WIDL and XML RPC

- Application interoperability
- Web Interface Definition Language (WIDL)
- XML Remote Procedure Call (RPC)
- WIDL specification on CD-ROM

WIDL and XML RPC

- Application interoperability
- Web Interface Definition Language (WIDL)
- XML Remote Procedure Call (RPC)
- WIDL specification on CD-ROM

Chapter 38

XML goes a long way toward allowing applications to interoperate, but some think it needs to go a WIDL further. Among them are webMethods, Inc., `http://www.webmethods.com`, who sponsor this chapter, and Joe Lapp, who prepared it.

Engineers numbered 12-345-68 through 23-457-89 at Oops E-Commerce Corporation say "XML is the solution to interoperability." These engineers gang up on the managers until the corporate gears succumb and reverse direction. Soon the sales reps are saying the words "universal data format" more often than the words "object-oriented." Oops XML-enables its popular Loops product, renames the product to Xoops, and then ships Xoops out the door.

Over the following weeks we eavesdrop on the support engineers: "Well, if you have Company Q's product you can use our XML feature with it... Well, to get it to talk to your purchasing system, you'll have to XML-enable the purchasing system... Well, their program uses a different DTD from ours, so Xoops won't interoperate with it."

Woops, Oops goofed with Xoops: XML alone is not quite enough.

38.1 | XML alone is not quite enough

A client that hands a server data must tell the server what to do with the data. The client does this by naming a service. A client must also understand the data that the service returns. Two applications may communicate only if they agree on the names of the services and on the types of the input and the output data.

Furthermore, applications must agree on how to represent this data in the messages that transfer between them. XML provides a way to represent the data, but it does not associate input data and output data with service names, and it does not provide a way to map between message types. Something is missing.

38.1.1 *The missing piece*

The obvious solution to the problem is to associate input DTDs with output DTDs and to give these associations service names. This does provide enough information for two applications to communicate, but it requires both applications to be XML-enabled and it requires the applications to conform to the same DTDs. While there may not be many XML-enabled applications right now, eventually there will be, but it is unlikely that all will agree on the same DTDs.

A better solution to the interoperability problem is to define application interfaces in an abstract way. CORBA, DCOM, and DCE have all taken this approach, and in these systems the abstractions are known as interface specifications.

Interface specifications allow developers to create different but compatible implementations of interfaces. In CORBA, DCOM, and DCE interface specifications allow applications written in different programming languages to communicate. We need to take this a step further. We must also bridge between applications whose XML messages conform to different DTDs.

The missing piece is an IDL - an *Interface Definition Language.* An IDL is a language in which interface specifications are written.

webMethods, Inc. has specified an IDL for this purpose, an IDL called WIDL. WIDL interface specifications enable middleware to map transparently between application interfaces and XML message DTDs. By delegating XML intelligence and accessibility issues to IDL-aware middleware, we also simplify the application. An IDL such as WIDL allows us to maximize an application's accessibility.

38.1.2 *The role of WIDL*

WIDL is an acronym for *Web Interface Definition Language.* It is an IDL that is expressed in XML. OMG IDL and Microsoft IDL are other examples of IDLs, but there are important differences between WIDL and conventional IDLs.

WIDL differs from other IDLs primarily because it satisfies the 80/20 rule. It provides 80% of the capability of a conventional IDL with only 20% of the complexity. WIDL is consequently easy to learn, easy to read, and relatively easy to implement.

This fact provides WIDL with a potentially large user base, but still leaves room for more sophisticated IDLs, including new ones based on XML. WIDL also goes a step further than conventional IDLs by requiring all data items to have names, which simplifies the process of translating documents into interfaces.

webMethods originally developed WIDL to wrap Web sites within APIs, thereby giving applications programmatic access to the Web. Consequently, the WIDL 1.x and 2.x specifications defined a single language that both specified interfaces and defined how interface specifications map onto a Web site.

WIDL 3.0 places the interface specification and the document-mapping implementation in separate XML documents. WIDL 3.0 therefore defines two components: an IDL component and a document-mapping component. Together these components allow applications to communicate over a network regardless of the programming languages in which the applications are written, regardless of whether the applications speak XML, and regardless of the DTDs to which XML-speaking applications conform.

38.2 | WIDL the IDL

Let's take a look at the IDL component of WIDL 3.0. Example 38-1 shows a short but complete example of a WIDL 3.0 interface specification.

Example 38-1. A WIDL 3.0 interface specification.

```
<WIDL NAME="com.Fortunes-R-Us.Purchasing" VERSION="3.0">
  <RECORD NAME="FortuneOrder">
    <VALUE NAME="accountID" TYPE="i4"/>
    <VALUE NAME="zodiacSign"/>
  </RECORD>
  <RECORD NAME="FortuneReceipt">
    <VALUE NAME="orderNumber" TYPE="i4"/>
    <VALUE NAME="fortune"/>
    <VALUE NAME="accountBalance" TYPE="r4"/>
  </RECORD>
  <METHOD NAME="orderFortune" INPUT="FortuneOrder"
      OUTPUT="FortuneReceipt" RETURN="orderNumber"/>
</WIDL>
```

A WIDL document specifies a single interface. Example 38-2 is a DTD that defines WIDL documents sufficiently for our purposes.

Interfaces should have names that are unique within their scope of use. Naming an interface relative to the reverse order of a domain name provides one way to accomplish this. A client may then identify interfaces by name.

A WIDL element contains one or more RECORD or METHOD elements.

38.2.1 *Methods*

The METHOD element identifies a service that the client may invoke.

Method names must be unique within the document. Methods may optionally have input and output parameters, as indicated by the optional INPUT and OUTPUT attributes.

The INPUT attribute provides a link to a RECORD element that enumerates the method's input parameters. The OUTPUT attribute provides a link to a RECORD element that enumerates the method's output parameters. The tag may optionally indicate that one of the output parameters is the return value of the method when the interface is implemented in a programming language. Methods may also identify the exceptions that they raise in order to report method invocation failures.

Example 38-2. WIDL interface DTD.

```
<!ELEMENT WIDL       (RECORD | METHOD)+ >
<!ATTLIST WIDL
          NAME       CDATA #REQUIRED
          VERSION    CDATA #FIXED "3.0"
>
<!ELEMENT METHOD     EMPTY>
<!ATTLIST METHOD
          NAME       CDATA #REQUIRED
          INPUT      CDATA #IMPLIED
          OUTPUT     CDATA #IMPLIED
          RETURN     CDATA #IMPLIED
>
<!ELEMENT RECORD     (VALUE | LIST | RECORDREF)* >
<!ATTLIST RECORD
          NAME       CDATA #REQUIRED
          BASE       CDATA #IMPLIED
>
     <!-- Parameters -->
<!ELEMENT VALUE      EMPTY >
<!ATTLIST VALUE
          NAME       CDATA #REQUIRED
          TYPE       CDATA "string"
          DIM        NMTOKEN 0
>
<!ELEMENT LIST       EMPTY >
<!ATTLIST LIST
          NAME       CDATA #REQUIRED
          DIM        NMTOKEN 0
>
<!ELEMENT RECORDREF EMPTY >
<!ATTLIST RECORDREF
          NAME       CDATA #REQUIRED
          RECORD     CDATA #IMPLIED
          DIM        NMTOKEN 0
>
```

38.2.1 *Records*

A RECORD element represents a record and conforms to the DTD shown in Example 38-2. Record names must be unique within a document. A record consists of a collection of zero or more parameter elements, each of which must have a unique name within the scope of the record. If the record provides a BASE attribute, the record inherits all of the named parameter elements found within the RECORD element to which the attribute points.

The parameter element types are VALUE, LIST, and RECORDREF.

VALUE
: An element that represents lexical data and has an optional TYPE attribute that identifies the datatype. Datatypes include strings ("string"), integers ("i4"), and floats ("r4").

LIST
: A LIST element represents a vector of arbitrary size consisting of an arbitrary set of types.

RECORDREF
: The RECORDREF element identifies a RECORD element that nests within the RECORDREF's parent record.

Parameters have an optional DIM attribute. When DIM has a value of "1" or "2" the parameter represents a single- or two-dimensional array. When the attribute is absent, the value defaults to "0" to indicate that the parameter is a single data item and not an array.

WIDL provides only a small number of simple data types. These data types are sufficient to represent most of the types available to programming languages. WIDL is compatible with other data definition languages such as XML-Data and Resource Description Framework (RDF), so WIDL may accommodate the sophisticated schema languages that are emerging. This allows WIDL to support complex data types without itself becoming complex.

38.3 | Remote procedure calls

WIDL provides the information that applications need to communicate, but it does not perform the actual communication. An application that requests a service of another application must issue a Remote Procedure Call, or RPC, to the other application. An application issues an RPC by packaging a message, sending the message to the other application, and then waiting for the reply message.

The RPC mechanism requires the applications to agree on the form of the messages and on the transfer protocol by which the messages travel. HTTP provides a POST method that allows a client to submit a document to a server and to receive a document in response, so HTTP is a candidate protocol. Since HTTP is nearly ubiquitous and since it tunnels through firewalls, it's obvious that we should use HTTP. The question is, should XML be the message form?

IIOP and DCE are both industry standards for RPC messages. Either of these would work, as it is possible to send them over HTTP. We might notice that these message representations are inflexible: senders and receivers must agree on how a message decomposes data into arguments, including the positions of the individual arguments and the structures of these arguments.

Yet if the message representation were XML, the applications would still have to agree on the DTDs to which the messages conformed. Just as applications that use different IIOP or DCE message types cannot communicate, applications that use different DTDs cannot communicate. Without looking more closely, we might be inclined to conclude that XML is all hype after all.

However, we *are* going to look more closely. These problems do afflict XML, IIOP, and DCE alike. No reneging here. When we take that closer look we find that, unlike IIOP and DCE, XML provides a way to solve the problem.

That is, XML provides a way to ensure that so long as two applications agree to conform to the same abstract interface specification, then those two applications may communicate – even if the applications are hard-coded to use different DTDs.

38.3.1 *Representing RPC messages in XML*

XML is an ideal notation for RPC messages because it allows us to label the individual data constituents of a message semantically. These labels are XML's tags.

The only semantic labels available in IIOP and DCE are the numeric positions of the constituents. IIOP and DCE do not allow data to move to new positions and they do not allow data to grow or shrink in unforeseen ways. They also do not allow applications to discover the absence of data from a message or to introduce new data items into a message independently.

But the greatest benefit that XML brings to RPC is that XML moves a significant amount of information about a message into the message itself. It is a benefit because it moves an equal amount of information out of the programs that process the messages. This simplifies the programs that integrate applications.

In all probability, industries will never completely agree on standard interfaces or standard DTDs, so it will always be necessary to translate between interfaces. XML provides interoperability by enabling a new class of middleware to serve as generic application integrators.

38.3.1 *Generic and custom message DTDs*

There are two ways to represent RPC messages in XML. A generic document type is capable of representing any message. The interface specification determines the form that a message takes in a generic document type.

More specifically, the definition of a method uniquely determines the DTDs of the request and reply messages that correspond to the method.

On the other hand, a custom document type is designed only to contain the inputs or the outputs of a particular kind of service. There are many possible custom document type definitions for a given interface method.

Let's look at a few examples that are based on the Fortunes-R-Us purchasing interface shown in Example 38-1. Example 38-3 contains three RPC messages.

The first portrays what an instance of a generic document type might look like for a message that invokes the "orderFortune" method. The same document type scheme might be used for the reply message, which is the second message of Example 38-3. The third message shown is an instance of a custom-DTD reply.

There are many possible generic XML document types, and we can expect to see industries creating them and using them. There are also many possible custom document types for any given method. We can also expect to see applications using custom document types to message other applications.

