Using UML To Define XML Document Types

W. Eliot Kimber
ISOGEN International, A DataChannel Company

Created On: 10 Dec 1999

Last Revised: 14 Jan 2000

Defines a convention for the use of UML to define XML documents. Uses stereotypes to map application-specific types to XML syntactic constructs. Shows how UML can be used in this way to map from abstract information data models to XML-specific implementation models in a natural and easy-to-understand way. Includes a sample program for generating XML DTD-syntax declaration sets from their corresponding UML model.

Table of Contents

1. Introduction

This paper presents an approach to using UML models to define XML document types. This use of UML serves as an alternative to other forms of "XML schema". The XML encoding of data is fundamentally an implementation representation of data that conforms to some higher-level abstract data model or object model. In this way, the use of UML to define the XML implementation of objects is exactly analogous to using UML to define the Java or CORBA or C++ or SQL implementations of those objects. Thus, for the purpose of this paper, there is assumed to be a more abstract data model of which the XML is an implementation, referred to as the "XML implementation representation" of the data.

In this paper, the term "DTD" is used in the "document type definition" sense, that is, the set of rules that governs the use of XML to represent sets of data that conform to a particular type, not in the "document type declaration" sense, that is, the syntactic component of XML documents that provides the document's local syntax rules used by the XML parser when parsing the document.

Using UML for the task of defining XML document types has a number of advantages, including:

- It enables the clear and formal binding of abstract data models to their XML implementation representations
- It provides clear graphical representations of the XML implementation model.
- True modularization of DTDs is provided through UML's normal package mechanism.
- It uses available tools with which most programmers have at least some passing familiarity
- The information modeling and implementation activities can take advantage of UML-based design and modeling methodologies such as the Catalysis method.
- The documentation for the element types can be more effectively bound directly to the model definition, rather than just being either comments in the declaration set or a completely separate document that is not tightly coupled to the formal DTD definition. Using XMI for the storage of the UML models, arbitrarily sophisticated XML structures can be used within the XMI data set to model the documentation.

The basic approach is fairly straight forward. Each XML syntactic construct is represented by a stereotype that is applied to the UML construct used to define the XML syntactic construct. For example, UML types that represent element types have the stereotype "<<element>>". These stereotypes provide the information needed to accurately and consistently render the UML model into whatever DTD syntax is desired

(DTD declarations, XML Schema, XDR, etc.). A typical "DTD model" is shown in Figure 1.

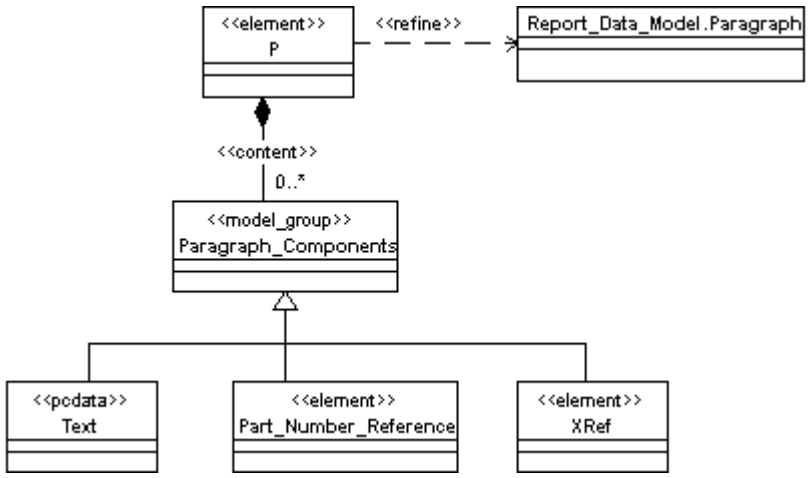

Figure 1 — Typical UML Element Type Definition

This paper first presents the definitions of the stereotypes, formally defined through a UML model in which the stereotypes are types, then demonstrates how those stereotypes are used using a simple demonstration DTD.

This paper assumes that you are familiar with both UML and XML syntax and concepts.

Note that as presented in this paper all the models are data models, not object models, meaning that they define static types. However, the data models could be further refined into object models that provide methods for the types. This paper does not address this aspect of using UML to model documents and document types.

2. Stereotypes for XML Document Constructs

The primary challenge in using something like UML to directly model XML document types is knowing how application-specific types are to be mapped to XML syntactic constructs: elements, attributes, notations, character data, etc. Another challenge is the mechanism by which detailed syntactic constraints are specified. For example, UML provides no obvious or direct way to model XML content models. These constraints must be specified through some sort of constraint language.

Content model constraints are specified through a combination of types with the stereotype "<<model-group>>" and content constraint specifications using normal XML model group syntax. Model-group types capture groups of related types that can satisfy a particular point in a content model while content constraint specifications further constrain occurrences of instances of those types within the "content" property of their container. Model-group types are roughly analogous to parameter entities used to parameterize content models in DTD declarations but can have a clear semantic because they are true types, not simple string macros[1]

An XML DTD defines two classes of thing: elements and notations. Notations have two required properties: a local name and a "public" (persistent) name. Elements have a local type name, a "public" (persistent) name, a possible unique identifier, possible content, and a set of zero or more attributes unique to the element type. In this approach, attributes are modeled either as simple type attributes or as relations between element types and types that represent the attribute value. Attributes that are semantically references are a special case, represented by the relation stereotype "<<reference>>". Because the effective value of an attribute may be specified indirectly through other attributes that are references, it is possible to relate one attribute to other attributes that address its effective value to create a "value reference".

The types for stereotypes refer to aspects of the metamodel for XML. This paper defines UML models of the necessary types. While not normative, these models are an accurate reflection of both the XML information set and SGML property set (published in ISO/IEC 10744:1997). It also includes concepts that are not syntactic in XML or SGML but that are common to most (if not all) XML applications and that are codified in the HyTime architecture (referential attributes, value references, and link types.

Figure 2 provides an overview of the UML packages defined in this document and their relationships to each other. The dependency relationship "<<refinement>>" means "refinement" as defined for the Catalysis method, that is, the types in the refining package refine types in the refined package. Refinement defines relationships between models at two different levels of abstraction.

