# Equivalence of XSD Constructs and its Exploitation in Similarity Evaluation

Irena Mlýnková

Department of Software Engineering, Charles University
Malostranské nám. 25, 118 00 Prague 1, Czech Republic
irena.mlynkova@mff.cuni.cz

**Abstract.** In this paper we propose a technique for evaluating similarity of XML Schema fragments. Firstly, we define classes of structurally and semantically equivalent XSD constructs. Then we propose a similarity measure that is based on the idea of edit distance utilized to XSD constructs and enables one to involve various additional similarity aspects. In particular, we exploit the equivalence classes and semantic similarity of element/attribute names. Using preliminary experiments we show the behavior and advantages of the proposal.

## 1 Introduction

The eXtensible Markup Language (XML) [3] has become a standard for data representation and, thus, it appears in most of areas of information technologies. A possible optimization of XML-based methods can be found in exploitation of similarity of XML data. In this paper we focus on similarity of XML schemas that can be viewed from two perspectives. We can deal with either *quantitative* or *qualitative* similarity measure. In the former case we are interested in the degree of difference of the schemas, in the latter one we also want to know how the schemas relate, e.g. which of the schemas is more general. In this paper we deal with quantitative measure that is the key aspect of schema mapping [4, 5], i.e. searching for (sub)schemas that describe the same reality.

In this area the key emphasis is currently put on the semantic similarity of element/attribute names reflecting the requirements of corresponding applications. And if the approaches consider schema structure, they usually analyze only simple aspects such as, e.g., leaf nodes or child nodes. In addition, most of the approaches deal with XML schemas expressed in simple DTD language [3]. Hence, in this paper we focus on similarity of XML schema fragments expressed in XML Schema language [11, 2]. In particular, we cover all key XML Schema constructs and we deal with their structural and semantic equivalence. We propose a similarity measure that is based on the idea of classical edit distance utilized to XSD[1] constructs and enables one to involve various additional similarity aspects. In particular, we exploit the equivalence classes of XML constructs

[1] XML Schema Definition

and semantic similarity of element/attribute names. Using various experiments we show the behavior and advantages of the proposed approach.

The paper is structured as follows: Section 2 describes the related works. In Section 3 we overview possible XML Schema constructs and we define their structurally and semantically equivalent classes. In Section 4 we describe the proposed approach and in Section 5 we overview results of related experiments. Finally, Section 6 provides conclusions and outlines future work.

## 2 Related Work

The number of existing works in the area of XML data similarity is nontrivial. We can search for similarity among XML documents, XML schemas or between the two groups. We can distinguish several levels of similarity, such as, e.g., structural level, semantic level or constraint level. Or we can require different precision of the similarity.

In case of document similarity we distinguish techniques expressing similarity of two documents $D_x$ and $D_y$ using edit distance, i.e. by measuring how difficult is to transform $D_x$ into $D_y$ (e.g. [9]) and techniques which specify a simple and reasonable representation of $D_x$ and $D_y$, such as, e.g., using a set of paths, that enables efficient comparison and similarity evaluation (e.g. [12]). In case of similarity of a document $D$ and a schema $S$ there are also two types of strategies – techniques which measure the number of elements which appear in $D$ but not in $S$ and vice versa (e.g. [1]) and techniques which measure the closest distance between $D$ and "all" documents valid against $S$ (e.g. [8]). And finally, methods for measuring similarity of two XML schemas $S_x$ and $S_y$ combine various supplemental information and similarity measures such as, e.g., predefined similarity rules, similarity of element/attribute names, equality of data types, similarity of schema instances or previous results (e.g. [4, 5]). But, in general, the approaches focus mostly on semantic aspects, whereas structural ones are of marginal importance. And what is more, most of the existing works consider only DTD constructs, whereas if the XML Schema language is supported, the constructs beyond DTD expressive power are often ignored.

## 3 XML Schema Constructs and their Equivalence

The most popular language for description of the allowed structure of XML documents is currently the Document Type Definition (DTD) [3]. For simple applications it is sufficient, but more complex ones the W3C proposed a more powerful tool – the XML Schema language [11, 2]. A self-descriptive example of an XSD is depicted in Figure 1.

The constructs of XML Schema can be divided into *basic*, *advanced* and *auxiliary*. The basic constructs involve simple data types (`simpleType`), complex data types (`complexType`), elements (`element`), attributes (`attribute`), groups of elements (`group`) and groups of attributes (`attributeGroup`). Simple data types involve both built-in data types (except for `ID`, `IDREF`, `IDREFS`),

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="employees">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="person" minOccurs="1" maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="person">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="name"/>
    <xs:element name="email" type="xs:string" minOccurs="0" maxOccurs="unbounded"/>
    <xs:element ref="relationships" minOccurs="0" maxOccurs="1"/>
   </xs:sequence>
   <xs:attribute name="id" type="xs:ID" use="required"/>
   <xs:attribute name="note" type="xs:string"/>
   <xs:attribute name="holiday" default="no">
    <xs:simpleType>
     <xs:restriction base="xs:string">
      <xs:enumeration value="yes"/>
      <xs:enumeration value="no"/>
     </xs:restriction>
    </xs:simpleType>
   </xs:attribute>
  </xs:complexType>
 </xs:element>

 <xs:element name="name">
  <xs:complexType>
   <xs:all>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="surname" type="xs:string"/>
   </xs:all>
  </xs:complexType>
 </xs:element>

 <xs:element name="relationships">
  <xs:complexType>
   <xs:attribute name="superior" type="xs:IDREF"/>
   <xs:attribute name="inferior" type="xs:IDREFS"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Fig. 1.** An example of an XSD of employees I

such as, e.g., `string`, `integer`, `date` etc., as well as user-defined data types derived from existing simple types using `simpleType` construct. Complex data types enable one to specify both content models of elements and their sets of attributes. The content models can involve ordered sequences (`sequence`), choices (`choice`), unordered sequences (`all`), groups of elements (`group`) or their allowable combinations. Similarly, they enable one to derive new complex types from existing simple (`simpleContent`) or complex types (`complexContent`). Elements simply join simple/complex types with respective element names and, similarly, attributes join simple types with attribute names. And, finally, groups of elements and attributes enable one to globally mark selected schema fragments and exploit them repeatedly in various parts using so-called *references*. In general, basic constructs are present in almost all XSDs.