The trick is to ensure that we can integrate applications that use different document types to represent the same information. Without this we do not have interoperability. XML makes it feasible to provide large-scale interop-

Example 38-3. Generic- and custom-DTD RPC messages.

```
<RPC TYPE="REQUEST">
  <VALUE NAME="accountID" TYPE="i4">2001</NUMBER>
  <VALUE NAME="zodiacSign">Aquarius</VALUE>
</RPC>

<RPC TYPE="REPLY">
  <VALUE NAME="orderNumber" TYPE="i4">438553</NUMBER>
  <VALUE NAME="fortune">You will use XML for RPC</VALUE>
  <VALUE NAME="accountBalance" TYPE="r4">65.00</NUMBER>
</RPC>

<FORTUNE-RECEIPT>
  <orderNumber>438553</orderNumber>
  <fortune>You will use XML for RPC</fortune>
  <accountBalance>65.00</accountBalance>
</FORTUNE-RECEIPT>
```

erability, but only if we design our messages so that integration middleware may robustly identify data constituents by label.

38.4 | Integrating applications

WIDL and XML RPC together enable middleware to integrate applications. We'll use the term integration server to refer to middleware that assumes this kind of responsibility.

A WIDL interface specification supplies an integration server with the information the server needs to map between XML RPC messages and native application interfaces. Interface specifications do not themselves define the mappings, but they provide a common language in which to express them.

Figure 38-1 shows how integration servers connect applications.

Integration servers need to integrate a wide variety of application interfaces. One application may implement an interface as a set of Java or C++ methods. Another may implement an interface as a set of functions in C.

Another application may input and output XML documents conforming to custom DTDs. Still another may input and output XML documents in the form of generic RPC messages. Integrating applications requires bridging between programming languages and document representations.

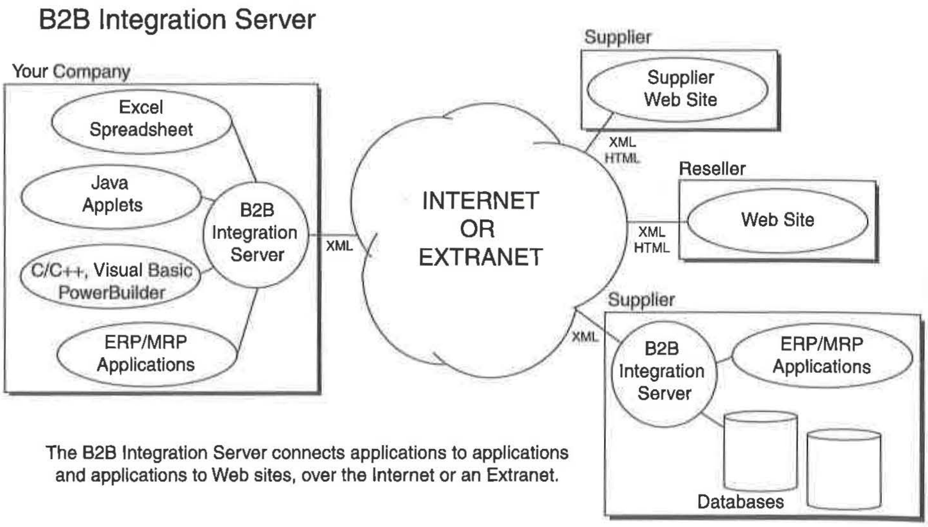


Figure 38-1 Connecting applications with XML RPC and integration servers.

38.4.1 *Stubs*

Conventional RPC bridges programming languages through code snippets known as *stubs*. A stub translates between the details of an interface and a common data representation. One side of a stub speaks the language that is native to an application and the other side speaks a common data representation.

By connecting the data representation ends of two stubs, one may bridge between any two programming languages. In a client stub, the language-specific side consists of a set of APIs (functions) that the client may call. In a server stub, the language-specific side calls APIs that the server itself exposes.

Figure 38-2 illustrates this property of stubs by portraying four stub pairings. Here, XML is the common data representation, but in the usual case intervening middleware will hide knowledge of XML from the stubs.

In diagram (a) an application written in Java is communicating with another application written in Java. Diagrams (b) and (c) show that the same application may also communicate with applications written in C++ or C. Diagram (d) depicts the Java application communicating with an application that speaks XML. In this last scenario the XML-speaking application has no stub, since the XML messages pass directly to the application.

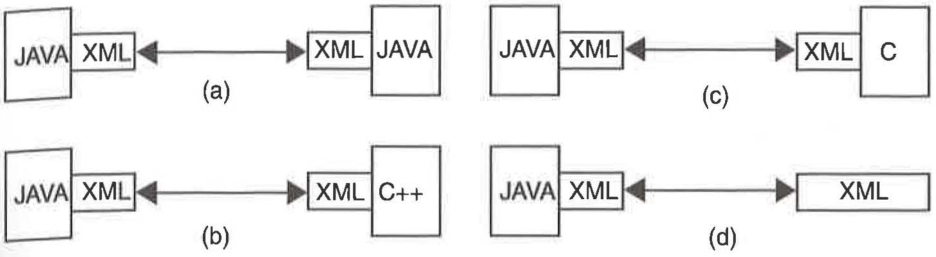


Figure 38-2 Using stubs to make applications interoperable.

Figure 38-3 portrays how a developer uses stubs to integrate applications. A developer generates an interface specification in WIDL and then runs the specification through a WIDL compiler.

The WIDL compiler generates two source files in a programming language of the developer's choice. Both files are stubs, but one file is a client stub and the other is a server stub. The developer then links the appropriate stub into the client or server application. The stubs free the application from knowledge of XML and allow middleware to map transparently between interfaces and different XML document types.

38.4.2 *Document mapping*

The document-mapping component of WIDL defines mappings between interfaces and XML or HTML documents. This is the portion that provides the bridge between XML RPC messages and application APIs; that is, the portion that makes the different XML document types indistinguishable to the application. webMethods originally developed this facility to encapsulate HTML-based Web sites within APIs, but because XML does a better job of labeling data than HTML does, the technology reaps more benefits from XML.

WIDL document-mapping does its job through *bindings*. A binding specifies how to map raw data into an RPC message or vice versa, where "raw data" means "data represented in a way that is natural to a program-

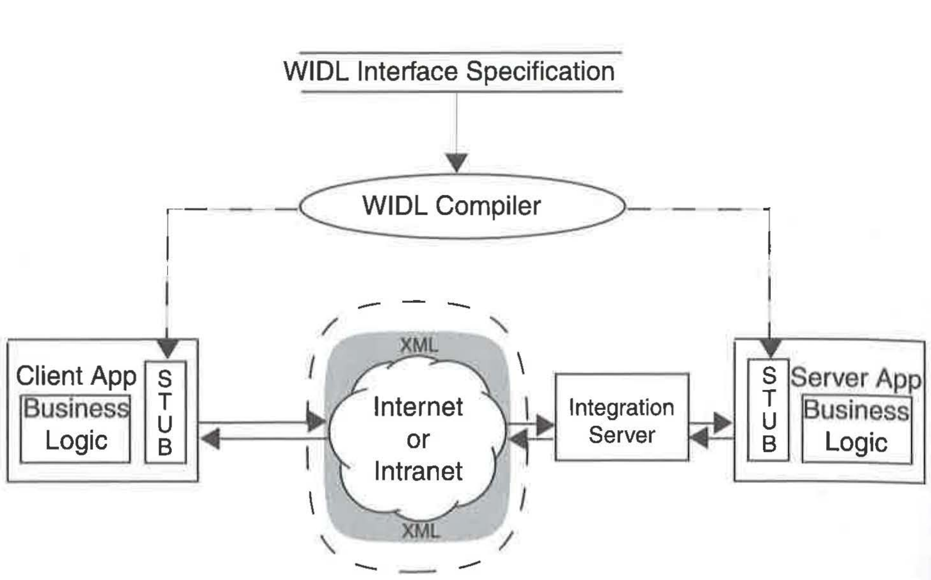


Figure 38-3 Using WIDL for RPC over the Web.

ming language". The best way to make sense of this is to look at an example, so consider Example 38-4.

Example 38-4. A WIDL binding.

```
<OUTPUT-BINDING NAME="OrderReplyBinding">
  <VALUE NAME="orderNumber" TYPE="i4">
            doc.orderNumber[0].text</VALUE>
  <VALUE NAME="fortune">doc.fortune[0].text</VALUE>
</OUTPUT-BINDING>
```

This binding applies to the custom-DTD reply message of Example 38-3. Each VALUE element corresponds to a data item that the binding extracts from the message. In this case the binding extracts two strings, but bindings may extract other data types, including records and even XML documents.

Upon receiving the reply message, middleware applies this binding and passes the two strings to the application. Since the application ordered the fortune by issuing a function call on a client stub, the stub returns the strings to the application as output parameters of the function. Middleware completely shields the application from knowledge of XML and from dependence on a specific XML document type.

In this example, the binding only retrieves the order number and the fortune from the reply message, indicating that the application cannot utilize the account balance. The content of each VALUE element is a query, expressed in a document query language, that specifies where to find these items within the message. In this particular case, the query uses the webMethods Object Model, but WIDL is compatible with other query languages as well.

A binding may also define how to translate data into an RPC message. WIDL supports several forms of messages. For request messages, the binding may have the data submitted via the HTTP GET or POST methods, thus providing the data as CGI query parameters. The binding may also have the data submitted as an XML or an HTML message, constructing the message from a particular template. Templates are a straightforward way to generate XML.

Bindings provide a simple way to make applications compatible with a variety of XML message DTDs. Bindings are most useful with custom document types, since it is possible to hard-code document-mapping for generic document types. Generic document types do not require the flexibility that bindings provide, and by hard-coding them middleware can provide more efficient document-mapping.

An integration server puts bindings to work by using them to mask differences in XML document types. By connecting the variable names of bindings to parameter names in interface specifications, an integration server may map any XML document type into any programming language.

To get a feel for the benefits of this capability, take a look at Figure 38-4. Here industries and businesses have defined a variety of DTDs to which different RPC document types conform. The interface defined with WIDL captures a superset of the services and data available through the DTDs. Although different client applications use different XML document types, the integration server is able to bridge these differences to make the application universally accessible.

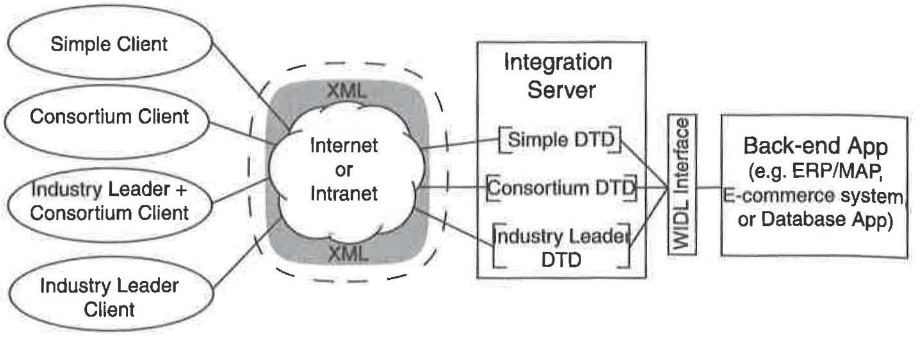


Figure 38-4 Using WIDL to make different XML messages interoperable.

38.5 | Interoperability attained

WIDL, XML RPC, and integration servers are the pieces that provide application interoperability. With them one can make any application accessible over a network via XML and HTTP.

One can also make a single application available to client applications that use different XML message formats. Or one can upgrade an application, or substitute one application for another, and still allow all previous clients to communicate with the new application.

These capabilities should give us second thoughts about hard-coding servers to use specific XML document types. Servers should leave document type decisions to middleware, empowering middleware to make the server widely accessible.