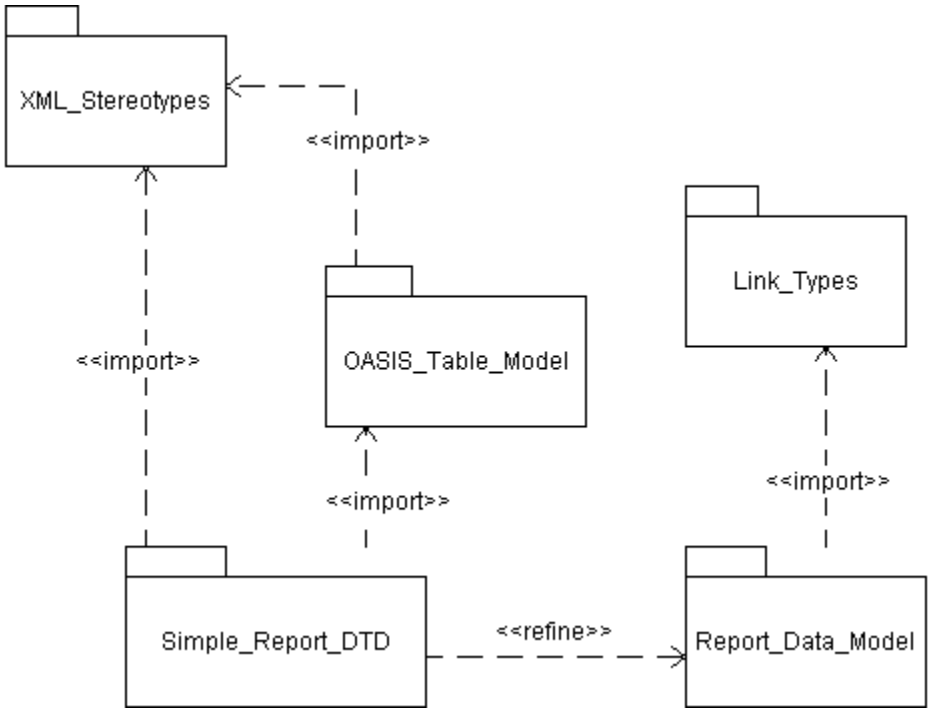

Figure 2 — Package Overview

2.1. XML Document Data Model

These types reflect the abstract data model for XML. They are defined in the XML stereotypes package. Figure 3 shows the formal model of an XML document. It's primary purpose is to establish some basic facts about XML documents, in particular, that every XML document establishes two name spaces (in the generic computer science sense), one for elements with unique IDs and one for data content notations. It also establishes the fact that a document always has exactly one root element[2] It also establishes the fact that every XML document has persistent identity (because an XML document is, by definition, represented by the storage object that contains the prolog and root element). The persistent name for the document is the storage object identifier for the document entity (whatever form that identifier might take).

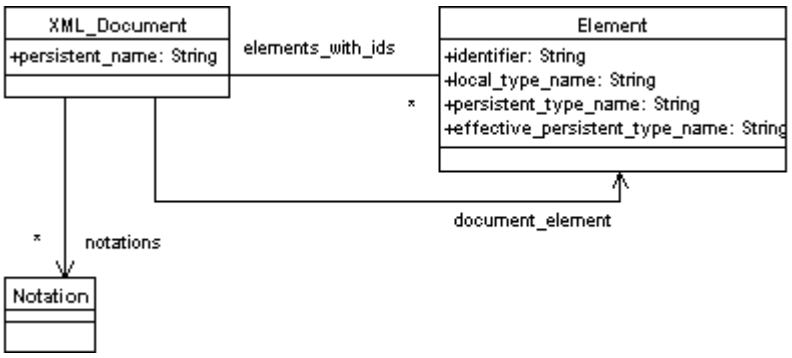

Figure 3 — Abstract Types for XML Documents

2.2. Definition of the XML Stereotypes

This section formally defines the types that are used as stereotypes in models that define application-specific DTDs. In order to avoid confusion, the types shown in this section are referred to generically as "stereotypes" as their function is to define stereotype names. These models also serve to define patterns or templates that the real types that use these stereotypes must conform to.

2.2.1. Element Content

Figure 4 shows the element stereotype and its relationship to the element's content. [An XML pedant might protest that the name for this stereotype should be "element_type", as that is what an ELEMENT declaration in a DTD defines. However, the modifier "type" is redundant in this context because the UML construct that the stereotype modifies is a type. Thus the stereotype name "element" applied to a type symbol produces an "element type", that is, a type that defines a type of element.]

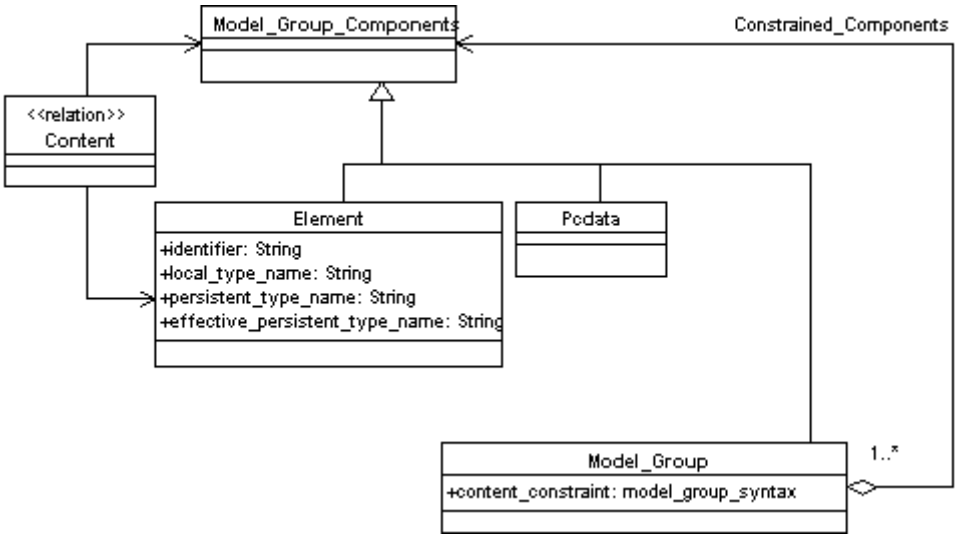

Figure 4 — Element Content

An element type may have a UML attribute whose stereotype is "<<content>>" that relates the element to those types that are allowed in its syntactic content. The content attribute is always represented by an aggregation relationship, reflecting the containment or ownership aspect of XML syntactic containment. The value of the content property can be any combinations of the types derived from Model_Group_Components, that is other elements, parsed character data, or subordinate model groups.

The stereotype "Pcdata" represents #PCDATA tokens in normal XML content models. In normal XML usage, only one Pcdata type will be used in a content model. However, other XML-like contexts may allow more flexibility with regard to how Pcdata types are used.[3] Pcdata types have no properties apart from their application-specific type name (which may be PCDATA, although it can reflect the actual semantic use of the character data

within a particular element content, e.g. the "part number" in a part number reference element).

