

Using XML to Support Media Types

Markus Kirchberg, Klaus-Dieter Schewe, Alexei Tretiakov

Massey University, Information Science Research Centre

Private Bag 11 222, Palmerston North, New Zealand

[m.kirchberg|k.d.schewe|a.tretiakov]@massey.ac.nz

Abstract: The *eXtensible Markup Language* (XML) has drawn significant attention in research and practice of databases and web-based information systems. At the same time, lots of work has been investigated in conceptual modelling for web-based information systems which is at most loosely coupled with XML. One such research direction has led to the theory of media types, which has proven itself to be useful for the design and development of large and maintainable web-based information systems.

The research reported in this article investigates how XML and the theory of media types could be brought together, i.e. how XML could be used to support media types. It turns out that some of the striking features of media types are not yet well supported by XML. Therefore, several extensions to XML such as subelements, update operations, schema updates, views, media elements and adaptivity will be suggested in this article in order to close this gap.

1 Introduction

The *eXtensible Markup Language* (XML) intends to combine and integrate the areas of databases and the world-wide web [2, 10]. Therefore, XML has drawn significant attention in research and practice of databases and web-based information systems. It should be expected that XML will play a significant role in the development of web-based information systems. If so, several problems have to be solved.

XML has to provide adequate query languages. In this area the database community has been quite active and defined several languages such as LOREL [3], UNQL [4], YAP [6], XML-QL [7] and XQuery [17]. However, recursive fixed-point queries that are quite well understood in the field of deductive databases [1] are not yet supported by these languages, though there are good reasons why languages of higher expressiveness are needed for the web [2, 9].

XML has to provide update languages. Some little work in this direction has also started and some update languages have been proposed [11, 12, 15], but not yet reached the discussion in the W3C consortium. One reason for this might be that updates may violate the structure that has been defined by the schema. If arbitrary updates are permitted, the usefulness of the schema may be questioned. If however updates are rejected because they violate the schema, we are confronted with the argument (given among others in [2]) that structures of data on the web are not very stable, so rejecting updates may be too restricted.

In order to solve this problem simultaneous implicit or explicit schema updates should be supported as well, hence, a schema definition language with structure suitable for schema evolution is also required.

The remainder of the article is organised as follows: In Section 2 we introduce some extensions to DTDs which capture the most relevant features of types and schemata. These extensions can be easily adapted to some of the XML schema languages such as XML-SCHEMA [18]. The most important extension is the introduction of subelements in order to support subtyping and to enable schema updates and adaptivity. In Section 3 we discuss update operations and schema updates, which lead to a semantics of evolving schemata. Section 4 is devoted to show that the extensions allow XML to be used for the implementation of media types: a construct proving to be highly useful in the design of large-scale web-based information systems [8, 9, 14, 16]. In this context we discuss views and the cohesion pre-orders as a way to support adaptivity.

2 Types and Schemata

We base our type definition language with support for evolving schemata on XML DTDs, because they are syntactically more compact, than it is the case for schema languages in XML format, such as XML SCHEMA [18].

2.1 Type Definitions

Types to be considered are (using abstract syntax) $t = ID \mid b \mid \epsilon \mid t^0 \mid t^* \mid t^+ \mid t_1, \dots, t_n \mid t_1 \oplus \dots \oplus t_n$. Here, b represents as usual a collection of base types. ID is a type representing a not further specified set of *identifiers*, and ϵ is a type representing just an empty sequence or tuple. t^* and t^+ represent arbitrary or non-empty sequences, respectively, with values of type t . t^0 represents values of type t or the empty sequence. Finally, t_1, \dots, t_n represents tuples and $t_1 \oplus \dots \oplus t_n$ represents a union.