Chapter 39

XML-Data is the name of a proposal for a DTD schema language, a new way to create and augment document type definitions. This chapter is sponsored by Microsoft Corporation, `http://www.microsoft.com`.

The Internet holds within it the potential for integrating all information into a global network (with many private but integrated domains), promising access to information any time and anywhere. However, this potential has yet to be realized. At present, the Internet is merely an access medium.

To realize the Internet's potential, we need to add intelligent search, data exchange, adaptive presentation, and personalization. The Internet must go beyond setting an information access standard and must set an information *understanding* standard, which means a standard way of representing data so that software can better search, move, display, and otherwise manipulate information currently hidden in contextual obscurity.

XML is an important step in this direction. XML is a standardized notation for representing structured information. It is well-founded theoretically and is based on extensive industry experience. Although XML documents are simple, readily-transmitted character strings, the notation easily depicts a tree structure. A tree is a natural structure that is richer than a simple flat list, yet also respectful of cognitive and data processing requirements for economy and simplicity.

Valid XML documents belong to classes – document types – that determine the tree structure and other properties of their member documents. The properties of the classes themselves comprise their document type definitions, or DTDs, which serve the same role for documents that schemas do for databases.

And that is where the potential for enhancing the Web lies.

Today, the only standardized method of creating document type definitions is through the use of markup declarations, a specialized syntax used only for this purpose. What is needed is a method of augmenting the existing set of DTD properties with additional properties that will enable the goal of true information understanding.

Fortunately, there are ways to accomplish this goal by using XML itself. The W3C XML Working Group has agreed to work on a *DTD schema language* for XML. The DTD schema language will provide a means of using XML instances to define augmented DTDs.

As a contribution to this effort, ArborText, DataChannel, Inso, and Microsoft have co-authored the *XML-Data* submission to the W3C.

XML-Data is a notation, in the form of an XML document, that is both an alternative to markup declarations for writing DTDs and a means of augmenting DTDs with additional capabilities. For example:

- *XML-Data* supports rich data types, allowing for tighter validation of data and reduced application effort. Developers can use a list of standard data types, such as numbers or ISO 8601 dates, or define their own.
- Through the namespaces facility, *XML-Data* improves expressiveness, ensuring the existence of uniquely qualified names.
- *XML-Data* provides for greater and more efficient semantic facilities by incorporating the concept of inheritance, enabling one schema to be based on another. For instance, a bookstore purchase order schema could be based on a general purpose electronic-commerce purchase order schema.

Since *XML-Data* uses XML instance syntax, there are a number of other benefits:

- The same tools that are used to parse XML can be used to parse the *XML-Data* notation.

- As the syntax is very similar to HTML, it should be easy for HTML authors to learn and read.
- It is easily extensible.

The text of the *XML-Data* proposal follows, as contained in *W3C Note 05 Jan 1998.* A browseable version, can be found on the CD-ROM and at `http://www.w3.org/TR/1998/NOTE-XML-data`. That version identifies the individual authors and others whose help and contributions to the proposal the authors acknowledged.

39.1 | Introduction

Schemas define the characteristics of classes of objects. This paper describes an XML vocabulary for schemas, that is, for defining and documenting object classes. It can be used for classes which as strictly syntactic (for example, XML) or those which indicate concepts and relations among concepts (as used in relational databases, KR graphs and RDF). The former are called "syntactic schemas;" the latter "conceptual schemas."

For example, an XML document might contain a "book" element which lexically contains an "author" element and a "title" element. An XML-Data schema can describe such syntax. However, in another context, we may simply want to represent more abstractly that books have titles and authors, irrespective of any syntax. XML-Data schemas can describe such conceptual relationships. Further, the information about books, titles and authors might be stored in a relational database, in which XML-Data schemas describe row types and key relationships.

One immediate implication of the ideas in this paper is that XML document types can now be described using XML itself, rather than DTD syntax. Another is that XML-Data schemas provide a common vocabulary for ideas which overlap between syntactic, database and conceptual schemas. All features can be used together as appropriate.

Schemas are composed principally of declarations for:

Concepts

Classes of objects

- Class hierarchies
- Properties

Relationships

- Indicated by primary key to foreign key matching
- Indicated by URI

XML DTD Grammars and Compatibility

- grammatical rules governing the valid nesting of the elements and attributes
- attributes of elements
- internal and external entities, represented by intEntityDecl and extEntityDecl
- notations, represented by notationDcl

Datatypes giving parsing rules and implementation formats.

Mapping rules allowing abbreviated grammars to map to a conceptual data model.

39.2 | The Schema Element Type

All schema declarations are contained within a schema element, like this:

```
<?XML version='1.0' ?>
<?xml:namespace
    name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/"
    as="s"/?>
<s:schema id='ExampleSchema'>
  <!-- schema goes here. -->
</s:schema>
```

The namespace of the vocabulary described in this document is named "urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/".

39.3 | The *ElementType* Declaration

The heart of an XML-Data schema is the *elementType* declaration, which defines a class of objects (or "type of element" in XML terminology). The *id* attribute serves a dual role of identifying the definition, and also naming the specific class.

```
<elementType id="author"/>
```

Within an elementType, the *description* subelement may be used to provide a human-readable description of the elements purpose.

```
<elementType id="author">
  <description>The person, natural or otherwise, who wrote
                the book.</description>
</elementType >
```

39.4 | Properties and Content Models

Subelements within *elementType* define characteristics of the classs members. An XML "content model" is a description of the contents that may validly appear within a particular element type in a document instance.

```
<elementType id="author">
    <string/>
</elementType>

<elementType id="Book">
    <element type="#author" occurs="ONEORMORE"/>
</elementType>
```

The example above defines two elements, author and book, and says that a book has one or more authors. The author element may contain a string of character data (but no other elements). For example, the following is valid:

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
</Book>
```

Within an elementType, various specialized subelements (element, group, any, empty, string etc.) indicate which subelements (properties) are allowed/required. Ordinarily, these imply not only the cardinality of the subelements but also their sequence. (We discuss a means to relax sequence later.)

39.4.1 *Element*

Element indicates the containment of a single element type (property). Each *element* contains an *href* attribute referencing another *elementType*, thereby including it in the content model syntacticly, or declaring it to be a property of the object class conceptually. The element may be required or optional, and may occur multiple times, as indicated by its *occurs* attribute having one of the four values "REQUIRED", "OPTIONAL", "ZEROORMORE" or "ONEORMORE". It has a default of "REQUIRED".

```
<elementType id="Book">
    <element type="#title" occurs="OPTIONAL"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>
```

The example above describes a book element type. Here, each instance of a book *may* contain a title, and *must* contain one or more authors.

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
</Book>
```

When we discuss type hierarchies, later, we will see that an element type may have subtypes. If so, inclusion of an element type in a content model permits elements of that type directly and all its subtypes.

39.4.2 *Empty, Any, String, and Mixed Content*

Empty and *any* content are expressed using predefined elements *empty* and *any*. (*Empty* may be omitted.) *String* means any character string not containing elements, known as "PCDATA" in XML. *Any* signals that any mixture of subelements is legal, but no free characters. *Mixed* content (a mixture of parsed character data and one or more elements) is identified by a *mixed* element, whose content identifies the element types allowed in addition to parsed character data. When the content model is mixed, any number of the listed elements are allowed, in any order.

```
<?XML version='1.0' ?>
<?xml:namespace
        name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/"
        as="s"/?>
<s:schema>

    <elementType id="name">
        <string/>
    </elementType>

    <elementType id="Person">
        <any/>
    </elementType>

    <elementType id="author">
        <string/>
    </elementType>

    <elementType id="titlePart">
        <string/>
    </elementType>

    <elementType id="title">
        <mixed><element type="#titlePart"/></mixed>
    </elementType>

    <elementType id="Book">
        <element type="#title" occurs="OPTIONAL"/>
        <element type="#author" occurs="ONEORMORE"/>
    </elementType>

</s:schema>

...

<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and<titlePart>Work</titlePart></title>
</Book>
```

Here, *book* is defined to have an optional *title* and one or more *authors.* The *name* element has content model of *any,* meaning that free text is not allowed, but any arrangement of subelements is valid. The *content model* of *title* is *mixed,* allowing a free intermixture of characters and any number of *titleParts*. The author, *name* and *titleParts* elements have a *content model* of *string.*

39.4.3 *Group*

Group indicates a set or sequence of elements, allowing alternatives or ordering among the elements by use of the groupOrder attribute. The group as a whole is treated similarly to an element.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <group occurs="OPTIONAL">
        <element type="#preface"/>
        <element type="#introduction"/>
    </group>
</elementType>
```

In the above example, if a preface or introduction appears, both must, with the preface preceding the introduction. Each of the following is valid:

```
<Book>
    <author>Henry Ford</author>
</Book>

<Book>
    <author>Henry Ford</author>
    <preface>Prefatory text</preface>
    <introduction>This is a swell book.</introduction>
</Book>
```

Sometimes a schema designer wants to relax the ordering restrictions among elements, allowing them to appear in any order. This is indicated by setting the groupOrder attribute to “AND”:

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <group groupOrder="AND" occurs="OPTIONAL">
        <element type="#preface"/>
        <element type="#introduction"/>
    </group>
</elementType>
```

Now the following is also valid:

```
<Book>
    <author>Henry Ford</author>
    <introduction>This is a swell book.</introduction>
    <preface>Prefatory text</preface>
</Book>
```

Finally, a schema can indicate that any one of a list of elements (or groups) is needed. For example, either a preface *or* an introduction. The groupOrder attribute value “OR” signals this.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <group groupOrder="OR">
        <element type="#preface"/>
        <element type="#introduction"/>
    </group>
</elementType>
```

Now each of the following is valid:

```
<Book>
    <author>Henry Ford</author>
    <preface>Prefatory text</preface>
</Book>

<Book>
    <author>Henry Ford</author>
    <introduction>This is a swell book.</introduction>
</Book>
```

39.4.4 *Open and Closed Content Models*

XML typically does not allow an element to contain content unless that content was listed in the model. This is useful in some cases, but overly in others in which we would like the listed content model to govern the cardinality and other aspects of whichever subelements are explicitly named, while allowing that other subelements can appear in instances as well.

The distinction is effected by the *content* attribute taking the values "OPEN" and "CLOSED." The default is "OPEN" meaning that all element types not explicitly listed are valid, without order restrictions. (This idea has a close relation to the Java concept of a final class.)

For example, the following instance data for a book, including the unmentioned element *copyrightDate* would be valid given the content models declared so far, because they have all been *open*.

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
    <copyrightDate>1922</copyrightDate>
</Book>
```

However, had the content model been declared closed, as follows, the *copyrightDate* element would be invalid.

```
<elementType id="Book" content="CLOSED">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <group groupOrder="SEQ" occurs="OPTIONAL">
        <element type="#preface"/>
        <element type="#introduction" occurs="REQUIRED"/>
    </group>
</elementType>
```

A closed content model does not allow instances to contain any elements or attributes beyond those explicitly listed in the elementType declaration.

39.5 | Default Values

An element with occurs of REQUIRED or OPTIONAL (but not ONEORMORE or ZEROORMORE) can have a default value specified.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <element type="#ageGrp" occurs="OPTIONAL">
        <default>adult</default>
    </element>
</elementType>
```

The default value is implied for all element *instances* in which it is syntactically omitted.

To indicate that the default value is the only allowed value, the *presence* attribute is set to "FIXED".

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <element type="#ageGrp" occurs="OPTIONAL" presence="FIXED">
        <default>ADULT</default>
    </element>
</elementType>
```

Presence has values of "IMPLIED," "SPECIFIED," "REQUIRED," and "FIXED" with the same meanings as defined in XML DTD.