The set of *advanced* constructs involves type substitutability and substitution groups, identity constraints (`unique`, `key`, `keyref`) as well as related simple data types (`ID`, `IDREF`, `IDREFS`) and assertions (`assert`, `report`). Type substitutability and substitution groups enable one to change data types or allowed location of elements. Identity constraints enable one to restrict allowed values of elemets/attributes to unique/key values within a specified area and to specify references to them. Similarly, assertions specify additional conditions that the

values of elements/attributes need to satisfy, i.e. they can be considered as an extension of simple types.

The set of *auxiliary* constructs involves wildcards (`any`, `anyAttribute`), external schemas (`include`, `import`, `redefine`), notations (`notation`) and annotations (`annotation`). Wildcards and external schemas combine data from various XML schemas. Notations bear additional information for superior applications. And annotations can be considered as a kind of advanced comments. Consequently, since these constructs do not have a key impact on schema structure or semantics, we will not deal with them in the rest of the text.

### 3.1 Structural Equivalence

As it is obvious from the above overview, there are sets of XML Schema constructs that enable one to generate XSDs that have different structure but are *structurally equivalent*.

**Definition 1.** *Let $S_x$ and $S_y$ be two XSD fragments. Let $I(S) = \{D$ s.t. $D$ is an XML document fragment valid against $S\}$. Then $S_x$ and $S_y$ are* structurally equivalent*, $S_x \sim S_y$, if $I(S_x) = I(S_y)$.*

Consequently, having a set $X$ of all XSD constructs, we can specify the quotient set $X/\sim$ of $X$ by $\sim$ and respective equivalence classes – see Table 1.

| **Class** | **Constructs** | **Canonical representative** |
|---|---|---|
| $C_{ST}$ | globally defined simple type, locally defined simple type | locally defined simple type |
| $C_{CT}$ | globally defined complex type, locally defined complex type | locally defined complex type |
| $C_{El}$ | referenced element, locally defined element | locally defined element |
| $C_{At}$ | referenced attribute, locally defined attribute, attribute referenced via an attribute group | locally defined attribute |
| $C_{ElGr}$ | content model referenced via an element group, locally defined content model | locally defined content model |
| $C_{Seq}$ | unordered sequence of elements $e_1, e_2, ..., e_l$, choice of all possible ordered sequences of $e_1, e_2, ..., e_l$ | choice of all possible ordered sequences of $e_1$, $e_2$, ..., $e_l$ |
| $C_{CTDer}$ | derived complex type, newly defined complex type | newly defined complex type |
| $C_{SubSk}$ | elements in a substitution group $G$, choice of elements in $G$ | choice of elements in $G$ |
| $C_{Sub}$ | data types $M_1, M_2, ..., M_k$ derived from type $M$, choice of content models defined in $M_1, M_2, ..., M_k, M$ | choice of content models defined in $M_1, M_2, ..., M_k, M$ |

**Table 1.** XSD equivalence classes of $X/\sim$

Classes $C_{ST}$ and $C_{CT}$ specify that there is no difference if a simple or a complex type is defined globally or locally as depicted in Figure 2.

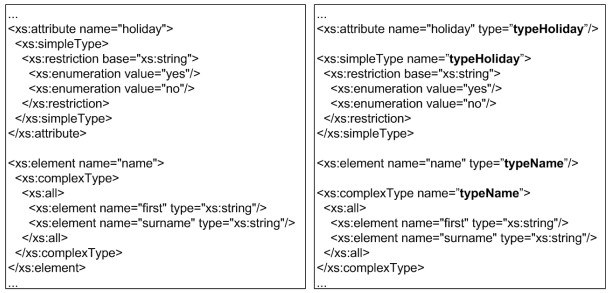


**Fig. 2.** Locally and globally defined data types

Similarly, classes $C_{El}$ and $C_{At}$ determine that locally defined elements/attributes and globally defined referenced elements/attributes are equivalent as depicted in Figure 3.

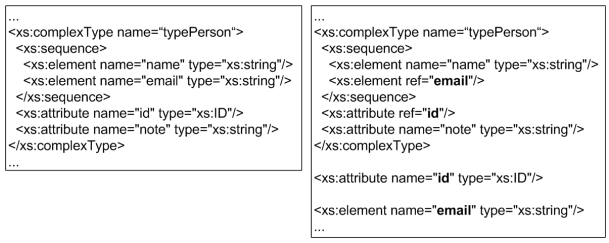


**Fig. 3.** Locally and globally defined elements and attributes

In addition, $C_{At}$ determines that also attributes referenced via attribute groups are equivalent to all other types of attribute specifications. And a similar meaning has also $C_{ElGr}$ class for content models referenced via groups or defined locally. Both situations are depicted in Figure 4.

Class $C_{Seq}$ expresses the equivalence between an unordered sequence of elements $e_1, e_2, ..., e_l$ and a choice of its all possible ordered permutations as depicted in Figure 5.

Class $C_{Inh}$ determines equivalence between a complex type derived from an existing one or a complex type that is defined newly as depicted in Figure 6.