The stereotype "Model_Group" represents a choice among a set of possible subcomponents within the content of an element. A model group has two properties: the set of element types, PCDATA, and model groups it constrains and, optionally, a constraint specification. By default, a model group is an OR group whose cardinality is defined by the cardinality of the of the content relation that contains it. If additional constraints are required, a constraint specification can be specified as a constraint named "content_constraint" on the model group type. The syntax of the constraint specification should be a normal XML DTD-syntax model group without the outer containing parens. For example, a model group that is a sequence of four subtypes, A, B, C, and D, where C is optional and D is required and repeatable, would have a content constraint of "A, B, C?, D+".

Types with a stereotype of "<<model_group>>" are always non-terminals in the sense that they are never directly reflected in either generated DTD-syntax declaration sets (except possibly as parameter entities) or in documents that conform to the DTDs.

The class names of element types serve as their element type names, which must be unique across all the element types in the DTD. This uniqueness constraint is more restrictive than UML's naming constraints, which only require type names to be unique within a package. If types are imported into a DTD package from another package, name clashes can be disambiguated by using UML's normal name mapping syntax.

An element type may also have a persistent, globally-unique name (i.e., it's "name space" name). If an element does not specify an explicit persistent name, it's effective persistent name is the combination of the persistent name (storage location or URN) for the document that contains it plus its local type name. Note that the persistent type name says nothing directly about what set of names the persistent name might be a member of, if any--it is simply a globally unique identifier for the element type. The binding of persistent element type names to defining vocabularies is outside the scope of this paper.

2.2.2. Element and Attributes

Figure 5 shows the element stereotype and its relationship to stereotypes used in the characterization of attributes for elements.

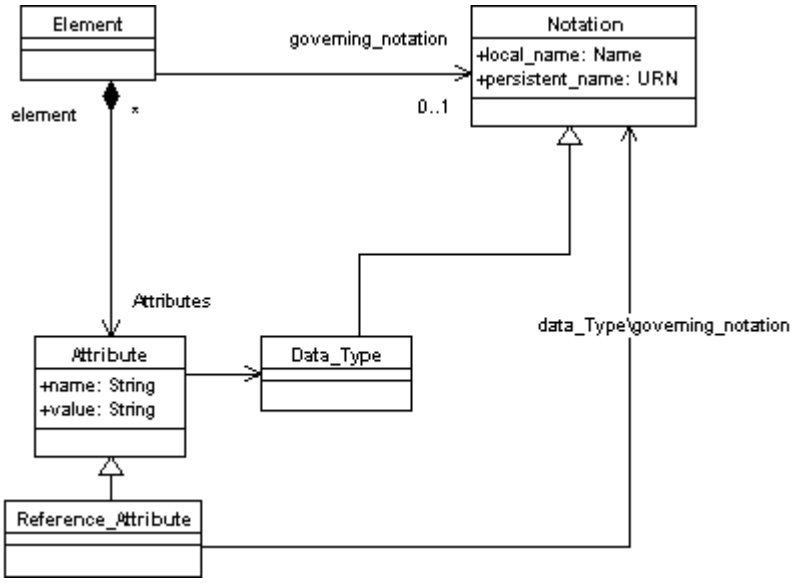

Figure 5 — Element and Attributes

An element may have an attribute whose stereotype is "<<identifier>>", indicating that it is unique within the document's "elements with IDs" name space (i.e., an attribute of type "ID"). As this attribute will always be a string (by the rules of XML), it can always be represented as a simple attribute of element types.

Attributes may also be represented by relations between the element type and types that define the allowed values of the attributes ("value types"). Relations that are attributes must have a stereotype of "<<attribute>>" and must be navigable from the element to the value type. The cardinality of the relationship defines the cardinality of the attribute value. Further constraints on the attribute value can be stated as constraints on the "attribute" role of the relationship or through the use of subtypes to define the value types. This paper does not define an attribute value constraint syntax.

The data type types are subclasses of the general notation stereotype. However, for the built-in XML-defined types the notation instance need not be declared.

A subclass of attributes are semantically referential (e.g., "href", attributes with a declared value prescription of "idref", HyTime-defined referential attributes, etc.). These attributes are identified by the stereotype "<<reference>>". The values of referential attributes are always governed by the notation that defines the semantics of the reference pointer itself.

In many cases the purpose of a referential attribute is to address the effective value of another attribute of the element (or its semantic content). For these attributes, the referential attribute is related to the referencing attribute by a "<<value_reference>>" relation.

2.2.3. Value Reference Attributes

A common semantic for referential attributes is to address the effective value of some property of the element. For example, the logical content of an element might be by reference to something else, such as a record in a database or a reference entry in a reference manual (such as mentions of commands or functions in technical documentation). In this semantic, the value of a property of the element is defined "by reference".

The referential attribute points to the thing that serves as the effective value of the property. To formally define this semantic in a document type the referential attribute must be bound to the property whose effective value is being referenced. This relation is defined as a "value reference", reflected through the relation type "Value_Reference", which is used as a template for relations between attributes and the things they get the value for. The referential attribute is given the stereotype "Value_Reference_Attribute", which is a specialization of the Reference_Attribute stereotype. These types are shown in Figure 6.