39.6 | Aliases and Correlatives

ElementTypes can be know be different names in different languages or domains. The equivalence of several names is effected by the sameAs attribute, as in

```
<elementTypeEquivalent id="livre" type="#Book"/>
<elementTypeEquivalent id="auteur" type="#author"/>
```

Elements are used to represent both primary object types (nouns) and also properties, relations and so forth. Relations are often known by two names, each reflecting one direction of the relationship. For example, husband and wife, above and below, earlier and later, etc. The *correlative* element identifies such a pairing.

```
<elementType id= "author">
    <string/>
</elementType>

<elementType id= "wrote">
    <correlative type="#author" />
    <string/>
</elementType>
```

This indicates that "wrote" is another name for the "author" relation, but from the perspective of the person, not the book. That is, the two fragments below express the same fact:

```
<Person>
    <name>Henry Ford</name>
</Person>

<Book>
    <title>My Life and Work</title>
    <author>Henry Ford</author>
</Book>

...

<Person>
    <name>Henry Ford</name>
    <wrote>My Life and Work</wrote>
</Person>

<Book>
    <title>My Life and Work</title>
</Book>
```

A correlative may be defined simply to document the alternative name for the relation. However, it may also be used within a content model where it permits instances to use the alternative name. Further it may to establish constraints on the relation, indicate key relationships, etc.

39.7 | Class Hierarchies

ElementTypes can be organized into categories using the *superType* attribute, as in

```
<elementType id="price">
    <string/>
</elementType>

<elementType id="ThingsIveBoughtRecently">
    <element type="#price"/>
</elementType>

<elementType id="PencilsIveBoughtRecently">
    <superType type="#ThingsIveBoughtRecently"/>
    <element type="#price"/>
</elementType>

<elementType id="BooksIveBoughtRecently">
    <superType type="#ThingsIveBoughtRecently"/>
    <element type="#price"/>
</elementType>
```

This simply indicates that, in some fashion, *PencilsIveBoughtRecently* and *BooksIveBoughtRecently*are subsets of *ThingsIveBoughtRecently.* It implies that every valid instance of the subset is a valid instance of the superset. The superset type must have an *open* content model.

There are restrictions that should be followed, based on the principle that all instances of the species (subtype) must be instances of the genus (supertype):

- The genus type must have content="OPEN".
- It must have either no groups or only groups with groupOrder="AND" (that is, no order constraints).
- You can add new elements and attributes.
- Occurs cardinality can be decreased but not increased.
- Ranges and other constraints are cummulative, that is, all apply (though the exact effect of this depends on the semantics of the constraint type).
- Default values can be made FIXED defaults.

To indicate that the content model of the subset should inherit the content model of a superset, we use a particular kind of superType called "genus" of which only one is allowed per ElementType. This copies the content model of the referenced element type and permits addition of new elements to it. Further, sub-elements occurring in the superset type, if declared again, are replaced by the newer declarations.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>

<elementType id="BooksIveBoughtRecently">
    <genus type="#Book"/>
    <superType type="#ThingsIveBoughtRecently"/>
    <element type="#price"/>
</elementType>
```

The above has the same effect as

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>

<elementType id="BooksIveBoughtRecently">
    <superType type="#Book"/>
    <superType type="#ThingsIveBoughtRecently"/>
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <element type="#price"/>
</elementType>
```

39.8 | Elements which are References

ElementTypes and the content model elements defined so far are sufficient to declare a tree structure of elements. However, some elements such as "author" are *not only* usable on their own, they also act as references to other elements. For example, "Henry Ford" is the value of the *author* subelement of a *book* element. "Henry Ford" is also the value of the *name* element in a *person* element, and it can be used to connect these two.

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
</Book>

<Person><name>Henry Ford</name></Person>

<Person><name>Samuel Crowther</name></Person>
```

In this capacity, such subelement are often referred to as *relations* when using "knowledge representation" terminology or "keys" when using database terms. (The meaning of "relation" and "key" are slightly different, but the fact which the terms recognize is the same.)

To make such references explicit in the schema, we add declarations for *keys* and *foreign keys.*

```
<elementType id="name">
    <string/>
</elementType>

<elementType id="Person">
    <element id="p1" type="#name"/>
    <key id="k1"><keyPart href="#p1"/></key>
</elementType>

<elementType id="author">
    <string/>
    <foreignKey range="#Person" key="#k1"/>
</elementType>

<elementType id="title">
    <string/>
</elementType>

<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>
```

The *key* element within *person* tells us that a person can be uniquely identified by his *name*. The *foreignKey* element within the *author* element definition says that the contents of an author element are a foreign key indentifying a person by *name.*

An uninformed user agent can still display the string "Henry Ford" even if it cannot determine that is supposed to be a person. A savvy agent that reads the schema can do more. It can locate the actual person.

This is the information needed for a *join* in database terminology.

This mechanism not only handles the typical way in which properties are expressed in databases, it also handles all cases in which the contents of an element are to be interpreted as strings from a restricted vocabulary, such as enumerations, XML nmtokens, etc.

```
<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
    <lccn>HD9710.U54 F58 1973</lccn>
    <dewey>629.2/092/4 B</dewey >
    <isbn>0405050887</isbn>
    <series>Business<series>
</Book>
```

Although not shown here, presumably *lccn*, *dewey* and *isbn* are declared in the schema to be foreign keys to corrresponding fields of catalog records. *Series* is a foreign key to a categorization of books, of which "Business" is one category.

Keys can contain URIs, as in

```
<Book>
    <author>http://SSA.gov/blab/people/Henry+Ford</author>
    <author>http://SSA.gov/blab/people/Samuel+Crowther</author>
    <title>My Life and Work</title>
</Book>
```

This is indicated in the schema by a datatype of "URI".

```
<elementType id="author">
    <string/>
    <datatype dt="uri"/>
</elementType>
```

39.8.1 *One-to-Many Relations*

Element relations are binary. That is, we never express an n-to-1 relationship directly. We do not, for example, list within *books* a single relation that somehow resolves to all the *authors*. Instead, we always write the relationship on the 1-to-n side, but allow multiple occurrances of the *subelement*, for example, allowing *books* to have multiple occurrences of *author*.

```
<Person><name>Henry Ford</name></Person>

<Person><name>Samuel Crowther</name></Person>

<Person><name>Harvey S. Firestone</name></Person>

<Book>
    <author>Henry Ford</author>
    <author>Samuel Crowther</author>
    <title>My Life and Work</title>
</Book>

<Book>
    <author>Harvey S. Firestone</author>
    <author>Samuel Crowther</author>
    <title>Men and Rubber</title>
</Book>
```

This example shows a book with several persons as author, and also a person who is author of several books. We discussed such many-to-many relations more under the topic of *correlations.*

39.8.2 *Multipart Keys*

When the foreignKey element does not have foreignKeyPart sub-elements (as it does not above) then the entirety of the elements contents (e.g. "Henry Ford") should be used as the key value. However, for multipart foreign keys, or cases where the element has several sub-elements, foreignKeyPart is used, as shown below.

```
<elementType id="firstName">
    <string/>
</elementType>

<elementType id="lastName">
    <string/>
</elementType>

<elementType id="Person">
    <element id="pp1" type="#firstName"/>
    <element id="pp2" type="#lastName"/>
    <key id="k1">
        <keyPart href="#pp1"/>
        <keyPart href="#pp2"/>
    </key>
</elementType>

<elementType id="author">
```

```
        <element id="ap1" type="#firstName"/>
        <element id="ap2" type="#lastName"/>
        <domain type="#Book"/>
        <range type="#Person"/>
        <foreignKey range="#Person" key="#k1">
            <foreignKeyPart href="#ap1"/>
            <foreignKeyPart href="#ap2"/>
        </foreignKey>
    </elementType>

    ...

    <Book>
        <title>My Life and Work</title>
        <author>
            <firstName>Henry</firstName>
            <lastName>Ford</lastName>
        </author>
    </Book>
```

39.9 | Attributes as References

An alternative way to express a reference is with an attribute.

```
<person id="person1"><name>Henry Ford</name></Person>

<person id="person2"><name>Samuel Crowther</name></Person>

<Book>
    <author name="Henry Ford"/>
    <author name="Samuel Crowther"/>
    <title>My Life and Work</title>
</Book>
```

This allows us to link a book to a person, through the author relation, using an attribute of the relation. This exactly parallels the construction we saw above under "multipart keys," where a *subelement* of author contained the authors name. Here, an *attribute* of author contains the name. We can express this in our schema as

```
<elementType id="author">
    <attribute name="name" id="authorname"/>
    <foreignKey range="#Person" key="#k1">
        <foreignKeyPart href="#authorname"/>
    </foreignKey>
</elementType>
```

A widely-used variant of this is to use a URI as a foreign key:

```
<Book>
    <author href="http://SSA.gov/blab/people/Henry+Ford"/>
    <author href="http://SSA.gov/blab/people/Samuel+Crowther"/>
    <title>My Life and Work</title>
</Book>
```

In this case, we are using the *href* attribute to contain a URI. This is a particular kind of foreign key, where the *range* is any possible resource, and where that resource is not identified by some combination of its properties but instead by a name-resolution service. We indicate this by using an attribute element, with dt= "URI".

```
<elementType id="author">
    <attribute name="href" id="authorhref" dt="uri"/>
</elementType>
```

39.10 | Constraints & Additional Properties

39.10.1 *Min and Max Constraints*

Elements can be limited to restricted ranges of values. The *min* and *max* elements define the lower and upper bounds.

```
<elementType id="age">
    <string/>
</elementType>

<elementType id="Person">
    <element hef="#age"><min>0</min><max>131</max></element>
</elementType>
```

Such intervals are *half-open* (that is, the *min* value is in the interval, and the *max* value is the smallest value not in the interval).

This rule leads to the simplest calculation in most cases, and is unambiguous with respect to precision. In the above example, it is clear by these rules the 130.9999 is in the interval and 131 is not. However, had we said "all numbers from 0 to 130.99," in practice we would have some ambiguity regarding the status of 130.9999. Or interpretation would depend on the precision that we inferred for the original statement. The issue is particularly ambiguous for dates. (What exactly does "From December 5 to December 8" mean? The use of half-open intervals for representation does not, however, put any requirements on how processors must display intervals. For example, dates in some contexts display differently than their storage. That is, the interval `<min>1997-12-05</min><max>1997-12-09</max>` might be displayed as "December 5 through December 8".

In certain cases this rule for a half-open interval is impractical (for example, what letter follows "z" in the latin alphabet?) If so, use *maxInclusive*:

```
<elementType id="student">
    <element type="#grade">
          <min>A</min><maxInclusive>Z</maxInclusive>
    </element>
</elementType>
```

39.10.1.1 Domain and Range Constraints

We can use the *domain* and *range* elements to add constraints to an elements use or value. The *domain* element, if present, indicates that the element may only be used as a property of certain other elements. That is, syntactically it may appear only in the content model of those other element types. It constrains the sorts of schemas that can be written with the element.

```
<elementType id="author">
    <string/>
    <domain type="#Book"/>
    <attribute name="href" dt="uri"/>
</elementType>
```

The *domain* property above permits *author* elements to be used only within elements which are either *books* or subsets of *books*. Use of domain is optional. If omitted, there is simply no restriction.

The *range* element is used with elements which are references and declares a restriction on the types of elements to which the relation may refer. Graphically, it describes the target end of a directed edge. Each *range* element references one elementType, any of which are valid. In this case, below, we have said that an *author* element must have an *href* attribute which is a URI reference to a *Person* or to an element type which is *Person* or a subset of *Person*.