Class $C_{SubSk}$ expresses that the mechanism of substitution groups is equivalent to the choice of respective elements, i.e. having elements $e_1$ and $e_2$ that are in substitution group of element $e_3$, it means that everywhere where $e_3$ occurs, also elements $e_1$ and $e_2$ can occur. The only exception is if $e_3$ is denoted as *abstract*

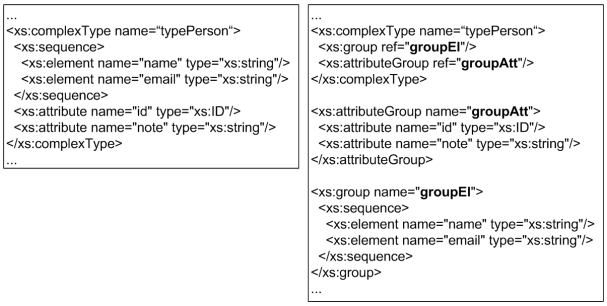


**Fig. 4.** Element and attribute groups

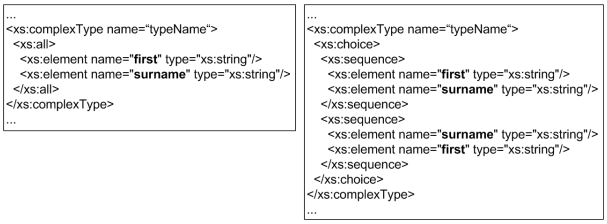


**Fig. 5.** Unordered and ordered sequences

(using attribute `abstract="true"`) and, hence, it must be always substituted. The structure of a substitution group can be also influenced via attributes `final` and `block` that disable substitution for a particular element anywhere it occurs or only at particular positions. An example of respective equivalent schemas is depicted in Figure 7, where we assume that types `typeBook` and `typeJournal` are derived from `typePublication`.

Similarly, class $C_{Sub}$ expresses the fact that having an element $e$ with type $M$, using the attribute `xsi:type` we can substitute $M$ with any of data types $M_1, M_2, ..., M_k$ derived from $M$. Consequently, the content model of $e$ is equivalent to choice of content models defined in $M_1, M_2, ..., M_k, M$. The only exception is when the type substitutability is blocked using the `block` attribute. An example of the equivalent schemas is depicted in Figure 8.

Each of the remaining XML Schema constructs not mentioned in Table 1 forms a single class. We will denote these classes as $C_1, C_2, ..., C_n$.

### 3.2 Semantic Equivalence

Apart from XSD constructs that restrict the allowed structure of XML data, we can find also constructs that express various semantic constraints. They involve identity constrains and simple data types `ID` and `IDREF(S)`. (Note that `ID`, `IDREF(S)` can be expressed using `key` and `keyref`.) The idea of semantic similarity is based on the following observation: A `keyref` construct refers to a particular part of the XSD – e.g. having an XSD containing a list of books

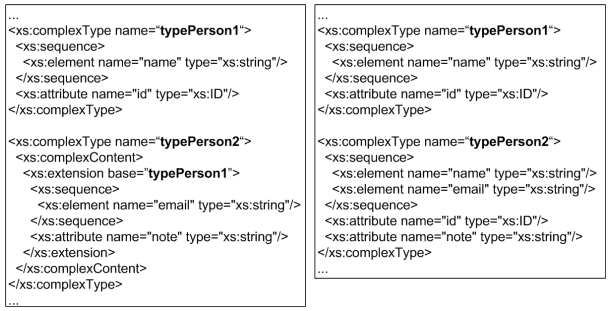


**Fig. 6.** Derived complex types

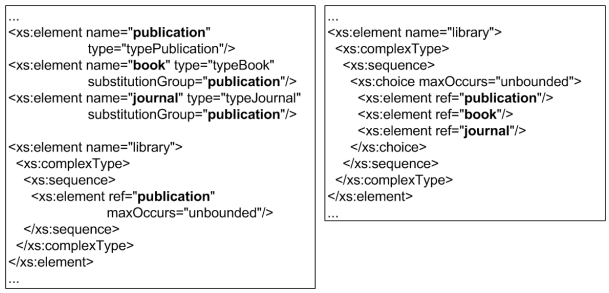


**Fig. 7.** Substitution groups

and a list of authors, each author can refer to his best book. And this situation described in a semantically equivalent manner occurs when the referenced fragment, i.e. the element describing the best book, is directly present within element author. Hence, these constructs enable one to generate XSDs that have different structure but are *semantically equivalent*.

**Definition 2.** *Let $S_x$ and $S_y$ be two XSD fragments. Then $S_x$ and $S_y$ are* semantically equivalent*, $S_x \approx S_y$, if they abstract the same reality.*

Having a set $X$ of all XSD constructs, we can specify the quotient set $X/\approx$ of $X$ by $\approx$ and respective equivalence classes – see Table 2. Classes $C'_{IdRef}$ and $C'_{KeyRef}$ express the fact that both `IDREF(S)` and `keyref` constructs, i.e. references to schema fragments, are semantically equivalent to the situation when we directly copy the referenced schema fragments to the referencing positions. An example of the equivalent schemas is depicted in Figure 9.