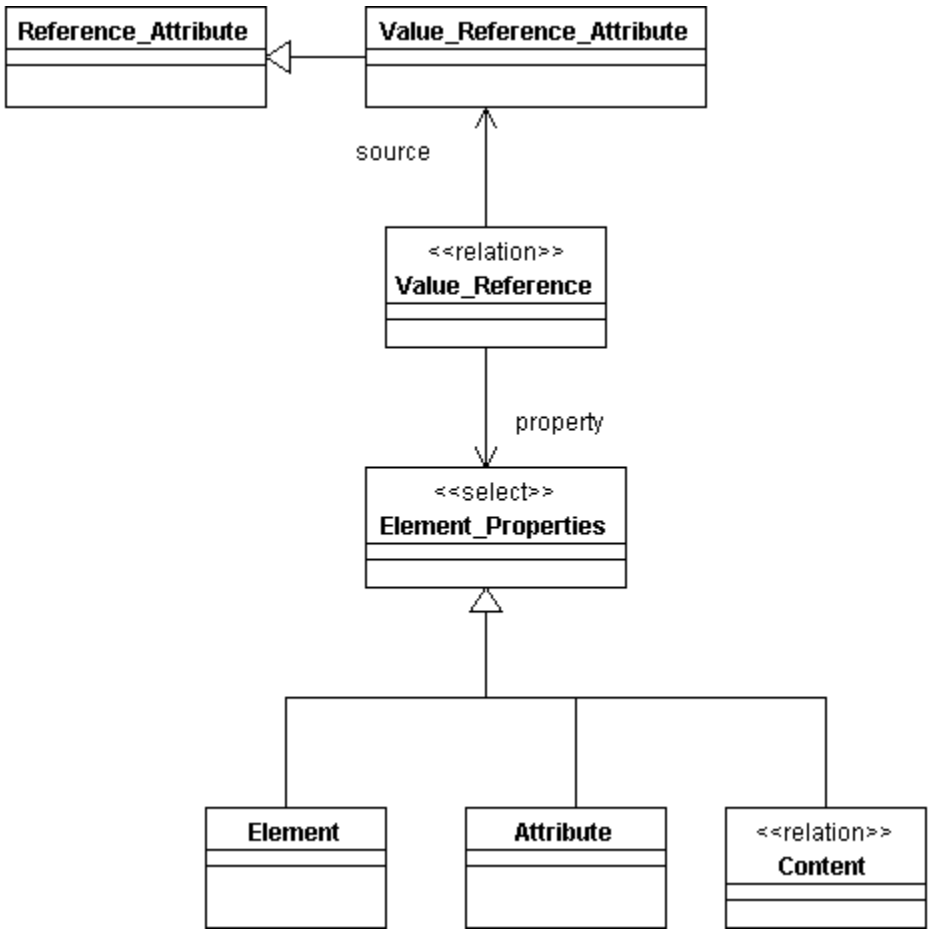

Figure 6 — Value Reference Stereotypes

A value reference can define the effective value of the content of an element, the effective value of any of its attributes (including the attribute that makes the reference), or of the element itself. Getting the effective content by reference is typically used for elements that are mentions or references, where the content can be derived from some property of the thing referenced (such as the title of a section or a field in a database). Getting the effective value of an entire element creates a "redirection" from the original element to the element pointed so, so that the target element is treated semantically (but not syntactically) by the processing application as though it had occurred at the point of reference. This is roughly analogous to the "show=embed" option for XLink, but is more precisely defined. Note that in all three cases, the resolution of the values is done after all the documents involved are parsed, that is, at the DOM or grove level, not at the syntactic level. For that reason, it is meaningful for the effective value of an element to be a database field, for example, because at the level of semantic processing, syntactic differences in how the data are stored are irrelevant.

This semantic is formally defined in the HyTime standard, which defines an attribute-based syntax for defining the relationships described by this template. The advantage of the template shown here is that it is independent of any implementation syntax, so you can map the concepts, which are more or less universal, into whatever document syntax the target processor will understand (if any). In any case, the intent of the DTD design will be clear to users and implementors.

2.2.4. Data Content Notations

Figure 7 shows the model for data content notations. A data content notation is nothing more than a binding between the definition of some data type, query syntax, or other application-specific thing and a persistent name for that thing that can be dependably used to both get to the documentation and as a key in lookup tables for associating processors with the notation. For example, a notation for a particular graphics format can be associated with a renderer for that notation or a notation for a particular addressing syntax can be associated with a function package that implements the syntax in the context of a larger processing system (e.g., a particular XSLT engine or HyTime system).

The document that a notation relates to is any form of documentation that might be available for that notation, from an international standard to a private specification to a couple of lines of comment in the model itself. The purpose of the documentation is to be an aid to humans who have to understand how to use the notation in information sets or implement support for it in systems.

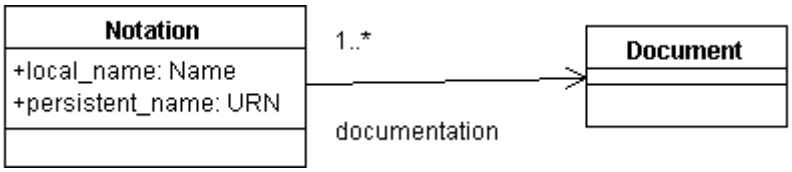

Figure 7 — Data Content Notation Type

3. Sample DTD Defined using UML

This section presents a sample DTD defined using UML and the stereotypes for XML. It is a trivial "report" DTD. It demonstrates the use of all of the stereotypes.

Like all DTDs, the report DTD is an implementation of a more abstract data model, in this example, the "Report Data Model", which would normally be the result of an information ("document") analysis. The model for this example is trivial but serves to demonstrate the modeling of formal relationships between the abstract model and implementation DTD model. The report data model imports a set of types that define relationship (link) types, reflecting the typical case of having a set of generic link types that are used by a variety of specialized information models.

The report DTD is formally a refinement of the report data model. Because the report data model requires some form of table (but does not specify their details), the report DTD is free to use any table model it wants. In this example, it imports the OASIS table model. This import is a formal semantic import described using UML's package import semantic, not the syntactic include of an external parameter entity reference.

3.1. Report Data Model Package

Figure 8 shows the abstract (analysis-level) model for simple reports. It defines the basic rules and structures for reports without saying anything about how reports might be represented syntactically. In particular, there is nothing about this model that says anything about the XML representation. For example, this model could just as usefully be applied to Framemaker templates or Java objects as to XML documents.

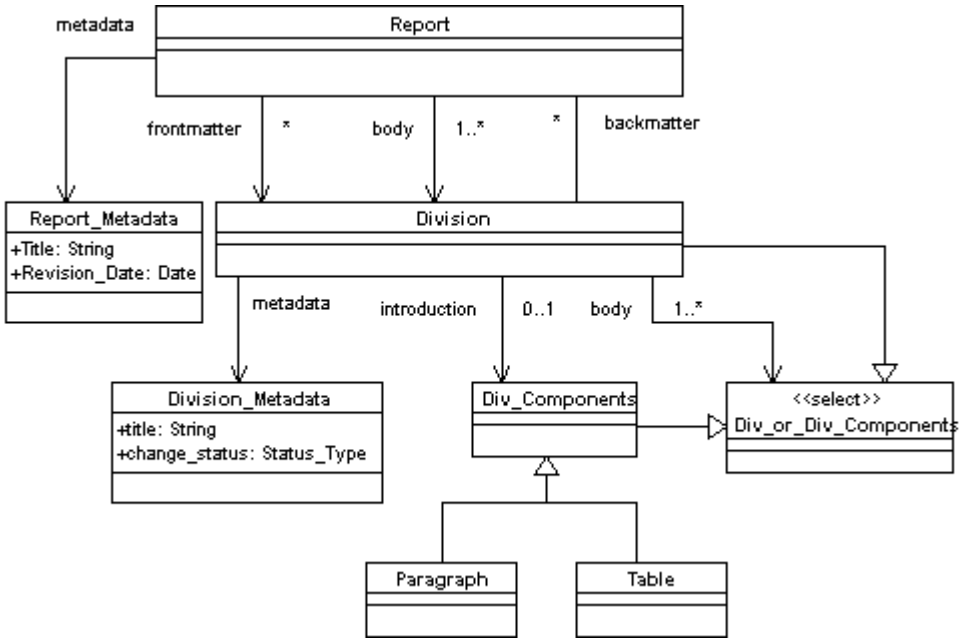

Figure 8 — Abstract Report Data Model

As defined in this model, a report has some metadata represented as a class that then holds all the metadata attributes, then three possible sections: frontmatter, body, and backmatter. In this model, these sections are represented as relations to a single type, division. A division has three components: its metadata, modeled using the same pattern as for report; an optional introduction, which can only consist of paragraphs and tables; and a body, which can consist of divisions or division components. The stereotype "<<select>>" represents a discriminated union in which the relationship end must be satisfied by one of the subtypes of the type div_or_div_components. [The name "select" is taken from the EXPRESS modeling language, defined in ISO 10303.]