```
<elementType id="author">
    <string/>
    <domain type="#Book"/>
    <attribute name="href" dt="uri" range="#Person" />
</elementType>
```

39.10.2 *Other useful properties*

Element and attribute types can have an unlimited amount of further information added to them in the schema due to the open nature of XML with namespaces.

39.11 | Using Elements from Other Schemas

A schema may use elements and attributes from other schemas in content models. For example, a subelement named "http://books.org/date" could be used within a *book* element as follows:

```
<?XML version='1.0' ?>
<?xml:namespace
       name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/"
       as="s"/?>
<s:schema>
    <elementType id="author">
        <string/>
    </elementType>

    <elementType id="title">
        <string/>
    </elementType>

    <elementType id="Book">
        <element type="#title" occurs="OPTIONAL"/>
        <element type="#author" occurs="ONEORMORE"/>
        <element href="http://books.org/date" />
    </elementType>
</s:schema>
```

This can be abbreviated by adopting the rule that namespace-qualified names may be used within the *href attribute* value of an *element* or *attribute* element.

```
<?XML version='1.0' ?>
<?xml:namespace
        name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/"
        as="s"/?>
<?xml:namespace name=" http://books.org/" as="bk"/?>
<s:schema>
    <elementType id="author">
        <string/>
    </elementType>

    <elementType id="title">
        <string/>
    </elementType>

    <elementType id="Book">
        <element type="#title" occurs="OPTIONAL"/>
        <element type="#author" occurs="ONEORMORE"/>
        <element href="bk:date" />
    </elementType>
</s:schema>
```

39.12 | XML-Specific Elements

39.12.1 *Attributes*

XML-Data schemas contain a number of facilities to match features of XML DTDs or to support certain characteristics of XML. The XML syntax allows that certain properties can be expressed in a form called "attributes." To support this, an elementType can contain attribute declarations, which are divided into attributes with enumerated or notation values, and all other kinds.

An attribute may be given a default value. Whether it is required or optional is signaled by *presence*. (Presence ordinarily defaults to IMPLIED, but if omitted and there is an explicit default, presence is set to the SPECIFIED.) See the DTD at the end of this document for syntactic details.

Attributes with enumerated (and notation) values permit a values attribute, a space-separated list of legal values. The values attribute is required when the atttype is ENUMERATION or NOTATION, else it is forbidden. In these cases, if a default is specified, it must be one of the specified values.

```
<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
    <attribute name="copyright" />
    <attribute name="ageGrp"
             atttype="ENUMERATION"
             values="child adult"
             default="adult" />
</elementType>
```

describes an instance such as

```
<book copyright="1922" ageGrp="adult">
    <title>My Life and Work</title>
    <author>
        <firstName>Henry</firstName>
        <lastName>Ford</lastName>
    </author>
</Book>
```

Attributes may also reference elementTypes, meaning that one may use the element type but with attribute syntax. This allows an attribute to explicitly have the same name and semantics even when used on different element types. There are of course some limits: The attribute can still occur only once in an instance, and it cannot contain other elements. However, this allows the semantics of the element type to be employed in attribute syntax.

```
<elementType id="Book">
    <attribute href="bk:title"/>
    <attribute href="bk:author"/>
    <attribute name="copyright" />
    <attribute name="ageGrp"
           type="ENUMERATION"
           values="children adult" default="adult" />
</elementType>
```

describes an instance such as

```
<book
     bk:author="Henry Ford"
     bk:title="My Life and Work"
     ageGrp="adult"/>
```

39.13 | Entity declaration element types

This and the next two declarations cover entities. Entities are a shorthand mechanism, similar to macros in a programming language.

```
<intEntityDcl name="LTG">
    Language Technology Group
</intEntityDcl>

<extEntityDcl name="dilbert" notation="#gif"
              systemId="http://www.ltg.ed.ac.uk/~ht/dilb.gif"/>
```

Here as elsewhere, following XML, systemId must be a URI, absolute or relative, and publicId, if present, must be a Public Identifier (as defined in ISO/IEC 9070:1991, Information technology – SGML support facilities – Registration procedures for public text owner identifiers). If a notation is given, it must be declared (see below) and the entity will be treated as binary, i.e., not substituted directly in place of references.

```
<notationDcl name="gif" systemId='http://who.knows.where/'/>
```

39.14 | External declarations element type

Although we allow an external entity with declarations to be included, we recommend a different declaration for schema modularization. The extDcls declaration gives a clean mechanism for importing (fragments of) other schemas. It replaces the common SGML idiom of declaring an external parameter entity and then immediately referring to it, and has the same import, namely, that the text referred to by the combination of systemId and publicId is included in the schema in place of the extDcls element, and that replacement text is then subject to the same validity constraints and interpretation as the rest of the schema.

Note that in many cases the desired effect may be better represented by referencing elements (and attributes) from the other schema or subclassing from them.

39.15 | Datatypes

A dataype indicates that the contents of an element can be interpreted as both a string and also, more specifically, as an object that can be interpreted more specifically as a number, date, etc. The datatype indicates that the elements contents can be parsed or interpreted to yeild an object more specific than a string.

That is, we distinguish the "type" of an element from its "datatype." The former gives the semantic meaning of an element, such as "birthday" indicating the date on which someone was born. The "datatype" represents the parser class needed to decode the element's contents into an object type more specific than "string." For example, "19541022" is the 22nd of October, 1954 in ISO 8601 date format. (That is, ISO 8601 parsing rules will decode "19541022" into a date, which can then be stored as a date rather than a string.

For example, we would like an XML author to be able to say that the contents of a "size" element is an integer, meaning that it should be parsed according to numeric parsing rules and that it can be stored in integer format. In some contexts an API can expose it as an integer rather than a string.

```
<item>
    <name>shirt</name>
    <size>8</size>
</item>
```

There are two main contexts for datatypes. First, when dealing with database APIs, such as ODBC, all elements with the same name typically contain the same type of contents. For example, all sizes contain integers or all birthdays contain dates. We will return to this case shortly.

Second, and by contrast, the type of the content may vary widely from instance to instance. The softer we make our software, the more often these flexible cases occur. For example, size could contain the integer 8, or the word "small" or even a formula for computing the size.

We expose the datatype of an element instance by use of a *dt:dt* attribute, where the value of the attribute is a URI giving the datatype. (The URI might be explicitly in URI format or might rely on the XML namespace facility for resolution.) For example, we might find a document containing something like:

```
<?namespace
        name="urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/"
        as="dt"?>
<?namespace name="http://zoosports.com/dt?" as="zoo"?>
<purchases>
  <item>
    <name>shirt</name>
    <size dt:dt="int">8</size>
  </item>
  <item>
```

```
    <name>shoes</name>
    <size>large</size>
  </item>
  <item>
    <name>suit</name>
    <size dt:dt="zoo:script">
        =(shirtsize*1.05) + 3
    </size>
  </item>
</purchases>
```

Clearly this technique works for the heterogeneous typing in the above example. It also works for the database case where all element's of the same type have the same datatype.

```
<item> <name>shirt</name> <size dt:dt="int">8</size>   </item>
<item> <name>shoes</name> <size dt:dt="int">6</size>   </item>
<item> <name>suit</name>  <size dt:dt="int">12</size>  </item>
```

As written above, this is inefficient. Fortunately, XML allows us in schemasto put attributes with default or fixed values, so we could say once that all *size* elements have a datatype with value "int". Having done so, our our instance just looks like:

```
<item> <name>shirt</name> <size>14</size>  </item>
<item> <name>shoes</name> <size>6</size>   </item>
<item> <name>suit</name>  <size>16</size>  </item>
```

In a DTD, we can set a *fixed* attribute value, so that all *size* elements have datatype "int" or we can set it as a *default* attribute value so that it is an integer except where explicitly noted otherwise.

```
<item> <name>shirt</name> <size>14</size>
       </item>
       <item> <name>shoes</name>
            <size dt:dt="string">large</size>
       </item>
       <item> <name>suit</name>  <size>16</size>
       </item>
```

XML DTDs today allow such attributes. For example, a DTD can say that all *shirt* elements have *integer datatype* by the following:

```
<!ELEMENT size PCDATA >
<!ATTLIST size   dt:dt "int" #FIXED >
```

XML-Data schemas allow the equivalent, though with specialized syntax:

```
<elementType id="size" >
    <datatype dt="int" />
</elementType>
```

Elements use *datatype* subelements to give the datatype so that an optional *presence* attribute of the datatype element can indicate whether the datatype is *fixed* or merely a *default*. Attributes can also have datatypes.

Because there is no possibility of their being anything other than a fixed type, the datatype of an attribute is signalled by a dt attribute:

```
<attribute id="size" dt="int" />
```

39.15.1 *How Typed Data is Exposed in the API*

Different APIs to typed data will use the datatype attribute differently. The basic XML parser API should expose all element contents as strings regardless of any datatype attribute. (It might also contain supplementary methods to read values as more specific types such as "integer," thereby getting more efficiency.) An ODBC interface could use the datatype attribute to expose each type of element as a column, with the column's datatype determined by the element type's datatype.

39.15.2 *Complex Data Types*

If a datatype requires a complex structure for storage, or an object-based storage, this is also handled by the dt:dt attribute, where the datatype's storage format can be a structure, Java class, COM++ class, etc. For example, if an application needed to have an element stored in a "ScheduleItem" structure and using some private format, it could note this like

```
<when dt:dt="zoo:ScheduleItem">M*D1W4B19971022;100</when>
```

The datatype does not require a private format. It could also use subelements and attributes such as

```
<when dt:dt="zoo:ScheduleItem2">
    <month>*</month>
    <day>1</day>
    <week>4</week>
    <begin>19971022</begin>
    <recurs>100</recurs>
</when>
```

In the case of the graph-oriented interfaces (e.g. XML/RDF) the mapping from the XML tree to a graph should add a *wrapping node* for each non-string data type. The datatype property gives the type of that node. For example, the following two are graphically equivalent:

```
<size dt:dt="int">8</size>
<size><dt:int>8</dt:int></size>
```

39.15.3 *Versioning of Instances*

Adding an attribute to an element does not change the other attributes or pose any special versioning problems. For example, an application written to expect an instance to contain "<birthday>19541022</birthday>" is not harmed if the schema reveals that this is ISO 8601 format. Versioning within datatypes should be handled by the author's making sure that that subtypes of datatypes retain all the characteristics of the supertype.

If a down-level application is given a datatype it cannot process, it should expose the element contents as a supertype of the indicated datatype. In practice, this will usually mean that unrecognized datatypes will be the same as "dt:string". However, there are cases in which a type will be promoted, for example exposing a boolean in a byte or word rather than a bit, exposing a floating point number in a language's native format, etc.

39.15.4 *The Datatypes Namespace*

The datatype attribute "dt" is defined in the namespace named "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/". (See the XML Namespaces Note at the W3C site for details of namespaces.) The full URN of the attribute is "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/dt".

You will have noticed that the value of the attribute, as used in the examples above, is not lexically a full URI. For example, it reads "int" or "string" etc. Datatype attribute values are abbreviated according to the following rule: If it does not contain a colon, it is a datatype defined in the datatypes namespace "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/". If it contains a colon, it is to be expanded to a full URI according to the same rules used for other names, as defined by the XML Namespaces Note. For example

```
<?namespace
         name="urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/"
         as="dt"?>
<?namespace name="http://zoosports.com/dt?" as="zoo"?>
<item>
    <size dt:dt="int">8</size>
    <name dt:dt="zoo:clothing">shirt</name>
</item>
```

has two datatypes whose full names are "urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/integer" and "http://zoosports.com/dt?clothing".

39.15.5 *What a datatype's URI Means*

Datatypes are identified by URIs. The URI as simply a reference to a section of a document that defines the appropriate parser and storage format of the element. To make this broadly useful, this document defines a set of common data types including all common forms of dates, plus all basic datatypes commonly used in SQL, C, C++, Java and COM (including strings).

The best form of such a document is that it should itself be an XML-Data schema where each datatype is an element declaration. For this purpose we define a *<Syntax>* subelement which can be used in lieu of a content model. We also define an *<objecttype>* subelement. Each has a URI as its value. This integrates data types with element types in general.

