

XMLDOM: DOM/Value Mapping Revised Submission



Submitted by

BEA Systems

Cape Clear Software Ltd

Hewlett-Packard Company

International Business Machines Corporation

IONA Technologies PLC

Oracle Corporation

PeerLogic, Inc.

Persistence Software

Rogue Wave Software

Unisys Corporation

Supported by

Sun Microsystems

August 21, 2000

OMG Document orbos/2000-08-10



Copyright 2000 BEA Systems
Copyright 2000 Cape Clear Software Ltd
Copyright 2000 Hewlett-Packard Company
Copyright 2000 IBM Corporation
Copyright 2000 IONA Technologies PLC.
Copyright 2000 Oracle Corporation
Copyright 2000 PeerLogic
Copyright 2000 Persistence Software
Copyright 2000 Rogue Wave
Copyright 2000 Unisys Corporation

All rights reserved.

The companies listed above hereby grant to the Object Management Group, Inc. (OMG) and OMG members, permission to copy this document for the purpose of evaluating the technology contained herein during the technology selection process by the appropriate OMG task force. Distribution to anyone not a member of the Object Management Group or for any purpose other than technology evaluation is prohibited.

The material in this document is submitted to the OMG for evaluation. Submission of this document does not represent a commitment to implement any portion of this specification in the products of the submitters.

WHILE THE INFORMATION IN THIS PUBLICATION IS BELIEVED TO BE ACCURATE, THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. The companies listed above shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material. The information contained in this document is subject to change without notice.

This document contains information which is protected by copyright. All Rights Reserved. Except as otherwise provided herein, no part of this work may be reproduced or used in any form or by any means—graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems— without the permission of one of the copyright owners. All copies of this document must include the copyright and other information contained on this page.

RESTRICTED RIGHTS LEGEND. Use, duplication, or disclosure by government is subject to restrictions as set forth in subdivision (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013.

Table of Contents



1 Preface	7
1.1 Cosubmitting Companies	7
1.2 Introduction	7
1.2.1 Dynamic Information Scenario	8
1.2.2 Static Information Scenario	9
1.2.3 Metadata	10
1.3 Submission Contacts	11
1.4 Status of this document	13
1.5 Guide to Submission	13
2 Proof of Concept	15
2.1 Copyright Waiver	15
2.2 Proof of Concept	15
3 Response to RFP Requirements	17
3.1 Mandatory Requirements	17
3.2 Optional Requirements	18
3.3 Issues to be discussed	18
4 DOM to Valuetype mapping	19
4.1 Introduction	19
4.2 DOM Mapping Convention	19
4.3 DOM Valuetype declarations	20
4.3.1 DOMException	20



4.3.2	DOMString	21
4.3.3	XMLString	23
4.3.4	DOMImplementation	23
4.3.5	Node	24
4.3.6	NodeList	26
4.3.7	DocumentFragment	27
4.3.8	Document	27
4.3.9	NamedNodeMap	30
4.3.10	Element	32
4.3.11	Attr	34
4.3.12	CharacterData	35
4.3.13	Text	37
4.3.14	Comment	37
4.3.15	CDATASection	37
4.3.16	DocmentType	37
4.3.17	Notation	38
4.3.18	Entity	38
4.3.19	EntityReference	39
4.3.20	ProcessingInstruction	39
4.4	Extended DOM Valuetype declarations	40
4.4.1	XMLParser	40
4.4.2	XMLFactory	43
4.4.3	XMLSerializer	43
4.4.4	XMLInit	43
4.4.5	XMLShutdown	44
4.4.6	Flyweight Metadata	44
4.4.7	Element Declarations:	48
4.4.8	Content Model	50
5	DOM Level 2 mapping	53
5.1	DOM extended declarations	53
5.1.1	Events	53
5.1.2	Traversal	58
5.1.3	Range	61
5.1.4	Views	65
6	Approach to Static Mapping	67
6.1	Mapping Principles	67
6.1.1	Mapping Concepts	68
7	Static Mapping from a DTD	71



7.1	Mapping XML DTDs to IDL	71
7.1.1	Document Scope	71
7.1.2	Document Specific Valuetype	71
7.1.3	Element Valuetypes	72
7.1.4	Conditional Sections	73
7.1.5	Entities and References	73
7.1.6	NOTATION	73
7.2	Mapping Element Content	73
7.2.1	Child Elements	73
7.2.2	#PCDATA Elements	74
7.2.3	EMPTY Elements	74
7.2.4	ANY Elements	74
7.2.5	“*” - zero or more	74
7.2.6	“+” - one or more	75
7.2.7	“?” - zero or one	76
7.2.8	“,” - Sequences	77
7.2.9	“ ” - Choice Lists	78
7.2.10	Duplicate Element Names	79
7.3	Mapping Attributes	80
7.3.1	CDATA	80
7.3.2	ID	81
7.3.3	IDREF	81
7.3.4	IDREFS	82
7.3.5	ENTITY	83
7.3.6	ENTITIES	83
7.3.7	NMTOKEN	83
7.3.8	NMTOKENS	84
7.3.9	#REQUIRED	84
7.3.10	#IMPLIED	84
7.3.11	#FIXED	85
7.3.12	Enumerations	85
7.3.13	Default Attributes	85
7.4	Parameter Entities	85
7.5	Factories	86
7.6	Marshaling Framework	86
7.6.1	Mapping Example	87

8 Compliance Points 91

8.1	Mandatory Compliance Points	91
8.2	Optional Compliance Points	91

9 References 93



1.1 Cosubmitting Companies

The following companies are pleased to submit the DOM-based XML/Value Mapping, hereafter called XMLDOM, in response to the ORBOS RFP - XML/Value Mapping:

- BEA Systems
- Cape Clear Software Ltd
- Hewlett-Packard Company
- International Business Machines Corporation
- IONA Technologies PLC
- Oracle Corporation
- PeerLogic
- Persistence Software
- Rogue Wave
- Unisys Corporation

The following companies support this submission:

- Sun Microsystems

1.2 Introduction

XML has become an important and widespread standard for representing hierarchical tagged data. So much so, that it has become a common requirement to pass XML documents in CORBA interface operations. While it is possible to pass XML documents as strings, it is cumbersome to do so, and it requires each recipient of the

string to parse its XML content. A better way is to create a data structure representing the XML document that can be traversed and manipulated in memory, and passed to a remote context without further processing by the sender or the receiver.

To address this problem, this submission provides a mapping from XML documents to IDL valuetype hierarchies, based on XML DTDs.

Note – This submission does not contain mappings to XML Schemas since XML Schemas will not be finalized by the W3C for several months.

This submission provides two essential scenarios for using XML to create IDL valuetypes. The first scenario, where dynamic information is present, leverages existing standards to provide access to the full contents of an XML document in terms of IDL valuetypes.

The second scenario builds upon the first where additional static information is present from XML DTDs and (in the future) XML Schemas. The DTDs / Schemas are metadata used to generate Valuetypes that match the types of information expected to be present in XML documents. The metadata from the DTDs / Schemas and Valuetypes may be imported into CORBA Interface Repositories and the Meta Object Facility, providing wide metadata distribution through OMG standards.

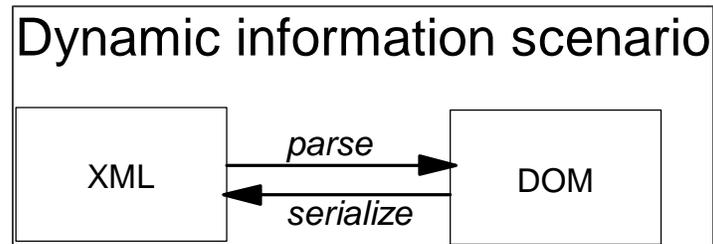
The dynamic information scenario is the processing of an XML document when the meaning of the XML elements found in the document is not defined. In this case, only minimal information is known - what is in the XML document and little else. The DOM is a standard representation for the complete contents of an XML document. The DOM satisfies the requirement of the W3C XML Information Set (Infoset) to provide an access mechanism to the document contents. By expressing the DOM in terms of IDL valuetypes, a CORBA implementation has practical, standardized, and direct access the full information in of the XML document.

1.2.1 Dynamic Information Scenario

The RFP requests "a standard way to represent XML values (documents) using OMG IDL non-object (value) types." This response provides an XML to IDL mapping leveraging the Document Object Model (DOM) technical recommendation from the World Wide Web Consortium (W3C). The DOM is an extensively used standard mechanism for defining access to XML content. The DOM includes a set of interfaces defined in IDL with mappings to Java and C++.

The purpose here is to enable IDL users to access XML content using IDL valuetypes while maintaining maximum DOM compatibility. To this end, DOM level 1 and level 2 interfaces are re-declared as IDL valuetypes instead of the IDL interfaces in the DOM standard.

The RFP does not request a mapping from IDL to XML. Mapping from IDL to XML is already accomplished using the MOF and XMI OMG standards.



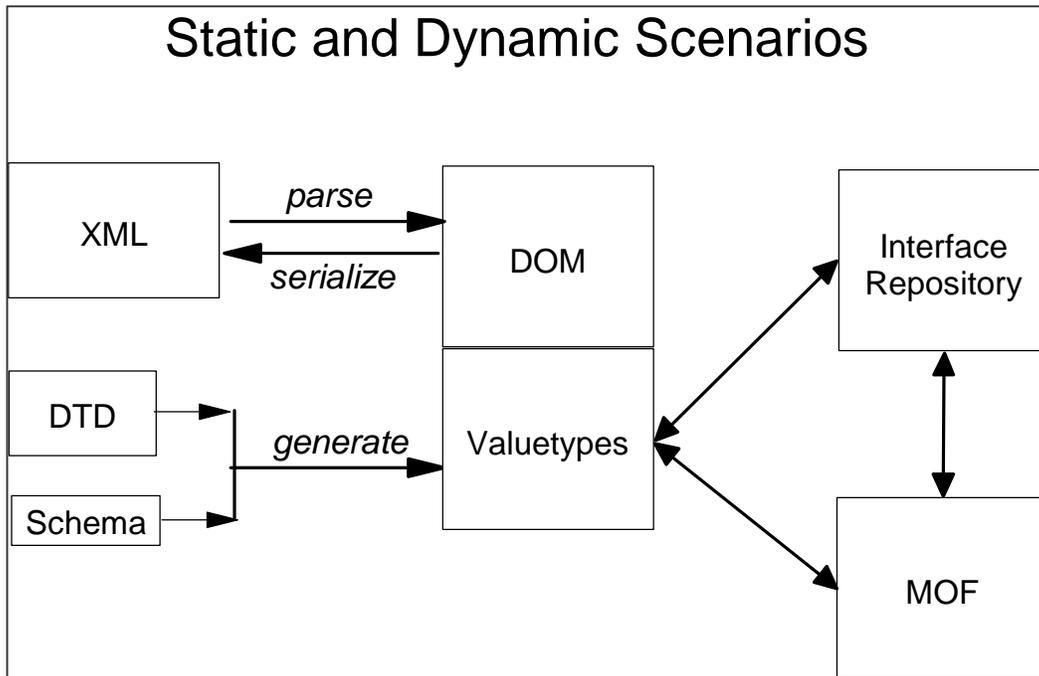
1.2.2 Static Information Scenario

If more information is known about an XML document it is possible to provide enhanced value and function from parsing an XML document. The additional information, in the form of XML DTDs and (in the future) Schemas, explain the meaning of the XML elements and enables a semantically richer operation. The DTDs and Schemas are the metadata for XML documents, the data that describes the XML document data. For clarity, we will consistently use the interchange of objects as primary example, although non-object information could also be used.

In the Static information scenario, the XML document contains a set of objects that have been serialized into that document, where the ultimate goal is to restore the objects as instances of classes. The additional information that enables the static scenario is some method for describing what the classes from which the objects were originally instantiated. This additional information is typically expressed in DTDs and (in the future) XML Schemas. If the DTDs or Schemas were generated from another source, they do not contain complete information, and additional metadata facilities such as the MOF provide further means of obtaining relevant metadata.

If the input document is in XMI format, restoring the original objects is straightforward since the XMI specification ensures that all the information required is conveniently present, with a consistent look-and-feel for all documents.

