ISTITUTO DI ANALISI DEI SISTEMI ED INFORMATICA
CONSIGLIO NAZIONALE DELLE RICERCHE

A. Formica

LEGALITY OF XML-SCHEMA TYPE HIERARCHIES

R. 629 Febbraio 2005

Anna Formica – Istituto di Analisi dei Sistemi ed Informatica "Antonio Ruberti" del CNR, Viale Manzoni 30 - 00185 Roma, Italy. Email : `anna.formica@iasi.cnr.it`

This paper appears in The Computer Journal 47(5), pp.591-601, 2004.

ISSN: 1128–3378

Collana dei Rapporti dell'Istituto di Analisi dei Sistemi ed Informatica, CNR

viale Manzoni 30, 00185 ROMA, Italy

tel. ++39-06-77161
fax ++39-06-7716461
email: `iasi@iasi.rm.cnr.it`
URL: `http://www.iasi.rm.cnr.it`

Abstract

The eXtensible Markup Language (XML) and XML-Schema are increasingly becoming the favorite languages to exchange and describe data across the internet, respectively. In this perspective, much of database research focus is shifting from the traditional database models to semistructured data and XML. In this paper, a formal framework supporting the design of XML-Schema type hierarchies has been proposed, by following and revisiting the database approach. In particular, the notion of a *legal* XML-Schema type hierarchy has been formally given and an algorithm for checking the legality of type hierarchies has been proposed.

Keywords: *XML-Schema, Type Hierarchy, Extension, Restriction.*

1. Introduction

The eXtensible Markup Language (XML) is rapidly emerging as the dominant standard for exchanging data across the Internet. The XML-Schema, since its elevation to W3C [1] Recommendation in May 2001, is increasingly becoming the favorite language to describe the structured XML data and, in the near future, it is expected to play a fundamental role. In this perspective, much of database research focus is shifting from the traditional database models to semistructured data and XML. However, most research on XML has so far largely neglected the analysis and definition of methodologies supporting XML-Schema design. In this regard, in this paper an important modeling notion of XML-Schemas has been addressed: the XML-Schema type hierarchy.

Type hierarchies, and the related *subtyping* relation [2] (also referred to as *subsumption*, *subclassing*, *refinement*, *ISA*, ..), have been extensively investigated in different fields of Computer Science, from Artificial Intelligence [3, 4], Programming Languages [5, 6, 7], to Databases [8, 9, 10, 11]. In particular, in the database area, checking the *correctness* (also referred to as *consistency*, or *satisfiability* [12]) of the type hierarchy of a schema is a key problem [13], which has been extensively investigated following two different approaches: the structural approach [6], and the naming approach [5]. Such different approaches have been integrated in the work presented in [14], conceived to support the database designer in the definition of a correct type hierarchy. In particular, in the mentioned paper an algorithm for identifying, and solving when possible, different kinds of inconsistencies of a type hierarchy has been proposed.

The contribution of this paper is inspired by the proposal presented in [14]. In particular, in this work, a formal framework supporting the design of XML-Schema type hierarchies has been presented, starting with the basic notions given in [15]. In particular, the approach presented in [14] has been extensively revisited and some of the notions presented therein, as for instance that of *subtyping* and *well-formed* schema, have been extended to deal with XML-Schemas. On the basis of these notions, the definition of a *legal* type hierarchy has been introduced and an algorithm for checking the legality of an XML-Schema type hierarchy has been presented.

The paper is organized as follows. In Section 2, the XML data model addressed in the paper has been presented via examples. In particular, the two mechanisms for defining an XML-Schema type hierarchy have been introduced, namely the *derivation by extension* and *derivation by restriction*. In Section 3, the proposed framework has been formally presented, starting with the definitions informally given in Section 2. In Section 4, the algorithm has been introduced, followed by Section 5 containing the concluding remarks. Below, the Related Work Section is given.

1.1. Related Work

We are assisting to a growing body of work concerning XML, and more recently XML-Schemas. Much of the work on XML-Schemas concerns expressive power, as for instance in [16], where a comparative analysis of the six noteworthy XML-Schema languages has been proposed (namely, XML-DTD, XML-Schema, XDR, SOX, Schematron, DSD). On the other hand, a big effort is being performed in different communities, in order to compare well-known problems and methodologies to the ones related to the new emerging XML. For instance, within the Logic Programming area, in [17] the Rule Markup Language has been proposed, with the aim of using Prolog as a semantic foundation for better understanding XML-Schemas. By following the description logics approach, in [18] a dialect of a description logic has been proposed which can be used to reason about structural equality in XML documents, with particular attention to the analysis of the computational complexity of the related implication problem. In the database community a lot of work has been done regarding the modeling capabilities of XML-Schemas. For instance, in [19] a proposal for the transformation of Object-Oriented Schemas into XML-Schemas has been presented in order to facilitate the management and retrieval of XML documents. In [20] a systematic approach to the data modeling capabilities of XML-schemas has been given, and a comparison with the ER model has been performed. In [21] XML-Schemas have been compared with the very popular method for software analysis and design UML.

A proposal concerning subsumption for XML types has been presented in [22]. Such a notion, which has been defined in the context of a type system rather than XML-Schemas, has been introduced to support type-related operations on XML data, as for instance type assignment and query processing (the design of XML-Schema type hierarchies, which is the focus of this paper, has not been addressed).

XML-Schema design has been investigated in [23], by making use of the Object Role Modelling (ORM) technique. In the mentioned paper, an algorithm for generating an XML-Schema from an ORM diagram has been presented, in order to discover data redundancy in the resulting XML-Schema by using ORM techniques. Consistency checking of XML-Schema constraints has been analyzed in [24]. In particular, the consistency of XML-Schemas in the presence of keys, foreign keys, and unique constraints has been addressed, and the related computational complexity has been investigated.

However, in all the above mentioned papers the problem of the legality of type hierarchies has not been addressed and, to our knowledge, there are no formal methodologies supporting the design of legal XML-Schema type hierarchies. It is worth recalling that a lot of work is being performed with regard to the XML *validation* and the XML *typechecking* [25, 26], very interesting problems which are not addressed here since they go beyond the scope of this work.

2. The XML Data Model

In this section the XML data model addressed in this paper is presented, which focuses on a subset of the key modeling notions defined by W3C [1].

An XML-Schema is defined by a set of *element* and *type* declarations. An element is declared by a *name* and a *type*. Types, which can be named or unamed, are simple (*simpleTypes*) or complex (*complexTypes*). SimpleTypes are atomic types as *string*, *decimal*, *integer*, *boolean*, *date* and *time*. ComplexTypes are defined by: (i) a sequence of elements, and/or (ii) a set of *attributes*, where an attribute is defined by a *name* and a simpleType.

