EXTENDING XML SCHEMA WITH DERIVED ELEMENTS*

Felipe Ibáñez, Oscar Díaz, Juan J. Rodríguez
Dpto. de Lenguajes y Sistemas Informáticos
University of the Basque Country
Apdo. 649 - 20080 San Sebastián (Spain)
{jipibanf, jipdigao, jibrojij}@si.ehu.es

Abstract XML is becoming the standard for document description, and XML Schema is gaining wide acceptance as the schema language to define the set of elements and attribute names that describe the content of a document. This work proposes both a knowledge model and an execution model to extend XML Schema with derived elements: the *XDerive* vocabulary. A derived element is an element whose content can be calculated by examining the content of other elements. The common presence of derived data in everyday documents supports this endeavour. The feasibility of this approach has been checked out by making the Oracles XML Parser able to interpret *XDerive* tags.

Keywords: Derived data, XML Schema, XML, Web Services

1. Introduction

The notion of document is at the heart of current business operations. Order, application or delivery forms are all examples of documents that integrate data which gives support to a certain business function, regardless of whether this data is finally stored in a database or in another remote repository. The number of applications that rely on XML standards for document description and transportation is increasing. XML is intended to be a self-describing data format, allowing authors to define a set of elements and attribute names that describe the content of a document. As XML allows the author such flexibility, a means is needed to restrict the number of valid element and attribute names for documents, so that the processing application knows what elements to expect

*This research was supported by the Secretaría de Estado de Política Científica y Tecnológica of the Spanish Government under contract TIC 1999-1048-C02-02. Founding was also received by the Departamento de Educación, Universidades e Investigación of the Basque Government under contract UE2000-32.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35614-3_21
C. Rolland et al. (eds.), *Engineering Information Systems in the Internet Context*

and how to process them. XML Schema serves this purpose. A schema defines the allowable structure and contents of a class of XML documents much in the same way as the "create table" statement defines the structure of the table tuples in SQL.

Documents frequently hold derived data, which is calculated by examining the content of other data elements. For example, the *totalAmount* of an order form can be calculated from the cost of each item included in the order. This calculation is part of the definition of the derived element. Besides, conceptual models have long supported this notion. The Entity/Relationship model depicts derived attributes as a dotted ellipse. Another semantic model that gives extensive support to derived attributes is SDM [7]. This model offers a small vocabulary of high-level attribute derivation primitives to compute the value using statistical, Boolean, arithmetic or ordering functions.

The frequent presence of derived data in both everyday documents and conceptual models, supports the endeavour of extending the XML Schema with a similar concept: the notion of derived elements. Currently, derived data semantics (e.g. the derived function) is not associated with the schema but hard-coded in the application which processes the XML instance document. As an example, consider a searching application where individual orders are rendered according to distinct criteria. An XSLT stylesheet can be available to indicate hoe an order is rendered using HTML. However, how the *totalAmount* is calculated will be implicit as a piece of scripting code or within the XSLT stylesheet, thus mixing derivation semantics with presentation logic. Not only does this approach hinder development, but the maintenance and coherence of the derivation semantic is jeopardized, too.

This work advocates for moving the derivation semantics to the document schema. The advantages that can be drawn from this migration are numerous. One such advantage is that it promotes code reusability. Rather than replicating code in distinct applications, the code (i.e. the derivation function) resides in a single place -the schema- from which it is implicitly used. Such centralization accounts for increasing consistency as all application share a common derivation semantics. Maintenance is also eased as changes to the derivation semantics are localized in a single place.

This paper presents a proposal for extending XML Schema with derived elements. This poses two questions: firstly, how can derived elements be specified (the knowledge model) and secondly, how behave derived elements at run time (the execution model). The derived function is described either by using XSLT or by calls to Web service operations. As for the execution model, derived elements can be calculated either at compile-time or run-time. This work focuses on the run-time option, where two possibilities can be distinguished: *onLoad* and *onDemand*. The former obtains the whole set of derived values at once. Although slightly faster, this option consumes both time and space resources

```
<?xml version="1.0" encoding="UTF-8"?>
<order xmlns:xlink="http://www.w3.org/1999/xlink" connection="myConnection" xmlns:xsql="urn:oracle-xsql">
  <orderDate>2002-02-22</orderDate>
  <shipDate>11111</shipDate>
  <customer id="18273" xlink:href="customers/18273.xml" />
  <lineItems>
    <xsql:query>
      SELECT productId, unitPrice, quantity
      FROM itemsTable
      WHERE id='TV-1' or id='Hi-Fi-1'
    </xsql:query>
  </lineItems>
  <deliveryCompany company="ups" xlink:href="http://db.atarix.org?getDeliveryCompanyDat('ups')"/>
</order>
```

