XDerive: a namespace for defining derived elements in *XML Schema*[⋆]

Oscar Díaz, Felipe I. Anfurrutia

Dpto. de Lenguajes y Sistemas Informáticos, University of the Basque Country
Apdo. 649 - 20080 San Sebastián (Spain)
{jipdigao, jipibanf}@si.ehu.es

Abstract. XML is becoming the standard for document description in a B2B setting, and *XML Schema* is gaining wide acceptance as the schema language to define the structure of these documents. Transactional documents as those currently found in B2B applications, frequently comprise derived elements, i.e. elements whose content can be calculated through deriving function that examines the content of other elements. Based on this observation, this work advocates incorporating the notion of derived element in *XML Schema*. Despite its wide presence, the notion of derived element is not yet supported in *XML Schema*. To this end two issues are addressed: (1) the implicitness of current approaches to deriving function support for XML documents, and (2) the externality of some deriving data which participate in the deriving function. These aspects are respectively tackled both by (1) moving the derivation semantics to the document schema, and (2) by proposing an attachment where the deriving elements are recorded. These ideas have been realized by extending the Oracle´s XML Parser. Now, this parser can be configured to become "derivation aware". That is, the deriving functions can now be found in the *XML Schema*. At parsing time, these functions are interpreted, and the returned values become the content of the associated derived elements.

Keywords: XML Schema, Derived data, B2B applications

1 Introduction

Problem statement. The notion of document is at the heart of current business operations, and plays an increasing important role in B2B applications. Order, billing or delivery forms are all examples of transactional documents that collect 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.

Transactional documents frequently hold derived data, i.e. data which is calculated from other data. For example, the *totalAmount* of an order can be obtained from the cost of each item included in the order minus some applicable discount. Such deriving

[⋆] This contribution is an excerpt of a paper already presented at the IFIP WG8.1 conference on Engineering Infromation Systems in the Internet Context, Kanazawa (Japan), 2002. 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.

functions support important business policies, including terms and conditions (e.g. policies for price discounting), service provisions (e.g. policies for refunds), or surrounding services (e.g. policies for lead time to place an order). Such business policies are the subjects of contracts among partners, and can normally be found in catalogs, storefronts and marketplaces, as well as in bids and requests communicated during negotiations, procurements and auctions.

XML is becoming the standard for document description in a B2B setting. And *XML Schema* is gaining wide acceptance as the schema language to define the structure of these documents. However, there is not yet mechanism to describe derived elements. This leads to deriving functions being hard-coded in the applications which process the document. For example, deriving functions can be hidden within XSLT templates, or as scripts within the application which processes the XML document. But hard-coding these functions not only hinders development, it also jeopardizes maintenance. Hard-coding, distributing and repeating these functions along the applications that process XML documents would certainly complicate the modification of deriving functions, and thus, of the business strategies these functions support.

As an additional difficulty, these deriving functions frequently depend both on facts explicitly stated on the document but also on other data, external to the document itself. That is, deriving data is not always recorded in the document . For instance, *applicableDiscount* can be obtained from distinct business policies, such as *"5 percent discount if buyer has a history of loyal spending at the store"* or *"20 percent discount if the product is on offer"*. In this example, *applicableDiscount* has two deriving facts (i.e. the consumer's loyalty history, and the product state), that are kept outside the *order* document realm. This makes business partners other than the issuer, have a partial picture of how the *applicableDiscount* has been obtained, since the information about the state of the product (i.e. whether it is on offer or not) is not recorded in the order. Such situation hampers trust construction between partners. Besides, in case of a conflict, it will be difficult to resolve about which discount was applied, as the business document misses important facts.

Such a situation could be tackled by incorporating the deriving data(e.g. whether the product is on offer or not) into the business document. However, this would increase the size of the document, and makes the document schema dependent on the business policies. A change in these policies (better said, on the deriving data consulted by these policies) would affect the document schema, as the new deriving data need to be integrated in the document.

Another alternative could be to let partners consult the database of the document issuer to check the deriving data. But this only makes sense if the query is posed at the time the document is issued. Otherwise, the deriving data could have been updated and thus, no longer reflects the context in which the order was processed. Even, if a temporal database is used that supports data versioning, this still does not solve the problem, should a conflict arise. If the case is brought to trial, the data kept on the database cannot be used as evidence as it could have been easily manipulated by the document issuer. Neither the business partners nor the court will trust the data. This situation is normally overcome by the existence of a "trusted third party" whose main task is to monitor the business transaction.

This implies that the business document as well as the business context at the time the document was processed (i.e. the deriving elements) should be made explicit and available.

Our contribution. Previous observations have highlighted two issues: (1) the implicitness of current approaches to deriving function support for XML documents, and (2) the externality of some deriving data that participates in the deriving function.

The first issue is tackled by moving the deriving function to the document schema. This is aligned with the strategy of externalization promoted by XML. That is, while XML documents let you externalize data representations in a platform independent manner, *DTD* or *XML Schema* let you externalize the types and structural constraints that govern XML documents. Instead of implementing these constraints programmatically in each applications processing a given type of document, the constraints are moved to the *XML Schema*. In this way, the constraints that dictate the structure and validity of data are *externalized*. This saves the trouble of implementing these constraints in each application dealing with this data.

Likewise, moving the deriving functions to the *XML Schema* (i.e. externalizing derivation) promotes code reuse. Furthermore, it enhances consistency as all applications share a common derivation semantics, and eases maintenance since changes to the derivation semantics is localized in a single place.

However, it is not only a software issue. As *XML Schema* enables integration across the board for business-to-business connectivity, externalization of the derivation semantics makes explicit important business rules that govern the interaction between business partners, thus enhancing partner trustworthiness and hindering repudiation.