Example 2.1. Consider the following declarations:

```
<xsd:element name="person" type="personInfo"/>

<xsd:element name="pet" type="petType"/>

<xsd:complexType name="personInfo">
    <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string"/>
        <xsd:element name="lastName" type="xsd:string"/>
    </xsd:sequence>
    <xsd:attribute name="age" type="xsd:integer"/>
    <xsd:attribute name="married" type="xsd:boolean"/>
  </xsd:complexType>

  <xsd:complexType name="petType">
    <xsd:sequence>
      <xsd:element name="breed" type="xsd:string"/>
      <xsd:element name="age" type="xsd:integer"/>
      <xsd:element name="ownedBy" type="personInfo"/>
    </xsd:sequence>
  </xsd:complexType>
```

In this example we have two elements, the former with name *person* and type *personInfo*, the latter with name *pet* and type *petType* (*xsd* stands for XML-Schema Definition). Both types are complexTypes. In particular, *personInfo* is defined by the sequence of elements (whose names are) *firstName* and *lastName*, both of type *string*, and two attributes, namely *age* and *married*, the former of type *integer* and the latter of type *boolean*. In the case of the complexType *petType*, besides the elements *breed* and *age*, both typed with a simpleType, we have *ownedBy* which is typed with the complexType *personInfo*. □

It is also possible to *inline* unamed complexTypes in the element declarations. In this way, the complexTypes cannot be referred by other elements or complexTypes. For instance, in the previous example, we can inline the complexType *personInfo* in the *person* declaration, without using its name, as follows:

```
  <xsd:element name="person">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="firstName" type="xsd:string"/>
        <xsd:element name="lastName" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute name="age" type="xsd:integer"/>
      <xsd:attribute name="married" type="xsd:boolean"/>
    </xsd:complexType>
  </xsd:element>
```

Elements which are defined by using type names are referred to as *flat* elements. In an XML-Schema, elements are organized according to an *element hierarchy*. This hierarchy, which essentially represents the element decomposition relation, has a root element, say *top*, which in the case of Example 2.1 is defined as follows:

```
  <xsd:element name="top">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="person" type="personInfo"/>
        <xsd:element name="pet" type="petType"/>
      </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Therefore, the element hierarchy of Example 2.1 is shown in Figure 1.

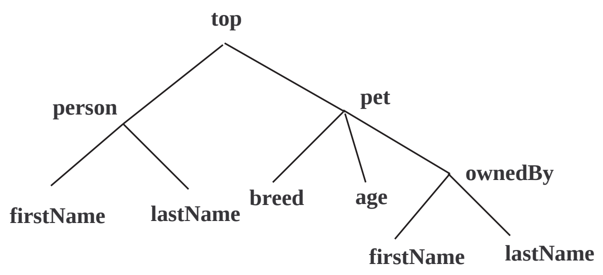

Figure 1: Element hierarchy of Example 2.1

It is important to recall that in XML-Schemas element hierarchies are always finite trees, that is, recursive complexTypes (i.e., defined by types which, directly or indirectly, refer to the type being defined) are not allowed. Notice that elements which are defined by inlining unamed types can always be transformed into flat elements by introducing the names of the types.

In XML-Schemas, it is possible to specify the minimum (*minOccurs*) and maximum (*maxOccurs*) number of values which an element can take, where the maximum number can also be *unbounded*. Attributes are always assumed to be single-valued (*minOccurs* = *maxOccurs* = 1). In this paper, for the sake of simplicity, also elements are assumed to be single-valued (notice that this choice does not affect the contribution of this paper, which can be extended to the general case by using a more complex notation).

In line with [15], in a complexType attributes with the same names are not allowed, whereas elements with the same names are allowed, subject to the restriction they have the same types. Furthermore, following the approach proposed in [14], we assume that in XML-Schemas 'type equivalence by name' holds, in the sense that types whose declarations differ for type names only are not equivalent. For instance, consider the following declarations:

```
<xsd:complexType name="dogType">
    <xsd:sequence>
        <xsd:element name="age" type="xsd:integer"/>
        <xsd:element name="ownedBy" type="personInfo"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="carType">
    <xsd:sequence>
        <xsd:element name="age" type="xsd:integer"/>
        <xsd:element name="ownedBy" type="personInfo"/>
    </xsd:sequence>
</xsd:complexType>
```

In this case, although defined by the same elements, *dogType* and *carType* are different types.

A mechanism to distinguish among structurally similar but semantically different types is well-known in programming languages, and consists in associating names with the structures of the types. Such a mechanism is referred to as *branding* [14].

SimpleTypes and complexTypes can also be defined in terms of other types, by organizing them according to a *type hierarchy*. Such a hierarchy, which has to be distinguished from the element hierarchy, is illustrated in the next subsection.

2.1. The Type Hierarchy

The *type hierarchy* of an XML-Schema is defined by means of the *extension* and *restriction base* constructors. The types defined by using such constructors will be referred to as *derived* types, whereas the types which are used to define the derived types are referred to as *base* types.

2.1.1. Derivation by Extension

The *extension base* constructor allows the incremental definition of complexTypes by adding elements and/or attributes to existing complexTypes. The derived complexType will have all the elements and attributes of the base type, plus the additional elements and/or attributes declared for it. Notationally, the *complexContent* constructor is used, as shown in the following example.

Example 2.2. Consider the complexType *personInfo* defined in Example 2.1. It is possible to define a derived complexType of name, for instance, *employeeInfo* by adding the *firstProject* and *secondProject* elements, and the *salary* and *company* attributes to *personInfo* as follows:

```
<xsd:complexType name=”employeeInfo”>
   <xsd:complexContent>
      <xsd:extension base=”personInfo”>
         <xsd:sequence>
            <xsd:element name=”firstProject” type=”xsd:string”/>
            <xsd:element name=”secondProject” type=”xsd:string”/>
         </xsd:sequence>
         <xsd:attribute name=”salary” type=”xsd:integer”/>
         <xsd:attribute name=”company” type=”xsd:string”/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

The type *personInfo* is the base type of *employeeInfo*. Notice that the complexType *employeeInfo* is equivalent to the following complexType defined without using the extension base constructor:

```
<xsd:complexType name=”employeeInfo”>
   <xsd:sequence>
      <xsd:element name=”firstName” type=”xsd:string”/>
      <xsd:element name=”lastName” type=”xsd:string”/>
      <xsd:element name=”firstProject” type=”xsd:string”/>
      <xsd:element name=”secondProject” type=”xsd:string”/>
   </xsd:sequence>
```

```
    <xsd:attribute name="age" type="xsd:integer"/>
    <xsd:attribute name="married" type="xsd:boolean"/>
    <xsd:attribute name="salary" type="xsd:integer"/>
    <xsd:attribute name="company" type="xsd:string"/>
  </xsd:complexType>
```

□

Notice that, according to [1], a complexType can be defined in terms of one base type at most, that is, by using the Object-Oriented terminology [8], in an XML-Schema multiple inheritance is not allowed.