Figure 1. Document modularization: an example for an order document.

regardless of whether the derived element is finally accessed or not. By contrast, the *onDemand* alternative delays the calculation till the element is requested. This work does not contemplate the possibility to update the derived element directly nor updating the propagation issue[1] [1].

The rest of the paper is structured as follows. Section 2 introduces current practices on XML document modularization which are somehow related with this work. Section 3 outlines XML Schema and WSDL as standards on which this work is based. The knowledge and execution model for derived elements are the topics of sections 4 and 5, respectively. And in the final section, conclusions are given. These ideas have been realized by creating a new "derivation aware" parser that extends the Oracle's XML Parser for Java [3].

2. Current practices on XML document modularization

The idea of self-contained documents is being abandoned in favour of a modular approach to document description. The idea is that the XML document holds what is to this document specific information, and leaving additional or contextual information to other related documents. DTD via entities [6, Chapter 33], XLink [15] and XInclude [18] are able to support such an approach. Figure 1 illustrates such approaches for an order.

The content which is specifically associated with an order is described in the order document itself (e.g. *<orderDate>* or *<shipDate>*). On the other hand, additional information, commonly shared with other orders, is indicated by means of "linked" documents (see figure 1). For instance, *<customer>* data is obtained from a separate documents after the customer id. Besides reusability

[1]The derived data update problem involves the transformation of modifications of derived data into corresponding changes to stored data and other derived data. The challenge lies on the ambiguity of this transformation as the propagation is not always unique.

and maintenance gains, this approach accounts for a more efficient and flexible document traversal: the consumer retrieves only the relevant data and additional information is available on demand by means of hyperlinks.

It is worth mentioning that related documents do not need to be static XML documents. Instead, function calls can be used to generate XML documents on the fly when required. This is the approach that has been followed for <*lineItems*> in figure 1. Data about the item is not statically stored in an XML document but retrieved on the fly from the database. Some DBMS manufacturers already provide support for easily generating XML from dynamic database queries via SQL. The <*lineItems*> element in figure 1 uses the Oracle's *XSQL Pages Publishing Framework* [10] to obtain its content on the fly. This framework is supported through the *XSQL Servlet Engine* to be installed on the web server. In the above code, a connection to the database is opened by using the SQL *connection* attribute whose value "myConnection" is the name of one of the pre-defined connections in the *XSQLConfig.xml* configuration file. This connection handle is used to execute a SQL query held by the *query* tag. The result is formated as an XML fragment whose root is a <*rowset*> element which contains <*row*> elements. These <*row*> elements hold in turn a sub-element for each attribute returned by the SQL query. In this approach, the XML file which contains the XSQL elements (e.g. the order document) is processed as a server page. The order document is copy to a directory under the web server's virtual directory hierarchy, an when invoked (e.g. by requesting the URL *http://yourcompany.com/order.xsql*) the web server returns the order document where the XSQL elements have been materialized automatically as XML fragments.

A possible drawback of this approach is that it does not achieve logical data independence[2] meaning that changes on how *item* data is stored in the database might require changes to the *order* document. However, such a situation can be overcome if the database is URL-addressable. In this case, the query is encapsulated within a stored procedure so that the hosting XML document is unaffected by changes to the database schema. This latter situation is exemplified by <*deliveryCompany*>. Data about the delivery company is obtained through the *getDeliveryCompanyData* procedure which has the name of the company as the input parameter. The requester is unaware of the base tables that ultimately store the data.

Summing up, the designer of the *order* document can opt for a static or a dynamic approach to support related content. The latter is preferable if the content has a relatively high volatility although it increases the transmission overhead.

[2]Logical data independence is a key concept for database practitioners. It indicates the capability to change the conceptual schema without having to change the application programs [4].