To this end, this work presents a model for extending *XML Schema* with derived elements. In this way, the schema is not only responsible for validating the instance document but it is also in charge of generating the derived values by applying the corresponding business policies. In so doing, this work eases the realization of the distinct *contract* vocabularies currently emerging [1], that focus on capturing the discovery-negotiation-execution life-cycle that models a business transaction. These agreements are frequently reflected in terms of business policies [1], of which deriving functions are a special realization.

The second issue refers to the externality of deriving data. Such externality can result in the loss of the “business context” where the document was generated (e.g. an order). This is solved by generating an attachment at the time the transaction (i.e. the order) occurs. Such an attachment holds a snapshot of the external deriving data which has been used to obtain the derived values. Such snapshots of the “business context” allows business partners other than the issuer’s one (included the trusted third party), to become aware of the state of the business at the time the transaction took place so as to validating the appropriate application of the business policies. Such a document is generated as a byproduct of the derivation process, and attached to the business document through an XML processing instruction. This approach has been realized by an implementation on the ORACLE’s XML Parser V2 for Java.

The rest of the paper is structured as follows. Section 2 introduces a running example that illustrates some the issues raised by this work. Section 3 addresses how *XML*

Schema can be extended to support derived elements. Section 4 tackles the externality of deriving elements. And in the final section, conclusions are given.

2 A running example

An *order* document is taken as an example. An order would usually contain information about the customer who placed the order, the order number and the individual line items on the order including their quantities and product references.

Transactional document such as the previous one, commonly, comprise derived elements. Next, we give a few examples, the business policies that regulate these derived elements. The examples have been taken or inspired by policies stated in *Amazon's* site. As we go, we will raise several issues such as the need for keeping track of the derived elements being used, and the need for prioritized conflict handling.

Example 1. Derived element: *subTotal*

– *(Rule) The subTotal is the sum of all the partial cost of the bought items*

This is an example where the deriving elements (i.e. *partialCost*) are local to the document, and only one rule is available to work out the derived value. Notice also that derived elements can be based on other derived elements (e.g. *partialCost* is in turn a derived element).

Example 2. Derived element: *partialCost*

– *(Rule) The partialCost is the multiplication of the quantity and the unitPrice of the current product*

Strictly speaking, a derived element should not provide any new information except that derived from other elements on the document. However, such an approach is too restrictive to reflect most of the policies that regulate current business transactions. Most policies consult a business state that is not always reflected in the document itself but in other data resources (e.g. databases, other XML documents such as catalogs, etc). Hence, we extend the potential deriving elements to any data resource available to the document. In this example, the *unitPrice* is not defined on the document itself but retrieved from the company's database.

This externality prevents the document receiver from having a complete picture of how the derived elements have been calculated, and hence, makes difficult the verification of the fulfillment of the agreement (reflected through the deriving function). Besides, such external elements evolve independently of the document itself so as to impede the re-creation of the business state at the time the document was generated (e.g. at the time the order took place).

This situation is tackled through the *derivationTrack* document. Such document records a snapshot of the external deriving elements at the time the derived elements were calculated. For the *partialCost* example, this implies that the *unitPrice* of the items is recorded in the *documentTrack*. The receiver of the order can then consult this attachment to verify the correct application of the business policies, which have been realized as deriving functions. Section 4 addresses this issue.

Example 3. Derived element: *deliveryTime*

- *(Rule A) If the shipping-speed is "Standard Shipping" then, the delivery time will be between 4 and 8 days*
- *(Rule B) If the shipping-speed is "Two-day Shipping", and at least three items are available within 24 hours, then the delivery time will be 3 days*
- *(Rule C) If the shipping-speed is "One-day Shipping" and at least three items are available within 24 hours, then the delivery time will be 2 days*
- *(Rule D) If the shipping-speed is either "One-day Shipping" or "Two-day Shipping", then the delivery time will be 4 days*
- *(Priority Rule) If both Rules B and D apply, then rule B "wins", i.e. the delivery time will be 3 days*
- *(Priority Rule) If both Rules C and D apply, then rule C "wins", i.e. the delivery time will be 2 days*

Here, the delivery policy is realized as a set of rules. We say a rule "applies" when the rule's body (i.e. the antecedent) is satisfied. If more than one rule applies simultaneously, then we can combine the values returned by each rule (value-based schema) or give priority to one of the rules (rule-based schema). The latter can be realized by assigning a priority to each rule, and then, the rule with the highest priority wins. However, this schema requires a global prioritization schema that is difficult to maintain for large or evolvable rule sets. Therefore, in a B2B setting we favor a partial prioritization schema where conflicts are resolved among smaller sets. This is the role played by the priority rules as the ones used in this example.

On the other hand,

Example 4. Derived element: *insuranceCost*

- *(Rule A) If any item exists whose category is "cd", then the insurance cost will be 1.99 euros*
- *(Rule B) If any item exists whose category is "book", then the insurance cost will be 2.99 euros*
- *(Rule C) If any item exists whose category is "electronic", then the insurance cost will be 5.99 euros*
- *(Rule D) If any item exists whose category is "electronic" and needs special hanging, then the insurance cost will be 6.99 euros*
- *(Priority Rule) If more than one rule applies, the final insurance cost will be the maximum of the distinct values returned*

These rules realize the insurance policy. Unlike the previous example, now a *value-based* prioritization schema has been followed. An order can contain distinct categories of items. Each category has a distinct insurance cost. The final insurance cost is not calculated as the result of a single rule but as the combination of the values returned by all applicable rules.