```
<schema:elementType id="int">
<syntax href=
"urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/num_to_int"/>
<objectType href=
"urn:uuid:C2F41010-65B3-11d1-A29F-00AA00C14882/integer32"/>
</schema:elementType>

<schema:elementType id="date.iso8601">
<syntax  href=
"urn:uuid:C2F41010-65B3-...882/date.iso8601_to_int32"/>
<objecttype href="urn:uuid:C2F41010-65B3-...882/integer32" />
</schema:elementType>
```

The objecttype sub-element can reference a structure, Java class, COM++ coClass, etc. The syntax subelement identifies a parser which can decode the element's content (and/or attributes) into the object type given the storage type URI. Input to the parser is the element object exposing all its attributes and content tree (that is, the subtree of the grove beginning with the element containing the dt attribute). The objectType attribute in particular is assumed available to the parser so that a single parser can support several objecttypes.

Having said this, all *basic* data types should be built into the parsers for efficiency and in order to ground the process. For these, the datatype elements serve only to formally document the storage types and parsers, and to give higher-level systems (such as RDF) a more formal basis for datatypes.

I do not currently propose that we attempt to write any universal notation for parsing rules. Certain popular kinds of formats, particularly dates, are not easily expressed in anything but natural language or code, and the parsers must be custom written code. In other words, the URIs for the basic syntax and objecttype elements probably resolve only to text descriptions.

39.15.6 *Structured Data Type Attributes*

Attributes in cannot XML have structure. I will separately propose some techniques to avoid this problem, specifically that the XML API should contain a method that treats attributes and subelements indistinguishably, and also that the content which is an element's value can be syntactically separated from content which is an element's properties.

39.15.7 *Specific Datatypes*

This includes all highly-popular types and all the built-in types of popular database and programming languages and systems such as SQL, Visual Basic, C, C++ and Java(tm).

Name	Parse type	Storage type	Examples
`string`	pcdata	string (Unicode)	[Greek letters: see CD-ROM version]

number	A number, with no limit on digits, may potentially have a leading sign, fractional digits, and optionally an exponent. Punctuation as in US English.	string	15, 3.14, -123.456E+10
int	A number, with optional sign, no fractions, no exponent.	32-bit signed binary	1, 58502, -13
float	Same as for "number."	64-bit IEEE 488	.314159265358979E+1
fixed.14.4	Same as "number" but no more than 14 dights to the left of the decimal point, and no more than 4 to the right.	64-bit signed binary	12.0044
boolean	"1" or "0"	bit	0, 1 (1=="true")
dateTime.iso8601	A date in ISO 8601 format, with optional time and no optional zone. Fractional seconds may be as precise as nanoseconds.	Structure or object containingyear, month, hour, minute, second, nanosecond.	19941105T08:15:00301

`dateTime.iso8601tz`	A date in ISO 8601 format, with optional time and optional zone. Fractional seconds may be as precise as nanoseconds.	Structure or object containing year, month, hour, minute, second, nanosecond, zone.	19941105T08:15:5+03
`date.iso8601`	A date in ISO 8601 format. (no time)	Structure or object containing year, month, day.	19541022
`time.iso8601`	A time in ISO 8601 format, with no date and no time zone.	Structure or object exposing day, hour, minute	
`time.iso8601.tz`	A time in ISO 8601 format, with no date but optional time zone.	Structure or object containing day, hour, minute, zonehours, zoneminutes.	08:15-05:00
`i1`	A number, with optional sign, no fractions, no exponent.	8-bit binary	1, 255
`i2`	"	16-bit binary	1, 703, -32768
`i4`	"	32-bit binary	
`i8`	"	64-bit binary	

ui1	A number, unsigned, no fractions, no exponent.	8-bit unsigned binary	1, 255
ui2	"	16-bit unsigned binary	1, 703, -32768
ui4	"	32-bit unsigned binary	
ui8	"	64-bit unsigned binary	
r4	Same as "number."	IEEE 488 4-byte float	
r8	"	IEEE 488 8-byte float	
float.IEEE.754 .32	"	IEEE 754 4-byte float	
float.IEEE.754 .64	"	IEEE 754 8-byte float	
uuid	Hexidecimal digits representing octets, optional embedded hyphens which should be ignored.	128-bytes Unix UUID structure	F04DA480-65B9-11d1-A29F-00AA00C14882

`uri`	Universal Resource Identifier	Per W3C spec	http://www.ics.uci.edu/pub/ietf/uri/draft-fielding-uri-syntax-00.txt http://www.ics.uci.edu/pub/ietf/uri/ http://www.ietf.org/html.charters/urn-charter.html
`bin.hex`	Hexidecimal digits representing octets	no specified size	
`char`	string	1 Unicode character (16 bits)	
`string.ansi`	string containing only ascii characters <= 0xFF.	Unicode or single-byte string.	This does not look Greek to me.

All of the dates and times above reading "iso8601.." actually use a restricted subset of the formats defined by ISO 8601. Years, if specified, must have four digits. Ordinal dates are not used. Of formats employing week numbers, only those that truncate year and month are allowed (5.2.3.3 d, e and f).

39.16 | Mapping between Schemas

Certain uses of data emphasize syntax, others "conceptual" relations. Syntactic schemas often have fewer elements compared to explicitly conceptual ones. Further, it is usually easier to design a schema that merely covers syntax rather than designing a well-thought-out conceptual data model. An effect of this is that many practical schemas will not contain all the elements that a conceptual schema would, either for reasons of economy or because the initial schema was simply syntactic. But is it useful to make the implicit explicit over time so that more generic processors can make use of data.

For example, the following schema is essentially syntax:

```
<elementType id="author">
    <string/>
</elementType>

<elementType id="title">
    <string/>
</elementType>

<elementType id="Book">
    <element type="#title"/>
    <element type="#author" occurs="ONEORMORE"/>
</elementType>
```

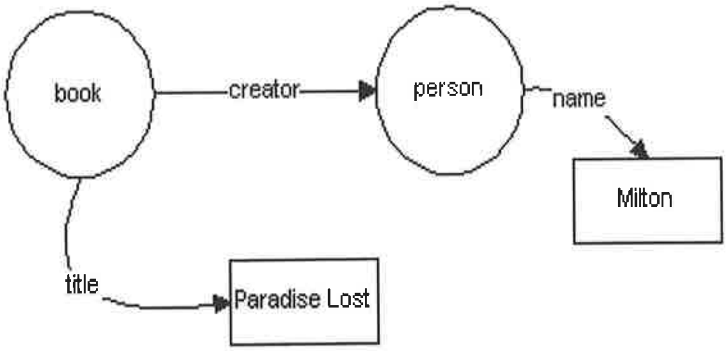
with instances looking like this

```
<Book>
    <title>Paradise Lost</title>
    <author>Milton</author>
</Book>
```

On the other hand, a conceptual schema could look like this:

```
<elementType id="name">
    <string/>
</elementType>

<elementType id="Person">
    <element type="#name/>
</elementType>

<elementType id="creator">
    <range type="#Person/>
</elementType>

<elementType id="title">
    <string/>
</elementType>

<elementType id="Book">
    <element type="#title"/>
    <element type="#creator" occurs="ONEORMORE"/>
</elementType>
```

If fully explicit, its instances would look something like this:

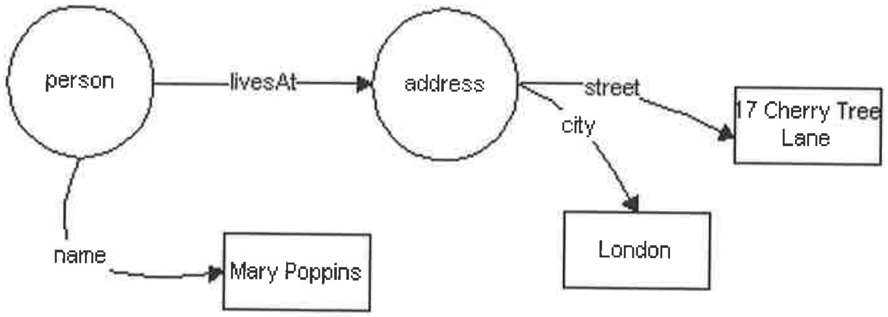


```
<Person id="thing1">
     <name>Milton</Person>
</Person>

<Book>
    <title>Paradise Lost</title>
    <creator>
        <Person>
            <name>Milton</name>
        </Person>
    </creator>
</Book>
```

In any case, what we want to express is a diagram such as this:

To do this, we will add mapping information into the syntactic schema which tells us how to interpolate the implied elements (and also to map *author* to *creator*) thereby creating a conceptual data model.

```
<?xml:namespace href="uri-to-the-conceptual-schema" as="c" ?>
<elementType id="author">
    <string/>
</elementType>

<elementType id="title">
    <string/>
</elementType>

<elementType id="Book">
    <mapsTo type="c:book"/>
    <element type="#title"> <mapsTo type="c:title"/> </element>
```



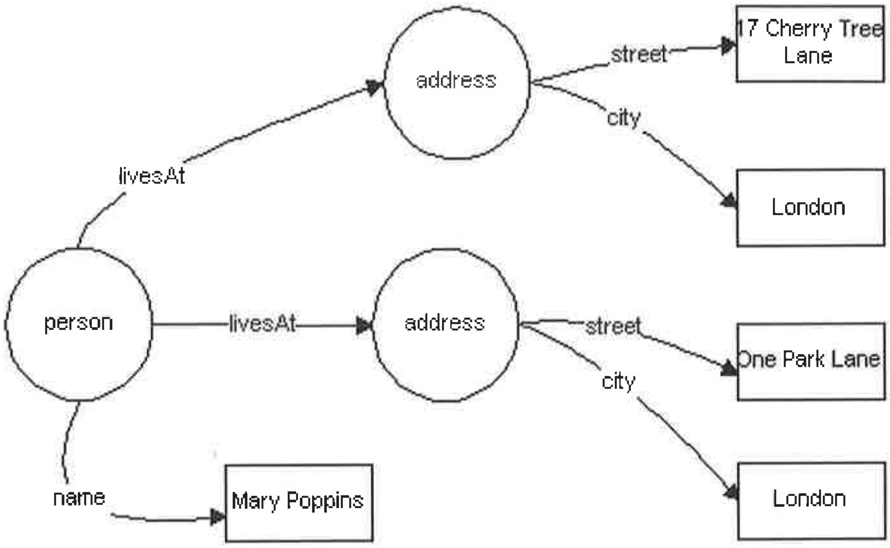
```
    <element type="#author" occurs="ONEORMORE">
        <mapsTo type="string">
            <implies type="c:name">
                <implies type="c:person">
                    <implies type="c:creator"/>
                </implies>
            </implies>
        </mapsTo>
    </element>
</elementType>
```

A more complex case could involve needing to map several properties to have a common implied node. For example, suppose we wanted that a *street* element and *city* element should both imply the **same** *address* node.