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
  <xs:annotation>
    <xs:documetation>
      <author>Felipe</author>
      <purpose>A simple example</purpose>
    </xs:documetation>
  </xs:annotation>
  <xs:element name="order">
    <xs:complexType name="orderType">
      <xs:sequence>
        <xs:element name="orderDate" type="xs:date" minOccurs="1" maxOccurs="unbounded"/>
        <xs:element name="shipDate" type="xs:date"/>
        <xs:element name="customer" type="customerType"/>
        <xs:element name="lineItem" type="lineItemType" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:complexType name="customerType">
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="customerData" type="customerDataType" />
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="lineItemType">
    <xs:sequence>
      <xs:element name="product" type="productType"/>
      <xs:element name="quantity" type="xs:int" minOccurs="0" />
      <xs:element name="applicableDiscount" type="xs:int" minOccurs="0" />
      <xs:element name="total" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="productType">
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="unitPrice" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 2. A partial schema for *order* documents.

Previous paragraphs have outlined current practice on XML document modularization. Might be due to its inspiration on the inclusion mechanisms available on most programming languages, these approaches are instance-based (as opposed to schema-based), and are rarely parameterized. By contrast, derived element semantics is defined at the schema level where derivation functions are parameterized by other elements of the document.

3. XML supporting standards

This section outlines the XML standards which were used to support the notion of derived element.

3.1. XML-Schema

An XML-Schema document is an XML document that describes structural and content-based constraints for a class of instance documents, e.g. which elements can occur and how they can be nested in the XML document [16]. Validating parsers can then be used to check conformance of documents claiming to be instances of a given schema.

Figure 2 presents a partial schema for *order* documents. The schema starts by declaring the distinct namespace used within the document. In this example, the "http://www.w3.org/2001/XMLSchema" namespace is used. The rest of the document consists of two distinct constructs: element declarations and type definitions. The *orderDate* element exemplifies an element of simple type that holds a *date* value. The *minOccurs* attribute indicates the minimum number of *orderDate* elements that must appear in an order document. Simple types define types of elements that have no attributes and only contain text. By contrast, complex types define types of elements that can have attributes and other elements as part of its content [16], [17]. The *customerData* element is a case in point. It holds both the *name* and *billingAddress* elements where the *sequence* construct indicates that these (sub) elements should be provided in the strict order specified by the schema.

This work addresses how XML Schema can be extended with derived elements. To this end, XML Schema offers the *annotation* element to provide further information about elements that are read by machines or humans. It may contain *appinfo*, or *documentation* elements, which are used to provide instructions for the processing application and schema documentation comments, respectively. In our example (see figure 2), a *documentation* element is added to the order element indicating both the author and the purpose of this schema. This information is intended for humans, hence a *documentation* element is used. By contrast, the derived element semantics is targeted at the XML parser, so it will be enclosed within *appinfo* tags.

3.2. Web Service Definition Language

Web services evolve the object-oriented vision of assembling software from component building blocks to the assembly of services that may or may not be built on object technology. Among the standards to describe, discover and integrate Web services, the Web Service Description Language (WSDL) is an XML vocabulary that indicates how a client program can connect to a Web service [14]. Therefore, a WSDL description specifies the common knowledge that must exist between a Web service requester and a provider of that service in order for a successful interaction to occur. Such knowledge refers to: *what* is offered by the service (and its signature), *how* is the service's signature mapped

to a specific representation in a given protocol and finally, *where* is the service located (for a gentle introduction see [11]).

As an example, consider a *customerInfo* Web service that offers a *getCustomerData* operation. A partial definition of this service follows (the complete specification can be found in appendix 6):

```
<portType name= "customerInfoPortType">
    <operation name= "getCustomerData">
        <input message= "tns:ssnMsg" />
        <output message= "tns:customerDataMsg" />
        <fault name= "incorrectSsnFault" message= "tns:incorrectSsnMsg"/>
        <fault name= "nonExistFault" message= "tns:nonExistMsg" />
    </operation>
</portType>
```

This operation takes a social security number of the *ssnType* as an input, and retrieves an XML fragment which conforms the *customerDataType*. During the execution of this operation, two faults are contemplated: *incorrectSsnFault* and *nonExistFault*.