Summing up, derived data can be local or remote to the document itself, while the deriving function can be rule-based or value-based. These examples also demonstrate the importance of an explicit prioritization schema (normally in the form of meta rules). Such meta-rules also convey important business policies. From an engineering viewpoint, prioritized conflict handling enables significantly easier modification and more modular revision, both key features to face the evolvable nature of the B2B setting. Indeed, it has been reported that business policies are among the most evolvable software artifacts found in current information systems. New rules (policies) can be added, or meta-rules (strategies) can be updated with no impact in the rest of the rules. Such aspects enhance modularity and facilitate "locality in revision" [1].

3 Extending *XML Schema* with derived elements

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 [2], much in the same way as the "*create table*" statement defines the structure of the table tuples in SQL. Validating parsers can then be used to check conformance of documents claiming to be instances of a given schema.

This work addresses how *XML Schema* can be extended to specify derived elements. To this end, *XML Schema* offers the *annotation* element which allows the designer to provide further information about elements. *Annotation*s may contain *appinfo* or *documentation* elements. The former provides instructions targeted at the processing application. On the other hand, *documentation* elements are for human consumption. Since derived element semantic is targeted at the XML parser, it is enclosed in *appinfo* tags. Next sections look at the knowledge model (i.e. how to specify derived elements) and the execution model (i.e. how derived elements behave).

3.1 The knowledge model

This section addresses the specification of derived elements, particularly, the cardinality issue, and the description of the deriving function. It extends a preliminary account reported in the paper titled "*Extending XML Schema with Derived Elements*" [3].

A first issue is the cardinality of derived elements. *XML Schema* provides the *minOccurs* and *maxOccurs* attributes to indicate the allowable occurrence interval of a given element. 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 re-interpret 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 deriving function [1].

[1] Another alternative would have been to raise an error that indicates this situation.

As for the specification of the deriving function, a rule-based approach has been followed [1]. As an overall representation approach, rules capture well many aspects of what one would like to describe in automated contracts where business policies are reflected. Broadly speaking, a rule includes an antecedent or condition describing the situation where the rule applies, and an action which indicates how to calculate the derived element if the antecedent is met. When more than one rule can be applicable, a prioritization schema must be provided. In this case, the literature favors two approaches, namely, "value-based" or "rule-based" [4,5]. The former provides a combining function to "merge" the values returned by the distinct applicable rules. This means that all the applicable rules are executed. By contrast, a "rule-based" prioritization schema resorts to meta-rules as a way to resolve potential conflicts and select only one rule to execute. A final aspect is whether the deriving data is local to the document or should be found in remote sources.

A namespace, *XDerive*, is introduced to specify all the above aspects. This namespace is presented here through examples.