At this level of analysis, nothing is said about the details of paragraphs and tables, meaning that those details are left up to implementations of this model. This is appropriate because the purpose of this analysis model is to establish general rules for reports.

3.2. Link Types Package

The link types package defines a set of general relationship types that are intended be used consistently in different documents. The definitions of useful link types would normally come out of a general document analysis. In this very simple example, the package defines a single link type representing cross references, as shown in Figure 9. The type "Cross Reference" defines a simple relationship of two ends, the reference "mark" and the reference subject. Both ends relate something called a "Resource", which in this example means "a thing you can point to" in the URI sense of "resource". This is a very simple, very generic model. It doesn't say anything about how this relationship type might represented in instances.

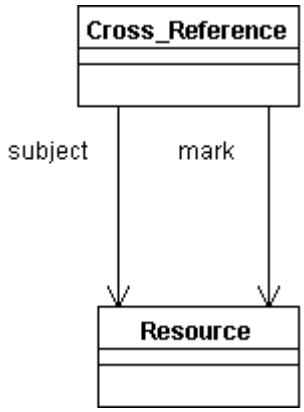

Figure 9 — Cross Reference Link Type

A package of such link types would represent a codification of the types of relationships that the analysts considered to be useful or allowed within a particular information management application or subject domain. They would often represent an independent analysis effort that is reusable across many other applications.

3.3. Oasis Table Model

The report analysis model says that divisions can contain tables but doesn't say anything about the details of tables. That leaves the choice of table model open to implementations of the analysis model. This model describes the table model defined by the OASIS consortium. This model is an example of a DTD "module" that is intended to be re-used in many different application DTDs. XML syntax doesn't provide a good way to do this because inclusion of declaration sets is simple syntactic inclusion--it doesn't provide any good facilities for name mapping, parameterization of content models, and so forth. In UML-based DTDs, the normal package import facilities in UML provides most of the modularization features needed to do true DTD modularization. More complete modularization is provided by things like Catalysis frameworks, which define additional semantics for modularization and use of modules. This package is formally imported by the report DTD implementation model.

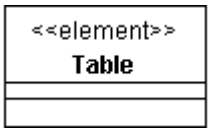

Figure 10 — The Oasis Table Model

[Editorial Note: Need to fill in the rest of this model, at least a minimal subset of it.]

3.4. Simple Report DTD

This model is an implementation model for the more general report data model, which might have any number of useful XML representations for different use environments.

This model is formally defined as a refinement of the report data model. It imports the Oasis table model package. It also imports the XML stereotypes package in order to define the stereotypes it uses.

[Editorial Note: It's not 100% clear that "import" is the correct dependency relationship for stereotype definition--it might be more correct to say that the stereotypes used in the DTD model are refinements of the types in the XML Stereotypes package.]

This section is organized around the class diagrams used to define the DTD model. The class diagrams are not the whole model. There is also documentation and constraints in the model that are not reflected in the diagrams but that are part of the model as authored in the modeling tool used to create the model initially. The organization of the class diagrams is somewhat arbitrary, largely driven by useful groupings of element types and what will fit comfortably in a single picture. A diagram showing the entire model is provided at the end of this section.

3.4.1. Top-Level Report Structures

Figure 11 defines the top-level structures for the simple report DTD. The type "Report" has the stereotype "<<element>>" indicating that it is defining an XML element type. It has a relation with a stereotype of "<<content>>", which indicates that the relation is defining the content of the Report element type. The target role of the content relation is a "<<model-group>>" type that defines the content rules for the report element type.

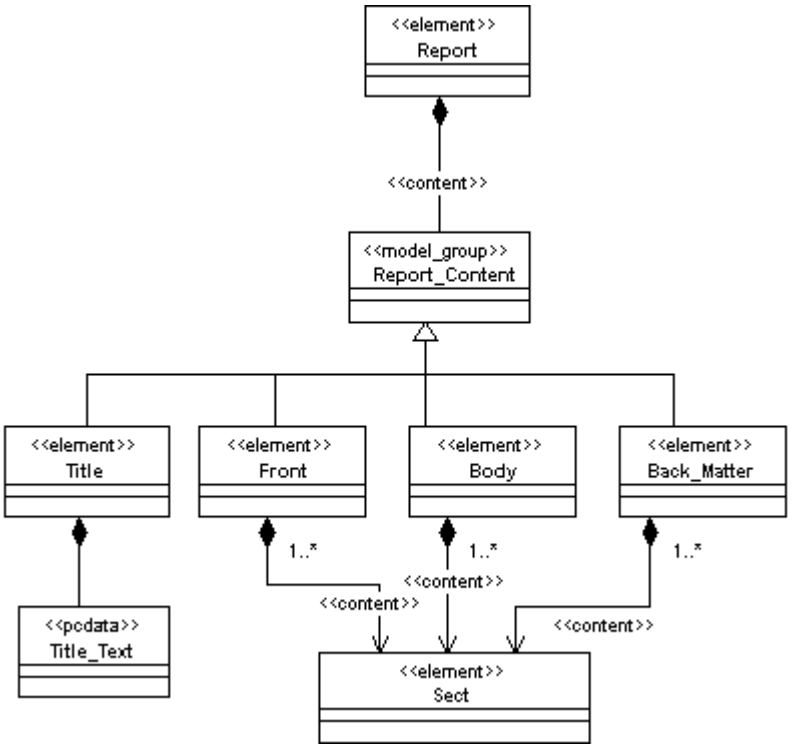

Figure 11 — Top-Level Element Types

The model group type plays essentially the same role as a parameter entity in an XML declaration set, but it has some key differences. The first difference is that it is a first-class object, so it can have unique properties distinct from its use contexts. The content tokens of the model group are defined by having classes that are a subtype of the model group class (that is, the model group represents a discriminated union of its subtypes, which is what a model group means). By application of the rule from the XML elements template model, by default a model group defines an OR group of its subtypes, with the cardinality of the group defined by the cardinality of the "<<content>>" relationship that uses the model group. For the `report` element type, the model group needs to be defined as a sequence. There is no way to do this graphically in UML, so the model group must have a constraint named "content_constraint" whose value is a normal content model group without the enclosing outer parentheses, i.e., "Title, Front, Body, Back_Matter".