Since every `key`/`keyref` constraint must contain one reference (`selector`) to a set of elements and at least one reference (`field`) to their subelements (descendants) and/or attributes expressed in the following grammar [11, 2]:

```
Selector ::= PathS ( '|' PathS )*
Field    ::= PathF ( '|' PathF )*
PathS    ::= ('.//')? Step ( '/' Step )*
```

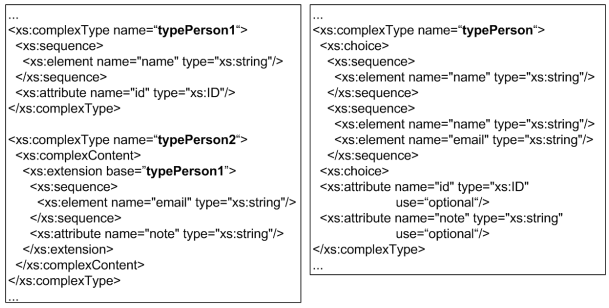


**Fig. 8.** Type substitutability

| Class | Constructs | Canonical representative |
|---|---|---|
| $C'_{IdRef}$ | locally defined schema fragment, schema fragment referenced via `IDREF` attribute | locally defined schema fragment |
| $C'_{KeyRef}$ | locally defined schema fragment, schema fragment referenced via `keyref` element | locally defined schema fragment |

**Table 2.** XSD equivalence classes of $X/\approx$

```
PathF    ::= ('.//')? ( Step '/' )* ( Step | '@' NameTest )
Step     ::= '.' | NameTest
NameTest ::= QName | '*' | NCName ':' '*'
```

the referenced fragments can be always easily copied to particular positions.

Similar to the previous case, each of the remaining XML Schema constructs not mentioned in Table 2 forms a single class. We will denote these classes as $C'_1, C'_2, ..., C'_m$.

Each of the previously defined classes of $\sim$ or $\approx$ equivalence can be represented using any of its elements. Since we want to simplify the specification of XSD for the purpose of analysis of its similarity, we have selected respective *canonical representatives* listed in Tables 1 and 2 as well. They enable one to simplify the structure of the XSD only to core constructs. (Note that since $C_1, C_2, ..., C_n$ and $C'_1, C'_2, ..., C'_m$ are singletons, the canonical representatives are obvious.)

## 4 Similarity Evaluation

The proposed algorithm is based mainly on the work presented in [9] which focuses on expressing similarity of XML documents $D_x$ and $D_y$ using tree edit distance. The main contribution of the algorithm is in introducing two new edit operations *InsertTree* and *DeleteTree* which allow manipulating more complex structures than only a single node. But, repeated structures can be found in an XSD as well, if it contains *shared* fragments or *recursive* elements.

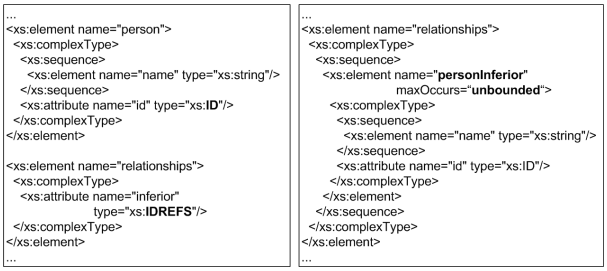


**Fig. 9.** Identity constraints

On the other hand, contrary to XML documents that can be modeled as trees, XSDs can, in general, form general cyclic graphs. Hence, procedures for computing edit distance of trees need to be utilized to XSD graphs. In addition, not only the structural, but also the semantic aspect is very important. Therefore, we will also concern both semantic equivalence of XSD fragments as well as semantic similarity of element/attribute names.

The whole method can be divided into three parts depicted in Algorithm 1.

**Algorithm 1** Main body of the algorithm

**Input:** XSDs $S_x$ and $S_y$
**Output:** Edit distance between $S_x$ and $S_y$
1: $T_x$ = ParseXSD($S_x$);
2: $T_y$ = ParseXSD($S_y$);
3: $Cost_{Graft}$ = ComputeCost($T_y$);
4: $Cost_{Prune}$ = ComputeCost($T_x$);
5: **return** EditDistance($T_x$, $T_y$, $Cost_{Graft}$, $Cost_{Prune}$);

Firstly, the input XSDs $S_x$ and $S_y$ are parsed (line 1 and 2) and their tree representations are constructed. Next, costs for tree inserting (line 3) and tree deleting (line 4) are computed. And in the final step (line 5) we compute the resulting edit distance, i.e. similarity, using classical dynamic programming.

### 4.1 XSD Tree Construction

The key operation of our approach is tree representation of the given XSDs. However, since the structure of an XSD can be quite complex, we firstly normalize and simplify it.

*Normalization of XSDs* Firstly, we normalize the given XSDs using the equivalence classes. In the first step we exploit structural equivalence $\sim$ and we iteratively replace each non-canonical construct (naturally except for the root element) with the respective canonical representative until there can be found

any. At the same time, for each element $v$ of the schema (i.e. XSD construct) we keep the set $v_{eq_\sim}$ of classes it originally belonged to.

In the second step we exploit semantic equivalence $\approx$ and we again replace each non-canonical construct with its canonical representative and we construct sets $v_{eq_\approx}$. Now the resulting schema involves elements, attributes, operators `choice` and `sequence`, intervals of allowed occurrences, simple types and assertions.

*Simplification of XSDs* Next we simplify the remaining content models. For this purpose we can use various transformation rules. Probably the biggest set was defined in [10] for DTD constructs, but these simplifications are for our purpose too strong. Hence, we use only a subset of them as depicted in Figures 10 and 11. They are expressed for DTD constructs, where "`|`" represents `choice`, "`,`" represents `sequence`, "`?`" represents interval $[0, 1]$, "`+`" represents intervals $[v_{low}, v_{up}]$, where $v_{low} > 0$ and $v_{up} > 1$, "`*`" represents intervals $[v_{low}, v_{up}]$, where $v_{low} \geqslant 0$ and $v_{up} > 1$ and empty operator represents interval $[1, 1]$.