A simple example with local data: the *subTotal* element. Its value is calculated from the partial costs of the items being ordered. Its specification using *XDerive* follows:

```
<xs:element name="subTotal" type="xs:decimal" minOccurs="0">
  <xs:annotation>
    <xs:appinfo>
      <xd:derivationPolicy>
        <xd:rule id= "rule-1" test="lineItem/partialCost">
          <xd:derivedFunction>
            <xsl:value-of select="sum(lineItem/partialCost)"/>
          </xd:derivedFunction>
            <xs:documentation>
               subTotal sums all the partialCost of the bought items.
            </xs:documentation>
        </xd:rule>
      </xd:derivationPolicy>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

The <*appinfo*> tag holds this specification. The business policy is described through the <*derivationPolicy*> tag[2]. Its content includes the rules (<*rule*> tag) that support the deriving function. The rule's antecedent is supported through the "*test*" attribute that holds an XPath expression, i.e. a predicate on the content of the instance document. In this case, the test checks that there exists *partialCost* elements. As for the rule's consequent, which is described by the <*derivedFunction*> tag, it is realized using standard XSLT instructions. The example uses <*value-of select= ".....">* to obtain the value of the derived element using the "*sum()*" XPath function with the *partialCost* elements as parameter. The latest instruction must be <*value-of ...*> for simple type values and <*copy-of ...*> for complex type values. Finally, a <*documentation*> tag has been added

[2] The prefix *xd:* indicates the namespace where this tag is defined.

for documentation purposes. Notice that no prioritization schema has been included as only one rule is provided.

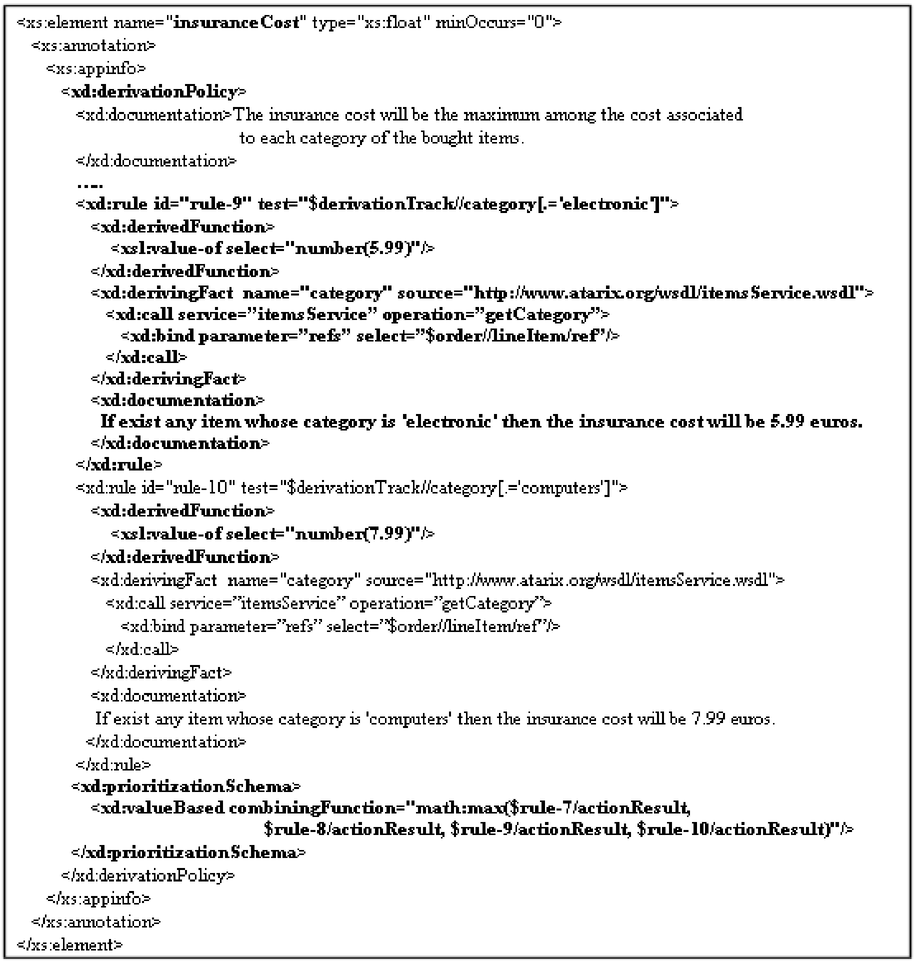
An example with remote data and a value-based prioritization schema: the *insuranceCost* element. Its value is the maximum of the costs associated with the *category* of distinct items included in the order. Its specification in *XDerive* is shown in figure 1. In this example, the derivation policy implies more than one rule. If the order comprises items of distinct categories, more than one rule can be applied. The <*prioritizationSchema*> tag indicates how to resolve this ambiguity. In this case, the designer chooses to combine the values returned by the distinct applicable rules (held by the *actionResult* element of the rule), and takes the highest one. This is supported by the mathematical *max* function specified at the *combiningFunction* attribute. This attribute is held by the <*valueBased*> tag.

This example also illustrates the use of remote data. Here, the rule's antecedent checks the item's *category*. As this data is not available in the base document, the track document is consulted (referred to through the variable *$derivationTrack*). The structure and generation of this document is postponed till section 4. Notice however, that the source of the external data should be indicated. The <*derivingFact*> tag serves this purpose. This tag holds the name of the external deriving data (*"name"* attribute), the location of the source to be used to retrieve this data, i.e. the WSDL document's URL (*"source"* attribute), and the calling statement to the specific operation which returns the deriving fact (the <*call*> element).

Although this approach includes the Web service to be used as part of the schema of the base document, it is debatable whether this information should have been better recorded in the track document itself. The rationale for our decision is that the track document is a *generated* document, not a specification document. So all information should be included in the schema of the base document. A second aspect is modularity and locality. By associating the source of the deriving data together with the rule that uses such data, the removal/addition of rules is greatly facilitated. Even if this leads to repeating the source several times (if the same external data is used in distinct rules), the benefits outweigh this drawback.

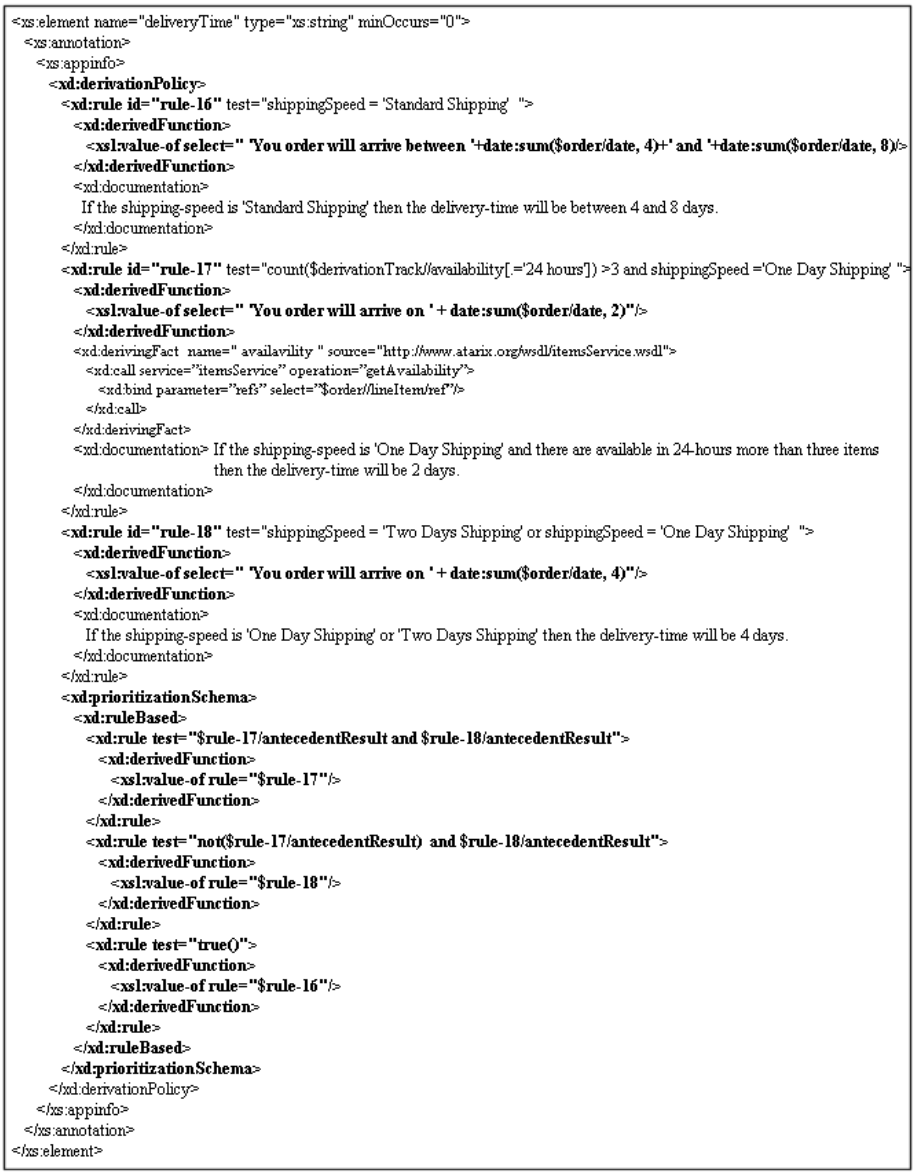
An example with remote data and a rule-based prioritization schema: the *deliveryTime* element. Its value depends on the *shippingSpeed* and the *availability* of the items in stock. Its specification in *XDerive* is shown in figure 2. In this example, the derivation policy also comprises several rules. However, unlike the previous example, only one rule is applied, even if the antecedent of several rules is met. That is, the prioritization schema is now rule-based. This implies that the system checks the rule's antecedents, keeps this result in a special element termed *antecedentResult*, and finally, applies the priority rules. The latter leads to the application of one of the base rules.

To this end, the <*prioritizationSchema*> tag now contains a <*ruleBased*> tag, which in turn contains the priority rules. Since a priority rule is a rule, its specification is similar to a base rule. But now the subject matter are the base rules. That is, a priority rule's test checks whether base rules can be applied or not, while its consequent indicates the base rule to be applied. An example follows:



```
<xs:element name="insuranceCost" type="xs:float" minOccurs="0">
  <xs:annotation>
    <xs:appinfo>
      <xd:derivationPolicy>
        <xd:documentation>The insurance cost will be the maximum among the cost associated
                                      to each category of the bought items.
        </xd:documentation>
        ....
        <xd:rule id="rule-9" test="$derivationTrack//category[.='electronic']">
          <xd:derivedFunction>
             <xsl:value-of select="number(5.99)"/>
          </xd:derivedFunction>
          <xd:derivingFact  name="category" source="http://www.atarix.org/wsdl/itemsService.wsdl">
            <xd:call service="itemsService" operation="getCategory">
              <xd:bind parameter="refs" select="$order//lineItem/ref"/>
            </xd:call>
          </xd:derivingFact>
          <xd:documentation>
           If exist any item whose category is 'electronic' then the insurance cost will be 5.99 euros.
          </xd:documentation>
        </xd:rule>
        <xd:rule id="rule-10" test="$derivationTrack//category[.='computers']">
          <xd:derivedFunction>
             <xsl:value-of select="number(7.99)"/>
          </xd:derivedFunction>
          <xd:derivingFact  name="category" source="http://www.atarix.org/wsdl/itemsService.wsdl">
            <xd:call service="itemsService" operation="getCategory">
              <xd:bind parameter="refs" select="$order//lineItem/ref"/>
            </xd:call>
          </xd:derivingFact>
          <xd:documentation>
           If exist any item whose category is 'computers' then the insurance cost will be 7.99 euros.
         </xd:documentation>
        </xd:rule>
       <xd:prioritizationSchema>
          <xd:valueBased combiningFunction="math:max($rule-7/actionResult,
                                    $rule-8/actionResult, $rule-9/actionResult, $rule-10/actionResult)"/>
       </xd:prioritizationSchema>
      </xd:derivationPolicy>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

Fig. 1. The specification of *insuranceCost* using *XDerive*.



```
<xs:element name="deliveryTime" type="xs:string" minOccurs="0">
  <xs:annotation>
    <xs:appinfo>
      <xd:derivationPolicy>
        <xd:rule id="rule-16" test="shippingSpeed = 'Standard Shipping'  ">
          <xd:derivedFunction>
            <xsl:value-of select=" 'You order will arrive between '+date:sum($order/date, 4)+' and '+date:sum($order/date, 8)/>
          </xd:derivedFunction>
          <xd:documentation>
           If the shipping-speed is 'Standard Shipping' then the delivery-time will be between 4 and 8 days.
          </xd:documentation>
        </xd:rule>
        <xd:rule id="rule-17" test="count($derivationTrack//availability[.='24 hours']) >3 and shippingSpeed ='One Day Shipping' ">
          <xd:derivedFunction>
            <xsl:value-of select=" 'You order will arrive on ' + date:sum($order/date, 2)"/>
          </xd:derivedFunction>
          <xd:derivingFact  name=" availavility " source="http://www.atarix.org/wsdl/itemsService.wsdl">
            <xd:call service="itemsService" operation="getAvailability">
              <xd:bind parameter="refs" select="$order//lineItem/ref"/>
            </xd:call>
          </xd:derivingFact>
          <xd:documentation> If the shipping-speed is 'One Day Shipping' and there are available in 24-hours more than three items
                             then the delivery-time will be 2 days.
          </xd:documentation>
        </xd:rule>
        <xd:rule id="rule-18" test="shippingSpeed = 'Two Days Shipping' or shippingSpeed = 'One Day Shipping'  ">
          <xd:derivedFunction>
            <xsl:value-of select=" 'You order will arrive on ' + date:sum($order/date, 4)"/>
          </xd:derivedFunction>
          <xd:documentation>
            If the shipping-speed is 'One Day Shipping' or 'Two Days Shipping' then the delivery-time will be 4 days.
          </xd:documentation>
        </xd:rule>
        <xd:prioritizationSchema>
          <xd:ruleBased>
            <xd:rule test="$rule-17/antecedentResult and $rule-18/antecedentResult">
              <xd:derivedFunction>
                <xsl:value-of rule="$rule-17"/>
              </xd:derivedFunction>
            </xd:rule>
            <xd:rule test="not($rule-17/antecedentResult)  and $rule-18/antecedentResult">
              <xd:derivedFunction>
                <xsl:value-of rule="$rule-18"/>
              </xd:derivedFunction>
            </xd:rule>
            <xd:rule test="true()">
              <xd:derivedFunction>
                <xsl:value-of rule="$rule-16"/>
              </xd:derivedFunction>
            </xd:rule>
          </xd:ruleBased>
        </xd:prioritizationSchema>
      </xd:derivationPolicy>
    </xs:appinfo>
  </xs:annotation>
</xs:element>
```