More formally, we can associate with each type t a *domain* $dom(t)$, defined as follows:

$$\begin{aligned}
dom(ID) &= OID \\
dom(b_i) &= V_i \text{ for all base types } b_i \\
dom(\epsilon) &= \{()\} \\
dom(t^*) &= \{(a_1, \dots, a_k) \mid k \in \mathbb{N}, a_i \in dom(t) \text{ for all } i = 1, \dots, k\} \\
dom(t^+) &= \{(a_1, \dots, a_k) \mid k \in \mathbb{N}, k \neq 0, a_i \in dom(t) \text{ for all } i = 1, \dots, k\} \\
dom(t^0) &= \{(a) \mid a \in dom(t)\} \cup \{()\} \\
dom(t_1, \dots, t_n) &= \{(a_1, \dots, a_n) \mid a_i \in dom(t_i) \text{ for all } i = 1, \dots, n\} \\
dom(t_1 \oplus \dots \oplus t_n) &= dom(t_1) \cup \dots \cup dom(t_n)
\end{aligned}$$

For our purposes we will need an extension of subtyping to the particular type system used

here. *Subtyping* is defined by a partial order \leq on types:

- any type is a subtype of ϵ ;
- we have $t^+ \leq t \leq t^0$ and $t^+ \leq t^* \leq t^0$ for all types t ;
- we have $t^* \leq (t')^*$, $t^+ \leq (t')^+$ and $t^0 \leq (t')^0$ for all types with $t \leq t'$;
- we have $t_1, \dots, t_m \leq t'_{\sigma(1)}, \dots, t'_{\sigma(n)}$ for a monotonic, injective $\sigma : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$ and $t_{\sigma(i)} \leq t'_{\sigma(i)}$ for all $i = 1, \dots, n$;
- we have $t'_{\sigma(1)} \oplus \dots \oplus t'_{\sigma(n)} \leq t_1 \oplus \dots \oplus t_m \leq$ for a monotonic, injective $\sigma : \{1, \dots, n\} \rightarrow \{1, \dots, m\}$ and $t'_{\sigma(i)} \leq t_{\sigma(i)}$ for all $i = 1, \dots, n$.

2.2 Elements and Schemata

Now we can define a *schema* basically as a sequence of element declarations, but we will extend this in the sequel. An *element declaration* has the form

`<!ELEMENT name expression KIND = kind>`,

where `name` is an arbitrarily chosen name for the element, `kind` can take one of the values 'db', 'media' or 'aux' indicating, whether the element represents a building block of the database, a media type, or an auxiliary element.

Note that the inclusion of `KIND` is merely syntactical sugar and can be omitted. Elements of type 'db' are meant to indicate a database type. Following the idea of object oriented databases [13] it is useful to restrict references in such a way that they only occur between database elements. Also update operations—at least transactions—could be attached to such database elements. We shall see in Section 4 that the situation is similar for media types. In particular, we will need a defining view for them. The type 'aux' refers to all the remaining elements. Thus, the type information can be inferred, but in the sense of providing checkable context conditions on the syntax they are still useful. `expression` is either a base type or a regular expression e made out of element names n : $e = n \mid \epsilon \mid (e)^* \mid (e)^+ \mid (e)^0 \mid (e_1, \dots, e_n)$.

We can extend domains in the obvious way to elements. For an element with the name n and the defining expression e we obtain the domain $dom(n)$ as a set of values $\langle n \rangle \bar{e} \langle /n \rangle$ with

$$\begin{aligned} \bar{e} &\in V_i, \text{ if } e \text{ is the base type } b_i \\ \bar{e} &\in \{v_1 \dots v_k \mid v_i \in dom(n_i) \text{ for } i = 1, \dots, k\}, \text{ if } e = (n_1, \dots, n_k) \\ \bar{e} &\in \{v_1 \dots v_k \mid k \in \mathbb{N}, v_i \in dom(n') \text{ for } i = 1, \dots, k\}, \text{ if } e = (n')^* \\ \bar{e} &\in \{v_1 \dots v_k \mid k \in \mathbb{N}, k \neq 0, v_i \in dom(n') \text{ for } i = 1, \dots, k\}, \text{ if } e = (n')^+ \end{aligned}$$

For $e = (n')^0$ \bar{e} must be either in $dom(n')$ or be simply omitted. For $e = \epsilon$ we always omit \bar{e} . As usual in XML we abbreviate empty elements by $\langle n / \rangle$.