It is also possible to define a complexType by extending a simpleType with some attributes. In this case, the complexContent constructor is replaced by *simpleContent*.

Example 2.3. For instance, consider the following type:

```
  <xsd:complexType name="elevationType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:integer">
        <xsd:attribute name="units" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
```

which allows the association of units, as for instance meters or feet, with an integer number. In this case, *integer* is the base type of *elevationType*. □

2.1.2. Derivation by Restriction

As an alternative to the extension base constructor, types can be organized according to a hierarchy by using the *restriction base* constructor. This constructor allows types to be derived by restricting the domains of already defined types. Similarly to the previous case, types can be derived by using one base type at most (that is only single inheritance is allowed, according to the Object-Oriented terminology). In particular, simpleTypes and complexTypes can be derived by restricting simpleTypes and complexTypes, respectively. In the case of simpleTypes, restrictions can be performed by enumeration or by specifying the interval extremes.

Example 2.4. Suppose we want to define two simpleTypes, namely *nameType* and *teenageType*, by restricting the set of all possible names and ages, respectively. This can be achieved by restricting the simpleTypes *string* and *integer*, respectively, the former by enumeration and the latter by specifying the interval extremes, as follows:

```
  <xsd:simpleType name="nameType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="John"/>
      <xsd:enumeration value="Tom"/>
      <xsd:enumeration value="Bob"/>
    </xsd:restriction>
  </xsd:simpleType>
```

```
<xsd:simpleType name="teenageType">
   <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="13"/>
      <xsd:maxInclusive value="19"/>
   </xsd:restriction>
</xsd:simpleType>
```
□

In the case of complexTypes, derivations by restriction can be achieved by replacing the types of one or more elements/attributes of the base type with derived types, therefore reducing the ranges of values that elements/attributes can assume with respect to the base type. As for the derivation by extension, the *complexContent* constructor is used.

Example 2.5. Consider again the *personInfo* complexType and, furthermore, the simpleTypes *nameType* and *teenageType* defined in the previous example. Then, it is possible to declare the *teenagerInfo* complexType by restricting the types of *firstName* and *age* as follows:

```
<xsd:complexType name="teenagerInfo">
   <xsd:complexContent>
      <xsd:restriction base="personInfo">
         <xsd:sequence>
            <xsd:element name="firstName" type="nameType"/>
            <xsd:element name="lastName" type="xsd:string"/>
         </xsd:sequence>
         <xsd:attribute name="age" type="teenageType"/>
         <xsd:attribute name="married" type="xsd:boolean"/>
      </xsd:restriction>
   </xsd:complexContent>
</xsd:complexType>
```
□

As opposed to the extension base constructor, in this case all the elements and attributes of the base type must be repeated in the derived type [15]. Notice that, although here we do not have more compact type declarations, derivation by restriction (as well as by extension) is a very useful technique in order to perform *type substitutability*, i.e., the replacement of a base type with any derived type [15].

As a final remark, we observe that derivations by restriction can also be achieved by reducing the number of values which an element can take. However, in this paper this kind of restriction is not addressed since we have assumed that elements are single-valued. As already mentioned, this choice is only due to readability reasons, since the contribution of this paper can be extended in that direction by using a more complex notation.

2.1.3. Legal Type Hierarchy

An XML-Schema type hierarchy is a tree having a root type referred to as *anyType*. The root type is the most general type, whose unique characteristic is that it can function either as a complexType or as a simpleType [1]. All the types which are not derived (including the atomic types) are directly connected to the root type. Furthermore, for each derived type there exists an arc in the tree connecting it to the related base type. According to the cases, the arc is labelled with one of the symbols *ext* or *res* standing for the *extension* and *restriction base* constructors, respectively. Notice that the type hierarchy of an XML-Schema is always a tree, since a type can be defined in terms of one base type at most (which corresponds to the notion of single inheritance in an Object-Oriented data model). In the case of Examples 2.1 - 2.5, the type hierarchy is represented in Figure 2.

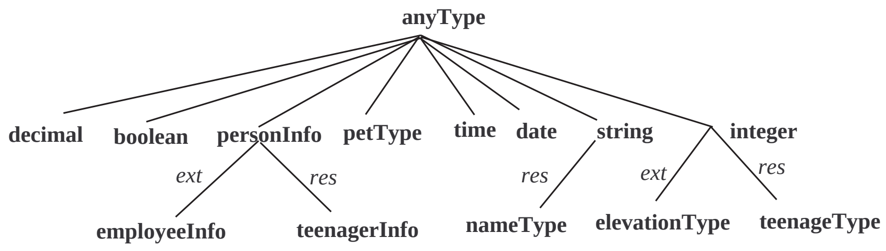

Figure 2: Type hierarchy of Examples 2.1 - 2.5

One of the difficulties encountered in designing a type hierarchy, concerns name conflicts. In fact, we have seen that in a complexType multiple occurrences of attribute names are not allowed, and element names can be repeated subject to the restriction they have the same types. Therefore, in deriving types by extension, a check on the additional attributes and elements is required. In particular, types derived by extending a base type with an already defined attribute name generate an illegal type hierarchy (as for instance, the addition of the attribute *age*, independently of the type, to the base type *personInfo* of Example 2.1). Similarly, the presence of an already defined element name, whose type is different from the one defined in the base type, generates an illegality (for instance, the element *firstName* can be added to the base type *personInfo*, if it is typed with *string*).

Regarding the derivation by restriction, elements or attributes with types whose domains are not restrictions of the corresponding type domains in the base type are not allowed. For instance, the complexType *teenagerInfo* of Example 2.5 is a legal restriction of *personInfo* since *firstName* is typed with *nameType*, which is derived from *string*, and *age* is typed with *teenageType*, which has *integer* as base type. Of course, if in place of *teenageType* we have had, for instance, the type *string*, we would have had an illegal type hierarchy.

Notice that in Object-Oriented data models (whose subtyping mechanism is more flexible than the one of XML data models) the same constructor can be used for deriving types either by extension or by restriction, whereas in XML-Schemas two different constructors (the *extension base* and *restriction base*) are present. In particular, each of them has been conceived for a specific kind of derivation, that is, the addition of attributes/elements (*extension*), and the restriction of type domains (*restriction*). Therefore, in designing a legal type hierarchy, a further check is required concerning the correct use of these constructors.