Fig. 2. The specification of *deliveryTime* using *XDerive.*

```
<xd:rule test= “$rule-17/conditionResult and $rule-18/conditionResult”>
  <xd:derivedFunction>
    <xsl:value-of select= “$rule-17” />
  </xd:derivedFunction>
</xd:rule>
```

This priority rule checks whether *rule-17* and *rule-18* are both met by the current order. If so, the priority rule resolves the conflict by applying *rule-1*7. In the current version, the rule engine assumes disjunctive test for the priority rules. An the end, a rule that is always activated (i.e. the antecedent is always true) is introduced to indicate the situation where none of the previous test are met.

3.2 The execution model

This model addresses *when* deriving functions are executed. In the database context, the answer is 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, with (2) the cost to calculate the derived data each time it is required [6].

Unlike derived data (as found in the database area), a transactional document should not be kept continuously consistent with the deriving data which participate in the deriving functions included in the document’s schema. Therefore, the dynamic option has not been contemplated. Derived elements are obtained as an aside process at parsing time. This process takes the “base document” as an input, and generates the “derived document” plus the “track document” (see section 4).

Due to the static nature of transactional documents, this work does not contemplate the possibility of updating the derived element directly nor the updating propagation issue. The latter involves how modifications on the derived data are propagated to the deriving data. The challenge lies on the ambiguity of this transformation as the propagation is not always unique [7].

4 The *derivationTrack* document

A transactional document reflects an agreement between the distinct partners involved in the transaction. Paper-based documents usually have an attachment (e.g. conditions on sale) that describes the context in which this agreement is set. Such an attachment plays an important legal role in case of any demand or complaint about the transaction. Likewise, “virtual documents” also need a similar attachment. This is particularly so in the presence of derived elements. Here, deriving data can be updated once the transaction occurs (e.g. the item price can be modified) so as to prevent the customer from recreating the context in which his order was processed.

To this end, this work proposes the *derivationTrack* document, a document which tracks the derivation process so that this derivation can be consulted at any moment by any of the partners involved. This document is automatically generated during the

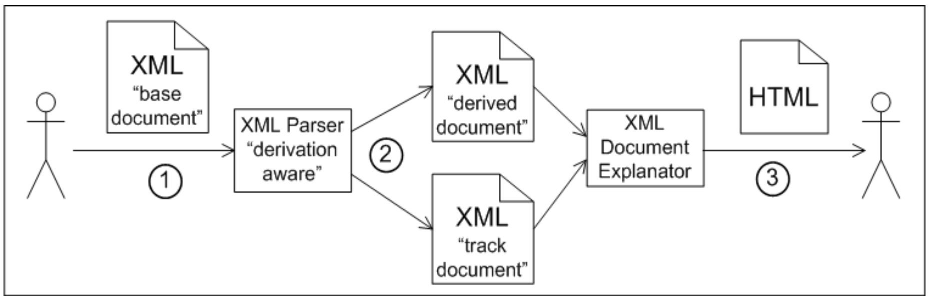


Fig. 3. Document life-cycle: the activating, parsing and explanation stages.

derivation process, and associated to the transactional document by means of the *<?xderive-derivationTrack ... ?>* processing instruction .

Although such an approach can sound bizarre at first glance, it is similar to current practices to indicate the presentation of an XML document. In this case, how a document is rendered is separately described in an XSLT file. This XSLT file is attached to the data document through the *<?xml-stylesheet ..>* tag. When the document is processed according to its stylesheet, the rendering is obtained.

Likewise, how a document has been derived can also be separately indicated in a distinct XML file: the *derivationTrack* file. When this document is processed according to its "track document", both the distinct business policies and the consulted deriving values are made explicit.