The Static information scenario uses the DTDs and (in the future) Schemas to generate new concrete IDL Valuetypes that match the metadata in the DTDs and Schemas. The valuetypes contain the same information as the XML elements that match the DTD and Schema definitions. The generation of the Valuetypes is related to the DOM defined valuetypes, so that an XML document is processed statically whenever static valuetypes are present and dynamically when new XML elements are encountered. This mixed-mode processing handles the especially common case where the XML DTD and Schema evolves at a different pace than the software deployment cycle. The flexibility to use both Dynamic and Static scenarios together in this mixed-mode processing allows the best features of both approaches to be used together.



1.2.3 Metadata

There are two fundamental sources of information in XML: DTDs and XML documents.

DTDs provide static information since they define XML elements for a class of XML documents.

XML documents provide dynamic information:

- The document contents may be instances of DTD declarations.
- The document contents may be instances of new types not declared in a DTD.
- A document may not have a DTD. The DTD may not exist or be referenced.
- The DTD is updated while the deployed software remains at a previous level, so information which could be available statically in a future software revision must be treated dynamically in the mean time.

Dynamic information is available through XML Parsing into a DOM tree. Knowing static information ahead of time supplements the dynamic information. If all the dynamic information is also available in a previously known DTD, this is the static scenario. If both static and dynamic information is used, this is the mixed scenario.

When static information from a DTD is available, the metadata defining XML documents can be extracted into IDL Valuetypes. The metadata in the valuetypes is made widely available through the CORBA Interface Repository (IR) and the Meta Object Facility (MOF).

The Corba Component Model describes the mappings from the valuetype declarations in the IR to the MOF. This provides a pathway from valuetypes to MOF metamodels.

Mapping from XML documents, DTDs, and XML Schema to the MOF is covered by the XMI production of XML Schemas RFP.

1.3 Submission Contacts

The following lists contact information for the submitters of this document. All questions concerning this submission should be directed to:

Stephen A. Brodsky, Ph.D.
International Business Machines Corporation
555 Bailey Ave., J8RA/F320
San Jose, CA 95141
Phone: +1 408 463 5659
Email: SBrodsky@us.ibm.com

Ed Cobb
BEA Systems
2315 North First St.,
San Jose, CA 95131
Tel: +1 408 570 8264
Email: ed.cobb@beasys.com

Sridhar Iyengar
Unisys Corporation
25725 Jeronimo Rd.
Mission Viejo, CA 92691
Phone: +1 949 380 5692
Email: sridhar.iyengar2@unisys.com

Manfred Koethe
IONA Technologies, Inc.
200 West Street
Waltham, MA 02451
phone: +1 781 902 8000
fax: +1 781 902 8001
Email: Manfred.Koethe@iona.com

Jeff Mischkinsky
Senior Software Architect
Persistence Software, Inc.
1720 S. Amphlett Blvd.
San Mateo, CA 04402
phone: +1 650 372 3604
fax: +1 650 341 8432
Email: jeff@persistence.com

Jishnu Mukerji
Hewlett-Packard EIAL,
300 Campus Drive, 2E-62,
Florham Park, NJ 07932, USA
Tel: +1 973 443 7528
Fax: +1 973 443 7422
Email: jis@fpk.hp.com

Nick Sharman
PeerLogic
Marlborough Court
Pickford Street
Macclesfield
Cheshire
SK11 6JD
United Kingdom
UK Tel: +44 (0) 161 333 4073
UK Fax: +44 (0) 161 333 4001
Email: nick.sharman@peerlogic.com

Patrick Thompson
Rogue Wave Software
815 NW 9th St.
Corvallis, OR 97330
USA
phone: +1 541 754 3189
fax: +1 541 758 4761
Email: thompson@roguewave.com

Jim Trezzo
Oracle Corporation
500 Oracle Parkway
Box 659504
Redwood Shores, CA 94065
Tel: +1 650 506 8240
Email: jtrezzo@us.oracle.com

1.4 *Status of this document*

This document is an update of the joint submission to the XML/Value RFP. Refer to the OMG web site, <http://www.omg.org> for additional information and the status of the adoption process.

This document has dependencies on DOM Level 2, which is under development at the W3C as a candidate recommendation. The Finalization Task Force will incorporate the final recommendations from the W3C for the DOM Level 2 specification. See <http://www.w3c.org> for current status of DOM.

1.5 *Guide to Submission*

This proposal is presented in the following chapters:

1) Preface

Introduces the submission and provides the context for the technology within the OMG architecture

2) Proof of Concept

Describes proof of concept efforts and results, in demonstration of the proposal's technical viability.

3) Response to RFP requirements

Identifies the specific RFP requirements and this proposal's response to each requirement.

4) DOM to Valuetype mapping

Describes how the DOM standard from the W3C is used to provide valuetypes for the contents of XML documents.

5) DOM Level 2 mapping

Describes the extended DOM Level 2 interfaces.

6) Approach to Static Mapping

Describes how the DTDs are mapped to Valuetypes.

7) Static Mapping from a DTD

Details the DTDs to Valuetype mapping.

8) Compliance Points

The compliance points for the submission.

9) References

References to existing and upcoming standards.

2.1 Copyright Waiver

In the event that this specification is adopted by OMG, the cosubmitters grant to the OMG, a non-exclusive, royalty-free, paid-up, worldwide license to copy and distribute this specification document and to modify the document and distribute copies of the modified version. For more detailed information, see the disclaimer on the inside of the cover page of this submission.

2.2 Proof of Concept

The cosubmitters and supporters have extensive experience in the areas of metadata repositories, modeling tools, CORBA and the related problems of interchange of metadata across tools in distributed heterogeneous environments. Representative portions of their experience are highlighted below:

- IBM has extensive experience in XML, XMI, enterprise architectures, Java, Enterprise Java Beans, CORBA, UML, MOF, IDL, and metadata. The WebSphere and VisualAge product lines provide sophisticated analysis, design, deployment, and execution functionality embodying all of the key representative technologies. AlphaWorks.ibm.com has extensive free XML tools, including a widely used XML parser.
- Unisys has implemented tools that forward and reverse engineer XML DTDs and XML Documents from CORBA Interfaces/Value Types and vice versa. The reverse engineering of IDL interfaces into XML is being used in applications that use XML and XSLT to web enable legacy systems that are CORBA enabled.

This section discusses how this submission addresses the RFP's Mandatory and Optional requirements.

3.1 Mandatory Requirements

- *Proposals shall specify a canonical way to represent XML documents as user-defined IDL, i.e. primitive types, constructed types, valuetypes, with the following restrictions:*
 - *Only values shall be used, i.e. no object types.*
 - *Only concrete value types shall be used, i.e. no abstract valuetypes, in order to ensure “on the wire” marshaling and interoperability.*

The IDL valuetypes defined herein can express all well-formed XML documents based on the W3C DOM element declarations. Valuetypes are mapped directly from the W3C DOM. Abstract valuetypes were not used.

- *Both DTD and DTDless XML document types shall be supported.*

The valuetypes defined herein handle documents specified with and without a DTD.

- *Facilities to construct XMLvalues from an XML character-based source (e.g.files,strings,streams,etc.) shall be specified.*

The valuetypes defined herein handle documents specified with and without a DTD.

- *Facilities to “walk” an XMLvalue directly (without having to parse it) shall be specified.*
 - *E.g. look up based upon tag values.*

The valuetypes defined herein include a full range of accessor functions. Navigation of an XMLvalue constructed of nodes is possible. The NamedNodeItem facility has name-based lookup.

3.2 *Optional Requirements*

- *Facilities to pattern match on XMLvalues may be specified.*

The NamedNodeItem facility has name-based lookup. Otherwise, not adressed.

- *Facilities to externalize XMLvalues into an XML character-based sink (e.g.files,strings,streams,etc.) may be specified.*

Addressed by the XMLSerializer.

- *Note that it is specifically a “non-goal” of this RFP to define a mapping from an arbitrary collection of instances of IDL types to an XML document. I*

This “non-goal” is respected.

3.3 *Issues to be discussed*

- *Submissions shall discuss the design choices that are made with respect to if, and how, the DTD associated with an XML document is made available.*

The DTD is is available through the MetaData facility

- *Submissions shall discuss the relationship between their proposal and the XML MetaData Interchange (XMI) specification.*

XMI is the OMG standard for the interchange of objects. This submission is focused on the exchange of data.

- *Submissions shall discuss the relationship between their proposal and the ongoing XML schema work occurring within the W3C.*

This proposal provides a mapping based on XML DTDs. We are comfortable that the approach taken in this mapping will apply to a W3C XML Schema based mapping. Both mappings are equally important: DTDs because they are already in widespread use; and W3C XML Schemas because they will be the standard in the future (the W3C XML Schema specification is still in the working draft stage at the time of this writing.)

DOM to Valuetype mapping

4

4.1 Introduction

This section describes how the DOM IDL interfaces are mapped to IDL valuetypes. A valuetype representation of the DOM provides information interoperability for both local and remote processing of XML documents, since the DOM is also defined as IDL interfaces for remote execution. Processing of XML documents through the XML Parser produces a DOM valuetype tree. An implementation uses the XML Factory to create specific Valuetypes that represent the XML document's contents.

Note – The DOM level 2 declarations are based on the current W3C DOM level 2 candidate recommendation. The declarations will need to be finalized when DOM level 2 becomes a W3C recommendation.

4.2 DOM Mapping Convention

The mapping from the W3C DOM is designed to provide support for valuetypes in a specification providing maximum flexibility for implementation. Valuetypes corresponding to interfaces are declared, with attribute accessor functions and a representation of state.

The convention for mapping the DOM interfaces to the submission is as follows

W3C DOM specification	Submission declarations
interface Node	valuetype Node
attribute DOMString name	attribute DOMString name private DOMString s_name

4.3 *DOM Valuetype declarations*

All the declarations in section 4.3 directly parallel the declarations from the DOM specification. The complete documentation for DOM declarations is found in the DOM specification found in the references chapter. This section documents the additional valuetype expressions and mappings.

The declarations in the submission are described below. The declarations are part of the XMLDOM module.

```
// File: value_dom.idl

#ifndef _VALUE_DOM_IDL_
#define _VALUE_DOM_IDL_

#pragma prefix "dom2.xmlvalue.omg.org"

module dom
{
// Declarations from this specification
}; /*! module dom */

#endif // _VALUE_DOM_IDL_
```

4.3.1 *DOMException*

A `DOMException` wraps the value of one of the exception codes below to indicate a DOM processing error. These codes do not indicate XML parsing errors.

```
// DOM Exception type
exception DOMException {
unsigned short    code;
};

// DOM Level 1 Exception Codes
//
const unsigned short INDEX_SIZE_ERR           = 1;
const unsigned short DOMSTRING_SIZE_ERR      = 2;
const unsigned short HIERARCHY_REQUEST_ERR   = 3;
const unsigned short WRONG_DOCUMENT_ERR      = 4;
const unsigned short INVALID_CHARACTER_ERR   = 5;
const unsigned short NO_DATA_ALLOWED_ERR     = 6;
const unsigned short NO_MODIFICATION_ALLOWED_ERR = 7;
const unsigned short NOT_FOUND_ERR           = 8;
const unsigned short NOT_SUPPORTED_ERR       = 9;
const unsigned short INUSE_ATTRIBUTE_ERR     = 10;

// Introduced in DOM Level 2:
//
const unsigned short INVALID_STATE_ERR       = 11;
const unsigned short SYNTAX_ERR              = 12;
const unsigned short INVALID_MODIFICATION_ERR = 13;
const unsigned short NAMESPACE_ERR          = 14;
const unsigned short INVALID_ACCESS_ERR     = 15;
```