We may add attributes to elements. An *attribute declaration* has the form `<!ATTRIBUTE element name type key>`, where `element` is the name of an element, `name` is an arbitrarily chosen attribute name, `type` is either a base type, *ID* or one of the three possibilities `REF name'`, `REFS name'` or `SUP name'` indicating single or multiple references, and `key` is one of `#REQUIRED` or `#IMPLIED` indicating, whether the attribute is mandatory or not.

Some comments are due for this form of attribute declarations. A minor change with respect to XML is that we prefer a separate declaration for each attribute instead of an attribute-list declaration. We also omitted further options for keys, but only for brevity. Similar to XML SCHEMA we added the possibility to specify base types other than `CDATA` or `PCDATA`, which is also only a minor extension. The major deviation from XML is with identifiers and references:

- For elements of kind 'db' or 'media' we require a unique attribute of type *ID*; for elements of kind 'aux' there should not be such an attribute. This is in line with the distinction between types and classes (or database types, respectively) in OODBs [13] (or web-based information systems, respectively [9, 14]).
- For references we specify the target element, which must have an attribute of type *ID*, thus will be of kind 'db' or 'media'. If the element is used within an element of kind 'db' or 'media', then the target element must have the same kind. This ensures that references are either between database elements or media elements.
- References introduced with the keyword `SUP` should be used for subclassing. This leads to the additional condition that the referenced target element should uniquely determine the referencing element.

Example 1 We now look at an example for a schema. As we still need more extensions to talk about media types, the schema will only contain elements of type 'db' and 'aux'.

```
<!SCHEMA db [
<!ELEMENT db (person|student|lecturer)*>
<!ELEMENT person (name, birthday, address) TYPE = db>
<!ATTRIBUTE person id ID #REQUIRED>
<!ELEMENT name (first_name0, last_name) TYPE = aux>
<!ELEMENT first_name STRING TYPE = aux>
<!ELEMENT last_name STRING TYPE = aux>
<!ELEMENT birthday DATE TYPE = aux>
<!ELEMENT address (street, city, zip0) TYPE = aux>
<!ELEMENT street STRING TYPE = aux>
<!ELEMENT city STRING TYPE = aux>
<!ELEMENT zip STRING TYPE = aux>
<!ELEMENT student (student_id, start) TYPE = db>
<!ATTRIBUTE student id ID #REQUIRED>
<!ATTRIBUTE student isa SUP person #REQUIRED>
<!ATTRIBUTE student supervisor REF lecturer #IMPLIED>
```

```

<!ELEMENT student_id NAT TYPE = aux>
<!ELEMENT start DATE TYPE = aux>
<!ELEMENT lecturer (staff_id, field) TYPE = db>
<!ATTRIBUTE lecturer id ID #REQUIRED>
<!ATTRIBUTE lecturer isa SUP person #REQUIRED>
<!ATTRIBUTE lecturer teaches REFS student #IMPLIED>
<!ELEMENT staff_id NAT TYPE = aux>
<!ELEMENT field STRING TYPE = aux }

```

Of course, an *instance* of a schema db is a value in $dom(db)$, such that for all elements with an attribute of type ID , the identifiers in the instance are globally unique, and identifiers used in references must appear as values of such attributes in the referenced element of type 'db' (or 'media'). For brevity we dispense with an example for such an instance.

2.3 Subelements

Instability of a schema means that updating an instance may violate the schema prescription. As the regular expressions already provide optionality, the major problem occurs with adding elements. For instance, in Example 1 we might add a country to an address, a title to a name, or we might obtain several first names. One way to deal with such additions is to use subelements.

A *subelement* of an element named n is specified by an element declaration with the same name n and an extended defining regular expression, i.e., if e and e' are the regular expressions in the declarations of the element and the subelement, respectively, then the type defined by e' must be a subtype of the type defined by e .

So, in order to obtain a subelement, we just have to add elements to a sequence at any level of nesting, or whenever we requested a single occurrence of some element (optional or not) we would allow multiple occurrences in the subelement. Schema changing updates should at most introduce subelements. Note that XML-Schema [18] also supports some form of subtyping. However, the form of subtyping introduced above also covers iteration, optionality and nesting.