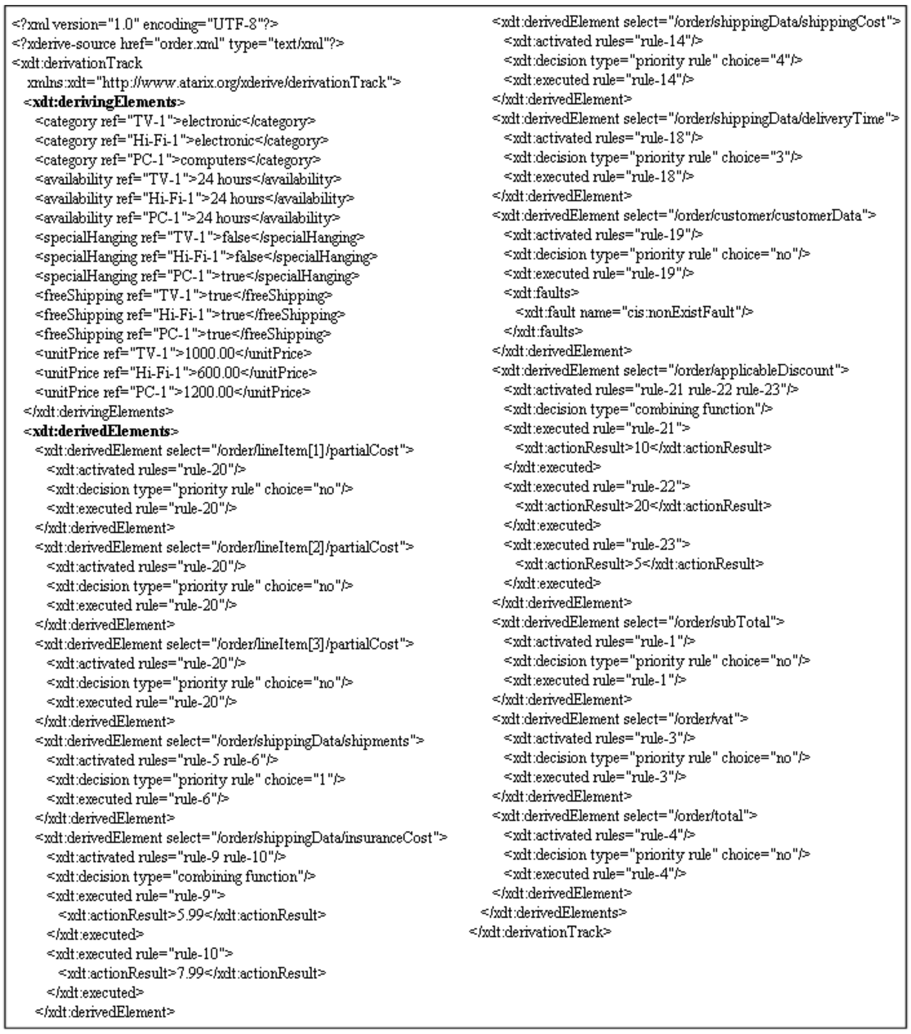
The envisioned situation is depicted in figure 3 where the following stages are contemplated:

1. activating stage. The activator (e.g. either a customer or an application that sends the order automatically if the stock goes below a certain threshold) kicks off the whole process by submitting the "base document",
2. parsing stage. At the issuer place, the derivation-aware parser takes the base document as an input and generates both the "derived document" and the "track document".
3. explanation stage. Finally, either the activator, the issuer or the trusted third party, can process the "derived document" in explanation mode so as to ascertain both the policies and deriving facts applied.

Therefore, the *derivationTrack* file records the values of the external deriving data (the *<derivingElements>* tag), and the rules which have been applied during the derivation process (the *<derivedElements>* tag). Figure 4 gives an example.

The *derivationTrack* document keeps each external deriving data as an element. It is worth emphasizing that the parser only records the deriving data that is used in the derivation process. An example follows:

<category ref= "TV-1"> electronic </category>



```
<?xml version="1.0" encoding="UTF-8"?>
<?xderive-source href="order.xml" type="text/xml"?>
<xdt:derivationTrack
   xmlns:xdt="http://www.atarix.org/xderive/derivationTrack">
  <xdt:derivingElements>
    <category ref="TV-1">electronic</category>
    <category ref="Hi-Fi-1">electronic</category>
    <category ref="PC-1">computers</category>
    <availability ref="TV-1">24 hours</availability>
    <availability ref="Hi-Fi-1">24 hours</availability>
    <availability ref="PC-1">24 hours</availability>
    <specialHanging ref="TV-1">false</specialHanging>
    <specialHanging ref="Hi-Fi-1">false</specialHanging>
    <specialHanging ref="PC-1">true</specialHanging>
    <freeShipping ref="TV-1">true</freeShipping>
    <freeShipping ref="Hi-Fi-1">true</freeShipping>
    <freeShipping ref="PC-1">true</freeShipping>
    <unitPrice ref="TV-1">1000.00</unitPrice>
    <unitPrice ref="Hi-Fi-1">600.00</unitPrice>
    <unitPrice ref="PC-1">1200.00</unitPrice>
  </xdt:derivingElements>
  <xdt:derivedElements>
    <xdt:derivedElement select="/order/lineItem[1]/partialCost">
      <xdt:activated rules="rule-20"/>
      <xdt:decision type="priority rule" choice="no"/>
      <xdt:executed rule="rule-20"/>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/lineItem[2]/partialCost">
      <xdt:activated rules="rule-20"/>
      <xdt:decision type="priority rule" choice="no"/>
      <xdt:executed rule="rule-20"/>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/lineItem[3]/partialCost">
      <xdt:activated rules="rule-20"/>
      <xdt:decision type="priority rule" choice="no"/>
      <xdt:executed rule="rule-20"/>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/shippingData/shipments">
      <xdt:activated rules="rule-5 rule-6"/>
      <xdt:decision type="priority rule" choice="1"/>
      <xdt:executed rule="rule-6"/>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/shippingData/insuranceCost">
      <xdt:activated rules="rule-9 rule-10"/>
      <xdt:decision type="combining function"/>
      <xdt:executed rule="rule-9">
        <xdt:actionResult>5.99</xdt:actionResult>
      </xdt:executed>
      <xdt:executed rule="rule-10">
        <xdt:actionResult>7.99</xdt:actionResult>
      </xdt:executed>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/shippingData/shippingCost">
      <xdt:activated rules="rule-14"/>
      <xdt:decision type="priority rule" choice="4"/>
      <xdt:executed rule="rule-14"/>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/shippingData/deliveryTime">
      <xdt:activated rules="rule-18"/>
      <xdt:decision type="priority rule" choice="3"/>
      <xdt:executed rule="rule-18"/>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/customer/customerData">
      <xdt:activated rules="rule-19"/>
      <xdt:decision type="priority rule" choice="no"/>
      <xdt:executed rule="rule-19"/>
      <xdt:faults>
        <xdt:fault name="cis:nonExistFault"/>
      </xdt:faults>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/applicableDiscount">
      <xdt:activated rules="rule-21 rule-22 rule-23"/>
      <xdt:decision type="combining function"/>
      <xdt:executed rule="rule-21">
        <xdt:actionResult>10</xdt:actionResult>
      </xdt:executed>
      <xdt:executed rule="rule-22">
        <xdt:actionResult>20</xdt:actionResult>
      </xdt:executed>
      <xdt:executed rule="rule-23">
        <xdt:actionResult>5</xdt:actionResult>
      </xdt:executed>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/subTotal">
      <xdt:activated rules="rule-1"/>
      <xdt:decision type="priority rule" choice="no"/>
      <xdt:executed rule="rule-1"/>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/vat">
      <xdt:activated rules="rule-3"/>
      <xdt:decision type="priority rule" choice="no"/>
      <xdt:executed rule="rule-3"/>
    </xdt:derivedElement>
    <xdt:derivedElement select="/order/total">
      <xdt:activated rules="rule-4"/>
      <xdt:decision type="priority rule" choice="no"/>
      <xdt:executed rule="rule-4"/>
    </xdt:derivedElement>
  </xdt:derivedElements>
</xdt:derivationTrack>
```