The notion of a legal type hierarchy is formally introduced in the next section, where a formal framework for checking the legality of an XML-Schema type hierarchy is presented.

3. Formal Basis

In this section the notions informally presented via examples are formally defined. To this end, for the sake of simplicity, in place of the previous XML-Schema syntax, a compact syntax will be used.

In the following $\mathcal{T}$, $\mathcal{E}$, and $\mathcal{A}$ are countable sets of type names, element names and attribute names, respectively. In the XML-Schema data model addressed in this paper types, which can be *simpleTypes* or *complexTypes*, are defined as follows.

Definition 3.1. **[SimpleType]**. A *simpleType* can be defined according to one of the following cases:

- an atomic type of the set:
  $\mathcal{B} = \{$`string, integer, decimal, boolean, date, time`$\}$;
- a declaration (of name) $\tau \in \mathcal{T}$ derived by restricting a simpleType $\sigma \in \mathcal{T}$ as follows:
  $\tau = \textbf{res}(\sigma)\ V$
  where **res** stands for the *restriction base* constructor, and $V \neq \emptyset$ is a set of values of one of the following forms:
  - $V = \{v_1, ..., v_n\}$ (enumerated set),
  - $V = (v_1\text{-}v_n)$ (interval). It is possible to define unbounded intervals, represented as $(v_1\text{-}...)$ or $(...\text{-}v_n)$.

The type σ is referred to as the *base type* of τ. □

Definition 3.2. **[ComplexType]**. A *complexType* (of name) $\tau \in \mathcal{T}$ is a declaration defined according to one of the following cases:

- $\tau = \textbf{ext}(\sigma)\ \{a_j : \lambda_j\}$, $j = 1...m$,
  where $\sigma \in \mathcal{T}$ is a simpleType, **ext** stands for the *extension base* constructor, $a_j \in \mathcal{A}$ are attribute names, and $\lambda_j \in \mathcal{T}$ are simpleTypes;
- $\tau = [\textbf{ext-res}]^*(\sigma) < e_i : \gamma_i > \{a_j : \lambda_j\}$, $i = 0...n$, $j = 0...m$,
  where $\sigma \in \mathcal{T}$ is a complexType, $e_i \in \mathcal{E}$ and $a_j \in \mathcal{A}$ are element and attribute names, respectively, which are optional ($i = j = 0$) but not at the same time, that is, $i = 0 \Rightarrow j \neq 0$, and $j = 0 \Rightarrow i \neq 0$. Furthermore, $\gamma_i \in \mathcal{T}$ are simpleTypes or complexTypes, $\lambda_j \in \mathcal{T}$ are simpleTypes, and:
  $e_h = e_k \Rightarrow \gamma_h = \gamma_k$, for h,$k = 1...n$;
  $a_h \neq a_k$ for h,$k = 1...m$.
  Finally, the notation $[\textbf{ext-res}]^*$ means that the extension (**ext**) and the restriction (**res**) base constructors are optional, and only one of them can be chosen.

As for simpleTypes, the type σ is referred to as the *base type* of τ. According to the case, we say that τ *is derived by extending/restricting* σ. Recursive types are not allowed. □

Example 3.1. For instance, according to the compact syntax introduced in this section, the simpleType *nameType* of Example 2.4 becomes:
  `nameType` = **res**(`string`) {`John,Tom,Bob`}
the complexType *elevationType* of Example 2.3, has the form:
  `elevationType` = **ext**(`integer`) {`units:string`}
and in Example 2.2, the type *employeeInfo* derived by extending *personInfo* becomes:
  `employeeInfo` = **ext**(`personInfo`) <`firstProject:string`,
    `secondProject:string`> {`salary:integer, company:string`} □

The notion of a *flat element* follows, taking into account that, as already mentioned in Section 2, elements which are not flat can always be transformed into a flat form by introducing names for the inlined types.

Definition 3.3. **[Element]**. A *flat element* (*element* for short) (of name) $e \in \mathcal{E}$ is a declaration of the form:
<e:τ>
where $\tau \in \mathcal{T}$ is a simpleType or a complexType. $\Box$

Before introducing the definition of an XML-Schema, in the rest of the paper a type τ is said to be in *derived form* (DF) if it is defined by means of the extension or restriction base constructors, otherwise it is said to be in *normal form* (NF).

Definition 3.4. **[XML-Schema]**. An XML-*Schema* (*Schema* for short) Σ is a 5-tuple:
$\Sigma = (T, E, A, T_{dec}, E_{dec})$
where $T \subset \mathcal{T}$, $E \subset \mathcal{E}$, $A \subset \mathcal{A}$, and T_{dec}, E_{dec} are sets of (declarations of) types and elements, respectively, with names in T, E, and A, such that the following holds:

- every type name in T has exactly one type declaration in T_{dec};
- every type name present in T_{dec} and E_{dec} is in T;
- consider the set $\mathcal{B}$ of atomic types and assume that $\mathcal{B}^+ = \mathcal{B} \cup \{anyType\}$.
  Let *dirDer* (*directDerivation*) be the ordered binary relation on $T \cup \mathcal{B}^+$ defined as follows:
  - *dirDer*(*anyType*,τ), for any atomic type $\tau \in \mathcal{B}$ and for any type $\tau \in T$ in NF;
  - *dirDer*(σ, τ), if $\tau \in T$ is in DF and $\sigma \in T$ is the base type of τ. In particular, if τ is derived by extending/restricting σ, the notations $dirDer^{ext}(\sigma, \tau)/dirDer^{res}(\sigma, \tau)$ will be used.

  Then ($T \cup \mathcal{B}^+$,*dirDer*) is a directed and labeled tree, of root *anyType*, which is referred to as the *type hierarchy* of Σ. Arcs, except for the ones connecting the root with the children, are labeled with *ext* or *res*. Let *derBy* (*derivedBy*) be the reflexive and transitive closure of *dirDer*. Then *derBy* is referred to as the *derivation* relation of Σ.

The sets T, E, and A stand for *symbol spaces*, which allow the definition of the XML-Schema *target namespace* [1, 15]. $\Box$

Similarly to the notion of a type in DF and NF, we will say that a schema is in *normal form* (NF) if all its types are in NF, otherwise it is in *derived form* (DF).

In the following the notion of *type components* is given. It allows us to directly refer to specific components defining a type, i.e., the set of values, the sequence of elements, or the set of attributes.