Formally, a schema should not allow more than one declaration of an element. So we indicate the relation of the subelement to the element by an index, i.e., we use *subelement declarations* of the form $\langle !SUBELEMENT \text{ name } \text{ expression } \text{ INDEX} = \text{ index} \rangle$, where name is the name of an existing element, expression is the new defining regular expression for the subelement leading to a subtype, and index is determined by an element of an index tree.

We do not have to repeat the type of the element. However, the index should appear in instances as an attribute value, as if we had an implicit attribute declaration $\langle !ATTRIBUTE \text{ name } \text{ index } \text{ INDEX } \#REQUIRED \rangle$.

An *index tree* is a non-empty set of sequences of positive integers $I \subseteq (\mathbb{N} - \{0\})^*$ such that $\alpha(k+1) \in I$ implies $\alpha k \in I$ and $\alpha \in I$. In particular, $\epsilon \in I$, which gives us the index

for the originally defined element. For subelements we chose indices (i_1, \dots, i_k) such that all prefixes could be associated with a super-type in such a way that no more intermediate types are possible.

Example 2 Take the element name from Example 1. Suppose an update requests that we allow multiple first names and a title. Thus, we obtain a subelement

```
<!SUBELEMENT name (first_name*, last_name, title)
                INDEX = (1,1)>.
```

Of course, we would also obtain a new element title, say

```
<!ELEMENT title STRING TYPE = aux>.
```

In this case, possible subelements with index (1) could be either

```
<!SUBELEMENT name (first_name0, last_name, title) INDEX = (1)>
or <!SUBELEMENT name (first_name*, last_name) INDEX = (1)>.
```

3 Evolving Schemata

In the last section we envisioned the possibility of schema changing updates and introduced subelements to cope with this. Now let us take a closer look into operations. In principle, we should allow updates on all elements. Precisely, an operation defined on an element named n should be allowed to change everything in the tree defined by the element declaration for n . This includes additions that would require the schema to be extended by new subelements and elements.

The language additions we propose for update operations are quite similar to the work by Tatarinov [15]. However, we make a clear distinction between updating a uniquely determined element or all elements matching a certain expression¹.

Following an argument in [2] we should assume that updates on XML documents may easily violate the schema. Of course, we could identify minimal schema changes and adapt the schema accordingly, but this violates the intention behind schemata to put explicit restrictions on updates. On the other hand, forbidding such updates is too restrictive. Therefore, we suggest some kind of a compromise approach. Allow only such updates, which only lead to additional subelements. In all other cases an explicit schema update is due.

3.1 Update Operations

An operation on elements `element` can be introduced by an *operation declaration* of the form `<!OPERATION element name MODE = mode>`, where `name` is an arbitrarily chosen name for the operation and `mode` can be one of ‘aux’ or ‘transaction’. However, operations of type ‘transaction’ are only allowed on elements of type ‘db’ or ‘media’.

¹The operators **I** and **@** we use for this have already been used by David Hilbert in the late 19th century.

Besides a name operations should have input- and output-parameters and an operation body, which we declare in the form

$\langle !IN \text{ operation expression} \rangle, \langle !OUT \text{ operation expression} \rangle$
 and $\langle !BODY \text{ operation} \dots \rangle$, respectively. Here, *expression* refers to the usual form of expressions used in element declarations. We have chosen a different form for the *body declarations*, as these might need a bit more space.

For operation bodies we use the following language:

- *assignments* in the form ‘identification’ := ‘expression’, where identification identifies one or more elements to be updated and expression provides a new value for this element;
- *non-deterministic selections* *NewID*, which selects a globally new identifier, or *NEW* (*element*, *element'*) with *element'* being a leaf in the tree for *element* and the defining expression for *element'* being a base type, which selects a new value of that base type that does not yet occur in instances of *element* with respect to the position indicated by *element'*;
- *control structures* for sequences, if-then-else, WHILE loops and calling operations defined on other elements—as these are standard, we omit further details;
- *local declarations* in the form
`LET variable : expression IN BEGIN ...END.`

The most important parts in here are the identification and the expressions used in assignments. For the former ones we use the following two operators:

- $Ix : n \bullet \varphi$ denotes the unique element with the name *n* satisfying formula φ ;
- $@x : n \bullet \varphi$ denotes any element with the name *n* satisfying formula φ .