Fig. 4. A possible *derivationTrack* document attached to the *order* document.

An attribute, *“ref”*, is added to relate the element on the *derivationTrack* document with the associated element in the base document. Here, the attribute *“ref”* indicates that this is the category of the item *“TV-1”* to be found in the *order* document. Therefore, this work is based on the understanding that the base document should provide the means to univocally identify the entities participating in the transaction. As an example, consider an order. The base document should clearly indicate the customer and the items bought. This is normally achieved through a reference or code. In our example, the customer is identified through the <*id*> tag while an item holds a <*ref*> tag.

Besides deriving elements, the track document keeps information about the derivation process through the <*derivedElement*> tag. An example follows:

```
<xdt:derivedElement select="/order/shippingData/insuranceCost">
  <xdt:activated rules="rule-9 rule-10"/>
  <xdt:decision type="combining function" />
  <xdt:executed rule="rule-9">
    <xdt:actionResult>5.99</xdt:actionResult>
  </xdt:executed>
  <xdt:executed rule="rule-10">
    <xdt:actionResult>7.99</xdt:actionResult>
  </xdt:executed>
</xdt:derivedElement>
```

The data recorded includes the element derived (*“select”* attribute), the rules activated (<*activated*> tag), the decision taken, if more than one rule was applicable (the <*decision*> tag), and the rules that were finally executed (<*executed*> tag). The example above states that *insuranceCost* has been calculated after applying a combining function (e.g. the mathematical *max* function, see figure 1) to the results of the execution of the rule being applied (i.e. *rule-9* and *rule-10*).

The rationale for having a *derivationTrack* document is to allow business partners to validate the conformance of the business transaction. As an example, consider an application that renders pending orders (e.g. similar to the one offered by *Amazon*). Partnership trustworthiness can be enhanced by allowing the customer not only to check the status and data about the order, but also ease the way for the customer to understand how the delivery time or insurance cost has been calculated. Currently, this involves the user reading through, potentially, several pages at the retailer’s site. This is lengthy and cumbersome, and can put the user off. The rationale here is similar the one found for the explanation mechanisms provided by some Expert Systems: improving the confidence of the user on the transparency and accuracy of the system. This is also so in a B2B setting.

5 Conclusions

This work provides some insights on how *XML Schema* can be extended with derived elements. To this end, two issues have been discussed: deriving function specification and the externality of deriving data. The former has been addressed by introducing the *XDerive* vocabulary, a rule-based approach to the definition of the deriving function.

As for the problems posed by the externality of deriving data, this work proposes the existence of a track document which records the state and business policies used at the time the transaction occurs.

References

1. Grosof, B., Labrou, Y., Chan, H.: A Declarative Approach to Business Rules in Contracts: Courteous Logic Programs in XML. In: Proceedings of the 1st ACM Conference on Electronic Commerce (EC99). (1999) 68–77
2. W3C: XML Schema Part 1:Structures (2001) at http://www.w3.org/TR/xmlschema-1/.
3. Ibanez, F., Díaz, O., Rodríguez, J.J.: Extending xml schema with derived elements. In: IFIP WG8.1 Working Conference on Engineering Information Systems in the Internet Context. (2001)
4. Ioannidis, Y., Sellis, T.: Supporting Incosistent Rules in Database Systems. Journal of Intelligent Information Systems (1992) 243–270
5. Hull, R., et al.: Declarative Workflows that Support Easy Modification and Dynamic Browsing. In: Proceedings ACM International Joint Conference on Work Activities Coordination and Collaboration (WACC'99). (1999) 69–78
6. Connolly, T., Begg, C.: Database Systems. Addison Wesley (1998)
7. Chen, I.A., McLeod, D.: Derived Data Update in Semantic Databases. In: Proceedings of the Fifteenth International Conference on Very Large Data Bases, Amsterdam, The Netherlands (1989) 225–235