| |
|---|
| I-a) $(e_1 \mid e_2)^* \rightarrow e_1^*, e_2^*$ |
| I-b) $(e_1, e_2)^* \rightarrow e_1^*, e_2^*$ |
| I-c) $(e_1, e_2)? \rightarrow e_1?, e_2?$ |
| I-d) $(e_1, e_2)^+ \rightarrow e_1^+, e_2^+$ |
| I-e) $(e_1 \mid e_2) \rightarrow e_1?, e_2?$ |

**Fig. 10.** Flattening rules

| | |
|---|---|
| II-a) $e_1^{++} \rightarrow e_1^+$ | II-b) $e_1^{**} \rightarrow e_1^*$ |
| II-c) $e_1^*? \rightarrow e_1^*$ | II-d) $e_1?^* \rightarrow e_1^*$ |
| II-e) $e_1^{+*} \rightarrow e_1^*$ | II-f) $e_1^{*+} \rightarrow e_1^*$ |
| II-g) $e_1?^+ \rightarrow e_1^*$ | II-h) $e_1^+? \rightarrow e_1^*$ |
| II-i) $e_1?? \rightarrow e_1^?$ | |

**Fig. 11.** Simplification rules

The rules enable one to convert all element definitions so that each cardinality constraint operator is connected to a single element. The second purpose is to avoid usage of `choice` construct. Note that some of the rules do not produce equivalent XML schemes and cause a kind of information loss. But this aspect is common for all existing XML schema similarity measures – it seems that the full generality of the regular expressions cannot be captured easily.

*XSD Tree* Having a normalized and simplified XSD, its tree representation is defined as follows:

**Definition 3.** *An* XSD tree *is an ordered tree $T = (V, E)$, where*

1. *$V$ is a set of nodes of the form $v = (v_{Type}, v_{Name}, v_{Cardinality}, v_{eq_\sim}, v_{eq_\approx})$, where $v_{Type}$ is the type of a node (i.e. attribute, element or particular simple data type), $v_{Name}$ is the name of an element or an attribute, $v_{Cardinality}$ is the interval $[v_{low}, v_{up}]$ of allowed occurrence of $v$, $v_{eq_\sim}$ is the set of classes of $\sim$ $v$ belongs to and $v_{eq_\approx}$ is the set of classes of $\approx$ $v$ belongs to,*
2. *$E \subseteq V \times V$ is a set of edges representing relationships between elements and their attributes or subelements.*

An example of tree representation of XSD in Figure 1 (after normalization and simplification) is depicted in Figure 12.

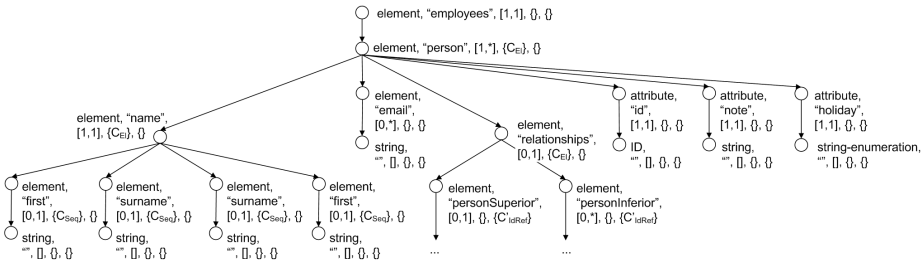


**Fig. 12.** An example of an XSD tree

*Shared and Recursive Elements* As we have mentioned, the structure of an XSD does not have to be purely tree-like. There can occur both shared elements which invoke undirected cycles and recursive elements which invoke directed cycles. The shared elements are eliminated in XSD normalization using canonical representatives, where all globally defined schema fragments are replaced with their locally defined copy. But, in case of recursive elements we cannot repeat the same idea since the recursion would invoke infinitely deep tree branches. However, in this case we exploit the observation of an analysis of real-world XML data [7] that the amount of recursive inclusions is on average less than 10. So we approximate the infinite amount with the constant one. Naturally, this is a kind of information loss, but based on the knowledge of real-world data.

### 4.2 Tree Edit Operations

Having the above described tree representation of an XSD, we can now easily utilize the tree edit algorithm proposed in [9]. For a given tree $T$ with a root node $r$ of degree $t$ and its first-level subtrees $T_1, T_2, ..., T_t$, the tree edit operations are defined formally as follows:

**Definition 4.** *$Substitution_T(r_{new})$ is a node substitution operation applied to $T$ that yields the tree $T'$ with root node $r_{new}$ and first-level subtrees $T_1, ..., T_t$.*

**Definition 5.** *Given a node $x$ with degree 0, $Insert_T(x, i)$ is a node insertion operation applied to $T$ at $i$ that yields the new tree $T'$ with root node $r$ and first-level subtrees $T_1$, ..., $T_i$, $x$, $T_{i+1}$, ..., $T_t$.*

**Definition 6.** *If the first-level subtree $T_i$ is a leaf node, $Delete_T(T_i)$ is a delete node operation applied to $T$ at $i$ that yields the tree $T'$ with root node $r$ and first-level subtrees $T_1$, ..., $T_{i-1}$, $T_{i+1}$, ..., $T_t$.*

**Definition 7.** *Given a tree $T_x$, $InsertTree_T(T_x, i)$ is an insert tree operation applied to $T$ at $i$ that yields the tree $T'$ with root node $r$ and first-level subtrees $T_1$, ..., $T_i$, $T_x$, $T_{i+1}$, ..., $T_t$.*

**Definition 8.** *$DeleteTree_T(T_i)$ is a delete tree operation applied to $T$ at $i$ that yields the tree $T'$ with root node $r$ and first-level subtrees $T_1, ..., T_{i-1}, T_{i+1}, ..., T_t$.*