Whenever $@x \dots$ is used on the left hand side of an assignment, all the *x* satisfying the condition φ will be updated. In addition to the identification operators we use regular path expressions as introduced in [2, Chap. 4] and the usual dot-notation for elements as well as for attributes.

Example 3 The assignment $(@x : \langle \text{staff} \rangle \bullet x.\text{birthday} < 1-1-1970).\text{salary} := x.\text{salary} * 1.04$ would increase the salary of all staff born before 1970 by 4%. This of course requires that salary is defined as an component element (or attribute) of the element staff and that it has been associated the base type DECIMAL.

The dot-notation for attributes allows us to use *x.a* for attributes *a* that have a type *ID*, *REF name'*, *REFS name'* or *SUP name'*, which gives us either an identifier or a sequence of identifiers, which would be treated in the same way as the @-operator.

The unique identifier of an element may be useful in navigating back to elements referencing it. For referenced elements, however, we would prefer to get the element(s) rather than their identifiers. Thus, we use the following shortcuts:

- $x!a = \mathbf{I}y : \langle n \rangle \bullet (y.\text{id} = x.a)$, if a is one of $\text{REF } n$ or $\text{SUP } n$ and id the attribute in the element n , which has the type ID ;
- $x!a = \mathbf{@}y : \langle n \rangle \bullet (y.\text{id} \in x.a)$, if a is $\text{REFS } n$ and id the attribute in the element n , which has the type ID .

Example 4 Let us take a look at Example 1. Here the assignment

$(\mathbf{@}s : \langle \text{student} \rangle \bullet (s!\text{supervisor!isa.name.last_name} = \text{"Einstein"})).\text{interest} := \text{"physics"}$

would update all students supervised by lecturers with last name “Einstein” adding a new element ‘interest’ and giving it the value “physics”. In particular, we would have to change the schema adding the new element

$\langle !\text{ELEMENT interest STRING TYPE} = \text{aux} \rangle$

and the new subelement

$\langle !\text{SUBELEMENT student (student_id, start, interest)} \\ \text{INDEX} = (1) \rangle$

of the element student. In addition, the students supervised by lecturers with last name “Einstein” would receive an attribute value $\text{index} = (1)$.

3.2 Schema Updates

So far we discussed schemata, subelements and update operations. In fact, schemata are pairs of a database and a media schema. Having defined an operation op of type transaction on some element of type ‘db’ or ‘media’, we may apply this operation to an instance I of the schema \mathcal{S} , which in fact is also a pair of instances for the database and the media schema. So, application of op will result in a new instance I' . This new instance I' will be the instance of a new schema \mathcal{S}' , which may extend \mathcal{S} by several subelements and new elements. In this case, however, the change of schema is implied by the application of an operation on the instance.

Thus, we have a transformation $(I, \mathcal{S}) \xrightarrow{(op, _)} (I', \mathcal{S}')$ with the underscore indicating that the change of schema is implied. So, in order to achieve greater flexibility and to avoid an inflation of new subelements we may think of making operations on the schema explicit. Suppose that we provide an operation Op on the schema, then we could apply (op, Op) to the pair (I, \mathcal{S}) , i.e., we get a transformation $(I, \mathcal{S}) \xrightarrow{(op, Op)} (I', \mathcal{S}')$ with an instance I' of the new schema \mathcal{S}' that results from applying Op . Of course, the possibility of using implied schema changes would still remain.

We will call a schema \mathcal{S} that is extended by a set of schema operations Op an *evolving schema*. Schema operations can be declared as $\langle !\text{OPERATION schema name [change}_1 \dots \text{change}_n] \rangle$, where *schema* is the name of the original schema to be updated, *name* is the name of the schema update operation, and $\text{change}_1 \dots \text{change}_n$ is a sequence of elementary changes to the schema.