4. Derived elements: a knowledge model

This section looks at how derived elements are specified. The *order* document is taken as an example. An order would usually contain information about the customer who placed the order, the order status and the individual line items on the order including their quantities and prices. Besides this raw data, some redundant elements can be introduced such as the <*subTotal*> element of each *lineItem* whose only purpose is to aggregate some data already available in the document.

Strictly speaking, a derived element should not provide any new information except information derived from other elements available in the document. However, this work extends the derivation context to any resource available to the document. Hence, derived elements can be calculated from data contained in the document itself, as well as from data stored in related resources. However, it should be underlined that the location of the data can have important implications on the efficiency of calculation as accessing external resources (e.g. databases) causes a transmission overhead.

Derived element specifications have posed four issues: (1) how the virtuality of derived elements is reflected in the schema, (2) how derivation functions are specified, (3) how derived function failures should be handle, and (4) meta-validation.

The virtuality issue. It refers to the fact that derived elements are not provided by the user but inferred from the other elements of the document. However, from the point of view of the processing application no distinction should be made between derived and "base" elements. Ideally, the processing

application should be unaware of the fact of whether an element is derived or not. To this end, the XML parser extends the original instance document with the derived content, so that the processing application ignores whether an element is derived or not. For instance, an XSLT template that processes the order document shown in figure 1, should refer to <*orderDate*> or <*subTotal*> in exactly the very same terms: using an XPath 1.0 expression [12]. This could look bizarre to XSLT programmers as they are designing templates for elements that do not appear in the associated instance document. From the XSLT programmer viewpoint, <*subTotal*> is a virtual element which come into existence at run-time.

The question is how this virtuality should be reflected in terms of the occurrence restrictions. XML Schema provides the *minOccurs* and *maxOccurs* attributes to indicate the allowable occurrence interval. However, it is not clear what this interval should be for derived elements. If *minOccurs* is set to 1 the user is forced to introduce the derived element. But this is not the expected behaviour. On the other hand, if *minOccurs* is set to 0 the associated element becomes optional, which is also not the right interpretation, particularly for the processing application.

As a result, the schema validator of the "derivation aware" parser should reinterpret the meaning of *minOccurs* based on the context of the derived element. In our implementation, derived elements are set to a *minOccurs* of zero. This means that an XML text editor would allow the user to introduce a value for a derived element. At parsing time however, this value is overridden by the output of the derivation function[3].

Derivation function specification. How the derived function is specified depends on whether the function is local or remote. A local function is described via the XSLT vocabulary [13]. As an example, consider the *subTotal* element. An <*appInfo*> tag is used to deliver to the XML Parser the derivation function (see figure 3 (a)). The <*derive*> tag is used for this purpose:

```
<xd:derive actuate="onLoad">
  <xsl:value-of select="quantity * product/uniPrice *
                        (1 - (applicableDiscount div 100))"/>
</xd:derive>
```

The <*derive*> tag is part of the *XDerive* vocabulary and keeps an XSLT fragment whose execution returns the derived value. XPath is used to address the distinct element's values within the instance document where paths are relative to the parent node. For instance, the paths "*quantity*", "*product/unitPrice*" and "*applicableDiscount*" address the quantity, the products price per unit and the

[3] Another alternative would have been to rise an error which indicates this situation.

applicableDiscount hanging from the *lineItem* node whose *subTotal* is being calculated. The product of these three elements values is the value of *subTotal*.

On the other hand, remote functions are supported by requests to Web Services. Figure 3 (b) gives an example for the *customerData* element. Its content is the XML fragment obtained as a result of invoking the *getCustomerData* operation. The <*call*> tag, part of the *XDerive* vocabulary, holds the attributes that identify the required service, port and operation. Parameter passing is achieved through the <*binding*> tag. This tag describes the mapping between the parameters of the Web service's operation ("parts" in WSDL terminology) and the elements of the instance document. In our example, the content of the *"id"* element in the *order* document is mapped to the *"ssn"* part of the Web service's operation. Again, elements of the document are addressed by relative paths using XPath.

Derivation function failures. *XDerive* forces the designer to address faulting situation through the <*onFault*> tag. This element holds information about the contingency action to be taken, should the function fails. This could be due to the absence of some deriving element (in the case of local functions) or to some operation faults (in the case of remote functions).