Transformation of a source tree $T_x$ to a destination tree $T_y$ can be done using a number of sequences of the operations. But, we can only deal with so-called *allowable* sequences, i.e. the relevant ones. For the purpose of our approach we only need to modify the original definition as follows:

**Definition 9.** *A sequence of edit operations transforming a source tree $T_x$ to a destination tree $T_y$ is* allowable *if it satisfies the following two conditions:*

1. *A tree $T$ may be inserted only if tree similar to $T$ already occurs in $T_x$. A tree $T$ may be deleted only if tree similar to $T$ occurs in $T_y$.*
2. *A tree that has been inserted via the InsertTree operation may not subsequently have additional nodes inserted. A tree that has been deleted via the DeleteTree operation may not previously have had nodes deleted.*

While the original definition requires exactly the same nodes and trees, we relax the requirement only to similar ones. The exact meaning of the similarity is explained in the following text and enables one to combine the tree edit distance with other approaches. Also note that each of the edit operations is associated with a non-negative cost.

### 4.3 Costs of Inserting and Deleting Trees

Inserting (deleting) a subtree $T_i$ can be done with a single operation *InsertTree* (*DeleteTree*) or with a combination of *InsertTree* (*DeleteTree*) and *Insert* (*Delete*) operations. To find the optimal variant the algorithm uses pre-computed cost for inserting $T_i$, $Cost_{Graft}(T_i)$ and deleting tree $T_i$, $Cost_{Prune}(T_i)$. The procedure can be divided into two parts: In the first part *ContainedIn* list is created for each subtree of $T_i$. In the second part $Cost_{Graft}$ and $Cost_{Prune}$ are computed for $T_i$. The procedure is described in [9], but in our approach it is modified to involve similarity of elements/attributes and their respective parameters.

*Similarity of Elements/Attributes* Similarity of two elements/attributes $v$ and $v'$ can be evaluated using various criteria. Since the structural similarity is solved via the edit distance, we focus on semantic and syntactic similarity of element/attribute names, cardinality-constraint similarity, structural/semantic similarity of schema fragments and similarity of simple data types.

*Semantic similarity of element/attribute names* is a score that reflects the semantic relation between the meanings of two words. We exploit procedure described in [5] which determines ontology similarity between two words $v_{Name}$ and $v'_{Name}$ by comparing $v_{Name}$ with synonyms of $v'_{Name}$.

*Syntactic similarity of element/attribute names* is determined by computing the edit distance between $v_{Name}$ and $v'_{Name}$. For our purpose the classical Levenshtein algorithm [6] is used that determines the edit distance of two strings using inserting, deleting or replacing single characters.

*Similarity of cardinality constraints* is determined by similarity of intervals $v_{Cardinality} = [v_{low}, v_{up}]$ and $v'_{Cardinality} = [v'_{low}, v'_{up}]$. It is defined as follows:

$$
\begin{aligned}
CardSim(v, v') &= 0 && ; (v_{up} < v'_{low}) \vee (v'_{up} < v_{low}) \\
&= 1 && ; v_{up}, v'_{up} = \infty \wedge v_{low} = v'_{low} \\
&= 0.9 && ; v_{up}, v'_{up} = \infty \wedge v_{low} \neq v'_{low} \\
&= 0.6 && ; v_{up} = \infty \vee v'_{up} = \infty \\
&= \tfrac{min(v_{up}, v'_{up}) - max(v_{low}, v'_{low})}{max(v_{up}, v'_{up}) - min(v_{low}, v'_{low})} && ; \text{otherwise}
\end{aligned}
$$

*Structural/semantic similarity of schema fragments* rooted at $v$ and $v'$ is determined by the similarity of sets $v_{eq_\sim}$, $v'_{eq_\sim}$ and $v_{eq_\approx}$, $v'_{eq_\approx}$ as follows:

$$
\begin{aligned}
StrFragSim(v, v') &= 1 && ; v_{eq_\sim}, v'_{eq_\sim} = \emptyset \\
&= \tfrac{|v_{eq_\sim} \cap v'_{eq_\sim}|}{|v_{eq_\sim} \cup v'_{eq_\sim}|} && ; \text{otherwise} \\
SemFragSim(v, v') &= 1 && ; v_{eq_\approx}, v'_{eq_\approx} = \emptyset \\
&= \tfrac{|v_{eq_\approx} \cap v'_{eq_\approx}|}{|v_{eq_\approx} \cup v'_{eq_\approx}|} && ; \text{otherwise}
\end{aligned}
$$

And, finally, *similarity of data types* is determined by similarity of simple types $v_{Type}$ and $v'_{Type}$. It is specified by *type compatibility matrix* that determines similarity of distinct simple types. For instance, similarity of `string` and `normalizedString` is 0.9, whereas similarity of `string` and `positiveInteger` is 0.5. Similarly, the table involves similarity of restrictions of simple types specified either via derivation of data types or assertions as well as similarity between element and attribute nodes. (We omit the whole table for the paper length.)

The overall similarity, $Sim(v, v')$ is computed as follows:

$$
\begin{aligned}
Sim(v, v') = {} & Max(SemanticSim(v, v'), SyntacticSim(v, v')) \times \alpha_1 \\
& + CardSim(v, v') \times \alpha_2 \\
& + StrFragSim(v, v') \times \alpha_3 \\
& + SemFragSim(v, v') \times \alpha_4 \\
& + DataTypeSim(v, v') \times \alpha_5
\end{aligned}
$$

where $\sum_{i=1}^{5} \alpha_i = 1$ and $\forall i : \alpha_i \geqslant 0$.

*Construction of ContainedIn Lists* The procedure for determining element/attribute similarity is used for creating *ContainedIn* lists which are then used for computing $Cost_{Graft}$ and $Cost_{Prune}$. The list is created for each node of the destination tree $T_y$ and contains pointers to similar nodes in the source tree $T_x$. The procedure for creating *ContainedIn* lists is shown in Algorithm 2.