Precisely, as we have two schemata, a database and a media schema, the operations Op mentioned before are indeed pairs of such schema update operations. The elementary schema changes have the form

+*declaration* or -*identification*,

where *declaration* is any valid declaration in XML except declarations for schemata, and *identification* identifies a schema component. The +-form means to add the declaration to the schema, and the --form means to remove the identified declaration from the schema.

Example 5 Taking the schema from Example 1, we could specify a schema update operation

```

<!OPERATION db add_graduates [
+<!ELEMENT graduate (interest+) TYPE = db>
+<!ATTRIBUTE graduate id ID #REQUIRED>
+<!ATTRIBUTE graduate isa SUP student #REQUIRED>
+<!ELEMENT interest STRING TYPE = aux>
]
```

This operation would also change the element declaration for db to

```

<!ELEMENT db (person|student|lecturer|graduate)*>
```

Note that if XML-SCHEMA is used, the schema itself is written in form of an XML-document. Thus, we could apply the update language from Section 3 to achieve schema updates.

4 Media Types in XML

In the introduction we described our goal to extend XML in such a way that the theory of media types would be supported. We shall now see that the extensions introduced so far will give us largely what we wanted to achieve.

4.1 The Structure of Media Types

The core of a media type is defined by a view (See [9, 14] instead for the details). A *view* V on a database schema \mathcal{S} consists of a view schema \mathcal{S}_V and a defining query q_V , which transforms databases over \mathcal{S} into databases over \mathcal{S}_V .

The underlying datamodel itself is not relevant. The defining query may be expressed in any suitable query language, e.g. query algebra, logic or an SQL-variant, provided that the queries are able to create links [9].

In order to introduce links, we must create identifiers in the result of a query. We may also want to provide escort information, which can be realized by a supertyping mechanism.

This leads to the definition of *raw media type*.

A *raw media type* has a name M and consists of a content data type $cont(M)$ with the extension that the place of a base type may be occupied by a pair $\ell : M'$ with a label ℓ and the name M' of a raw media type, a finite set $sup(M)$ of raw media type names M_i , each of which will be called a supertype of M , and a defining query q_M with create-facility such that $(\{t_M\}, q_M)$ defines a view. Here t_M is the type arising from $cont(M)$ by substitution of URL for all pairs $\ell : M'$.

In order to model functionality we add operations to raw media types. An *operation* on a raw media type M consists of an operation signature, i.e., name, input-parameters and output-parameters, a selection type which is a supertype of $cont(M)$, and a body which is defined via operations accessing the underlying database.

In order to allow the information content to be tailored to specific user needs and presentation restrictions, we must extend raw media types.

For many of the values we have to provide not only the type, but also the *measure unit*, e.g. Joule or kcal, PS or kW, cm, mm or m, etc. There exist fixed means for the calculation between the different units. Formally, each base type b should come along with a set $unit(b)$ of possible measure units. Each occurrence of b in the database or the raw media types has to be accompanied by one element from $unit(b)$. This leads to an implicit extension of the defining queries q_M . We shall talk of a *unit-extended* raw media type.

Since the raw media types are used to model the content of the information service, order is important. Therefore, we claim that the set constructor should no longer appear in content expressions. Then we need an *ordering-operator* ord_{\leq} which depends on a total order \leq defined on a type t and is applicable to values v of type $\{t\}$. The result $ord_{\leq}(v)$ has the type $[t]$. We shall tacitly assume that ordering operators are used in the defining queries q_M . In this case we talk of an *order-extended* raw media type.

Cohesion introduces a controlled form of information loss. Formally, we define a partial order \leq on content data types, which extends subtyping in a straightforward way such that references and superclasses are taken into consideration.

If $cont(M)$ is the content data type of a raw media type M and $sup(cont(M))$ is the set of all content expressions exp with $cont(M) \leq exp$, then a total pre-order \preceq_M on $sup(cont(M))$ extending the order \leq on content expressions is called a *cohesion pre-order*. Clearly, $cont(M)$ is minimal with respect to \preceq_M .

Small elements in $sup(cont(M))$ with respect to \preceq_M define information to be kept together, if possible. An alternative to cohesion pre-orders is to use *proximity values*, but we will not consider them here.

