration diagram is used, the objects on the diagram can be from any model that is an instance of any metamodel meeting the assumptions stated earlier.

The first step in the algorithm is to locate the outermost ICI(s) in the model. An outermost ICI is one that is not contained via a Composite Reference. In terms of an instance diagram, they are the instances that have only outbound Composite Reference links. In other words, the outermost ICI(s) are the overall containers in the model. Hence, they become the main container elements in the DTD. In Figure 2, the Company:Model item is the only such entity. The algorithm works by starting at these outermost ICI(s) and working inward.

Once the outermost ICI(s) have been located, the DTD generation algorithm would begin filling these containers by working inward. This is done by following the outbound links from the outermost ICI(s) to

other ICIs and so on from there, until each of the links has been traversed once. As each ICI is encountered, it is used as the basis for the definition of an XML Element. Its tag name is the value of the naming attribute of the CI of which the ICI is an instance, as qualified relative to the ICI in which it (ultimately) contained. An example in Figure 2A of a naming attribute value is “Person”. The content model for XML Element for an ICI is then generated from the outbound links of the ICI. Some of these links represent Composite References; the rest represent Non-composite References. They are handled differently, as described below.

If a link is a Composite Reference, then the content model item is qualified formed by the qualified name of the start of the link and the value of the naming attribute of the ICI at the end of the link. This content model item indicates that an ICI is contained directly in the XML Element, as would be implied by a Composite Reference.

An example in Figure 2A of a content model for Person:Class, which has three Composite References would be as shown below:

```
<!ELEMENT Person (Person.name | Person.address | Person.specialization )* />
```

If a link represents a Non-composite References, the content model item is the qualified name of the Reference itself. This form of content model entry indicates that what is contained in the XML Element is not the XML element of the ICI itself, but an XML element that references the ICI at the end of the link.

An example of a content model from Figure 2A for address:Attribute, incorporating a Non-composite Reference, would be as shown below:

```
<!ELEMENT Person.address(Person.address.type)* />
```

The reference element having been named in the content model of the ICI, it is then necessary to generate the reference element itself. The content model of such XML Elements is the qualified name of the ICI that is at the end of the Non-composite Reference’s link. For example, in Figure 2A, the Element for the Person.address.type Non-composite Reference shown above would be:

```
<!ELEMENT Person.address.type(String)* />
```

Eventually, ICIs are reached that have no outbound links. In this case, the content model for the XML Element of the ICI is #PCDATA, indicating that it is a data type, as indicated in the assumptions at the beginning of the section. In the example of Figure 2, the String:DataType ICI has no navigable references, so its element definition is:

```
<!ELEMENT String (#PCDATA) />
```

The Appendix of this paper contains a complete DTD for the model of Figure 2, as produced by the prototype implementation of this algorithm.

4.3.1.1. Comments

This section includes some additional comments about the algorithm. They are in a separate section in order to keep the flow of the algorithm description as smooth as possible.

Composite References are treated differently from Non-composite References. This is an arbitrary decision based on the fact that the composite references typically are used to implement what amount to “attributes” of the metamodel CIs, whereas the non-composite references are used to implement “associations” of the metamodel CIs.

The fact that it is not possible to directly determine which items in the metamodel are data types results in string values being the element content of a ICI naming the data type rather than in the element content of the “class” ICI. This is somewhat space-inefficient. For example a “Person” element have the following actual data:

```
<Person>
  <Person.name>
    <Person.name.type>
      <String>Joe Smith</String>
    </Person.name.type>
  </Person.name>
  etc.
</Person>
```

5. Conclusions

This paper has discussed approaches to the problem of creating XML Document Type Definitions for the purpose of exchanging model instance data. It covered three approaches, in differing levels of detail.

The ad hoc approach can yield compact, efficient DTDs, but, since they lack a formal foundation, they can be difficult to create and maintain and are not likely to be re-usable or to work well with other DTDs.

The MOF Analogue approach makes use of the existing work that has been done generating DTDs for metamodels. For a class of metamodels, the DTDs produced are grounded in a formal foundation and are likely to be compact, efficient and, perhaps, re-usable. However, not all metamodels are MOF Analogues.

The general approach makes few assumptions about the metamodel in question, instead working directly on the MOF characteristics of the elements of the model. Since there is a loss of information in going from the MOF level through the metamodel level to the model level, the DTDs produced are not as compact and efficient as those produced by other means. However, since a single algorithm produces all M1-level DTDs, they are more likely to be both interoperable and re-usable.