Since creating of lists starts from leaves and continues to root, there is recursive calling of procedure at line 2. At line 4 we find all similar nodes of $r$ in tree $T_x$ and add them to a temporary list. If $r$ is a leaf node, the *ContainedIn* list is created. For a non-leaf node we have to filter the list with lists of its descendants (line 6). At this step each descendant of $r$ has to be found at corresponding position in descendants of nodes in the created *ContainedIn* list. More precisely, let $u \in r_{ContainedIn}$, $children_u$ is the set of $u$ descendants and $v$ is a child of $r$. Then $v_{ContainedIn} \cap children_u \neq \emptyset$, otherwise $u$ is removed from $r_{ContainedIn}$.

*Costs of Inserting Trees* When the *ContainedIn* list with corresponding nodes is created for node $r$, the cost for inserting the tree rooted at $r$ can be assigned.

**Algorithm 2** CreateContainedInLists($T_x$, $r$)

**Input:** tree $T_x$, root $r$ of tree $T_y$
**Output:** $CointainedIn$ lists for all nodes in tree $T_y$

```
1: for all child of r do
2:     CreateContainedInLists(T_x, child);
3: end for
4: r_ContainedIn = FindSimilarNodes(T_x, r);
5: for all child of r do
6:     r_ContainedIn = FilterLists(r_ContainedIn, child_ContainedIn);
7: end for
8: Sort(r_ContainedIn);
```

The procedure is shown in Algorithm 3. The *forall* loop computes sum $sum_0$ for inserting node $r$ and all its subtrees. If *InsertTree* operation can be applied (*ContainedIn* list of $r$ is not empty), $sum_1$ is computed for this operation at line 8. The minimum of these costs is finally denoted as $Cost_{Graft}$ for node $r$.

**Algorithm 3** ComputeCost(r)

**Input:** root $r$ of tree $T_y$
**Output:** $Cost_{Graft}$ for tree $T_y$

```
1: sum_0 = 1;
2: for all child of r do
3:     ComputeCost(child);
4:     sum_0 += Cost_Graft(child);
5: end for
6: sum_1 = ∞;
7: if r_ContainedIn is not empty then
8:     sum_1 = ComputeInsertTreeCost(r);
9: end if
10: Cost_Graft(r) = Min(sum_0,sum_1);
```

*Costs of Deleting Trees* Since the rules for deleting a subtree from the source tree $T_x$ are the same as rules for inserting a subtree into the destination tree $T_y$, costs for deleting trees are obtained by the same procedures. We only switch tree $T_x$ with $T_y$ in procedures $CreateContainedInLists$ and $ComputeCost$.

### 4.4 Computing Edit Distance

The last part of the algorithm, i.e. computing the edit distance, is based on dynamic programming. At this step the procedure decides which of the operations defined in Section 4.2 will be applied for each node to transform source tree $T_x$ to destination tree $T_y$. This part of algorithm does not have to be modified for XSDs so the original procedure presented in [9] is used.

### 4.5 Complexity

In [9] it was proven that the complexity of transforming tree $T_x$ into tree $T_y$ is $O(|T_x||T_y|)$. In our method we have to consider procedures for constructing XSD trees and mainly for evaluating similarity. Constructing an XSD tree can be done in $O(|T_x|)$ for tree $T_x$. Complexity of similarity evaluation depends on procedures *SemanticSim*, *SyntacticSim*, *CardSim*, *StrFragSim*, *SemFragSim* and *DataTypeSim*. Syntactic similarity is computed for each pair of elements in $T_x$ and $T_y$, so its complexity is $O(|T_x||T_y||\omega|)$, where $\omega$ is maximum length of an element/attribute label. Similarity of cardinality, similarity of simple types and structural/semantic similarity of schema fragments is also computed for each pair of elements, however, it is an operation with constant complexity, i.e. their complexity is $O(|T_x||T_y|)$. Complexity of finding semantic similarity depends on the size of the thesaurus and on the number of iterations of searching synonyms. Since it is reasonable to search synonyms only for a few steps, the overall complexity is $O(|T_x||T_y||\Sigma|)$, where $\Sigma$ is the set of words in the thesaurus. And it also determines the overall complexity of the algorithm.

## 5 Experiments

For the purpose of experimental evaluation of the proposal we have created next two synthetic XSDs that are from various points of view more or less similar to the XSD depicted in Figure 1. They are depicted in Figures 13 and 14.

At first glance the XSD II is structurally highly different from the original XSD (denoted as XSD I). But, under a closer investigation, we can see that the difference is only within classes of $\sim$ equivalence. On the other hand, XSD III differs in more aspects, such as, e.g., simple types, allowed occurrences, globally/locally defined data types, exploitation of groups, element/attribute names, attributes vs. elements with simple types etc.