Another possibility to tailor the information content of raw media types is to consider dimension hierarchies as in OLAP systems. Flattening of dimensions results in information growth, its converse in information loss. Such a hierarchy is already implicitly defined by the component or link structures, respectively. Formal details on such hierarchies can be found in [9].

For a raw media type M let $\bar{H}(M)$ be the set of all raw media types occurring in the hierarchy of M . A *set of hierarchical versions* of M is a finite subset $H(M)$ of $\bar{H}(M)$

with $M \in H(M)$. Each cohesion pre-order \preceq_M on M induces a cohesion pre-order $\preceq_{M'}$ on each element $M' \in H(M)$.

A *media type* is a unit-extended, order-extended raw media type M together with a cohesion pre-order \preceq_M and a set of hierarchical versions $H(M)$.

Details on the theory of media types have been published in [9, 14].

4.2 Defining Queries of Media Types

The view of a media type can be declared in the form `<!VIEW element query >`, where `query` is the defining query. Such queries can be defined by using one of the XML query languages discussed before [3, 4, 7, 17]. However, in order to support the create-facility the following extensions can be used:

- Allow unspecified variables to appear in the specification of the query result. For each possible binding of other variables these variables will be bound to a new identifier.
- Use an iteration construct such as `ITER (q1; ... ; qn)` with queries q_i to define an inflationary fixed-point.
- Use a sequence operator `;` for queries.

4.3 Cohesion

When we introduced subelements, we exploited a subtype order. So whenever we are given a regular expression e in an element declaration, we know all super-types for this. With respect to the theory of media types we would use such an element to represent the content type $cont(M)$, so we know $sup(cont(M))$. On this the subtype order \leq is naturally defined. Cohesion depends on defining a total pre-order \preceq on $sup(cont(M))$ that extends the subtype order \leq .

In the context of XML we may think of a solution based on attributes. As we extend \leq , we do not have to repeat the subtype order in the definition of \preceq . As we are only looking for a pre-order, a missing specification $e_1 \preceq e_2$ can automatically be interpreted as $e_2 \preceq e_1$.

Suppose we have an element named n defined by the expression e . For all the components appearing in e we may simply declare a *cohesion index* in the form

```
<!COHESION element name number>
```

where `element` is an element of kind ‘media’, `name` occurs directly or indirectly in `element`, and `number` is a positive integer subject to the condition that whenever $v(n)$ denotes the cohesion index of n and the components of n are n_1, \dots, n_k , then we must have $\sum_{i=1}^k v(n_i) \leq v(n)$.

For all e occurring in $\text{sup}(\text{cont}(M))$ we associate a number $v(e)$, if possible. If all e_1, \dots, e_k with $e \leq e_i$ have been assigned a value $v(e_i)$, but e has not, then we set $v(e) = \sum_{i=1}^k v(e_i)$. Then we require $e_1 \preceq e_2$, if we have $v(e_1) \geq v(e_2)$. The default rules above completely define the cohesion pre-order \preceq .

Example 6 Suppose we have an element declaration with the name n and the regular expression (a^*, b, c) . We could declare cohesion indices for a , b and c :

```
<!COHESION n a 3> <!COHESION n b 2> <!COHESION n a 1>
```

4.4 XML Support for Media Types

We can now bring together the extensions discussed in the previous sections and describe how to support media types in XML. The fundamental part is to declare an element of type 'media'. In such a declaration, say `<!ELEMENT name expression KIND =3D media>`, the regular expression `expression` is used to describe the content type of the media type named `name`. The view of a media type can be declared in the form `<!VIEW element query >`, where `query` is the defining query as discussed above. The order extension required in media types has to be dealt with in this query.

The unit extension can be easily dealt with adding declarations of the form `<!UNIT element1 element2 unit>`, where `element1` refers to the element defining the media type, `element2` refers to a leaf element in the tree for `element1`, which is associated with a base type b_i , and `unit` is the name of a measure unit that is suitable for values in V_i .