4.3.2 *DOMString*

A `DOMString` wraps DOM string data, providing encapsulation of implementation and basic string accessibility functions. DOM Level 1 and 2 define `DOMString` as our `XMLString`, however, this does not give us the opportunity to standardize a minimal subset of operations which should be available on `DOMStrings` under CORBA in order to guarantee a minimal level of interoperability.

```
// Modified for XMLValues  
valuetype DOMString  
{  
// Attributes  
attribute XMLString data;  
  
// State  
private XMLString s_data;  
  
// Operations  
void appendData(  
    in DOMString source  
);  
  
void insertData(  
    in unsigned long pos,  
    in DOMString source  
);  
  
void deleteData(  
    in unsigned long pos,  
    in unsigned long count  
);  
  
DOMString substringData(  
    in unsigned long pos,  
    in unsigned long count  
);  
  
DOMString clone();  
  
unsigned short at(  
    in unsigned long pos  
);  
  
unsigned long length();  
  
short compare(  
    in DOMString other  
);  
  
boolean equals(  
    in DOMString other  
);  
  
}; /*! valuetype DOMString */
```

4.3.3 XMLString

A XMLString is a sequence of 16-bit quantities with the UTF-16 encoding following the same case sensitivity rules as in the DOM specification.

```
// Introduced for XMLValues  
typedef sequence<unsigned short> XMLString;
```

4.3.4 DOMImplementation

The DOM Implementation is for operations that are independent of a specific DOM tree instance, including implementation specific information.

```
valuetype DOMImplementation  
{  
  
// DOM1 State  
private sequence<DOMString> s_features;  
private sequence<DOMString> s_versions;  
  
// DOM1 Operations  
//  
  
boolean hasFeature(  
    in DOMString feature,  
    in DOMString version  
);  
  
// DOM2 Operations  
//  
  
DocumentType createDocumentType(  
    in DOMString qualifiedName,  
    in DOMString publicId,  
    in DOMString systemId  
)  
    raises(DOMException);  
  
Document createDocument(  
    in DOMString namespaceURI,  
    in DOMString qualifiedName,  
    in DocumentType doctype  
)  
    raises(DOMException);  
  
};
```

4.3.5 *Node*

The DOM represents the contents of an XML document as a tree of Nodes. The meaning of each declaration is identical to the DOM specification. The state representation is optimized. The Document maintains the cache of metadata used to look up the complete Node information, such as the Node name field.

The Node datatype is the base class upon which the entire Document Object Model is built. The Node provides methods for: location of the nodes parent node, iteration through a nodes children, addition and removal of nodes, etc. All other XML types represented in the DOM inherit these common services. Not all nodes, however may contain child nodes, and different Node types may only allow insertion of a subset of the full set of Node types.

Node, Element and Attribute datatypes in this submission minimize the storage of state of Node, Element and Attribute names by storing the names in the reference counted Flyweight in the Document node.

```

valuetype Node
{

    // XML Node Types
    const unsigned short ELEMENT_NODE           = 1;
    const unsigned short ATTRIBUTE_NODE        = 2;
    const unsigned short TEXT_NODE             = 3;
    const unsigned short CDATA_SECTION_NODE    = 4;
    const unsigned short ENTITY_REFERENCE_NODE = 5;
    const unsigned short ENTITY_NODE          = 6;
    const unsigned short PROCESSING_INSTRUCTION_NODE = 7;
    const unsigned short COMMENT_NODE         = 8;
    const unsigned short DOCUMENT_NODE        = 9;
    const unsigned short DOCUMENT_TYPE_NODE   = 10;
    const unsigned short DOCUMENT_FRAGMENT_NODE = 11;
    const unsigned short NOTATION_NODE        = 12;

    // DOM1 Attributes
    readonly attribute DOMString  nodeName;
    attribute   DOMString  nodeValue;
        // raises(DOMException) on setting
        // raises(DOMException) on retrieval

    readonly attribute unsigned short .nodeType;
    // NOTE: nodetype computable via repository id
    readonly attribute Node      parentNode;
    readonly attribute NodeList  childNodes;
    readonly attribute Node      firstChild;
    readonly attribute Node      lastChild;
    readonly attribute Node      previousSibling;
    readonly attribute Node      nextSibling;
    readonly attribute NamedNodeMap attributes;
    readonly attribute Document  ownerDocument;

    // DOM2 Attributes
    readonly attribute DOMString  namespaceURI;
    attribute   DOMString  prefix;
        // raises(DOMException) on setting
    readonly attribute DOMString  localName;

    // DOM1 State
    private StringKeyType  s_nodeName_key;
    private DOMString  s_nodeValue;
    private Node      s_parentNode;
    private NodeList  s_childNodes;
    private NamedNodeMap  s_attributes;
    private Document  s_ownerDocument;

    // DOM2 State
    private DOMString  s_namespaceURI;
    private DOMString  s_prefix;

```

```
private DOMString s_localName;

// DOM1 Operations
//

Node insertBefore(
    in Node newChild,
    in Node refChild
)
    raises(DOMException);

Node replaceChild(
    in Node newChild,
    in Node oldChild
)
    raises(DOMException);

Node removeChild(
    in Node oldChild
)
    raises(DOMException);

Node appendChild(
    in Node newChild
)
    raises(DOMException);

boolean hasChildNodes();

Node cloneNode(
    in boolean deep
);

// DOM2 Operations
//

void normalize();

boolean _supports(
    in DOMString feature,
    in DOMString version
);

}; /*! valuetype Node */
```

4.3.6 *NodeList*

The `NodeList` valuetype provides the abstraction of an ordered collection of nodes. The meaning of each declaration is found in the DOM specification.

```

valuetype NodeList
{
  // DOM1 Attributes
  readonly attribute unsigned long length;

  // DOM1 State
  private sequence<Node> s_nodes;

  // DOM1 Operations
  //

  Node item(
    in unsigned long index
  );
};

```

4.3.7 *DocumentFragment*

DocumentFragment is a "lightweight" or "minimal" Document object.

```

valuetype DocumentFragment : Node
{
  // Empty
};

```

4.3.8 *Document*

The Document valuetype represents the entire document. The document also contains the optimization and metadata references. The Factory controls the actual optimizations used.

The Document valuetype represents the 'top level' interface for manipulating parsed XML documents. A Document must contain one root element in order to be well-formed. The Document also contains factory methods for creating most of the other Node types available in the DOM. Each created Node is owned by the Document instance which created it.

The Document node caches Element, Node and Attribute Name state in a reference counted state member in this submission. Where the DOM is being used in the static mapping, however, this Metadata is not stored. Alternatively a means to retrieve the metadata if necessary is provided. Whether or not a document contains metadata or a callback object can be determined through a state variable.

There are two main reasons for optimizing data storage in CORBA based XML documents:

- Transport size is reduced overall.
- In the case of static mapped document types inheriting from the DOM base, the metadata is never used (cf: XML DTD mapping) and the callback not invoked.

```
valuetype Document : Node
{
    // DOM1 Attributes
    readonly attribute DocumentType doctype;
    readonly attribute DOMImplementation implementation;
    readonly attribute Element documentElement;

    // Introduced XMLValue Attributes
    readonly attribute DocumentOptimizationType xv_docOptimizationType;
    readonly attribute DocumentMetadata DocMetadata;

    // DOM1 State
    private DocumentType s_doctype;
    private DOMImplementation s_implementation;
    private Element s_documentElement;

    // Introduced XMLValue State
    private DocumentOptimizationType docOptimizationType;

    // Document Metadata
    private MetadataSwitch s_docMetadata;

    // DOM1 operations
    //

    Element createElement(
        in DOMString tagName
    )
        raises(DOMException);

    DocumentFragment createDocumentFragment();

    Text createTextNode(
        in DOMString data
    );

    Comment createComment(
        in DOMString data
    );

    CDATASection createCDATASection(
        in DOMString data
    )
        raises(DOMException);

    ProcessingInstruction createProcessingInstruction(
        in DOMString target,
        in DOMString data
    )
        raises(DOMException);
}
```

```
Attr createAttribute(  
    in DOMString name  
)  
    raises(DOMException);  
  
EntityReference createEntityReference(  
    in DOMString name  
)  
    raises(DOMException);  
  
NodeList getElementsByTagName(  
    in DOMString tagname  
)  
);  
  
// DOM2 operations  
//  
  
Node importNode(  
    in Node importedNode,  
    in boolean deep  
)  
    raises(DOMException);  
  
Element createElementNS(  
    in DOMString namespaceURI,  
    in DOMString qualifiedName  
)  
    raises(DOMException);  
  
Attr createAttributeNS(  
    in DOMString namespaceURI,  
    in DOMString qualifiedName  
)  
    raises(DOMException);  
  
NodeList getElementsByTagNameNS(  
    in DOMString namespaceURI,  
    in DOMString localName  
)  
);  
  
Element getElementById(  
    in DOMString elementId  
)  
);  
  
}; /*! valuetype Document */
```

4.3.9 *NamedNodeMap*

Objects implementing the `NamedNodeMap` datatype are used to represent collections of nodes that can be accessed by name.

```
valuetype NamedNodeMap
{
  // DOM1 Attributes
  readonly attribute unsigned long   length;

  // DOM1 State
  private sequence<Node> s_nodes;

  // DOM1 Operations
  //
  Node getItem(
    in DOMString name
  );

  Node setItem(
    in Node arg
  )
    raises(DOMException);

  Node removeItem(
    in DOMString name
  )
    raises(DOMException);

  Node item(
    in unsigned long index
  );

  // DOM2 Operations
  //
  Node getItemNS(
    in DOMString namespaceURI,
    in DOMString localName
  );

  Node setItemNS(
    in Node arg
  )
    raises(DOMException);

  Node removeItemNS(
    in DOMString namespaceURI,
    in DOMString localName
  )
    raises(DOMException);
};
```

4.3.10 Element

Elements correspond to XML elements.

```
valuetype Element : Node
{
  // DOM1 Attributes
  readonly attribute DOMString   tagName;

  // DOM1 State
  private StringKeyType  s_tagName_key;

  // DOM1 Operations
  //

  DOMString getAttribute(
    in DOMString name
  );

  void setAttribute(
    in DOMString name,
    in DOMString value
  )
    raises(DOMException);

  void removeAttribute(
    in DOMString name
  )
    raises(DOMException);

  Attr getAttributeNode(
    in DOMString name
  );

  Attr setAttributeNode(
    in Attr newAttr
  )
    raises(DOMException);

  Attr removeAttributeNode(
    in Attr oldAttr
  )
    raises(DOMException);

  NodeList getElementsByTagName(
    in DOMString name
  );

  // DOM2 Operations
  //

  DOMString getAttributeNS(
    in DOMString namespaceURI,
    in DOMString localName
  );
};
```

```
void setAttributeNS(  
    in DOMString namespaceURI,  
    in DOMString qualifiedName,  
    in DOMString value  
)  
    raises(DOMException);  
  
void removeAttributeNS(  
    in DOMString namespaceURI,  
    in DOMString localName  
)  
    raises(DOMException);  
  
Attr getAttributeNodeNS(  
    in DOMString namespaceURI,  
    in DOMString localName  
);  
  
Attr setAttributeNodeNS(  
    in Attr newAttr  
)  
    raises(DOMException);  
  
NodeList getElementsByTagNameNS(  
    in DOMString namespaceURI,  
    in DOMString localName  
);  
  
boolean hasAttribute(  
    in DOMString name  
);  
  
boolean hasAttributeNS(  
    in DOMString namespaceURI,  
    in DOMString localName  
);  
  
}; /*! valuetype Element */
```

4.3.11 *Attr*

Attrs correspond to attributes of XML elements.

```
valuetype Attr : Node
{
  // DOM1 Attributes
  readonly attribute DOMString  name;
  readonly attribute boolean    specified;
  attribute  DOMString  value;
  // raises(DOMException) on setting
  readonly attribute Element  ownerElement;

  // DOM1 State
  private StringKeyType  s_name_key;
  private boolean  s_specified;
  private DOMString  s_value;

  // DOM2 State
  private Element  s_ownerElement;
};
```

4.3.12 *CharacterData*

The *CharacterData* valuetype extends *Node* with a set of attributes and methods for accessing character data in the DOM.

```
valuetype CharacterData : Node
{
// DOM1 Attributes
    attribute DOMString data;
    // raises(DOMException) on setting
    // raises(DOMException) on retrieval
    readonly attribute unsigned long length;

// DOM1 State
    private DOMString s_data;

// DOM1 Operations
//

DOMString substringData(
    in unsigned long offset,
    in unsigned long count
)
    raises(DOMException);

void appendData(
    in DOMString arg
)
    raises(DOMException);

void insertData(
    in unsigned long offset,
    in DOMString arg
)
    raises(DOMException);

void deleteData(
    in unsigned long offset,
    in unsigned long count
)
    raises(DOMException);

void replaceData(
    in unsigned long offset,
    in unsigned long count,
    in DOMString arg
)
    raises(DOMException);

}; /*! valuetype CharacterData */
```