6. References

- ^{Carlson} Carlson, David A., *Modeling XML-Based Applications with UML*, Addison Wesley Longman, in preparation.
- ^{CWM} Object Management Group, *Common Warehouse Metamodel*, OMG Document ad/2000-01-03, February 11, 2000.
- ^{MOF} Object Management Group, *Meta Object Facility (MOF) Specification*, version 1.3 RTF, September 1999.
- ^{Tolbert} Tolbert, Doug, Common Warehouse Metamodel development group, private communication.
- ^{UML} Object Management Group, *OMG Unified Modeling Language Specification*, Version 1.3, 1999.
- ^{XMI} Object Management Group, *XML Metadata Interchange (XMI)*, Version 1.1, OMG Document ad/99-10-02, October 25, 1999.

7. Appendix

This DTD corresponds to the model of Figure 1 and was generated by a program implementing the algorithm described in this paper for the general approach to M1-level XMI DTD generation.

Generating MOF M1-level XMI Document Type Definitions

```
<!ELEMENT Company (Person| String| Vehicle| Employee| Date| Contractor|
                  Currency| Pet| CarsOwned| Pets )* />

<!ELEMENT Person (Person.name| Person.address| Person.specialization )* />
<!ELEMENT Person.name(Person.name.type) *>
<!ELEMENT Person.name.type(String)* />
<!ELEMENT Person.address(Person.address.type)* />
<!ELEMENT Person.address.type(String)* />
<!ELEMENT Person.specialization (Employee.Employee_isa_Person |
                                 Contractor.Contractor_isa_Person )* />

<!ELEMENT Vehicle (Vehicle.make| Vehicle.model )* />
<!ELEMENT Vehicle.make(type)* />
<!ELEMENT Vehicle.make.type(String)* />
<!ELEMENT Vehicle.model(type)* />
<!ELEMENT Vehicle.model.type(String)* />

<!ELEMENT Employee (Employee.ssn| Employee.hireDate| Employee.Employee_isa_Person|
                   Employee.generalization )* />
<!ELEMENT Employee.ssn(type)* />
<!ELEMENT Employee.ssn.type(String)* />
<!ELEMENT Employee.hireDate(type)* />
<!ELEMENT Employee.hireDate.type(Date)* />
<!ELEMENT Employee.generalization (Employee.Employee_isa_Person)* />

<!ELEMENT Employee.Employee_isa_Person (Employee.Employee_isa_Person.child |
                                         Employee.Employee_isa_Person.parent )* />
<!ELEMENT Employee.Employee_isa_Person.child (Employee)* />
<!ELEMENT Employee.Employee_isa_Person.parent (Person)* />

<!ELEMENT Contractor (Contractor.dailyRate| Contractor.Contractor_isa_Person|
                     Contractor.generalization )* />
<!ELEMENT Contractor.dailyRate(type)* />
<!ELEMENT Contractor.dailyRate.type(Currency)* />
<!ELEMENT Contractor.generalization (Contractor.Contractor_isa_Person)* />

<!ELEMENT Contractor.Contractor_isa_Person (Contractor.Contractor_isa_Person.child |
                                             Contractor.Contractor_isa_Person.parent )* />
<!ELEMENT Contractor.Contractor_isa_Person.child (Contractor) />
<!ELEMENT Contractor.Contractor_isa_Person.parent (Person) />

<!ELEMENT Pet (Pet.type| Pet.name )* />
<!ELEMENT Pet.type(type)* />
<!ELEMENT Pet.type.type(String)* />
<!ELEMENT Pet.name(type)* />
<!ELEMENT Pet.name.type(String)* />

<!ELEMENT CarsOwned (CarsOwned.autoOwner | CarsOwned.ownedAuto )* />
<!ELEMENT CarsOwned.autoOwner (CarsOwned.autoOwner.type)* />
<!ELEMENT CarsOwned.autoOwner.type (Person) * />
<!ELEMENT CarsOwned.ownedAuto (CarsOwned.ownedAuto.type)* />
<!ELEMENT CarsOwned.ownedAuto.type (Vehicle)* />

<!ELEMENT Pets (Pets.ownedPet | Pets.petOwner )* />
<!ELEMENT Pets.ownedPet (Pets.ownedPet.type)* />
<!ELEMENT Pets.ownedPet.type (Pet)* />
<!ELEMENT Pets.petOwner (Pets.petOwner)* />
<!ELEMENT Pets.petOwner.type (Person)* />

<!ELEMENT String (#PCDATA) />

<!ELEMENT Date (#PCDATA) />

<!ELEMENT Currency (#PCDATA) />
```