For local functions, a contingency action should be specified for each deriving element that is optional (i.e. *minOccurs* = *"0"*). As an example, consider that the *subTotal* element is calculated from the *quantity, product/unitPrice* and *applicableDiscount* elements. If these elements are optional, a different contingency action can be set to tackle the absence of each element. The "*test*" attribute is used for this purpose. The value of this attribute is an XPath expression that checks the absence of those elements. For instance, the expression

```
<xd:onFault test= "count(quantity)=0">
    <xsl:value-of select= "product/unitPrice *
                           (1 - (applicableDiscount div 100))"/>
</xd:onFault>
```

checks whether no <*quantity*> elements appear on the document. If so, a default of one is assumed. Another example follows:

```
<xd:onFault test= "count(applicableDiscount)=0 and
                           /order/customer/id= '2345' ">
    <xsl:message terminate = "yes">
      The applicable discount for customer 2345 must be provided
    </xsl:message>
</xd:onFault>
```

In this case, there is no way to recover the situation which is described by the *test* predicate. It is worth noticing how the expressiveness of XPath allows to describe complex faulting situations. Figure 3 shows the complete specification.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:xd="http://www.atarix.org/xderive" elementFormDefault="qualified"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform" xmlns:cis="http://www.atarix.org/services/customerInfoService.wsdl" >
  <xs:complexType name="lineItemType">
    <xs:sequence>
      <xs:element name="product" type="productType"/>
      <xs:element name="quantity" type="xs:int"/>
      <xs:element name="applicableDiscount" type="xs:int"/>
      <xs:element name="subTotal" type="xs:decimal" minOccurs="0" >
        <xs:annotation>
          <xs:appinfo>
            <xd:derive actuate="onLoad">
              <xsl:value-of select="quantity * product/unitPrice * (1- (applicableDiscount div 100))"/>
              <xd:onFault test="count(quantity)=0">
                <xsl:value-of select=" product/unitPrice *  (1- (applicableDiscount div 100))"/>
              </xd:onFault> ...
(a)           <xd:onFault test="count(applicableDiscount)=0 and /order/customer/id='2345' ">
                <xsl:message terminate="yes">
                  The applicable discount for customer 2345 must be provided
                </xsl:message>
              </xd:onFault>  ...
            </xd:derive>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="customerType">
    <xs:sequence>
      <xs:element name="id" type="xs:string"/>
      <xs:element name="customerData" type="customerDataType" minOccurs="0" >
        <xs:annotation>
          <xs:appinfo>
            <xd:derive actuate="onDemand">
              <xd:call service="cis:customerInfoService" port="cis:customerInfoPort" operation="cis:getCustomerData">
                <xd:binding wsPart="cis:ssn" value="id/text()"/>
              </xd:call>
(b)           <xd:onFault name="cis:incorrectSsnFault">
                <xsl:message terminate="yes">
                  <xsl:value-of select="cis:incorrectSsn"/>
                </xsl:message>
              </xd:onFault>
              <xd:onFault name="cis:nonExistFault"> ... </xd:onFault> ...
            </xd:derive>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Figure 3. Extending the XML Schema to accommodate derived elements: the *order* document example.

It should be underlined that contingency actions convey important business policies which are currently hidden and distributed among the consuming applications. *XDerive* promotes an explicit, centralized and declarative description of such policies for easy maintenance. Indeed, it has been reported that business policies are among the most evolvable software artifacts found in current information systems.

As for remote functions, the failures should map to the "faults" associated with the Web service operation, as described in the WSDL file (see section 3.2.). For instance, the expression checks the happening of the "incorrectSsnFault" error. The value of the "name" attribute should coincide with a fault in the WSDL file. Finally, the severity of the error is indicated through the *"terminate"* attribute of the *<message>* element. If this attribute is set to *"no"*, the error does not interrupt the parsing process; instead, a message is generated and the parsing continues. By contrast, the otherwise value *"yes"* will stop the parser, if a fault arises.

Meta-validation. In the same way that an instance document should conform to its schema, the schema itself should obey some (meta) rules to ensure that the schema conforms to the intended semantics of the schema constructs. Below are some examples for the *XDerive* elements:

- Derived elements which have a Web service operation as its derived function. In this case, the type of the derived element must coincide with the type of the operation's result as specified in the WSDL file. This scenario is illustrated by the *<customerData>* element in our running example.
- Derived elements which have a Web service operation as its derived function. In this case, the name of the fault must correspond to one of the faults associated with the operation as specified in the WSDL file.
- A derived element must always have a minimum occurrence of zero.