4.3.13 *Text*

Text corresponds to content of XML elements and attributes.

valuetype Text : CharacterData

```
{
// DOM1 Operations
//

Text splitText(
    in unsigned long offset
)
    raises(DOMException);
};
```

4.3.14 *Comment*

The Comment represents the contents of XML comment elements.

valuetype Comment : CharacterData

```
{
// Empty
};
```

4.3.15 *CDATASection*

CDATA sections represent escape blocks of characters that may otherwise appear as XML markup.

valuetype CDATASection : Text

```
{
// Empty
};
```

4.3.16 *DocumentType*

Each Document has a doctype attribute whose value is either null or a DocumentType object.

```
valuetype DocumentType : Node
{
  // DOM1 Attributes
  readonly attribute DOMString    name;
  readonly attribute NamedNodeMap entities;
  readonly attribute NamedNodeMap notations;

  // DOM2 Attributes
  readonly attribute DOMString    publicId;
  readonly attribute DOMString    systemId;
  readonly attribute DOMString    internalSubset;

  // DOM1 State
  private DOMString    s_name;
  private NamedNodeMap s_entities;
  private NamedNodeMap s_notations;

  // DOM2 State
  private DOMString    s_publicId;
  private DOMString    s_systemId;
  private DOMString    s_internalSubset;

};
```

4.3.17 Notation

Notation represents a notation declaration from a DTD.

```
valuetype Notation : Node
{
  // Attributes
  readonly attribute DOMString    publicId;
  readonly attribute DOMString    systemId;

  // State
  private DOMString    s_publicId;
  private DOMString    s_systemId;
};
```

4.3.18 Entity

Entity represents the use of an XML entity, as opposed to the definition.

```
valuetype Entity : Node
{
  // Attributes
  readonly attribute DOMString   publicId;
  readonly attribute DOMString   systemId;
  readonly attribute DOMString   notationName;

  // State
  private DOMString  s_publicId;
  private DOMString  s_systemId;
  private DOMString  s_notationName;
};
```

4.3.19 *EntityReference*

A reference to an Entity for use during Entity substitution.

```
valuetype EntityReference : Node
{
  // Empty
};
```

4.3.20 *ProcessingInstruction*

Processing Instructions are processor-specific information defined in XML.

```
valuetype ProcessingInstruction : Node
{
    // Attributes
    readonly attribute DOMString target;
    attribute DOMString data;
    // raises(DOMException) on setting

    // State
    private DOMString s_target;
    private DOMString s_data;

}; /*! valuetype ProcessingInstruction */
```

4.4 *Extended DOM Valuetype declarations*

This section describes the functions that are extensions beyond the DOM specification. The functions are interfaces for the XML Parser and XML Serializer that parse and write an XML document to/from Valuetypes. The XML Factory determines exactly which Valuetypes are created when parsing a document, and selects between the dynamic and static mapping case by the implementation of the factory. The Init and Shutdown interfaces are hooks for implementations to handle implementation-specific issues such as resource allocation. The flyweight metadata describes the mechanism for handling metadata efficiently for both the dynamic and static cases.

4.4.1 *XMLParser*

XMLParser represents an XML parser that converts an XML stream into a document. In the first form, the parse() operation will return a new DOM tree using DOM nodes. In the second form, the parse() operation will call the XMLFactory to create each node in the DOM tree.

This declaration is an extension to the DOM.

```
local interface XMLParser {
    Document parse(in DOMString XMLStream)
                raises(XMLEException);
    Document parse(in DOMString XMLStream,
                  in XMLFactory selectedFactory)
                raises(XMLEException);
};
```

The XML Exception codes indicate error codes that may occur during the parsing of a document. An implementation may produce a subset of these codes. Codes above 1000 are considered implementation-specific errors.

```
// XML Exceptions
exception XMLException {
    unsigned short code;
};

// XML Exception Codes
const short NO_ERROR = 0,
const short ALREADY_EXISTS = 2,
const short ENCODING_ERROR = 3,
const short CONTENT_MODEL_ERROR = 4,
const short INDEX_BOUNDS_ERROR = 5,
const short NODE_UNEXPECTED_ERROR = 6,
const short INVALID_DECLARATION = 7,
const short ATTR_LIST_ERROR = 8,
const short UNSUPPORTED_ENCODING = 9,
const short MISSING_XML_DECLARATION = 10,
const short MARKUP_SYNTAX_ERROR = 11,
const short INVALID_DOCUMENT_STRUCTURE = 12,
const short UNSUPPORTED_XML_PARSER = 13,
const short INVALID_CHARACTER = 14,
const short UNEXPECTED_NAME = 15,
const short UNEXPECTED_VALUE = 16,
const short INVALID_AFTER_CONTENT = 17,
const short EXPECTED_WHITESPACE = 18,
const short NOT_LEGAL_HERE = 19,
const short ENTITY_NOT_FOUND = 20,
const short ENTITY_RECURSION_ERROR = 21,
const short EXPECTED_CONTENT_MODEL = 22,
const short EXPECTED_OPEN_PAREN = 23,
const short EXPECTED_CLOSE_PAREN = 24,
const short UNEXPECTED_CLOSE_PAREN = 25,
const short EXPECTED_OCCURANCE_CHARACTER = 26,
const short EXPECTED_SEPARATOR = 27,
const short UNBALANCED_TAGS_IN_MARKUP = 28,
const short ILLEGAL_REFERENCE_ERROR = 29,
const short UNEXPECTED_END_OF_ELEMENT = 30,
const short EXPECTED_SQUARE_OPEN = 31,
const short UNEXPETED_SQUARE_OPEN = 32,
const short EXPECTED_SQUARE_CLOSE = 33,
const short INVALID_XML_DECLARATION = 34,
const short AMBIGUOUS_CONTENT_MODEL = 35,
const short NESTED_CDATA = 36,
const short INVALID_PI = 37,
const short SYSTEM_EXCEPTION = 38,
const short UNEXPECTED_EOF = 39
};
```

4.4.2 XMLFactory

The XMLFactory is a design pattern for producing concrete IDL subtypes of the DOM types. When a document is prepared for parsing, an implementation may select a specific factory to instantiate specific Node subtypes based on criteria such as the kind of DTD. For example, given an XML document and a Car DTD, a Factory may be selected that would return a Car valuetype when an element named "Car" is found in the document. If there is no user-defined type, the factory should return one of the DOM types.

The Factory can control the optimization style by returning a new Document with the desired optimization options set.

This declaration is an extension to the DOM.

```
// XML Node Factory  
local interface XMLFactory {  
    Node createType(in DOMString type);  
};
```

4.4.3 XMLSerializer

XMLSerializer represents an XML serializer that converts a Document tree of Nodes to an XML stream. This function is the reverse of the XMLParser.

This declaration is an extension to the DOM.

```
local interface XMLSerializer {  
    DOMString serialize(in Document theDocument);  
};
```

4.4.4 XMLInit

This function is called to initialize XML processing. It should be called before other operations defined in this specification to allow implementations to perform initialization. Implementations may use this call to allocate resources.

```
local interface XMLInit{  
    static void init();  
};
```

4.4.5 *XMLShutdown*

This interface is used to terminate XML valuetype processing. It should be called after all other operations are called. Implementations may use this call to deallocate resources.

This declaration is an extension to the DOM.

```
local interface XMLShutdown{  
    static void shutdown();  
};
```

4.4.6 *Flyweight Metadata*

The metadata for the document structure is kept in optimized data structures for reducing transmission and memory requirements. The strings for element and attribute names are managed using the flyweight design pattern. Nodes contain keys corresponding to the full string names maintained in the metadata so that only the keys need be present in the document nodes. If metadata is needed, it may be obtained via the MetadataCallback interface. An implementation may have the flyweight pattern enabled or disabled.

This submission optimizes state stored in parsed XML Document hierarchies by storing Element, Attribute and Node names in reference counted Flyweights. This reduces the transport overhead when sending DOM based XML document between DOM based XMLValue systems. In the case of the static mapping, rather than storing this metadata, which can be hardcoded in the static mapped documents, a query object which returns document metadata is provided as an alternative (through utilizing a union switching on a boolean) to switch in the metadata or the metadatacallback object.

In this way transport optimizations can be further improved in the static mapping, and systems sending XML documents exclusively to static systems can store data using mapped CORBA types. This increases both the transport and processing efficiency of the static mappings whilst preserving interoperability with DOM based systems through the provision of the metadata callback object.

The metadata for the document structure is stored in optimized data structures in order to a) reduce transmission and memory requirements, b) enable static mappings to disable components of the metadata in favour of caching metadata knowledge at

compile time, c) reducing runtime overhead, and d) allowing validation or exposing methods preservant of content model semantics manipulating underlying DOM data structures.

The DOMStrings for element/attribute names and values are managed using the flyweight design pattern. Nodes contain keys corresponding to the full string names maintained in the metadata so that only the keys need be present in the document nodes. For example, for all element nodes named 'foobar' each node instance contains a copy of the key. The element/attribute names and attribute values are stored efficiently in the flyweight reducing the amount of duplicated information stored (and hence transmitted) in the node instances.

The flyweight also provides a callback facility, allowing metadata to be retrieved by valuetypes that utilize caching, where the originating system has the flyweight disabled. Usage of the callback facility is transparent to endpoint developers and designed for use by XML/Valuetype implementations.

In the case of static mappings, rather than storing all metadata, typically metadata should be hardcoded at compile-time. When a static mapped document is received by a DOM based XML/Valuetype system, however, a mechanism to retrieve the metadata must be provided. The callback query object provides this functionality through utilizing a boolean discriminated union metadata switch. Effectively, an XML/Valuetype instance contains either a) no metadata or b) a means by which to retrieve metadata if required. This mechanism further reduces communication by sending metadata only when necessary.

The flyweight metadata store and metadata callback query objects increase both the transport and processing efficiency of XML/valuetypes whilst preserving full interoperability between dynamic DOM based and static document preservant XML/Valuetype systems.

This feature is a CORBA-specific optimization introduced for the efficient transport and processing of XML documents in distributed environments. The facilities provided by the solution are designed for utilization by XML/Valuetypes implementations, and places no restrictions or variation on document processing, manipulation or transport onto XML/Valuetype developers.