Definition 3.5. **[Type components]**. Consider a simpleType or a complexType $\tau \in \mathcal{T}$ (in NF or DF). The set V, the sequence of elements $< e_i : \gamma_i >$, and the set of attributes $\{a_j : \lambda_j\}$, when present, are referred to as the *value*, *element*, and *attribute components* of τ, and they are indicated as V_τ, E_τ, A_τ, respectively. Then, the *type components* of τ are defined by the triple:
$comp(\tau) = (V_\tau, E_\tau, A_\tau)$
where, if absent, V_τ, E_τ, A_τ are assumed to be empty. In the case of an atomic type $\delta \in \mathcal{B}$, $comp(\delta)$ is defined by the only non-empty component V_δ, corresponding to the domain of the type. $\Box$

Example 3.2. For instance, in Example 2.2, the type components of the type *employeeInfo* in DF are the following:
$comp(\texttt{employeeInfo}) = (V_{\texttt{employeeInfo}}, E_{\texttt{employeeInfo}}, A_{\texttt{employeeInfo}})$,
where:
$V_{\texttt{employeeInfo}} = \emptyset$
$E_{\texttt{employeeInfo}} = <$`firstProject:string, secondProject:string`$>$
$A_{\texttt{employeeInfo}} = \{$`salary:integer, company:string`$\}$
whereas, in the case of *employeeInfo* in NF, we have:
$V_{\texttt{employeeInfo}} = \emptyset$
$E_{\texttt{employeeInfo}} = <$`firstName:string, lastName:string,`
`firstProject:string, secondProject:string`$>$
$A_{\texttt{employeeInfo}} = \{$`age:integer, married:boolean, salary:integer,`
`company:string`$\}$
In the case of *elevationType* of Example 2.3 we have:
$V_{\texttt{elevationType}} = \emptyset$
$E_{\texttt{elevationType}} = \emptyset$
$A_{\texttt{elevationType}} = \{$`units:string`$\}$,
and, regarding *nameType* of Example 2.4:
$V_{\texttt{nameType}} = \{$`John,Tom,Bob`$\}$
$E_{\texttt{nameType}} = \emptyset$
$A_{\texttt{nameType}} = \emptyset$ □

Starting with the notion of type components, we have the following.

Definition 3.6. [Schema components] Given a schema $\Sigma = (T, E, A, T_{dec}, E_{dec})$ the *schema components* of Σ, indicated as $comp(\Sigma)$, are defined as:
$comp(\Sigma) = \bigcup_{\tau \in T} comp(\tau)$ □

In line with [14], proposed within the Object-Oriented data model, the goal of this paper is to transform a schema in DF into a schema in NF, and in this process to discover if the type hierarchy of the schema is legal. Notice that the *derBy* relation is defined according to the extension/restriction base constructors which, in the transformation process, have to be removed. For this reason, the notion of a *hierarchical schema* follows, where the type hierarchy is defined as an independent component of a schema in NF.

Definition 3.7. [Hierarchical schema] Consider a schema $\Sigma = (T, E, A, T_{dec}, E_{dec})$ in NF and a labeled, directed tree $(T \cup \mathcal{B}^+, dirDer)$, of root *anyType*, where $\mathcal{B}^+$ is defined as in Definition 3.4, and *dirDer* is an ordered relation defining the set of labeled arcs of the tree. A *hierarchical schema* $\mathcal{S}$ is a pair:
$\mathcal{S} = (comp(\Sigma), derBy_{\mathcal{S}})$
where $comp(\Sigma)$ are the schema components of Σ, and $derBy_{\mathcal{S}}$ is the reflexive and transitive closure of *dirDer*, referred to as the *derivation* relation of $\mathcal{S}$. The tree $(T \cup \mathcal{B}^+, dirDer)$ is referred to as the *type hierarchy* of $\mathcal{S}$. □

In order to avoid confusion, the *dirDer* relation of the type hierarchy of a hierarchical schema $\mathcal{S}$ will be indicated as $dirDer_{\mathcal{S}}$. Furthermore, as for the Definition 3.4, the notation $dirDer^{l}_{\mathcal{S}}(\sigma, \tau)$ is used to denote the arc (σ, τ) with label l of the tree $(T \cup \mathcal{B}^+, dirDer_{\mathcal{S}})$. Analogously, if Σ is a schema in DF, $dirDer_{\Sigma}$ and $derBy_{\Sigma}$ will indicate the relations *dirDer* and *derBy*, respectively, derived from its declarations. We expect that if a schema in DF is successfully transformed into NF, then this relation will be the derivation relation of the resulting hierarchical schema and therefore the type hierarchy of the schema is legal. This is accomplished by the algorithm presented the next section.

In order to formally present the notion of a legal type hierarchy, the definition of *subtype* relation follows. In such a definition, the set of types *over* a schema $\Sigma = (T, E, A, T_{dec}, E_{dec})$ is the smallest set which contains T, the set of atomic types $\mathcal{B}$, the sets of values, and which is closed under sequences of elements and sets of attributes formation.

Definition 3.8. [Subtype relation] Given a hierarchical schema $\mathcal{S}$, the *subtype* relation, denoted as $\preceq$, is defined on types over Σ as the smallest relation such that for any:

1. type:
   $\tau \preceq \tau$;

2. sets of values:
   $\lambda_1 \preceq \lambda_2$ if $\lambda_1 \subseteq \lambda_2$

3. sequences of elements:
   $< e_i : \zeta_i >_{1 \leq i \leq n+m} \preceq < e_i : \eta_i >_{1 \leq i \leq n}$ if $\zeta_i \preceq \eta_i$ for $1 \leq i \leq n$, $n, m \geq 0$

4. sets of attributes:
   $\{a_j : \gamma_j\}_{1 \leq j \leq h+k} \preceq \{a_j : \delta_j\}_{1 \leq j \leq h}$ if $\gamma_j \preceq \delta_j$ for $1 \leq j \leq h$, $h, k \geq 0$

5. type names and atomic types:
   $\tau_1 \preceq \tau_2$ if $derBy_{\mathcal{S}}(\tau_2, \tau_1)$ and $V_{\tau_1} \preceq V_{\tau_2}$, $E_{\tau_1} \preceq E_{\tau_2}$, $A_{\tau_1} \preceq A_{\tau_2}$,
   where:
   $comp(\tau_1) = (V_{\tau_1}, E_{\tau_1}, A_{\tau_1})$
   $comp(\tau_2) = (V_{\tau_2}, E_{\tau_2}, A_{\tau_2})$
   and $V_{\tau_2} \neq \emptyset \Rightarrow V_{\tau_1} \neq \emptyset$.