```
<Person>
    <name>Mary Poppins</name>
    <street>17 Cherry Tree Lane</street>
    <city>London</city>
</Person>
```

That is, rather than creating two *address* nodes, we want to create only a single onc, and subordinate both the *street* and *city* to it. If the conceptual schema has elements *livesAt*, *address*, *street* and *city*, we could write a mapping thus:

```
...definitions of name, street and city...
<elementType id="Person">
    <mapsTo type="c:person"/>
    <element type="#name">
            <string/>
            <mapsTo type="c:name"/>
    </element>
    <element type="#street">
        <string/>
        <mapsTo type="c:street">
              <implies type="c::address" id="livesAtAddress">
                  <implies type="c:livesAt"/>
```



```
            </implies>
        </mapsTo>
    </element>
    <element type="#city">
        <string/>
        <mapsTo type="c:city">
            <implies type="#livesAtAddress"/>
        </mapsTo>
    </element>
</elementType>
```

Elements may be repeated, so mapping rules need to accommodate repetitions. Suppose that someone has two addresses in the grammatical syntax, this needs to map to two addresses in the graph while still keeping the structure correct.

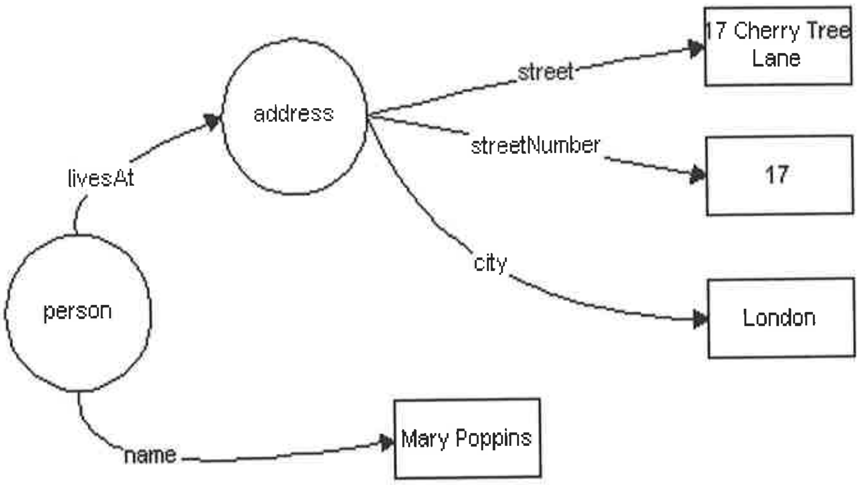
```
<Person>
    <name>Mary Poppins</name>
    <street>17 Cherry Tree Lane</street>
    <city>London</city>
    <street>One Park Lane</street>
    <city>London</city>
</Person>
```

```
<elementType id="Person">
    <mapsTo type="c:person"/>
    <element type="#name">    <string/>
              <mapsTo type="c:name"/>
    </element>
    <group occurs="ZEROORMORE"/>
        <element type="#street">
            <string/>
            <mapsTo type="c:street">
                <implies type="c::address" id="livesAtAddress">
                    <implies type="c:livesAt"/>
                </implies>
            </mapsTo>
        </element>
        <element type="#city">
            <string/>
            <mapsTo type="c:city">
                <implies type="#livesAtAddress"/>
            </mapsTo>
        </element>
    </group>
</elementType>
```

Mappings within groups are handled together. Since *street* and *city* are in a single group, each occurrence of such a group results in one *address*.

Text markup can also be handled by mapping. Suppose that for some reason we choose to markup the number portion of a street address:

```
<Person>
    <name>Mary Poppins</name>
    <street>< streetNumber>17</ streetNumber >
    Cherry Tree Lane</street>
    <city>London</city>
</Person>
```



If this should be reflected in the graph,
We can do that with mapping such as:

```
<elementType id="streetNumber">
    <string/>
</elementType>

<elementType id="street>
    <mixed>
        <element type="# streetNumber">
            <mapsTo type="c: streetNumber">
                <implies type="#livesAtAddress"/>
            </mapsTo>
        </element>
    </mixed>
</elementType>

...Person defined as before...
```

39.17 | Appendix A: Examples

Some data:

```
<?xml:namespace name="http://company.com/schemas/books/" as="bk"/>
<?xml:namespace name="http://www.ecom.org/schemas/dc/" as="ecom" ?>

<bk:booksAndAuthors>
    <Person>
        <name>Henry Ford</name>
        <birthday>1863</birthday>
    </Person>

    <Person>
        <name>Harvey S. Firestone</name>
    </Person>

    <Person>
        <name>Samuel Crowther</name>
    </Person>

    <Book>
        <author>Henry Ford</author>
        <author>Samuel Crowther</author>
        <title>My Life and Work</title>
    </Book>

    <Book>
        <author>Harvey S. Firestone</author>
        <author>Samuel Crowther</author>
        <title>Men and Rubber</title>
        <ecom:price>23.95</ecom:price>
    </Book>
</bk:booksAndAuthors>
```

The schema for http://company.com/schemas/books:

```
<?xml:namespace
     name="urn:uuid:BDC6E3F0-6DA3-11d1-A2A3-00AA00C14882/"
     as="s"/?>
<?xml:namespace
    href="http://www.ecom.org/schemas/ecom/" as="ecom" ?>

<s:schema>

    <elementType id="name">
        <string/>
    </elementType>

    <elementType id="birthday">
        <string/>
        <dataType dt="date.ISO8601"/>
    </elementType>

    <elementType id="Person">
        <element type="#name" id="p1"/>
    <element type="#birthday" occurs="OPTIONAL">
     <min>1700-01-01</min><max>2100-01-01</max>
    </element>
    <key id="k1"><keyPart href="#p1" /></key>
  </elementType>

  <elementType id="author">
      <string/>
      <domain type="#Book"/>
      <foreignKey range="#Person" key="#k1"/>
  </elementType>

  <elementType id="writtenWork">
      <element type="#author" occurs="ONEORMORE"/>
  </elementType>

  <elementType id="Book" >
     <genus type="#writtenWork"/>
     <superType
        href="http://www.ecom.org/schemas/ecom/commercialItem"/>
     <superType
       href="http://www.ecom.org/schemas/ecom/inventoryItem"/>
     <group groupOrder="SEQ" occurs="OPTIONAL">
         <element type="#preface"/>
          <element type="#introduction"/>
      </group>
   <element href="http://www.ecom.org/schemas/ecom/price"/>
   <element href="ecom:quantityOnHand"/>
  </elementType>

   <elementTypeEquivalent id="livre" type="#Book"/>
   <elementTypeEquivalent id="auteur" type="#author"/>

</s:schema>
```

39.18 | Appendix B: An XML DTD for XML-Data schemas

```
<!ENTITY % nodeattrs 'id ID #IMPLIED'>

<!-- href is as per XML-LINK, but is not required unless
      there is no content -->

<!ENTITY % linkattrs
                'id ID #IMPLIED
                href CDATA #IMPLIED'>

<!ENTITY % typelinkattrs
                'id ID #IMPLIED
                type CDATA #IMPLIED'>

<!ENTITY % exattrs
                'name CDATA #IMPLIED
                content (OPEN|CLOSED) "OPEN" >

<!ENTITY % elementTypeElements1
                genus? correlative? superType*>

<!ENTITY % elementTypeElements2
                description,
                (min|minExclusive)?,
                (max | maxInclusive)?,
                domain*,
                key*,
                foreignKey*,
                (datatype | ( syntax?, objecttype+ ) )?
                mapsTo?>

<!ENTITY % elementConstraints
                'min? max? default?'>

<!ENTITY % elementAttrs
                'occurs
                   (REQUIRED|OPTIONAL|ONEORMORE|ZEROORMORE)
                "REQUIRED" '>

<!ENTITY % rangeAttribute
                'range CDATA #IMPLIED'  >

<!-- The top-level container -->
<!element schema      ((elementType|linkType|
                         extendType|
                         intEntityDcl|extEntityDcl|
                    notationDcl|extDcls)*)>
<!attlist schema %nodeattrs;>

<!-- Element Type Declarations -->

<!element elementType (%elementTypeElements1;,
                  ((element|group)*|empty|any|string|mixed)?,
                  attribute*
                  %elementTypeElements2 )>

<!attlist elementType %nodeattrs;
                      %exattrs >

<!-- Element types allowed in content model -->

<!-- Note this is just short for a model group with only
          one element in it -->
<!element element  (%elementConstraints;) >

<!-- The type is required -->
<!attlist element   %typelinkattrs;
                    %elementAttrs;
                    presence (FIXED) #IMPLIED >

<!-- A group in a content model: and, sequential
                          or disjunctive -->
<!element group     ((group|element)+)>
<!attlist group     %nodeattrs;
                    %elementattrs;
                    presence (FIXED) #IMPLIED
                    groupOrder (AND|SEQ|OR) 'SEQ'>

<!element any       EMPTY>
<!element empty     EMPTY>
<!element string    EMPTY>

<!-- mixed content is just a flat, non-empty
                               list of elements -->
<!-- We don't need to say anything about
                   <string/> (CDATA), it's implied -->