This declaration is an extension to the DOM.

```

// Introduced for XMLValues
exception KeyNotExist {};
exception NonZeroReferenceCount {};
exception FlyweightDisabled {};

// Introduced for XMLValues - for efficient storage
// Element/Attribute string names. First 1000 key
// names are reserved and can only be assigned by
// the OMG for identifying 'anonymous' nodes such as
// #cdata-section, #comment etc...
//
typedef
union SKT switch(boolean)
{
    case TRUE: unsigned long key;
    case FALSE: DOMString name;
} StringKeyType;

valuetype StringFlyweight
{
// Declarations
typedef
    struct SRT
    {
        DOMString data;
        unsigned long ref_count;
        StringKeyType key;
    } StringRegistryType;

union optimized_registry switch(boolean)
{
    case TRUE: sequence<StringRegistryType> registry;
};

// State
private optimized_registry store;

// Operations
unsigned long get_key_by_name(
    in DOMString query
) raises(FlyweightDisabled);

DOMString get_string_by_key(
    in unsigned long key
) raises(FlyweightDisabled);

DOMString get_builtin_name_by_nodeType(
    in unsigned short type
)
    raises(KeyNotExist);

```

```
    unsigned long register(  
        in DOMString candidate  
    ) raises(FlyweightDisabled);  
  
    DOMString unregister(  
        in unsigned long key,  
        in boolean force  
    )  
        raises(NonZeroReferenceCount,FlyweightDisabled);  
  
};  
  
// Introduced in DOM2  
typedef unsigned long long DOMTimeStamp;  
  
// Introduced for XMLValues  
enum DocumentOptimizationType  
{  
    FLYWEIGHT_ENABLED,  
    FLYWEIGHT_DISABLED  
};  
  
struct DocumentMetadata  
{  
    StringFlyweight element_name_map;  
    StringFlyweight attr_name_map;  
    StringFlyweight node_name_map;  
};  
  
struct MetadataProxyDetails  
{  
    MetadataCallback metadata_query_object;  
    DOMString docId;  
};  
  
union MetadataSwitch switch(boolean)  
{  
    case TRUE: DocumentMetadata docMetadata;  
    case FALSE: MetadataProxyDetails docProxyInfo;  
};  
  
interface MetadataCallback  
{  
    DocumentMetadata get_metadata(  
        in DOMString document_id  
    );  
};
```

4.4.7 Element Declarations:

Arbitrary Element declarations are supported by the following IDL constructs.

```
//  
// Element Content  
//  
  
// Element Types  
//  
enum ElementType  
{  
EMPTY,  
PCDATA,  
ANY  
};  
  
// Occurences of Any, Choice or Sequence Elements  
// may occur multiply  
//  
enum ElementOccurenceType  
{  
SIMPLE, // One ie: Empty / PCDATA  
ZERO_OR_ONE,  
ZERO_OR_MANY,  
ONE_OR_MANY  
};  
  
// Element Content Descriptor  
struct ElementDeclaration  
{  
DOMString name;  
ElementType type;  
sequence<ElementDeclaration> children;  
// Length == 0 for SIMPLE  
// Length <= 1 for ZERO_OR_ONE  
// Length >= 0 for ZERO_OR_MANY  
// Length >= 1 for ONE_OR_MANY  
// ...determined by occurences  
ElementOccuranceType occurences;  
sequence<AttrDeclaration> attributes;  
unsigned long duplicates; // Handles duplicate Element instances  
};  
  
//  
// Content Model Representation  
//  
  
enum DeclarationType  
{  
ELEMENTDECL,  
NOTATIONDECL,
```

```
PARAMENTITYDECL,  
ATTRDECL  
// etc..  
};  
  
typedef unsigned long DeclarationIndex;  
  
struct DeclarationInstanceType  
{  
    DeclarationType decltype;  
    DeclarationIndex index;  
};
```

4.4.8 Content Model

The ContentModel struct represents the DTD tree which defines structure of an XML document. Currently, only Element and Attribute Declarations are supported.

```

typedef sequence <DeclarationInstanceType> ContentModelIndex;

struct ContentModelBase
{
  ElementDeclaration rootElement;
  // etc..
};

struct ContentModel
{
  ContentModelBase internal_subset;
  ContentModelBase external_subset;
};

//
// Node Types supporting arbitrary Content Models
//

// Forward Declarations
valuetype ElementBase;

// IDElementMap - maps id's to idrefs as per roguewave 4.1.6
struct IDtoElement
{
  DOMString id;
  ElementBase element_instance;
};

valuetype IDElementMap
{
  // State
  private sequence<IDtoElement> map;

  // Operations

  ElementBase getElement(in string id);

  void setElement(in string id, in ElementBase element_instance);
};

// Base for Content Model Enhanced Attributes
valuetype AttrBase : truncatable DOM_Value::Attr
{
  // State
  private AttrType attr_type;
  private DeclarationInstanceType decl_instance;

  // Operations

  AttrType getAttributeType();
};

```

```
    boolean mayInsertAttribute(in AttrBase candidate);

    boolean isValid();

    AttrDeclaration getAttributeDeclaration();
};

// Base for Content Model Enhanced Elements
valuetype ElementBase : truncatable DOM_Value::Element
{
    // State
    private ElementType element_type;
    private DeclarationInstanceType decl_instance;

    // Operations

    ElementType getElementType();

    boolean mayInsertElement(in ElementBase candidate);

    boolean mayInsertAttribute(in AttrBase candidate);

    boolean isValid();

    ElementDeclaration getElementDeclaration();
};

// Base for Content Model Enhanced Documents
valuetype DocumentBase : truncatable DOM_Value::Document
{
    // State
    private IDElementMap id_element_map;
    private ContentModelIndex doc_index;
    private ContentModelBase internal_subset;

    // Operations

    IDElementMap getIDElementMap();

    ContentModel getContentModel();

    ContentModelIndex getContentModelIndex();

    boolean validate();
};
};
```

5.1 DOM extended declarations

Note – This chapter is based on the current draft of the W3C DOM level 2. Updates will be made when the final DOM Level 2 specification is completed.

The DOM Level 2 consists of both the core declarations in Chapter 4 and the Chapter 5 optional declarations for processing additional XML-style information sources. The following mappings are for Events, Traversal, Range, and Views. The other optional components HTML (HyperText Markup Language), StyleSheets, and CSS (Cascading Style Sheets) are not mapped.

5.1.1 Events

The DOM Level 2 Event Model is a generic event system which allows registration of event handlers, describes event flow through a tree structure, and provides basic contextual information for each event.

```
// File: value_events.idl

#ifndef _VALUE_EVENTS_
#define _VALUE_EVENTS_

#include "value_dom.idl"
#include "value_views.idl"

#pragma prefix "dom2.xmlvalue.omg.org"

// The Events module was introduced in DOM Level 2
// as an optional module
module events
{

    // Convenience Declarations
    typedef dom::DOMString DOMString;
    typedef dom::Node Node;

    valuetype EventListener;
    valuetype Event;

    exception EventException
    {
        unsigned short code;
    };

    // EventExceptionCode
    const unsigned short UNSPECIFIED_EVENT_TYPE_ERR = 0;

    valuetype EventTarget
    {
        void addEventListener(
            in DOMString type,
            in EventListener listener,
            in boolean useCapture
        );

        void removeEventListener(
            in DOMString type,
            in EventListener listener,
            in boolean useCapture
        );

        boolean dispatchEvent(
            in Event evt
        )
    }
}
```

```
        raises(EventException);
    };

    valuetype EventListener
    {
        void handleEvent(
            in Event evt
        );
    };

    valuetype Event
    {
        // PhaseType
        const unsigned short CAPTURING_PHASE = 1;
        const unsigned short AT_TARGET     = 2;
        const unsigned short BUBBLING_PHASE = 3;

        // State
        private DOMString type;
        private EventTarget target;
        private Node currentNode;
        private unsigned short eventPhase;
        private boolean bubbles;
        private boolean cancelable;

        // Operations
        DOMString getType();

        EventTarget getTarget();

        Node getCurrentNode();

        unsigned short getEventPhase();

        boolean getBubbles();

        boolean getCancelable();

        void stopPropagation();

        void preventDefault();

        void initEvent(
            in DOMString eventTypeArg,
            in boolean canBubbleArg,
            in boolean cancelableArg
        );
    };

    valuetype DocumentEvent
    {
```

```
    Event createEvent(  
        in DOMString eventType  
    )  
        raises(dom::DOMException);  
};  
  
valuetype UIEvent : Event  
{  
    // State  
    private views::AbstractView view;  
    private long detail;  
  
    // Operations  
    views::AbstractView getView();  
  
    long getDetail();  
  
    void initUIEvent(  
        in DOMString typeArg,  
        in boolean canBubbleArg,  
        in boolean cancelableArg,  
        in views::AbstractView viewArg,  
        in long detailArg  
    );  
};  
  
valuetype MouseEvent : UIEvent  
{  
    // State  
    private long screenX;  
    private long screenY;  
    private long clientX;  
    private long clientY;  
    private boolean ctrlKey;  
    private boolean shiftKey;  
    private boolean altKey;  
    private boolean metaKey;  
    private unsigned short button;  
    private Node relatedNode;  
  
    // Operations  
    long getScreenX();  
  
    long getScreenY();  
  
    long getClientX();  
  
    long getClientY();  
  
    boolean getCtrlKey();
```

```
boolean getShiftKey();

boolean getAltKey();

boolean getMetaKey();

unsigned short getButton();

Node getRelatedNode();

void initMouseEvent(
in DOMString typeArg,
  in boolean canBubbleArg,
  in boolean cancelableArg,
  in views::AbstractView viewArg,
  in unsigned short detailArg,
  in long screenXArg,
  in long screenYArg,
  in long clientXArg,
  in long clientYArg,
  in boolean ctrlKeyArg,
  in boolean altKeyArg,
  in boolean shiftKeyArg,
  in boolean metaKeyArg,
  in unsigned short buttonArg,
  in Node relatedNodeArg
);
};

valuetype MutationEvent : Event
{
// State
private Node relatedNode;
private DOMString prevValue;
private DOMString newValue;
private DOMString attrName;

// Operations
Node getRelatedNode();

DOMString getPrevValue();

DOMString getNewValue();

DOMString getAttrName();

void initMutationEvent(
in DOMString typeArg,
  in boolean canBubbleArg,
  in boolean cancelableArg,
```

```
        in Node relatedNodeArg,  
        in DOMString prevValueArg,  
        in DOMString newValueArg,  
        in DOMString attrNameArg  
    );  
};  
  
}; /*! module events */  
  
#endif // _VALUE_EVENTS_  
}
```

5.1.2 Traversal

The optional TreeWalker, NodeIterator, and Filter interfaces for traversing DOM Node trees are described.

```
// File: value_traversal.idl

#ifndef _VALUE_TRAVERSAL_
#define _VALUE_TRAVERSAL_

#include "value_dom.idl"

#pragma prefix "dom2.xmlvalue.omg.org"

// The traversal module was Introduced in DOM Level 2
// as an optional module

module traversal
{

    // Convenience Declarations
    typedef dom::Node Node;

    interface NodeFilter;

    valuetype NodeIterator
    {
        // State
        private long    whatToShow;
        private NodeFilter filter;
        private boolean  expandEntityReferences;

        // Operations
        long getWhatToShow();

        NodeFilter getFilter();

        boolean getExpandEntityReferences();

        Node nextNode()
            raises(dom::DOMException);

        Node previousNode()
            raises(dom::DOMException);

        void detach();
    };

    valuetype NodeFilter
    {
        // Constants returned by acceptNode
        const short    FILTER_ACCEPT    = 1;
    };
};
```

```
const short    FILTER_REJECT      = 2;
const short    FILTER_SKIP        = 3;

// Constants for whatToShow
const unsigned long SHOW_ALL      = 0x0000FFFF;
const unsigned long SHOW_ELEMENT  = 0x00000001;
const unsigned long SHOW_ATTRIBUTE = 0x00000002;
const unsigned long SHOW_TEXT     = 0x00000004;
const unsigned long SHOW_CDATA_SECTION = 0x00000008;
const unsigned long SHOW_ENTITY_REFERENCE = 0x00000010;
const unsigned long SHOW_ENTITY   = 0x00000020;
const unsigned long SHOW_PROCESSING_INSTRUCTION =
0x00000040;
const unsigned long SHOW_COMMENT  = 0x00000080;
const unsigned long SHOW_DOCUMENT = 0x00000100;
const unsigned long SHOW_DOCUMENT_TYPE = 0x00000200;
const unsigned long SHOW_DOCUMENT_FRAGMENT = 0x00000400;
const unsigned long SHOW_NOTATION = 0x00000800;

// Operations
short acceptNode(
in Node n
);
};

valuetype TreeWalker
{
// State
private long    whatToShow;
private NodeFilter filter;
private boolean expandEntityReferences;
private Node    currentNode;

// Operations
long getWhatToShow();

NodeFilter getFilter();

boolean getExpandEntityReferences();

Node getCurrentNode();

void setCurrentNode(
in Node currentNode
)
raises(dom::DOMException);

Node parentNode();

Node firstChild();
```

```

    Node lastChild();

    Node previousSibling();

    Node nextSibling();

    Node previousNode();

    Node nextNode();
};

valuetype DocumentTraversal
{
    Nodelerator createNodelerator(
    in Node root,
    in long whatToShow,
    in NodeFilter filter,
    in boolean entityReferenceExpansion
    );

    TreeWalker createTreeWalker(
    in Node root,
    in long whatToShow,
    in NodeFilter filter,
    in boolean entityReferenceExpansion
    )
    raises(dom::DOMException);
};

}; /*! module traversal */

#endif // _VALUE_TRAVERSAL_