□

Notice that the condition $derBy_{\mathcal{S}}(\tau_2, \tau_1)$ has been introduced in line with the notion of *branding* presented [14]. As already mentioned in Section 2, branding allows the introduction of 'type equivalence by name' in order to distinguish between structurally similar but semantically different types. In the absence of such a condition, structurally similar types which are not hierarchically related, as for instance *dogType* and *carType* shown in Section 2, result equivalent. It is important to observe that the designer is not required to explicitly use branding. In fact, it is possible to implicitly assume that the type declarations are branded with the related type names. As a final remark, notice that in the cases $n = 0$ and $h = 0$, the sequence of elements and the set of attributes on the right hand sides of points 3 and 4 of Definition 3.8 degenerate in the empty set and empty list, respectively.

Example 3.3. For instance, consider the type components of *personInfo* of Example 2.1, that are:

$V_{\texttt{personInfo}} = \emptyset$
$E_{\texttt{personInfo}} = $ <`firstName:string, lastName:string`>
$A_{\texttt{personInfo}} = $ {`age:integer, married:boolean`}

and the type components of *employeeInfo* in NF of Example 3.2. Then:
`employeeInfo` $\preceq$ `personInfo`.

As a further example, consider the type components of two types in NF of a hierarchical schema $\mathcal{S}$, namely *bikeColors* and *carColors*, for which $derBy_{\mathcal{S}}(carColors, bikeColors)$ holds, defined as follows:

$V_{\texttt{bikeColors}} = \{$red,green,yellow$\}$
$E_{\texttt{bikeColors}} = \emptyset$
$A_{\texttt{bikeColors}} = \{$`metalSpray:boolean, multipleColors:boolean`$\}$

and:

$V_{\texttt{carColors}} = \{$red,green,blue,yellow$\}$
$E_{\texttt{carColors}} = \emptyset$
$A_{\texttt{carColors}} = \{$`metalSpray:boolean`$\}$

Then:
`bikeColors` $\preceq$ `carColors`. □

Now, we are able to introduce the notion of a *well-formed* type hierarchy.

Definition 3.9. [**Well-formed type hierarchy**] The type hierarchy of a hierarchical schema $\mathcal{S} = (comp(\Sigma), derBy_{\mathcal{S}})$ is *well-formed* if, for all types $\tau_1, \tau_2 \in T \cup \mathcal{B}$, the following holds:
$derBy_{\mathcal{S}}(\tau_2, \tau_1) \Rightarrow \tau_1 \preceq \tau_2$. □

Due to the presence of two different constructors for deriving types (extension and restriction base), each conceived for a specific kind of derivation, the notion of a well-formed type hierarchy has to be refined with that of a *legal* type hierarchy, as follows.

Definition 3.10. [**Legal type hierarchy**] The type hierarchy of a hierarchical schema $\mathcal{S} = (comp(\Sigma), derBy_{\mathcal{S}})$ is *legal* if it is well-formed and for each pair of types τ_1, τ_2 in T the following holds:

$$dirDer_{\mathcal{S}}(\tau_2, \tau_1) \Rightarrow E_{\tau_2} = < e_i : \zeta_i >_{1 \le i \le n},\ E_{\tau_1} = < e_i : \eta_i >_{1 \le i \le n+m}$$
$$A_{\tau_2} = \{a_j : \delta_j\}_{1 \le j \le h},\ A_{\tau_1} = \{a_j : \gamma_j\}_{1 \le j \le h+k}$$

where $n, m, h, k \ge 0$, and there are no *name conflicts*, that is:
- $e_i = e_u$ in $E_{\tau_2} \Rightarrow \zeta_i = \zeta_u$, for $i, u = 1...n$,
- $e_i = e_u$ in $E_{\tau_1} \Rightarrow \eta_i = \eta_u$, for $i, u = 1...n+m$,
- $a_j \neq a_v$, for $j, v = 1...h+k$.

Furthermore:

- $dirDer^{ext}_{\mathcal{S}}(\tau_2, \tau_1) \Rightarrow m > 0$, and/or $k > 0$;
- $dirDer^{res}_{\mathcal{S}}(\tau_2, \tau_1) \Rightarrow m, k = 0$.

□

4. The Algorithm

In this section, an algorithm for the transformation of a schema in DF into a hierarchical schema is presented. The transformation essentially consists in the replacement of the extension/restriction base constructors with the type components of the base types, along the type hierarchy. If the algorithm succeeds, a hierarchical schema is provided, otherwise it fails. Failures are essentially due to illegal derivations of types either by extending or by restricting the related base types. Before introducing the algorithm, a few notions have to be presented. Notice that in this section, for the sake of simplicity, the term *property* will be used to indicate both an element or an attribute name.

Definition 4.1. [ValueSet] Consider a schema $\Sigma = (T, E, A, T_{dec}, E_{dec})$, and a type $\tau \in T$. Then $valueSet(\tau)$ is a set of values defined as follows:

- τ is a type in NF or an atomic type. Then:
  $valueSet(\tau) = V_\tau$
  where V_τ is the value component of τ. Therefore, in the case of an atomic type, as for instance, *integer*, $valueSet(\tau)$ is the set of integer numbers;

- τ is a simpleType in DF and δ is the base type of τ. Then:
  $valueSet(\tau) = valueSet(\delta) \cap V_\tau$,
  where V_τ is the value component of τ;

- τ is a complexType in DF and δ is the base type of τ. Then:
  $valueSet(\tau) = valueSet(\delta)$.

□

Example 4.1. For instance, consider the types *nameType, elevationType, employeeInfo* of Example 3.1. Then:
$valueSet(\texttt{nameType}) = \{\texttt{John,Tom,Bob}\}$
$valueSet(\texttt{elevationType}) = \mathbf{Z}$
$valueSet(\texttt{employeeInfo}) = \emptyset$
where $\mathbf{Z}$ stands for the set of integers, and taking into account that $V_{\texttt{personInfo}} = \emptyset$. Suppose now to define one more type, namely *firstNameType*, by deriving it from *nameType* as follows:
$\texttt{firstNameType} = \mathbf{res}(\texttt{nameType})\ \{\texttt{Tom,Bob,Hans}\}$.
In this case:
$valueSet(\texttt{firstNameType}) = \{\texttt{Tom,Bob}\}$
that, similarly to the case of `elevationType` (see Example 3.2), does not coincide with its value component:
$V_{\texttt{firstNameType}} = \{\texttt{Tom,Bob,Hans}\}$. □