<!element mixed         (element+)>
<!attlist mixed         %nodeattrs;>

<!element superType  EMPTY>
<!attlist superType %linkattrs;>

<!element genus  EMPTY>
<!attlist genus %typelinkattrs;>
```

The XML SPECtacular

- International Standards
- W3C Recommendations
- XML applications
- ... and More!

Chapter 40

For those of you who like to dig into the source material, our CD-ROM has plenty for you. Here's a full description.

Welcome to the XML SPECtacular, a collection of the relevant standards and specifications that you can browse, search, and print. This collection was compiled for The XML Handbook by Lars Marius Garshol.

For each document, we've included a link to a web site where you can learn more about the underlying project and obtain the latest version. Where copyright and production considerations allowed, we've also included a browseable copy on the CD-ROM.

All documents categorized as W3C recommendations or W3C work in process are subject to the W3C document use policy, which you can find on the CD-ROM and on the Web.

Not all specifications were available in HTML, so some of them are included as Adobe Portable Document Format (or PDF). A PDF viewer is available for free from Adobe for Mac, MS Windows, DOS, Unix and OS/2.

40.1 | Base standards

40.1.1 *International Standards*

40.1.1.1 Approved standards

SGML: Standard Generalized Markup Language

Charles F. Goldfarb
Information on web:
http://www.sil.org/sgml

This standard is really the ancestor of nearly all the other standards listed here. SGML is the mother tongue of most markup languages and the "big brother" of XML.

HyTime

Charles F. Goldfarb
Steven R. Newcomb
Eliot Kimber
Peter Newcomb
Information on web:
http://www.hytime.org/

HyTime is from the SGML family of International Standards. It describes many different things. There of the most important are architectural forms, hyperlinking, and structuring of time-based media like sound and film. Architectural forms is a technique for describing common semantics among different DTDs and is widely used. (The XLink standard uses something like it.)

DSSSL

Sharon Adler
Anders Berglund
Jon Bosak
James Clark

Information on web:
http://www.jclark.com/dsssl/

DSSSL is a powerful (and elegant!) style sheet language for SGML. DSSSL can be thought of as the "big brother" of XSL, but with a different syntax.

Unicode

Information on web:
http://www.unicode.org/

Unicode is an advanced and very complete character coding system. Using 16 bits (and various coding tricks), Unicode aims to encompass all human scripts, both those in use today as well as archaic ones. Unicode provides XML's character set.

40.1.2 *W3C recommendations*

40.1.2.1 Approved recommendations

Extensible Markup Language (XML)

Tim Bray
Jean Paoli
C.M. Sperberg-McQueen

Information on web:
http://www.w3.org/TR/REC-xml
Document included on CD-ROM:
./specs/w3c/rec-xml.html

Here it is: the XML standard itself. For a standard it is mercifully short and readable, and nicely unambiguous. This is definitely recommended reading!

Cascading Style Sheets (CSS2)

Håkon Wium Lie
Bert Bos

Information on web:
http://www.w3.org/Style/CSS/
Document included on CD-ROM:
./specs/w3c/pr-css2/index.html

CSS is the style sheet standard that is implemented in browsers today and can be used right now. It is simple, but effective and elegant.

40.1.2.2 Work in progress

XML Linking Language (XLink)

Steve DeRose
Eve Maler

Information on web:
http://www.w3.org/TR/WD-xlink
Document included on CD-ROM:
./specs/w3c/WD-xlink.html

XLink is a crucial part of the XML standards family as it describes hyperlinking in XML documents and takes major steps beyond the hyperlinking provided by HTML.

XML Pointer Language (XPointer)

Eve Maler
Steve DeRose

Information on web:
http://www.w3.org/TR/WD-xptr
Document included on CD-ROM:
./specs/w3c/WD-xptr.html

XPointer is a companion standard to XLink that describes mechanisms for addressing a particular part of a document.

Extensible Style Language (XSL)

Sharon Adler
Anders Berglund
James Clark
Istvan Cseri
Paul Grosso
Jonathan Marsh
Gavin Nicol
Jean Paoli
David Schach
Henry S. Thompson
Chris Wilson

Information on web:
http://www.w3.org/Style/XSL/
Document included on CD-ROM:
./specs/w3c/note-xsl-970910.html

XSL is what has been produced so far in phase 3 of the XML effort: a proposal for the style sheet language for XML. The document included here, though already implemented in products, is just a proposal, and it seems likely that it will undergo considerable changes before it becomes a recommendation.

Note that XSL incorporates Standard ECMA-262 ECMAScript: A general purpose, cross-platform programming language, which can be found below.

Document Object Model (DOM)

Lauren Wood
Jared Sorensen
Lauren Wood
Steve Byrne
Mike Champion
Rick Gessner
Scott Isaacs
Arnaud Le Hors
Gavin Nicol
Peter Sharpe
Jared Sorensen

Bob Sutor
Vidur Apparao
Bill Smith
Chris Wilson

Information on web:
http://www.w3.org/DOM/
Document included on CD-ROM:
./specs/w3c/wd-dom/cover.html

The Document Object Model is a very important related standard. It is to be the standard API for accessing and manipulating XML and HTML documents in browser, editors and other applications.

Namespaces in XML

Tim Bray
Dave Holander
Andrew Layman

Information on web:
http://www.w3.org/TR/WD-xml-names
Document included on CD-ROM:
./specs/w3c/wd-xml-names.html

This namespace proposal sketches a way to ensure that names used in XML DTDs are unique, so that names from different DTDs can be combined in a single document when need be.

40.2 | XML applications

These are XML document types that have been designed for specific purposes.

40.2.1 *W3C recommendations*

40.2.1.1 Approved recommendations

Mathematical Markup Language (MathML)

Patrick Ion
Robert Miner
Stephen Buswell
Stan Devitt
Angel Diaz
Nico Poppelier
Bruce Smith
Neil Soiffer
Robert Sutor
Stephen Watt

Information on web:
http://www.w3.org/Math/
Document included on CD-ROM:
./specs/w3c/rec-mathml/index.html

MathML is the long-awaited solution to a problem many scientists and teachers have struggled with: how to publish mathematical formulae on the web.

40.2.1.2 Work in progress

Channel Definition Format (CDF)

Information on web:
http://www.w3.org/TR/NOTE-CDFsubmit.html
Document included on CD-ROM:
./specs/w3c/NOTE-CDFsubmit.html

CDF is a DTD proposed by Microsoft for describing push channels. One interesting aspect of this format is that it is already in use in MSIE 4.0, so millions of CDF files already reside on the hard disks of users all over the world.

Web Interface Definition Language (WIDL)

Information on web:
http://www.w3.org/TR/NOTE-widl
Document included on CD-ROM:
./specs/w3c/note-widl.html

WIDL is a proposed metalanguage for descriptions of web service interfaces, from which client code can be generated automatically.

Resource Description Framework (RDF) Schemas

Information on web:
http://www.w3.org/TR/WD-rdf-schema
Document included on CD-ROM:
./specs/w3c/wd-rdf-schema/index.html

RDF provides a standard framework for describing resource metadata and as such is very important for the future development of search engines and other web navigation applications.

XML-Data

Andrew Layman
Edward Jung
Eve Maler
Henry S. Thompson
Jean Paoli
John Tigue
Norbert H. Mikula
Steve De Rose

Information on web:
http://www.w3.org/TR/1998/NOTE-XML-data-0105
Document included on CD-ROM:
./specs/w3c/note-xml-data.html

XML-Data is a proposal to use XML documents, rather than markup declarations, to describe DTDs. With XML-Data, the document type definitions can be augmented with additional properties, such as inheritance and datatypes.

Precision Graphics Markup Language (PGML)

Information on web:
http://www.w3.org/TR/1998/NOTE-PGML
Document included on CD-ROM:
./specs/w3c/note-pgml.html

PGML is a scalable vector graphics language based on the imaging model of PostScript and PDF, with hooks for animation and dynamic behavior.

Standard Multimedia Integration Language (SMIL)

Stephan Bugaj
Dick Bulterman
Bruce Butterfield
Wo Chang
Guy Fouquet
Christian Gran
Mark Hakkinen
Lynda Hardman
Peter Hoddie
Klaus Hofrichter
Philipp Hoschka
Jack Jansen
George Kerscher
Rob Lanphier
Nabil Layaïda
Stephanie Leif
Jonathan Marsh
Sjoerd Mullender
Didier Pillet
Anup Rao
Lloyd Rutledge
Patrick Soquet
Warner ten Kate
Jacco van Ossenbruggen
Michael Vernick
Jin Yu

Information on web:

http://www.w3.org/TR/1998/PR-smil-19980409/
Document included on CD-ROM:
./specs/w3c/pr-smil/index.html

SMIL is a language for describing multimedia presentations. It allows for the integration of independent multimedia objects into these presentations.

40.2.2 *Other initiatives*

40.2.2.1 Approved standards

ECMAScript (ECMA-262)

Information on web:
http://www.ecma.ch/stand/ecma-262.htm
Document included on CD-ROM:
./specs/e262-pdf.pdf

ECMAScript is a merger of JavaScript and JScript, standardized and described in detail. It is included here because it is the programming language used in XSL.

40.2.2.2 Work in progress

Simple API for XML (SAX)

David Megginson
A cast of thousands

Information on web:
http://www.microstar.com/XML/SAX/
Document included on CD-ROM:
./specs/sax.html

SAX is an event-based API for XML parsers written in object-oriented languages. Using SAX enables application programmers to switch XML parsers without changing their applications.

SAX is not presently being standardized by an official standards body. It is a defacto standard developed by the participants of the xml-dev mailing list. You should visit the web page, since SAX was due for an update when we went to press.

Guidelines for using XML for Electronic Data Interchange (XML-EDI)

Martin Bryan
Benoít Marchal
Norbert Mikula
Bruce Peat
David RR Webber

Information on web:
http://www.geocities.com/WallStreet/Floor/5815/xmlediindex.htm
Document included on CD-ROM:
./specs/edi/index.html

XML-EDI describes the use of XML in online commerce for exchanging transaction information. This isn't a complete specification, but more of a guideline to function as a precursor to a formal specification.

Index

Robin Short
Prentice Hall PTR
One Lake Street
Upper Saddle River, New Jersey 07458

About the CD-ROM

The CD-ROM is packed with useful XML tools and information. There are three main areas:

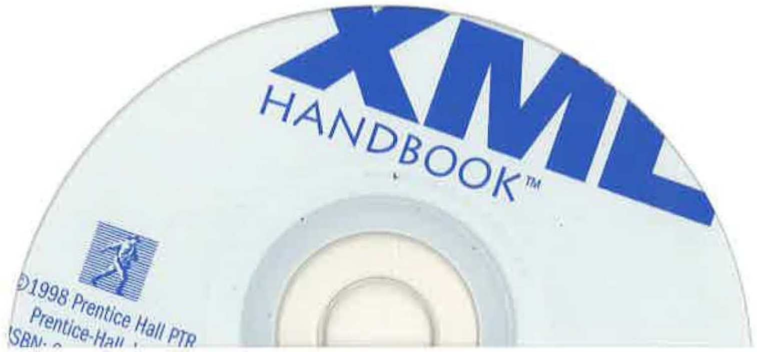
- A hand-picked collection of genuine, productive, no-time-limit XML free software. There are over 55 titles. A full description can be found in the *Free XML software* chapter.
- A showcase for leading XML software and service providers. It features in-depth product and service information, white papers, XML samples, live demos, and trialware.
- The XML SPECtacular, a collection of the relevant specifications that you can browse, search, and print.

How to Use the CD-ROM

The CD-ROM supports Windows 95, Windows NT, and UNIX systems. Simply load index.htm, located in the root directory of the CD-ROM, into your Web browser.

License Agreement

Use of *The XML Handbook*™ CD-ROM is subject to the terms of the License Agreement and Limited Warranty on the preceding pages.



WARRANTY LIMITS

READ AGREEMENT FOLLOWING THE INDEX AND THIS LABEL BEFORE OPENING SOFTWARE MEDIA PACKAGE.

BY OPENING THIS SEALED SOFTWARE MEDIA PACKAGE, YOU ACCEPT AND AGREE TO THE TERMS AND CONDITIONS PRINTED BELOW. IF YOU DO NOT AGREE, **DO NOT OPEN THE PACKAGE.** SIMPLY RETURN THE SEALED PACKAGE.

The software media is distributed on an "As IS" basis, without warranty. Neither the authors, the software developers nor Prentice Hall make any representation, or warranty, either express or implied, with respect to the software programs, their quality, accuracy, or fitness for a specific purpose. Therefore, neither the authors, the software developers nor Prentice Hall shall have any liability to you or any other person or entity with respect to any liability, loss, or damage caused or alleged to have been caused directly or indirectly by programs contained on the media. This includes, but is not limited to, interruption of service, loss of data, loss of classroom time, loss of consulting or anticipatory profits, or consequential damages from the use of these programs. If media is defective, you may return it for a replacement.

XML/HTML

Charles F. Goldfarb Series on Open Information Management

The XML Handbook™

"This book is an excellent starting point where you can learn and experiment with XML. As the inventor of SGML, Dr. Charles F. Goldfarb is one of the most respected authorities on structured information. Charles and I share a common vision: that the most valuable asset for the user or for a corporation, namely the data, can be openly represented in a simple, flexible, and human-readable form. This vision can now be realized through XML."

From the Foreword by Jean Paoli, Microsoft XML architect and co-editor of the W3C XML specification

Learn the secrets of successful real-world business applications using XML products from Microsoft® and these leading XML companies:

Adobe · SoftQuad · Chrystal software · Interleaf · Microstar · POET · ArborText · Russell · Junglee · webMethods · Inso · Isogen International corp. · Texcel

The definitive resource for the Brave New Web of smart structured data and electronic commerce

- Start by understanding what XML is, why it came to be, how it differs from HTML, and the handful of vital concepts that you *must* understand to apply XML quickly and successfully
- Experience what it's like to use XML, through illustrated walk-throughs of XML tools—including hot new Web servers for e-commerce, content management, structuring, creation, and presentation
- Master the details of the XML language and related technologies from reader-friendly, in-depth presentations

The accompanying CD-ROM brings together an amazing set of XML resources:

- An expert selection of free XML software—over 55 packages!
- XML Sponsor Showcase: leading vendors present trialware, demos, in-depth information, examples, and more
- XML SPECtacular—complete, browseable, printable copies of vital XML-related standards and specs

About the Authors

CHARLES F. GOLDFARB is the inventor of SGML, the International Standard (ISO 8879) on which both XML and HTML are based. He is the author of *The SGML Handbook* and co-author of the *SGML Buyer's Guide*.

PAUL PRESCOD is a leading XML consulting engineer and a member of the W3C XML team that is working on the development of XML and Xlink.

PRENTICE HALL | Upper Saddle River, NJ 07458 | http://www.phptr.com

$44.95 U.S./$63.00 Canada

0 76092 00334 2

ISBN 0-13-081152-1

9 780130 811523 90000