```

5.1.3 Range

A Range identifies a range of content in a Document, DocumentFragment or Attr. It is contiguous in the sense that it can be characterized as selecting all of the content between a pair of boundary-points.

```
// File: value_range.idl

#ifndef _VALUE_RANGE_
#define _VALUE_RANGE_

#include "value_dom.idl"

#pragma prefix "dom2.xmlvalue.omg.org"

// Range module introduced in DOM Level 2 as
// an optional module

module ranges
{

    // Convenience Declarations
    typedef dom::Node Node;
    typedef dom::DocumentFragment DocumentFragment;
    typedef dom::DOMString DOMString;

    exception RangeException
    {
        unsigned short code;
    };

    // RangeExceptionCode
    const unsigned short BAD_BOUNDARYPOINTS_ERR = 1;
    const unsigned short INVALID_NODE_TYPE_ERR = 2;

    valuetype Range
    {
        // State
        //
        private Node startContainer;
        private long startOffset;
        private Node endContainer;
        private long endOffset;
        private boolean isCollapsed;
        private Node commonAncestorContainer;

        Node getStartContainer()
            raises(dom::DOMException);

        long getStartOffset()
    }
}
```

```
        raises(dom::DOMException);

Node getEndContainer()
    raises(dom::DOMException);

long getEndOffset()
    raises(dom::DOMException);

boolean getIsCollapsed()
    raises(dom::DOMException);

Node getCommonAncestorContainer()
    raises(dom::DOMException);

void setStart(
    in Node refNode,
    in long offset
)
    raises(RangeException, dom::DOMException);

void setEnd(
    in Node refNode,
    in long offset
)
    raises(RangeException, dom::DOMException);

void setStartBefore(
    in Node refNode
)
    raises(RangeException, dom::DOMException);

void setStartAfter(
    in Node refNode
)
    raises(RangeException, dom::DOMException);

void setEndBefore(
    in Node refNode
)
    raises(RangeException, dom::DOMException);

void setEndAfter(
    in Node refNode
)
    raises(RangeException, dom::DOMException);

void collapse(
    in boolean toStart
)
    raises(dom::DOMException);
```

```
void selectNode(
    in Node refNode
)
    raises(RangeException, dom::DOMException);

void selectNodeContents(
    in Node refNode
)
    raises(RangeException, dom::DOMException);

typedef enum CompareHow_ {
    StartToStart,
    StartToEnd,
    EndToEnd,
    EndToStart
} CompareHow;

short ompareBoundaryPoints(
    in CompareHow how,
    in Range sourceRange
)
    raises(dom::DOMException);

void deleteContents()
    raises(dom::DOMException);

DocumentFragment extractContents()
    raises(dom::DOMException);

DocumentFragment cloneContents()
    raises(dom::DOMException);

void insertNode(
    in Node newNode
)
    raises(dom::DOMException, RangeException);

void surroundContents(
    in Node newParent
)
    raises(dom::DOMException, RangeException);

Range cloneRange()
    raises(dom::DOMException);

DOMString toString()
    raises(dom::DOMException);

void detach()
    raises(dom::DOMException);
};
```

```

    valuetype DocumentRange
    {
        Range createRange();
    };

};

#endif // _VALUE_RANGE_

```

5.1.4 Views

Views are introduced in DOM Level 2 as a means for representations of documents after transformations are applied.

```

// File: value_views.idl

#ifndef _VALUE_VIEWS_
#define _VALUE_VIEWS_

#include "value_dom.idl"

#pragma prefix "dom2.xmlvalue.w3c.org"

// The Views module was introduced in DOM
// Level 2 as an optional module
module views
{
    // Forward Declarations
    valuetype DocumentView;

    valuetype AbstractView
    {
        // State
        private DocumentView document;

        // Operations
        DocumentView getDocument();
    };

    valuetype DocumentView
    {
        // State
        private AbstractView defaultView;
    };
};

```

```
    // Operations
    AbstractView getDefaultView();
};

}; /*! module views */

#endif // _VALUE_VIEWS_
```

The static mapping of XML to IDL valuetypes creates a hierarchical set of types that directly reflect the document structure. The mapping is driven by a document DTD definition which describes the structure of a class of documents. The static mapping provides a document specific set of abstractions. In contrast, the dynamic mapping provides a generic hierarchy of nodes that can be used to represent any XML document. In general, the static mapping is easier to use than the dynamic mapping, however it is less flexible.

The static mapping of XML to IDL valuetypes is driven by a document DTD. A document DTD defines the format of a document: what set of XML elements and attributes make up a valid document, and the ordering of elements that constitute valid document content. Defining a mapping based on DTDs allows us to define a mapping that closely reflects the document content model.

The balance of this chapter discusses the general approach taken in statically mapping XML documents to valuetypes based on DTD information.

6.1 Mapping Principles

The format of an XML document is represented by a DTD. A DTD specifies a vocabulary for a class of documents. XML document instances which follow the DTD are said to be valid with respect to the DTD.

XML documents and IDL valuetypes are similar in that each have a sort of “class” definitions and instances. An XML DTD represents a class of documents, as an IDL valuetype definition represents a class of valuetype. XML documents are instances of XML DTDs as valuetype instances are instances of valuetype interface definitions. In mapping XML to valuetypes, we map XML DTDs to Valuetype IDL, and XML document instances to valuetype instances. The generated valuetype IDL and its implementation provides a hierarchical structure of valuetypes that parallels the hierarchical structure of the XML document structure.

Along with the valuetype definition and implementation, a marshalling framework is generated. The marshalling framework provides a mechanism by which documents can be read into a valuetype hierarchy and written out again. Documents may be created from scratch in memory or read in from an external source. Before a document is written out, or at anytime, the validity of the document in memory can be verified.

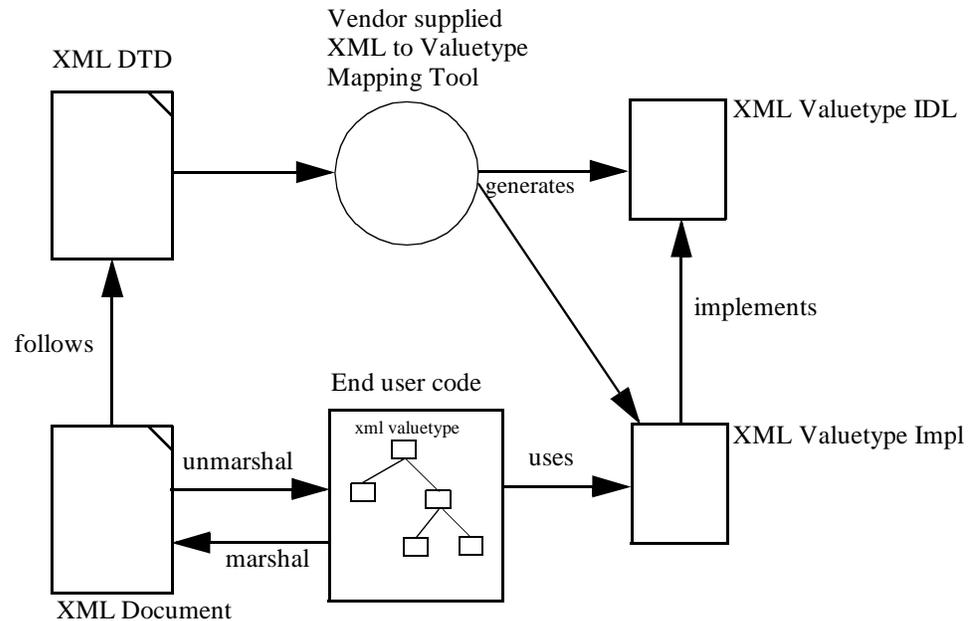


Figure 6-1 High level perspective of XML to Valuetype static mapping

6.1.1 Mapping Concepts

The static mapping creates a hierarchical set of valuetypes which parallels the document's hierarchical element hierarchy. As elements can contain attributes and other elements, corresponding element valuetypes may contain attributes and other element valuetypes. The element valuetypes are derived from the dynamic element type described by the dynamic mapping; all element state is stored in the dynamic element base valuetype.

Note – Storing state in the dynamic base elements allows for the static document to be passed to context that does not know about the static structure, allowing the document to be traversed using the dynamic mapping.

If an element contains other elements then the valuetypes representing those elements may be accessed through operations on the containing element valuetype.

All static XML valuetype declarations are scoped by a module that corresponds to the name of the document's DTD.

Elements are represented as IDL valuetypes. All element valuetypes inherit from the `dom::Element` valuetype described in the dynamic mapping portion of this submission.

Embedded elements and attributes are stored in the `dom::Element` base valuetype and accessed via `get` and `set` operations. The names of the `get` and `set` operations are determined by the name of the element or attribute.

Element lists are represented as a sequence and a set of access operations.

Choice or alternate lists are represented as a sequence of a contrived type, where the contrived type represents the choice or alternate statement.

Document structures represented as valuetypes may be navigated from parent elements to child elements through element valuetype operations. As primitive element content and attributes are represented as private state of the element's base type, XML valuetype documents can be marshalled by the ORB, like any other valuetype.

Static Mapping from a DTD

7

This section describes a static mapping of XML documents to valuetypes based on XML DTDs. The mapping defines a hierarchy of valuetypes that mirror an XML document's structure. Specific valuetypes are used to represent elements that may themselves contain other elements.

Valuetypes representing document elements inherit from generic valuetypes defined in the DOM module.

7.1 Mapping XML DTDs to IDL

7.1.1 Document Scope

A DTD maps to an IDL module scope. The module is named after the DTD. The contents of the DTD map to IDL declarations in the module scope corresponding to the DTD.

For example, a DTD named "Inventory.dtd" would map to an IDL module named "Inventory".

7.1.2 Document Specific Valuetype

Associated with each document mapping is a document specific valuetype, which inherits from the generic `dom::Document`. The document specific valuetype has a type specific operation to return the document root element and a set of factory operations for each element type defined in the document.

The name of the document specific valuetype is the name of the document concatenated with "Doc".

The name of the type specific document root accessor operation is `get<DocumentName>Root()`.

The name of the type specific document root accessor operation is `get<DocumentName>Root()`.

The element factory operations names are patterned as:
`create<ElementName>Element()`.

For example in mapping a DTD named `Personnel.dtd` with a root element called `HR`, a module named `Personnel` would be created with a valuetype definition called `PersonnelDoc`, as such:

```

module Personnel {
  valuetype PersonnelDoc : truncatable dom::Document {
    HR getPersonnelRoot();
    void setPersonnelRoot(in HR docRoot);
    HR createHRElement();
    Employee createEmployeeElement();
  }
}

```

7.1.2.1 *get<RootElementName>Root*

Returns the type specific root of the document.

7.1.2.2 *setDocRoot*

Sets the document root element.

7.1.3 *Element Valuetypes*

Each XML element in a document maps to an IDL valuetype. The valuetype is named after the element from which it is derived and it is defined in the scope of the module corresponding to the DTD. All element valuetypes inherit from `dom::Element`.

For example, the element `Employee` defined in `Personnel.dtd`:

```
<!ELEMENT Employee (...)>
```

would map to:

```

module Personnel {
  valuetype Employee : truncatable dom::Element {
    ...
  }
}

```

Element valuetypes provide type specific operations for accessing the XML element's attributes and content. Content can be either character data or child elements. The set of valuetypes corresponding to a document form a hierarchy matching the document structure. Operations on each valuetype allows attribute and element content to be retrieved and set.

Note – An element valuetype manages the elements and attributes that it contains. When the element valuetype is destroyed the contained element valuetypes and attributes are also destroyed.

7.1.4 Conditional Sections

It is assumed that conditional sections will be resolved before the mapping to valuetypes takes place.

7.1.5 Entities and References

Note – See dynamic mapping.

7.1.6 NOTATION

Note – See dynamic mapping.

7.2 Mapping Element Content

This section describes in detail how element definitions are mapped to valuetype definitions.

7.2.1 Child Elements

A mandatory child element maps to get and set operations on the mapped XML valuetype. Note that the child element state is stored in the `dom::Element` base class.