Such rules can be enforced by a "derivation aware" XML editor that verifies the correctness of the schema at editing time.

5. Derived elements: an execution model

This model addresses when derivation functions are executed. In the database context, exists a tradeoff between memory usage and processing overhead which depends on whether derived attributes are stored in the database or calculated on demand. The designer should balance: (1) the additional cost to store the derived data and keep it consistent with the base data from which it is derived, with (2) the cost to calculate it each time it is required [2]. Similar concerns arise when reviewing the modularization options for XML documents. In this

case, the related document could be either static or dynamically generated (see section 2). This work does not yet contemplate the static option. Instead in the current implementation, derived elements are always calculated at run-time.

Even within the dynamic option, the derived value can be worked out at parsing time or on demand. The former processes the derived element when the document is parsed, i.e. transformed to a DOM structure. The latter postpones the calculation till the element is required. The decision is held in the *"actuate"* attribute of the <*derive*> which can be set to *onLoad* or *onDemand*.

A derived element with a local function is normally processed on load, since the data is locally available and the calculation overhead can be neglected. This case is illustrated by the *subTotal* element in figure 3a. An exception to this situation would be if the associated derivation function implies complex XSLT expressions or, if the derived element is hardly used. In these cases, the cost to derive its value at parsing time would be wasted in situations where the value is not used. An enhancement of the current implementation would be to let the parser decide when to execute the derivation function depending on whether the derived element is used or not. This could be possible if the required elements are known in advance. For instance, if an XML document aims at an XSLT stylesheet that will be applied to it, the parser can first check which derived elements are referred to in the XSLT stylesheet, and restrict the on-load calculation only to those elements.

From the above discussion follows, that derived elements with an associated remote function tend to follow an *onDemand* pattern. If this is the case, the cost of invoking -and constructing- the call to a remote provider is much higher, and the value should be always stored locally.

6. Conclusions

This work promotes the notion of derived elements as a useful construct for document description. The *XDerive* vocabulary has been introduced which extends XML Schema with derived elements. The *appinfo* facility has been used for this purpose. *XDerive* supports both the knowledge model and the execution model that define the semantics of derived elements.

XDerive has been realized by extending the Oracle XML Parser for Java. To this end, a "parse enhancing" processing model has been used which addresses how new extensions to the XML Schema are supported. Some of these extensions (e.g. Schematron [9]) follow a two-step approach whereby the semantics of the new constructs are first compiled into a run-time validator, -normally using XSLT. Next, this validator is used to check the compliance of the instance document against the new constructs.

This work follows an alternative way. Rather than building a new validator that complements the XML Schema parser, the parser itself is extended to be-

come "derivation aware". Applications can use the extended parser to validate the instance document as they did before except that now, the new constructs are also validated. Such an approach provides "parser transparency" in the sense that, parser enhancements do not imply changes to the applications using the parser: the application ignores whether the processed documents contains the new constructs (in this case, derived elements) or not. Of course, this option is feasible only for open and modular parsers. However, such parses seem to increase in popularity since the publication of JAXP (Java API for XML processing) [8]. This recommendation proposes an architecture to create plugable and configurable parsers. Among the parsers implementing this approach are Xerces2 [5](the XML Parser of the Apache Project) and the Oracle's XML Parser for Java. *XDerive* is supported in the Oracle's XML Parser.

We are currently working on the possibility of a derived element to be used in the computation of another derived element(with cycle detection in the expressions of elements). Moreover, we are tackling the situation where several derivation functions are available to compute the same derivation element.