Definition 4.2. [TypeOf] Let Σ be a schema and τ a complexType of the schema whose type components E_τ and A_τ are defined as follows:
$E_\tau = < e_i : \gamma_i >$ and $A_\tau = \{a_j : \lambda_j\}$, $i = 1...n$, $j = 1...m$.
For each property of the schema, say q, $typeOf(\tau, q)$ is a type of the schema defined as follows:
$typeOf(\tau, q) = \gamma_i$ if $q = e_i$ for some i,
$typeOf(\tau, q) = \lambda_j$ if $q = a_j$ for some j,
and $typeOf(\tau, q)$ is undefined otherwise. □

Example 4.2. Consider again the type *employeeInfo* of Example 3.1, the element *firstProject* and the attribute *salary.* Then:
$typeOf(\texttt{employeeInfo,firstProject}) = \texttt{string}$
$typeOf(\texttt{employeeInfo,salary}) = \texttt{integer}$. □

H-Transform

// input: a schema $\Sigma = (T, E, A, T_{dec}, E_{dec})$ in DF
// output: a hierarchical schema $\mathcal{S}$, or 'fail'

```
for each type $\tau \in T$, compute $valueSet(\tau)$
if, for some simpleType $\tau$ in $DF$, $valueSet(\tau) \neq V_\tau$
   then 'fail' and exit
else
   for each complexType $\tau \in T$ in $DF$, starting at
         the root of $(T \cup \mathcal{B}^+, dirDer_\Sigma)$
      $Check(\tau)$ (this call may not return)
      case $\tau$ of
      (i) $\tau$ is derived by $extending$ the base type $\sigma$
         $E_\tau$ = $E_\tau \dot\cup E_\sigma$, $A_\tau$ = $A_\tau \cup A_\sigma$
      (ii) $\tau$ is derived by $restricting$ the base type $\sigma$
         if not($elemOf(\sigma) \leftrightarrow elemOf(\tau)$)
               or not($attrOf(\sigma) \leftrightarrow attrOf(\tau)$)
            then 'fail' and exit
      $V_\tau$ = $valueSet(\tau)$
      $comp(\tau)$ = $(V_\tau, E_\tau, A_\tau)$
$comp(\Sigma)$ = $\bigcup_{\tau \in T} comp(\tau)$
$derBy_\mathcal{S}$ = $derBy_\Sigma$
$\mathcal{S} = (comp(\Sigma), derBy_\mathcal{S})$
```

Figure 3: The *H-Transform* algorithm

Check
// input: a complexType τ
// output: return or ‘fail’

```
let $\sigma$ be the base type of $\tau$
case $\tau$ of
(1) $\tau$ is derived by *extending* the base type $\sigma$
   for each property $q \in elemOf(\tau)$
      if $q \in elemOf(\sigma)$ and $typeOf(\tau, q) \neq typeOf(\sigma, q)$
         then ‘fail’ and exit
   for each property $q \in attrOf(\tau)$
      if $q \in attrOf(\sigma)$
         then ‘fail’ and exit
   return
(2) $\tau$ is derived by *restricting* the base type $\sigma$
   if $\exists$ a property $q$ of $\tau$ s.t.
      ($q \in elemOf(\tau) \dot{\cap} elemOf(\sigma)$ or $q \in attrOf(\tau) \cap attrOf(\sigma)$)
         and not($derBy_{\Sigma}(typeOf(\sigma, q), typeOf(\tau, q))$)
      then ‘fail’ and exit
   else return
```

Figure 4: The *Check* procedure

In Figure 3 the *H-Transform* algorithm for transforming a schema in *DF* into a hierarchical schema is presented. It starts by checking the legal restrictions of the value components of types. In particular, it computes the *valueSet* for each type of the schema and, in the presence of simpleTypes whose *valueSet* does not coincide with the related value component, the algorithm fails. Clearly, *valueSet* can be easily computed by a recursive procedure that analyzes types starting at the root of the type hierarchy of Σ.

Successively, all complexTypes in *DF* are analyzed, starting at the root of the type hierarchy, and the *Check* procedure is called, which is defined in Figure 4. Notationally, $elemOf(\tau)$ and $attrOf(\tau)$ denote the bag of element names and the set of attribute names of the type τ, respectively. The *Check* procedure has been conceived to verify: (1) the absence of name conflicts (in particular, the absence of elements with the same names and different types, and attributes with the same names), and (2) the legal restrictions of types (the symbol $\dot{\cap}$ stands for *bag intersection* [27]). If the *Check* procedure succeeds, in the case (i) of derivations by extension the *H-Transform* algorithm performs the bag/set union of the element/attribute components of the type with the corresponding ones of the base type (the symbol $\dot{\cup}$ stands for *bag union* [27]), whereas in the case (ii) of derivations by restriction the verification concerning the one-to-one correspondence ($\leftrightarrow$) among the properties of the type and the properties of the corresponding base type is performed. Finally, the value component of the type has to be set to its *valueSet* (which, in general, do not coincide, as shown for instance in the case of *elevationType* of Example 4.1).

The following proposition can be easily proved.

Proposition 4.1. If the *H-Transform* algorithm succeeds, then the type hierarchy of the resulting schema is legal. $\square$

Example 4.3. For instance, consider our running example, i.e., the schema defined starting at the types introduced in Subsections 2.1.1, and 2.1.2, whose type hierarchy is shown in Figure 2. In this case the algorithm succeeds, and the type hierarchy of the resulting schema is legal. In particular, the schema components are defined by the following type components:

comp(`personInfo`)= ($\emptyset$, <`firstName:string, lastName:string`>,
{`age:integer, married:boolean`})
comp(`employeeInfo`) = ($\emptyset$, <`firstName:string, lastName:string,`
`firstProject:string, secondProject:string`>,
{`age:integer, married:boolean, salary:integer,`
`company:string`})
comp(`elevationType`) = (**Z**, $\emptyset$, {`units:string`})
comp(`nameType`) = ({`John,Tom,Bob`}, $\emptyset$, $\emptyset$)
comp(`teenageType`) = ({`13-19`}, $\emptyset$, $\emptyset$)
comp(`teenagerInfo`) = ($\emptyset$, <`firstName:nameType, lastName:string`>,
{`age:teenageType, married:boolean`})

It is easy to see that the type hierarchy is well-formed, that is, according to Definition 3.9, for each pair of hierarchically related types subtyping holds and, in particular:

`employeeInfo` $\preceq$ `personInfo`
`teenagerInfo` $\preceq$ `personInfo`
`nameType` $\preceq$ `string`
`elevationType` $\preceq$ `integer`
`teenageType` $\preceq$ `integer`

Furthermore, according to Definition 3.10, the type components of such types do not contain name conflicts, and contain additional properties, in the case of extensions, and exactly the same properties of the base types, in the case of restrictions.