The Title type has a content property that is a "<<pcdata>>" type. Types with a stereotype of "<<pcdata>>" represent #PCDATA tokens in content models. Because the token is represented by a type, the type can have a descriptive name and associated documentation, constraint specifications, and so on. In this case, the type name "Title_Text" just serves to more explicitly describe what the purpose of the content of the title element type is. This degree of explicitness is not required but it provides the opportunity to be more descriptive than DTD syntax provides.

The other types represent the major organizational sections of the report. They all have content properties that require at least one Sect element. The Sect element is a refinement of Division type within the report data model.

The effective DTD declarations represented by this diagram are:

```
<!ELEMENT report
  (Title,
   Front?,
   Body,
   Back_Matter?)
>

<!ELEMENT Title
  (#PCDATA)
>

<!ELEMENT Front
  (Sect+)
>

<!ELEMENT Body
  (Sect+)
>
<!ELEMENT Back_Matter
  (Sect+)
>
```

3.4.2. Section Model

Figure 12 defines the structure for sections. The Intro_and_Subsects model group has a content constraint with the value "intro?, sect+". The Sect_Content model group doesn't have an explicit content constraint because the default implementation is a non-repeating OR group between the Intro_and_Subsects model group and the Sect_Body element type.

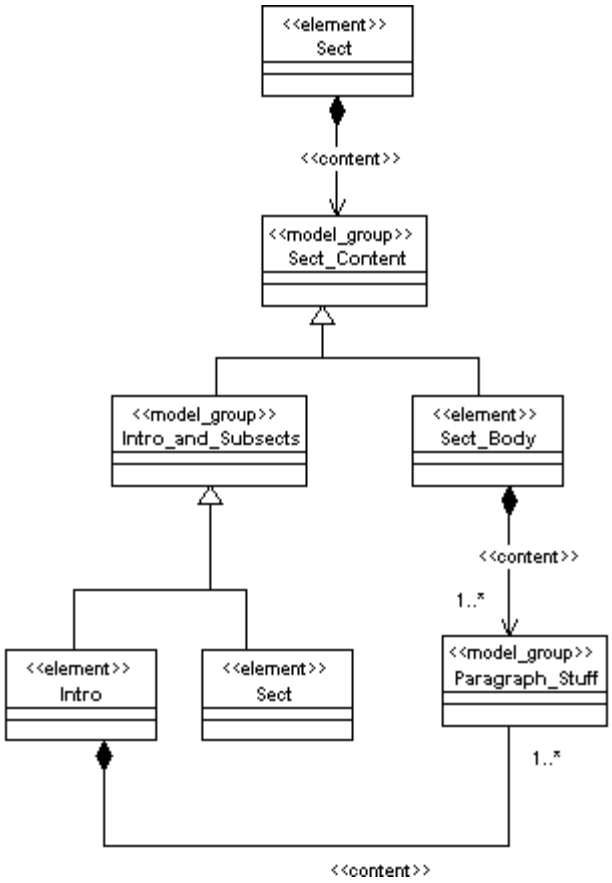

Figure 12 — Section Element Model

The effective DTD declarations represented by this diagram are:

```
<!ELEMENT Sect
  ((Intro?,
    Sect+) |
   Sect_Body)
>

<!ELEMENT Sect_Body
  (%Paragraph_Stuff;)+
>
<!ELEMENT Intro
  (%Paragraph_Stuff;)+
>
```

3.4.3. Paragraph-Level Elements Model

Figure 13 defines the rules for paragraph-level elements, that is, elements that can occur inside sections. Note that the table part of the model (the requirement for which is originally defined in the report data model) is a type imported from the OASIS table model package.

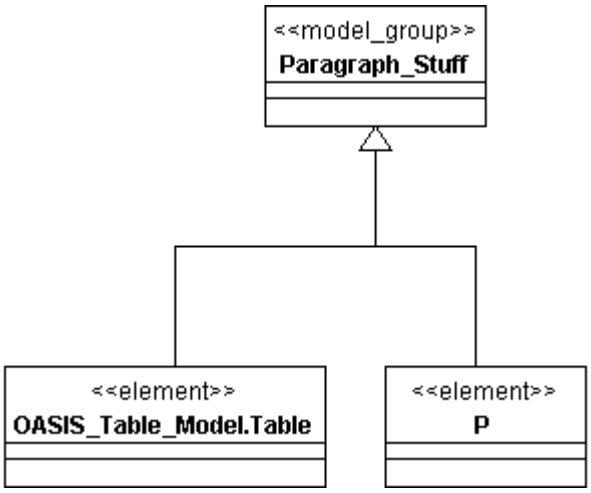

Figure 13 — Paragraph-Level Elements

The effective DTD declarations represented by this diagram are:

```
<!ENTITY % Paragraph_Stuff
  "P |
   Table"
>
<!-- P element declaration shown in next section -->
<!ELEMENT Table
  (row+)
>
<!-- Add rest of table element declarations here -->
```

3.4.4. Paragraph Model

Figure 14 defines the rules for paragraph content. Note the dependency relationship between the P type and the Paragraph type from the report data model. The stereotype "<<refine>>" indicates that the dependency is a refinement dependency. This provides a formal and trackable relationship between the P type and the thing it implements. (The full set of refinements is given in 3.4.7. *Refinements From Simple Report to Report Model*, page 20.)

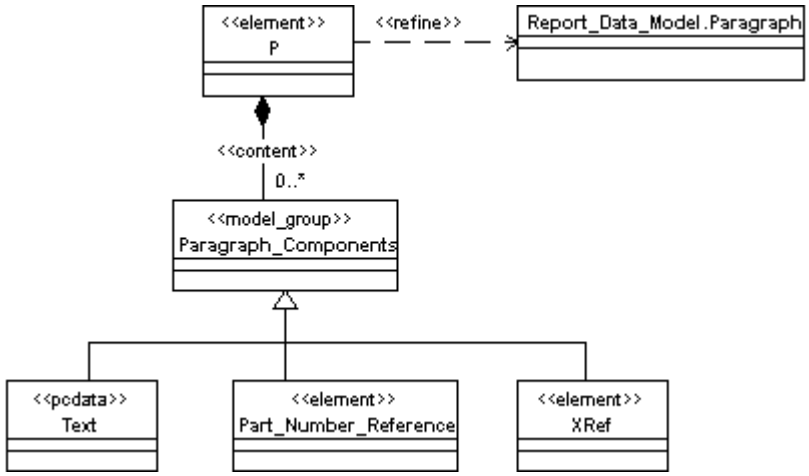

Figure 14 — Paragraph Element Model

The effective DTD declarations represented by this diagram are:

```
<!ELEMENT P
  (#PCDATA |
   Part_Number_Reference |
   XRef)*
>
```