Appendix: The *customerInfo* Web service definition.

```
<definitions name="customerInfo" targetNamespace="http://www.atarix.org/services/customerInfoService.wsdl"
          xmlns="http://schemas.xmlsoap.org/wsdl/" xmlns:tns="http://www.atarix.org/services/customerInfoService.wsdl"
          xmlns:xs="http://www.w3.org/2001/XMLSchema" xmlns:ci="http://www.atarix.org/schemas/customerInfo.xsd"
          xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://schemas.xmlsoap.org/wsdl/ http://www.atarix.org/external/wsdl.xsd">
  <types>
    < xs:schema targetNamespace="http://www.atarix.org/schemas/customerInfo.xsd"
                xmlns: xs ="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
      <xs:simpleType name="ssnType">
        <xs:restriction base="xs:string">
          <xs:pattern value="[0-9]{8} - [A-Z]"/>
        </xs:restriction>
      </xs:simpleType>
      <xs:complexType name="customerDataType">
        <xs:sequence>
          <xs:element name="name" type="xs:string"/>
          <xs:element name="billingAddress" type="addressType"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="addressType">
        <xs:sequence>
          <xs:element name="address" type="xs:string"/>
          <xs:element name="city" type="xs:string"/>
          <xs:element name="zip" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:schema>
  </types>
  <message name="customerDataMsg">
    <part name="customerData" type="ci:customerDataType"/>
  </message>
  <message name="ssnMsg">
    <part name="ssn" type="ci:ssnType"/>
  </message>
  <message name="incorrectSsnMsg">
    <part name="incorrectSsn" type="xs:string"/>
  </message>
  <message name="nonExistMsg">
    <part name="nonExist" type="xs:string"/>
  </message>
  <portType name="customerInfoPortType">
    <operation name="getCustomerData">
      <input message="tns:ssnMsg"/>
      <output message="tns:customerDataMsg"/>
      <fault name="incorrectSsnFault" message="tns:incorrectSsnMsg"/>
      <fault name="nonExistFault" message="tns:nonExistMsg"/>
    </operation>
  </portType>
  <binding name="customerInfoBinding" type="tns:customerInfoPortType">
  ....
  </binding>
  <service name="customerInfoService">
    <port name="customerInfoPort" binding="tns:customerBinding">
      <soap:address location="http://www.atarix.org:8888/soap/servlet/soaprouter"/>
    </port>
  </service>
</definitions>
```

References

[1] I. Amy Chen and D. McLeod. Derived Data Update in Semantic Databases. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 225–235, Amsterdam, The Netherlands, August 1989.

[2] T. Connolly and C. Begg. *Database Systems*. Addison Wesley, 1998.

[3] Oracle Corporation. The Oracles XML Parser for Java, 2002. at http://otn.oracle.com/tech/xml/xdk_java/content.html.

[4] R. Elmasri and S.B. Navathe. *Fundamentals of database systems*. Addison Wesley, 2000.

[5] The Apache Software Foundation. Xerces2 Java Parser 2.0.1 Release, 2001. at http://xml.apache.org/xerces2-j/index.html.

[6] C.F. Goldfarb and P. Prescod. *The XML Handbook*. Prentice Hall, Inc., 1998.

[7] M. Hammer and D. McLeod. Database description with SDM: A Semantic Database Model. *ACM Transactions on Database Systems*, 6(3):351–386, 1981.

[8] Sun Microsystems Inc. JavaTM API for XML Processing (JAXP) 1.2. at http://java.sun.com/xml/jaxp/.

[9] R. Jelliffe and Academia Sinica Computing Centre. The Schematron: An XML Structure Validation Language using Patterns in Trees, 2001. at http://www.ascc.net/xml/resource/schematron/schematron.html.

[10] Oracle. XSQL Pages Publishing Framework, 2001. http://download-west.oracle.com/otndoc/oracle9i/ 901_doc/appdev.901/a88894/adx10xsq.htm.

[11] S. Simenov. Web services description language part. 1. *XML Journal*, 2(2):20–24, 2001.

[12] W3C. XML Path Language (XPath) Version 1.0 at http://www.w3.org/TR/xpath.html, 1999.

[13] W3C. XSL Transformations (XSLT) Version 1.0, 1999. at http://www.w3.org/TR/xslt/.

[14] W3C. Web Services Description Language (WSDL) 1.1, 2000. at http://www.w3.org/TR/wsdl/.

[15] W3C. XML Linking Language (XLinking) Version 1.0, 2001. at http://www.w3.org/TR/xlink/.

[16] W3C. XML Schema Part 1:Structures, 2001. at http://www.w3.org/TR/xmlschema-1/.

[17] W3C. XML Schema Part 2:Datatypes, 2001. at http://www.w3.org/TR/xmlschema-2/.

[18] W3C. XML Inclusions (XInclude) Version 1.0, 2002. at http://www.w3.org/TR/xinclude/.