An element defined as:

```
<!ELEMENT Parent ( Child ) >
```

would map to:

```
valuetype Parent : truncatable dom::Element {  
    Child getChild();  
    void setChild(in Child arg0);  
}
```

7.2.1.1 *get*<ElementName>

Returns the child element. If there is no child element defined then it returns null.

7.2.1.2 *set*<ElementName>

Sets the child element.

7.2.2 #PCDATA Elements

Character data content is mapped to an element that inherits from the `dom:Text` datatype.

An element defined as:

```
<!ELEMENT Name ( #PCDATA ) >
```

would map to:

```
valuetype Name : truncatable dom::Text {}
```

7.2.3 EMPTY Elements

Elements with an EMPTY content specification map to a datatype which inherits from the `dom::Element` datatype.

An element defined as:

```
<!ELEMENT HR EMPTY >
```

would map to:

```
valuetype HR : truncatable dom::Element {}
```

7.2.4 ANY Elements

Elements with an ANY content model map to a datatype which inherits from the `dom::Element` datatype.

An element defined as:

```
<!ELEMENT AnyAndAll ANY >
```

would map to:

```
valuetype AnyAndAll : truncatable dom::Element {}
```

7.2.5 “*” - zero or more

The * character following an element, sequence, or choice indicates that it occurs zero or more times. That piece of the content specification maps to an IDL sequence within the mapped XML datatype. Operations are defined to access get, set, and manipulate the elements in the sequence.

For example:

```
<!ELEMENT CardShelf ( Card* ) >
```

would map to:

```

typedef sequence<Card> CardSeq;
valuetype CardShelf : truncatable dom::Element {

    //Element access operations
    CardSeq getCardSeq();
    Card getCardSeqAt(in long index);
    long getCardSeqSize();
    void setCardSeq(in CardSeq arg0);
    void replaceCardSeqAt(in Card arg0,
                          in long index);
    void appendCardSeq(in Card arg0);
    void insertCardSeqAt(in Card arg0,
                        in long index);
    void removeFromCardSeq(in Card arg0);
    void removeFromCardSeqAt(long index);
    void clearCardSeq();
}

```

7.2.6 “+” - *one or more*

The + character following an element, sequence, or choice indicates that it occurs one or more times. That piece of the content specification maps to an IDL sequence within the mapped XML valuetype. Operations are defined to access get, set, and manipulate the elements in the sequence. The mapping is the same as the mapping for “*”, the exception being that the isValid() operation will test that there is at least one element defined in the sequence.

For example:

```
<!ELEMENT Employees ( Employee+ ) >
```

would map to:

```

typedef sequence<Employee> EmployeeSeq;
valuetype EmployeeShelf : truncatable dom::Element {

    //Element access operations
    EmployeeSeq getEmployeeSeq();
    Employee getEmployeeSeqAt(in long index);
    long getEmployeeSeqSize();
    void setEmployeeSeq(in EmployeeSeq arg0);
    void replaceEmployeeSeqAt(in Employee arg0,
                              in long index);
    void appendEmployeeSeq(in Employee arg0);
    void insertEmployeeSeqAt(in Employee arg0,
                             in long index);
    void removeFromEmployeeSeq(in Employee arg0);
    void removeFromEmployeeSeqAt(long index);
    void clearEmployeeSeq();
}

```

7.2.7 “?” - zero or one

The ‘?’ character following an element, sequence, or choice indicates that it occurs zero or one time. That piece of the content specification maps to operations to get, set, and remove the item.

An element defined as:

```
<!ELEMENT ABC ( XYZ? ) >
```

would map to:

```

valuetype ABC : truncatable dom::Element {
    XYX getXYZ();

    void setXYZ(in XYZ arg0);
    void removeXYZ();
}

```

7.2.7.1 *get<ElementName>*

Returns the child element. If there is no child element defined then it returns null.

7.2.7.2 *set<ElementName>*

Sets the child element.

7.2.7.3 *remove<ElementName>*

Removes the child element if one is set.

7.2.8 “,” - Sequences

A sequence is an ordered group of content particles. A content particle can be an element, a sequence list, or a choice list. The content particles in a sequence are separated by commas. For example the following is a sequence of child elements:
<ELEMENT Date (Day, Month, Year)>

For the purposes of this mapping we will characterize sequences as being either simple or complex. A simple sequence list is as shown in the Date element above. A complex sequence is a simple sequence that is followed by a “*”, a “+”, or a “?”.

7.2.8.1 Simple Sequence Lists

A simple sequence maps to a valuetype with operations to get and set the individual elements or content particles. The operations and isValid constraints are as defined in section 7.2.1.

For example, the Date element above would map to:

```
valuetype Date : truncatable dom::Element {
  Day getDay();
  void setDay(in Day arg0);

  Month getMonth();
  void setMonth(in Month arg0);

  Year getYear();
  void setYear(in Year arg0);
}
```

7.2.8.2 Complex Sequence Lists

In mapping complex sequences we must contrive an element valuetype that represents the sequence. Then the mapping for “*”, “+”, or “?” is applied to that contrived element valuetype.

The name of the contrived valuetype is constructed from the names of the content particles separated by “**And**”.

For example the element:

```
<!ELEMENT ComplexSeq (One, Two)* >
```

would map to the following:

```

valuetype OneAndTwo : truncatable dom::Element {
    One getOne();
    void setOne(in One arg0);

    Two getTwo();
    void setTwo(in Two arg0);
}

typedef sequence<OneAndTwo> OneAndTwoSeq;
valuetype ComplexSeq : truncatable dom::Element {
    //Element access operations
    OneAndTwoSeq getOneAndTwoSeq();
    OneAndTwo getOneAndTwoSeqAt(in long index);
    long getOneAndTwoSeqSize();
    void setOneAndTwoSeq(in OneAndTwoSeq arg0);
    void replaceOneAndTwoSeqAt(in OneAndTwo arg0,
        in long index);
    void appendOneAndTwoSeq(in OneAndTwo arg0);
    void insertOneAndTwoSeqAt(in OneAndTwo arg0,
        in long index);
    void removeFromOneAndTwoSeq(in OneAndTwo arg0);
    void removeFromOneAndTwoSeqAt(long index);
    void clearOneAndTwoSeq();
}

```

7.2.9 “|” - Choice Lists

A choice list is a set of alternates from a group of content particles. A content particle can be an element, a sequence list, or a choice list. The content particles in a choice list are separated by the “|” character. For example the following is a choice list of child elements:

```
<ELEMENT ChoiceList ( One | Two | Three)>
```

As with sequences, we characterize choice lists as being either simple or complex. A simple sequence list is as shown in the ChoiceList element above. A complex choice list is a simple choice list that is followed by a “*”, a “+”, or a “?”.

7.2.9.1 Simple Choice Lists

A simple choice list maps to operations to get and set the individual elements or content particles. The operations are as defined in section 7.2.1, with the exception that a set overrides any previous set. Previously set choices will be set to null when a new choice is set. The isValid operation implementation ensures that a choice is set.

For example, the ChoiceList element above would map to:

```

valuetype ChoiceList : truncatable dom::Element {
  One getOne();
  void setOne(in One arg0);

  Two getTwo();
  void setTwo(in Two arg0);

  Three getThree();
  void setThree(in Three arg0);
}

```

7.2.9.2 *Complex Choice Lists*

In mapping complex choice lists (as with complex sequence lists) we must contrive an element valuetype that represents the choice lists. Then the mapping for “*”, “+”, or “?” is applied to that contrived element valuetype.

The name of the contrived valuetype is constructed from the names of the content particles separated by “Or”.

For example the element:

```
<!ELEMENT ComplexChoice (One | Two)? >
```

would map to the following:

```

valuetype OneOrTwo : truncatable dom::Element {
  One getOne();
  void setOne(in One arg0);

  Two getTwo();
  void setTwo(in Two arg0);
}

valuetype ComplexChoice : dom::Element {
  OneOrTwo getOneOrTwo();
  void setOneOrTwo(in OneOrTwo arg0);
  void removeOneOrTwo();
}

```

7.2.10 *Duplicate Element Names*

If the same element type is used in a sequence or choice list then these elements must be differentiated in the generated element valuetype operations. The elements are differentiated by appending a number to the generated valuetype get and set operation names. The numbering is sequential starting from 1.

For example:

```
<!Element Duplicates (One, Two, One) >
```

would map to:

```
valuetype Duplicates : truncatable dom::Element {
    One getOne1();
    void setOne1(in One arg0);

    Two getTwo();
    void setTwo(in Two arg0);

    One getOne2();
    void setOne2(in One arg0);
}
```

7.3 Mapping Attributes

Element attributes map to state in the base element valuetype and get and set operations on the element specific valuetype.

7.3.1 CDATA

An attribute of type CDATA maps to a string state member and get and set operations on the element in which it is contained.

An element defined as:

```
<!ELEMENT OS EMPTY >
<!ATTLIST OS
    Name CDATA #REQUIRED>
```

would map to:

```
valuetype OS : truncatable dom::Element {
    dom::DOMString getName();
    void setName(in dom::DOMString arg0);
}
```

7.3.1.1 *get<AttributeName>*

Returns the attribute as a DOM string. If there is no attribute set, null is returned.

7.3.1.2 *set<AttributeName>*

Sets the attribute to the given DOMString parameter.

7.3.2 ID

An ID attribute maps to a string state member and operations to get and set the ID. When the ID is set, the ID and the element that it is associated with are added to the document's IDElementMap.

```
<!ELEMENT Employee (...) >
<!ATTLIST Employee
    EmpNumber ID #REQUIRED>
```

would map to:

```
valuetype Employee : truncatable dom::Element {
    dom::DOMString getEmpName();
    void setEmpName(in dom::DOMString arg0)
        raises(dom::XMLException);
}
```

7.3.2.1 get<AttributeName>

Returns the string value box ID attribute. If there is no ID attribute set, null is returned.

7.3.2.2 set<AttributeName>

Sets the ID attribute to the string parameter and associates the id with its associated element in the document's IDElementMap.

Raises an XML exception if ID value is not unique in the scope of the document.

7.3.3 IDREF

An IDREF attribute maps to an attribute state member, operations to get and set the IDREF, and an operation to set and get the element that it points to. This attribute adds the constraint to document validation that the IDREF must point to a valid element. That is, there must be an ID entry in the document associated with the IDREF.

```
<!ELEMENT Manager (...) >
<!ATTLIST Manager
    EmpNumber IDREF #REQUIRED>
```

would map to:

```
valuetype Manager : truncatable dom::Element {
    dom::DOMString getEmpNumber();
    dom::Element getEmpNumberElement();
    void setEmpNumber(in dom::DOMString arg0);
    void setEmpNumberElement(in dom::Element arg0);
}
```

7.3.3.1 get<AttributeName>

Returns the string value box IDREF attribute. If there is no IDREF attribute set, null is returned.

7.3.3.2 *get<AttributeName>Element*

Returns the element to which the IDREF refers. Returns null if not found.

7.3.3.3 *set<AttributeName>*

Sets the IDREF attribute to the given string parameter. The IDREF must refer to an element valuetype in the document with an ID matching IDREF in order for the document to be valid.

7.3.3.4 *set<AttributeName>Element*

Sets the Element associated with the IDREF.

7.3.4 IDREFS

Like IDREF, except can set and get a sequences of IDREFs and the elements that they point to. The same constraints apply: all IDREFs must be valid for `isValid()` to return true. Maps to a sequence of strings, operations to get and set the IDREFs sequence, and operations to get and set the sequence of the elements to which the IDREFs refer.

```
<!ELEMENT WorkGroup (...) >
<!ATTLIST WorkGroup
    EmpNumbers IDREFS #REQUIRED>
```

would map to:

```
valuetype WorkGroup : dom::Element {

    sequence<string> getEmpNumbersSeq();
    sequence<Element> getEmpNumbersElementSeq();

    void setEmpNumbersSeq(in sequence<string> arg0);
    void setEmpNumbersElementSeq(in sequence<string> arg0);
}
```

7.3.4.1 *get<AttributeName>Seq*

Returns the sequence of IDREFS.

7.3.4.2 *get<AttributeName>ElementSeq*

Returns a sequence of the elements to which the IDREFS refer.