Of course, the presence of *firstNameType* as defined in Example 4.1 would generate a failure due to an illegal value restriction. □

Given a schema with n types, where each type contains at most m properties, the algorithm has complexity $O(nm)$. Notice that, since we start with a schema in DF, we assume that types are defined according to Definitions 3.1 and 3.2. Therefore verifications concerning, for instance, the absence of declarations of complexTypes derived by extending a simpleType with elements (rather than attributes) are not addressed.

5. Conclusion

In this paper a formal framework for dealing with the legality of XML-Schema type hierarchies has been proposed. It has been inspired by the work presented in [14], whose approach has been extensively revisited and extended. We have seen that the proposed framework focuses on a subset of XML-Schema constructors, which does not include for instance the *choice*, *union*, *redefine mechanism* or *substitution groups* (and therefore the related type hierarchy constraints). As a future work, there are no obstacles in extending the proposed results to a wider XML-Schema data model including such constructors. In fact, this activity essentially requires the definition of a more elaborated, and probably less intuitive, formalization of both the data model and the algorithm.

References

[1] The World Wide Web Consortium (W3C); http://www.w3.org/XML/Schema.

[2] Borgida, A. and Mylopoulos, J. and Wong, H.K.T. (1984) Generalization/Specialization as a Basis for Software Specification. In Brodie, M.L. and Mylopoulos, J. and Schmidt, J.W. (eds), On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases and Programming Languages. Springer-Verlag, Berlin.

[3] Quillian, M.R. (1968) Semantic Memory. In Minsky, M. (ed), Semantic Inforamtion Processing. MIT Press, Cambridge, MA.

[4] Sowa, J.F. (1991) Principles of Semantic Networks. Morgan Kaufmann, San Mateo, CA.

[5] Ait-Kaci, H. and Nasr, R. (1986) LOGIN: A Logic Programming Language with Built-In Inheritance. Journal of Logic Programming (JLP), 3(3), 185-215.

[6] Cardelli, L. (1988) A Semantics of Multiple Inheritance. Info.&Comp., 76, 138-164.

[7] Dahl, O. and Nygaard, K. (1966) SIMULA - an ALGOL-based simulation language. Communications of the ACM (CACM), 9(9), 671-678.

[8] Beeri, C. (1990) A formal approach to object-oriented databases. Data & Knowledge Engineering (DKE), 5, 353-382.

[9] Khoshafian, S. and Abnous, R. (1990) Object-Orientation - Concepts, Languages, Databases, User Interfaces. Wiley, New York.

[10] Formica, A. and Groger, H.D. and Missikoff, M. (1997) Object-Oriented Database Schema Analysis and Inheritance Processing: a Graph-Theoretic Approach. Data & Knowledge Engineering (DKE), 24(2), 157-181.

[11] Formica, A. and Groger, H.D. and Missikoff, M. (1998) An Efficient Method For Checking Object-Oriented Database Schema Correctness. ACM Transactions on Database Systems (TODS), 23(3), 333-369.

[12] Formica, A. (2002) Finite Satisfiability of Integrity Constraints in Object-Oriented Database Schemas. IEEE Transactions on Knowledge & Data Engineering (TKDE), 14(1), 123-139.

[13] Zicari, R. (1992) A Framework for Schema Updates in an Object-Oriented Database System. In Bancilhon, F. and Delobel, C. and Kanellakis, P. (eds), Building an Object-Oriented Database System - The story of O2. Morgan Kaufmann, San Mateo, CA.

[14] Beeri, C. and Formica, A. and Missikoff, M. (1999) Inheritance Hierarchy Design in Object-Oriented Databases. Data & Knowledge Engineering (DKE), 30(3), 191-216.

[15] Costello, R. (2002) XML Schema Tutorial, http://www.w3.org/XML/Schema.

[16] Lee, D. and Chu, W.W. (2000) Comparative Analysis of Six XML Schema Languages. SIGMOD Record 29(3), 76-87.

[17] Boley, H. (2001) The Rule Markup Language: RDF-XML Data Model, XML Schema Hierarchy, and XSL Transformations. Proc. of International Conference on Applications of Prolog (INAP), Tokyo, Japan, October 20-22, 124-139. Lecture Notes in Computer Science 2543, Springer, Berlin.

[18] Toman, D. and Weddell, G.E. (2003) On Reasoning about Structural Equality in XML: A Description Logic Approach. Proc. of International Conference on Database Theory (ICDT), Siena, Italy, January 8-10, 96-110. Lecture Notes in Computer Science 2572, Springer, Berlin.

[19] Xiaou, R. and Dillon, T.S. and Chang, E. and Feng, L. (2001) Modeling and Transformation of Object-Oriented Conceptual Models into XML Schema. Proc. of Database and Expert Systems Applications (DEXA), Munich, Germany, September 3-5, 795-804. Lecture Notes in Computer Science 2113, Springer, Berlin.

[20] Mani, M. and Lee, D. and Muntz, R.R. (2001) Semantic Data Modeling Using XML Schemas. Proc. of International Conference on Conceptual Modeling (ER), Yokohama, Japan, November 27-30, 149-163. Lecture Notes in Computer Science 2224, Springer, Berlin.

[21] Routledge, N. and Bird, L. and Goodchild, A. (2002) UML and XML Schema. Proc. of Australasian Database Conference (ADC), Melbourne, Australia, January 28 - February 1. Australian Computer Society Inc., Melbourne.

[22] Kuper, G.M. and Simeon, J. (2001) Subsumption for XML Types. Proc. of International Conference on Database Theory (ICDT), London, UK, January 4-6, 331-345. Lecture Notes in Computer Science 1973, Springer, Berlin.

[23] Bird, L. and Goodchild, A. and Halpin, T.A. (2000) Object Role Modelling and XML-Schema. Proc. of International Conference on Conceptual Modeling (ER), Salt Lake City, Utah, USA, October 9-12, 309-322. Lecture Notes in Computer Science 1920, Springer, Berlin.

[24] Arenas, M. and Fan, W. and Libkin, L. (2002) What's Hard about XML Schema Constraints? Proc. of Database and Expert Systems Applications (DEXA), Aix-en-Provence, France, September 2-6, 269-278. Lecture Notes in Computer Science 2453, Springer, Berlin.

[25] Hosoya, H. and Pierce, B.C. (2003) XDuce: A statically typed XML processing language. ACM Transactions on Internet Technology (TOIT), 3(2), 117-148.

[26] Suciu, D. (2002) The XML Typechecking Problem. SIGMOD Record 31(1), 89-96.

[27] Ullman, J.D. and Widom, J. (2001) A First Course in Database Systems. Prentice Hall, Upper Saddle River, New Jersey.