3.4.5. Cross Reference Element Model

The element type `XRef` implements the link type "Cross_Reference" defined in the link types package. The abstract design for the link type allows a number of implementation choices. In this case, the implementation makes the cross reference an inline link where the link element itself plays the "reference mark" role and the reference subject is pointed to using a URI. The implementation model shown in Figure 15 defines the element type `XRef` and the attribute `href`, with a relation between them indicating that the element and the attribute establish the "Cross_Reference" relation. The relation shows its refinement dependency on the Cross_Reference type from the link types package. The attribute type `href` has the stereotype "<<reference>>" indicating that this attribute is semantically a reference. All referential attributes must be related to the notation or set of notations that defines what form or forms of address the attribute uses. In this model, the `href` attribute is bound to the notation "uri", formally defining the normal expectation for an attribute named "href".

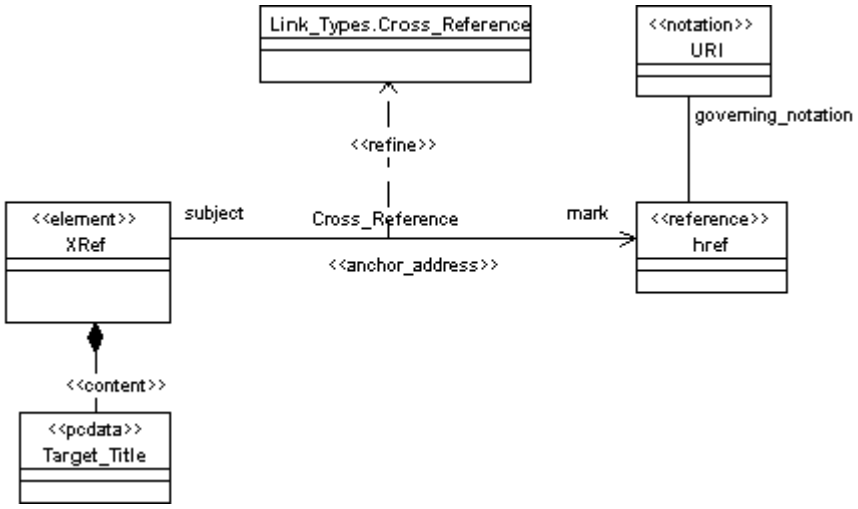

Figure 15 — Cross Reference Element Model

Notice that at the DTD level, the ends of the link are represented by the syntactic constructs that either serve as an end (the `XRef` element) or act as a "proxy" for an by pointing to it (the `href` attribute).

The effective DTD declarations represented by this diagram are shown below. They use the HyTime syntax for expressing the constraints shown in the model.

```
<!NOTATION uri SYSTEM "www.w3.org/Recs/URI.xml"
>
<!ELEMENT XRef
  (#PCDATA)
>
<!ATTLIST XRef
  href
    CDATA
    #REQUIRED
  linktype
    CDATA
    #FIXED "Cross_Reference"
  anchrole
    CDATA
    #FIXED "mark subject"
  anchcstr
    CDATA
    #FIXED "self required"
  valueref
    CDATA
    #FIXED "#CONTENT href"
  loctype
    CDATA
    #FIXED "href queryloc uri"
  HyTime
    NAME
    #FIXED "hylink"
>
```

3.4.6. Value Reference Attribute Model

The element type `Part_Number_Reference` is used to create mentions of part numbers where the part number itself is retrieved from a parts database. Logically, the part number value is the content of the element but the intent of this design is that the part number text never exists in the document source but is always retrieved as needed. Because the part number is used by reference, there must be an attribute that makes the reference and there must be a formal relationship between this attribute and the content property of the `Part_Number_Reference` element. Figure 16 shows this model.

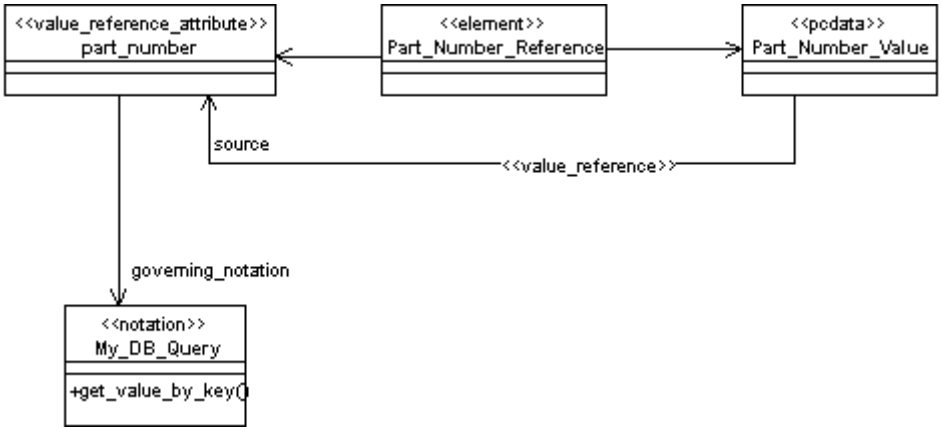

Figure 16 — Referential Attribute Model

The model shows that the `Part_Number_Reference` element has a PCDATA content property that is, semantically, the part number value. The `part_number` attribute has the stereotype "<<value_reference_attribute>>", indicating that it is a referential attribute whose purpose is to address some property of the element. The relation between the `part_number` attribute and the Part_Number_Value type has the stereotype "<<value_reference>>", which formally states the relationship between the value reference attribute and the content property of the element. In addition, the `part_number` attribute, as a referential attribute, is bound to the notation that governs its interpretation as an address. In this case, the notation represents some purpose-built database query that gets part numbers from a parts database.