7.3.4.3 *set<AttributeName>Seq*

Sets the private state member holding the sequence of IDREF strings to the given sequence of strings. All IDREFs in the given sequence must refer to elements in the document with an associated unique ID.

7.3.4.4 *set<AttributeName>ElementSeq*

Sets the attribute state member holding the sequence of elements associated with the IDREFs in the IDREF sequence.

7.3.5 *ENTITY*

Note – See dynamic mapping.

7.3.6 *ENTITIES*

Note – See dynamic mapping.

7.3.7 *NMTOKEN*

An attribute of type NMTOKEN maps to a private string value box attribute and get and set operations on the element in which it is contained. The name of the private string value member is the name of the attribute. The value of the attribute must conform to the Nmtoken production in the XML 1.0 specification.

An element defined as:

```
<!ELEMENT OS EMPTY >
<!ATTLIST OS
    Name NMTOKEN #REQUIRED>
```

would map to:

```
valuetype OS : truncatable dom::Element {
    dom::DOMString getName();
    void setName(in dom::DOMString arg0)
        raises(dom::XMLException);
}
```

7.3.7.1 *get<AttributeName>*

Returns the string value box attribute. If there is no attribute set, null is returned.

7.3.7.2 *set<AttributeName>*

Sets the attribute to the given string value box parameter.

Raises XML exception if not conforming NMTOKEN.

7.3.8 NMTOKENS

Like NMTOKEN, except can set and get a sequence of NMTOKENs. The same constraints apply: all NMTOKENs must be valid for `isValid()` to return true. Maps to a sequence of strings and operations to get and set the NMTOKEN sequence.

```
<!ELEMENT Name (...) >
<!ATTLIST Name
    Aliases NMTOKENS #REQUIRED>
```

would map to:

```
valuetype Name : truncatable dom::Element {
    sequence<dom::DOMString> getAliasesSeq();
    void setAliases(in dom::DOMString arg0)
        raises(dom::XMLException);
    void setAliasesSeq(in sequence<dom::DOMString> arg0)
        raises(dom::XMLException);
}
```

7.3.8.1 *get<AttributeName>Seq*

Returns the sequence of strings that represents the NMTOKENS attribute.

7.3.8.2 *set<AttributeName>*

Adds an NMTOKEN string attribute to the NMTOKEN sequence. The given NMTOKEN must refer to be a valid NMTOKEN according to the XML 1.0 specification.

Raises exception if not conforming NMTOKEN.

7.3.8.3 *set<AttributeName>Seq*

Sets the base valuetype state member holding the sequence of NMTOKEN strings to the given sequence of strings. All NMTOKENs in the given sequence must be valid NMTOKEN strings as specified in the XML 1.0 specification.

Raises exception if not conforming NMTOKEN.

7.3.9 #REQUIRED

If an attribute is annotated as “#REQUIRED” then document validation must validate that the attribute has been set.

7.3.10 #IMPLIED

If an attribute is annotated as “#IMPLIED” then its presence, or lack thereof, has no bearing on the validity of the element in which it is contained.

7.3.11 #FIXED

If the attribute is annotated as “#FIXED” then the attribute maps to “get” operations only. No “set operations are generated. The mapped get operation is hardcoded to return the fixed attribute value.

7.3.12 Enumerations

An enumerated attribute maps to an attribute stored in the element base valuetype and get and set operations on the element in which it is contained. If an attempt is made to set the attribute with a value that is not in the enumeration then an exception will be thrown. The default value is set in the element’s initializer.

An element defined as:

```
<!ELEMENT CD (... ) >
<!ATTLIST CD
    Style (Jazz | Rock | Classical | Folk) "Jazz">
```

would map to:

```
valuetype CD : truncatable dom::Element {
    dom::DOMString getStyle();
    void setStyle(in dom::DOMString arg0)
        raises(dom::XMLException);
}
```

7.3.12.1 get<AttributeName>

Returns the string value box attribute.

7.3.12.2 set<AttributeName>

Sets the attribute to the given string parameter. If the given value is not in the attribute enumeration specification then an exception is thrown.

Raises XML exception if enum string is invalid.

7.3.13 Default Attributes

If a default argument is specified then the initializer of the element will initialize the attribute to the default value.

7.4 Parameter Entities

Note – See dynamic mapping.

7.5 Factories

Individual type specific nodes are created using the `create<ElementTag>` operations defined on the document valuetype. See Section 7.1.2 and 7.6.

7.6 Marshaling Framework

Documents are read into memory, into a valuetype representation, using a parser and written out using a serializer. For each document type, concrete parsers and serializers are defined which inherit from the more general `dom::XMLParser` and `dom::XMLSerializer` defined earlier in this document. A customized factory extending the `dom::XMLFactory` creates instances of the specific generated valuetypes.

For a document named *Invoice* the following interfaces would be generated within the scope of the generated document module.

```
local interface InvoiceParser : dom::XMLParser {  
    InvoiceDoc parseInvoice(in dom::DOMString XMLStream)  
        raises(dom::XMLException);  
    InvoiceDoc parseInvoice(in dom::DOMString XMLStream,  
        in InvoiceFactory)  
        raises(dom::XMLException);  
}  
  
local interface InvoiceSerializer : dom::XMLSerializer {  
    dom::DOMString serializelInvoice(in InvoiceDoc doc)  
        raises(dom::XMLException);  
}  
  
local interface InvoiceFactory : dom::XMLFactory {  
    Node createType(in DOMString type);  
}
```

7.6.1 Mapping Example

Consider the following XML document.

```
<CDCatalog>
  <CD DiscID="00756BF6">
    <Artist>Lee Konitz</Artist>
    <Title>Another Shade of Blue</Title>
    <TrackTitle>Another Shade Of Blue</TrackTitle>
    <TrackLength>10:50</TrackLength>
    <TrackTitle>Everything Happens To Me</TrackTitle>
    <TrackLength>12:15</TrackLength>
    <TrackTitle>What's New</TrackTitle>
    <TrackLength>15:49</TrackLength>
  </CD>
  <CD DiscID="007F93CC">
    <Artist>Keith Jarrett</Artist>
    <Title>Standards, Vol. 2</Title>
    <TrackTitle>So Tender</TrackTitle>
    <TrackLength>7:15</TrackLength>
    <TrackTitle>Moon And Sand</TrackTitle>
    <TrackLength>8:54</TrackLength>
    <TrackTitle>In Love In Vain</TrackTitle>
    <TrackLength>7:06</TrackLength>
  </CD>
</CDCatalog>
```

It's metadata, as represented by a DTD, might look like the following:

```
<!ELEMENT Artist ( #PCDATA ) >
<!ELEMENT Title ( #PCDATA ) >
<!ELEMENT TrackLength ( #PCDATA ) >
<!ELEMENT TrackTitle ( #PCDATA ) >

<!ELEMENT CDCatalog ( CD* ) >

<!ELEMENT CD
  ( Artist
    , Title
    , (TrackTitle, TrackLength)+
  ) >
<!ATTLIST CD DiskID CDATA #REQUIRED>
```

The IDL produced by a mapping tool, would look like:

```

module cdcatalog {
  valuetype CDCatalogDoc : truncatable dom::Document {
    CDCatalog getCDCatalogRoot();
    void setCDCatalogRoot(in CDCatalog docRoot);
    CDCatalog createCDCatalogElement();
    CD createCDElement();
    Artist createArtistElement();
    Title createTitleElement();
    TrackTitle createTrackTitleElement();
    TrackLength createTrankLengthElement();
  }

  local interface CDCatalogParser : dom::XMLParser {
    CDCatalogDoc parseCDCatalog(in dom::DOMString XMLStream)
    raises(dom::XMLException);
  }

  local interface CDCatalogSerializer : dom::XMLSerializer {
    dom::DOMString serializeCDCatalog(in CDCatalogDoc doc)
    raises(dom::XMLException);
  }

  typedef sequence<CD> CDSeq;
  typedef sequence<TrackTitleAndTrackLength>
    TrackTitleAndTrackLengthSeq;

  valuetype Artist : truncatable dom::Text {};
  valuetype Title : truncatable dom::Text {};
  valuetype TrackLength : truncatable dom::Text {};
  valuetype TrackTitle : truncatable dom::Text {};

  valuetype CDCatalog : truncatable dom::Element {
    // State declaration
    private CDSeq theCDSeq;

    //Element access operations
    CDSeq getCDSeq();
    CD getCDSeqAt(in long index);
    long getCDSeqSize();
    void setCDSeq(in CDSeq arg0);
    void replaceCDSeqAt(in CD arg0, in long index);
    void appendCDSeq(in CD arg0);
    void insertCDSeqAt(in CD arg0, in long index);
    void removeFromCDSeq(in CD arg0);
    void removeFromCDSeqAt(long arg0);
    void clearCDSeq();
  }

  valuetype TrackTitleAndTrackLength : truncatable dom::Element {
    TrackTitle getTrackTitle();
    void setTrackTitle(in TrackTitle arg0);
  }
}

```

```
    TrackLength getTrackLength();
    void setTrackLength(in TrackLength arg0);
};

valuetype CD : truncatable dom::Element {
    // Attribute access operations
    dom::DOMString getDiskID();
    void setDiskID(in dom::DOMString arg0);

    // Element access operations
    Artist getArtist();
    void setArtist(in Artist arg0);

    Title getTitle();
    void setTitle(in Title arg0);

    TrackTitleAndTrackLengthSeq getTrackTitleAndTrackLengthSeq();
    TrackTitleAndTrackLengthSeq getTrackTitleAndTrackLengthSeqAt(
        in long index);
    long getTrackTitleAndTrackLengthSeqSize();
    void setTrackTitleAndTrackLengthSeq(
        in TrackTitleAndTrackLengthSeq arg0);
    void replaceTrackTitleAndTrackLengthSeqAt(
        in TrackTitleAndTrackLength arg0,
        in long index);
    void appendTrackTitleAndTrackLengthSeq(
        in TrackTitleAndTrackLength arg0);
    void insertTrackTitleAndTrackLengthSeqAt(
        in TrackTitleAndTrackLength arg0,
        in long index);
    void removeFromTrackTitleAndTrackLengthSeq(
        in TrackTitleAndTrackLength arg0);
    void removeFromTrackTitleAndTrackLengthSeqAt(long arg0);
    void clearTrackTitleAndTrackLengthSeq();
}
}
```


This section describes the compliance points for the submission.

8.1 Mandatory Compliance Points

- An implementation may be compliant with either of the two DOM Level compliance points:
 - The IDL declarations for interfaces and valuetypes for DOM Level 1 functionality described in Chapter 4.3.
 - The IDL declarations for interfaces and valuetypes for both DOM Level 1 and Level 2 functionality described in Chapter 4.3 and Chapter 5. DOM Level 2 is a superset of DOM Level 1.
- The DOM extensions described in Chapter 4.4 and 4.5.
- The static mapping declarations described in Chapter 7.

8.2 Optional Compliance Points

None.

References

9

[CORBA] CORBA OMG standard. <http://www.omg.org>

[CORBA COMPONENT MODEL] Corba specification in the finalization task force. OMG document Orbos/99-07-02. Section 2.4.1.7 contains the Valuetypes MOF metamodel.

[DOM 1] Document Object Model Level 1 W3C recommendation.
<http://www.w3.org/TR/REC-DOM-Level-1>. See also the errata
<http://www.w3.org/DOM/updates/REC-DOM-Level-1-19981001-errata>

[DOM 2] Document Object Model Level 2 W3C candidate recommendation.
<http://www.w3.org/TR/DOM-Level-2/>

[MOF] Meta Object Framework OMG standard.
<http://cgi.omg.org/techprocess/meetings/schedule/tech2a.html#mod>

[XMI] XML Metadata Interchange OMG standard.
<http://cgi.omg.org/techprocess/meetings/schedule/tech2a.html#mod>

[XML Value RFP] OMG RFP orbos/99-08-20.

[XMI production of XML Schema RFP] OMG RFP ad/00-01-04.

[Schema] XML Schema working drafts:

- Part 0 (primer) <http://www.w3.org/TR/xmlschema-0/>
- Part 1 (structures) <http://www.w3.org/TR/xmlschema-1/>
- Part 2 (data types) <http://www.w3.org/TR/xmlschema-2/>