Finally, a media schema would be declared analogously to a database schema, say

```
<!MEDIA schema [(!ELEMENT schema ...) ...]>
```

5 Conclusion

In this article we compared the theory of media types with XML. Media types have proven themselves to be a useful tool for conceptual modelling of large-scale web-based information systems. XML on the other hand is currently attracting enormous attraction in research and practice as a standard for data on the web.

We could show that with some extensions made XML could well be used to support media types. In its current form important features such as hierarchies and adaptivity are not yet supported. The suggested extensions comprise subelements, operations and cohesion indices.

In addition, subelements and operations bring some more clarity to the problem of updating XML-documents. We argued that each document is associated naturally with a schema or DTD, and that updates should refer to the pair consisting of the document and the schema. Subelements allow the schema after an update to be easily detected. This

leads to a semantics of evolving schemata.

References

- [1] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases*. Addison-Wesley 1995.
- [2] S. Abiteboul, P. Buneman, D. Suciu. *Data on the Web: From Relations to Semistructured Data and XML*. Morgan Kaufmann Publishers 2000.
- [3] S. Abiteboul, D. Quass, J. McHugh, J. Widom, J. Wiener. The LOREL Query Language for Semi-Structured Data. *Int. Journal on Digital Libraries*, vol. 1(1): 68-88. 1997.
- [4] P. Buneman, S. Davidson, G. Hillebrand, D. Suciu. A Query Language and Optimization Techniques for Unstructured Data. *Proc. SIGMOD '96*: 505-516.
- [5] P. Buneman, S. Davidson, M. Fernandez, D. Suciu. Adding Structure to Unstructured Data. *Proc. ICDT '97*.
- [6] S. Cluet, C. Delobel, J. Siméon, K. Smaga. Your Mediators need Data Conversion! *Proc. SIGMOD '98*: 177-188.
- [7] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, D. Suciu. A Query Language for XML. *International World Wide Web Conference* 1999.
- [8] T. Feyer, K.-D. Schewe, B. Thalheim. Conceptual Modelling and Development of Information Services. in T.W. Ling, S. Ram (Eds.). *Conceptual Modeling – ER '98*: 7-20. Springer LNCS 1507, Berlin 1998.
- [9] T. Feyer, O. Kao, K.-D. Schewe, B. Thalheim. Design of Data-Intensive Web-Based Information Services. In *Proc. 1st International Conference on Web Information Systems Engineering*. Hong Kong (China) 2000.
- [10] C.F. Goldfarb, P. Prescod. *The XML Handbook*. Prentice Hall. New Jersey 1998.
- [11] T. Grabs, K. Böhm, H.-J. Schek. Scalable Distributed Query and Update Service Implementations for XML Document Elements. *RIDE-DM 2001*: 35-42.
- [12] A. Marotta, R. Motz, R. Ruggia. Managing Source Schema Evolution in Web Warehouses. *Workshop on Information Integration on the Web 2001*: 148-155. Rio de Janeiro, April 2001.
- [13] K.-D. Schewe, B. Thalheim. Fundamental Concepts of Object Oriented Databases. *Acta Cybernetica*, vol. 11 (4), 1993, 49-84.
- [14] K.-D. Schewe, B. Thalheim. Modeling Interaction and Media Objects. In E. Métails (Ed.). *Proc. 5th Int. Conf. on Applications of Natural Language to Information Systems (NLDB 2000)*. Versailles (France) 2000. Springer LNCS.
- [15] I. Tatarinov, Z. Ives, A. Halevy, D. Weld. Updating XML. *SIGMOD'01*: 413-424.
- [16] A. Tretiakov and S. Hartmann. Mobile Content Adaptation as an Optimisation Problem. *Lecture Notes in Computer Science, Web Information Systems =96 WISE 2004 Workshops: WISE 2004 International Workshops, Brisbane, Australia, November 22-24, 2004*.
- [17] The World Wide Web Consortium (W3C). *XQuery*. <http://www.w3c.org/TR/xquery>
- [18] The World Wide Web Consortium (W3C). *XML Schema*. Working Draft, 2001. <http://www.w3c.org/TR/xmlschema-i> ($i = 0, 1, 2$)