The effective DTD declarations represented by this diagram are shown below. They use the HyTime syntax for expressing the constraints defined in the model.

```
<!ELEMENT Part_Number_Reference
  EMPTY
>

<!NOTATION My_DB_Query
  SYSTEM "my database query"
>

<!ATTLIST Part_Number_Reference
  part_number
    CDATA
    #REQUIRED
```

```
  valueref
    CDATA
    #FIXED "#CONTENT part_number"
  loctype
    CDATA
    #FIXED "part_number QUERYLOC My_DB_Query"
  HyTime
    CDATA
    #FIXED "HyBrid"
>
```

3.4.7. Refinements From Simple Report to Report Model

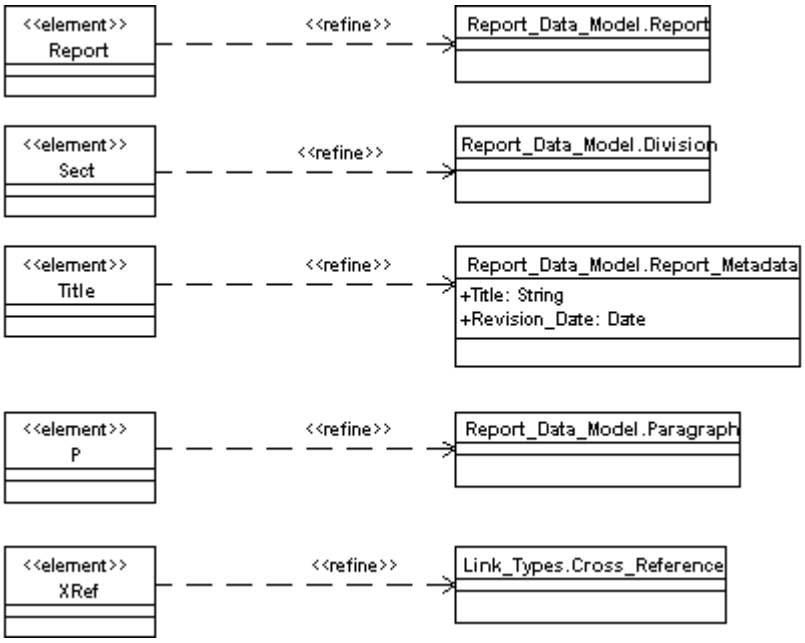

Figure 17 — Refinement Relationships for Simple Report DTD

3.5. Diagram of Complete DTD Model

Sample DTD Defined using UML

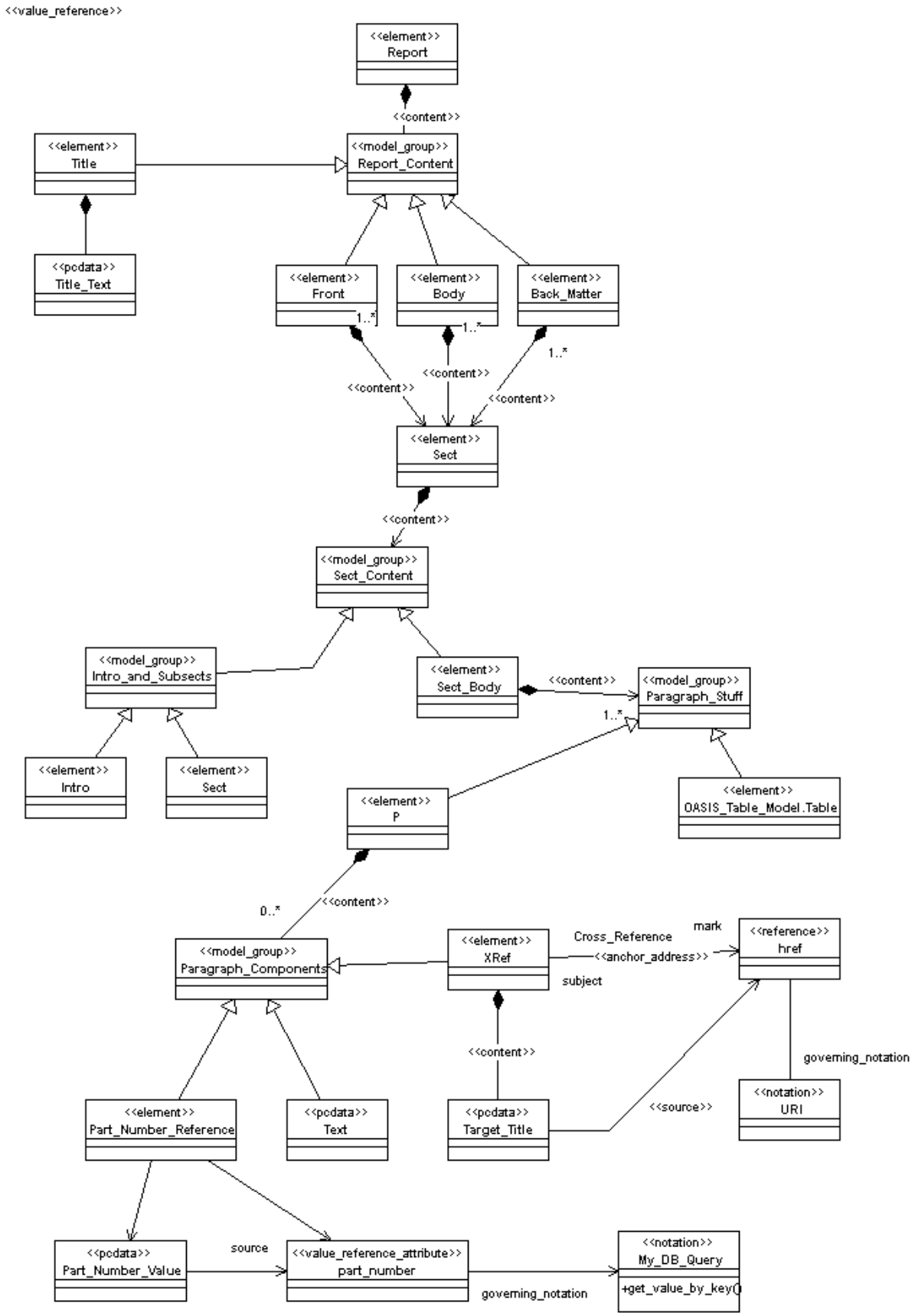

Figure 18 — Full Simple Report DTD Model

4. Sample Code for Generating DTD Declarations from the UML Model

This section presents a set of Python functions used to automatically generate XML DTD declarations from UML DTDs. This code was written for the commercial tool ObjectDomain (www.objectdomain.com), a Java-based UML modeling tool. ObjectDomain is available for free evaluation from the ObjectDomain Web site.

4.1. Code Stuff

[Editorial Note: Put code stuff here.]

5. Conclusions and Further Work To Be Done

It should be clear from the sample document that UML can be used productively to directly model XML DTDs using normal UML techniques.

Additional work to be done includes:

- Refinement of the use of specific UML syntactic facilities and conventions. For example, is an element type a refinement of a more abstract type or is it an implementation?
- Refinement of the content constraint language.
- Gain more practical experience with using this technique in real-world situations.
- Refine the use of Catalysis methods and conventions (templates, refinement, etc.) in this model.
- Propose a syntax for attribute value constraints.

Related Information

Catalysis

Hedge Automata

Colophon

This document generated from the original SGML using DSSSL and the Jade DSSSL engine (www.jclark.com/jade).