As we can see in Table 3 which depicts the results in case we set $\alpha_3 = \alpha_4 = 0$, i.e. we ignore the information on original constructs of XML Schema, the similarity of XSD I and XSD II is 1.0, because they are represented using identical XSD trees. Similarity between XSD I vs. XSD III and XSD II vs. XSD III are for the same reason equivalent, though naturally lower.

| | **XSD I** | **XSD II** | **XSD III** |
|---|---|---|---|
| **XSD I** | 1.00 | 1.00 | 0.82 |
| **XSD II** | 1.00 | 1.00 | 0.82 |
| **XSD III** | 0.82 | 0.82 | 1.00 |

**Table 3.** Similarity for $\alpha_3 = \alpha_4 = 0$

| | **XSD I** | **XSD II** | **XSD III** |
|---|---|---|---|
| **XSD I** | 1.00 | 0.89 | 0.66 |
| **XSD II** | 0.89 | 1.00 | 0.70 |
| **XSD III** | 0.66 | 0.70 | 1.00 |

**Table 4.** Similarity for $\alpha_3 \neq 0$

If we set $\alpha_3 \neq 0$ (according to our experiments it should be $> 0.2$ to influence the algorithm), the resulting similarity is influenced by the difference between

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:simpleType name="typeHoliday">
  <xs:restriction base="xs:string">
   <xs:enumeration value="yes"/>
   <xs:enumeration value="no"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:simpleType name="typeEmail">
  <xs:restriction base="xs:string"/>
 </xs:simpleType>

 <xs:group name="groupContact">
  <xs:sequence>
   <xs:element ref="name"/>
   <xs:element name="email" type="typeEmail"
                minOccurs="0" maxOccurs="unbounded"/>
   <xs:element ref="relationships" minOccurs="0"
                maxOccurs="1"/>
  </xs:sequence>
 </xs:group>

 <xs:element name="employees">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="person" minOccurs="1"
                 maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="person">
  <xs:complexType>
   <xs:group ref="groupContact"/>
   <xs:attribute name="id" type="xs:ID" use="required"/>
   <xs:attribute name="note" type="xs:string"/>
   <xs:attribute name="holiday" type="typeHoliday"/>
  </xs:complexType>
 </xs:element>

 <xs:element name="name">
  <xs:complexType>
   <xs:all>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="surname" type="xs:string"/>
   </xs:all>
  </xs:complexType>
 </xs:element>

 <xs:element name="relationships">
  <xs:complexType>
   <xs:attribute name="superior" type="xs:IDREF"/>
   <xs:attribute name="inferior" type="xs:IDREFS"/>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Fig. 13.** An example of an XSD of employees II

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:simpleType name="typeVacation">
  <xs:restriction base="xs:string">
   <xs:enumeration value="yes"/>
   <xs:enumeration value="no"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:simpleType name="typeEmail">
  <xs:restriction base="xs:string">
   <xs:maxLength value="256"/>
  </xs:restriction>
 </xs:simpleType>

 <xs:group name="groupContact">
  <xs:sequence>
   <xs:element ref="name"/>
   <xs:element name="email" type="typeEmail"
                minOccurs="1" maxOccurs="unbounded"/>
   <xs:element ref="connections" minOccurs="0"
                maxOccurs="1"/>
   <xs:element name="note" type="xs:string"
                minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
 </xs:group>

 <xs:element name="employees">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="subject" minOccurs="1"
                 maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:element name="subject">
  <xs:complexType>
   <xs:group ref="groupContact"/>
   <xs:attribute name="id" type="xs:ID" use="required"/>
   <xs:attribute name="vacation" type="typeVacation"/>
  </xs:complexType>
 </xs:element>

 <xs:element name="name">
  <xs:complexType>
   <xs:all>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="surname" type="xs:string"/>
   </xs:all>
  </xs:complexType>
 </xs:element>

 <xs:element name="connections">
  <xs:complexType>
   <xs:sequence>
    <xs:element ref="subject" minOccurs="0"
                 maxOccurs="unbounded"/>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

**Fig. 14.** An example of an XSD of employees III

the used XML Schema constructs. The results are depicted in Table 4, where we can see more precise results. In particular, the similarity of XSD I and II is naturally $\neq 1.0$, and similarity of XSD II and III is higher due to the respective higher structural similarity of constructs.

On the other hand, if we set $\alpha_4 \neq 0$ and $\alpha_3 = 0$, i.e. we are interested in semantic similarity of schema fragments, the results have the same trend as results in Table 3, because we again omit structural similarity of XSD constructs, but in this case the semantic similarity of schema fragments `relationships` and `connections` is high.

As we have mentioned in Section 4.5, the most time consuming operation of the approach which determines the overall complexity of the algorithm is searching the thesaurus. Hence, in the last test we try to omit evaluation of *SemanticSim*. If we consider the first situation, i.e. when $\alpha_3 = \alpha_4 = 0$, it influences similarity with XSD III (which drops to 0.33), whereas similarity of XSD I and II remains the same because the respective element/attribute names are the same. The results in case $\alpha_3 \neq 0$ are depicted in Table 5. As we can see, the similarity of XSD I and II remains the same again, whereas the other values are much lower.

| | **XSD I** | **XSD II** | **XSD III** |
|---|---|---|---|
| **XSD I** | 1.00 | 0.89 | 0.24 |
| **XSD II** | 0.89 | 1.00 | 0.255 |
| **XSD III** | 0.24 | 0.255 | 1.00 |

**Table 5.** Similarity without *SemanticSim*

In general, the experiments show that various parameters of the similarity measure can highly influence the results. On the other hand, we cannot simply analyze all possible aspects, since some applications may not be interested, e.g., in semantic similarity of used element/attribute names or the "syntactic sugar" (i.e. structurally equivalent constructs) XML Schema involves. Consequently, a reasonable approach should enable one to exploit various aspects as well as temporarily omit the irrelevant ones.

## 6 Conclusion

The aim of this paper was a proposal of an algorithm for evaluating similarity of XML Schema constructs which enable one to specify the structure and semantics of XML data more precisely. For this purpose we have defined structural and semantic equivalence of XSD constructs and we have proposed similarity measure based on classical edit distance strategy that enables one to analyze their structure more precisely and to involve additional similarity aspects. In particular, we have exploited the proposed equivalence classes and semantic similarity of element/attribute names.

In our future work we will focus mainly on further improvements of our approach. We will deal with other edit operations (e.g. moving a node or adding/deleting a non-leaf node), improvements of efficiency of supplemental algorithms, especially the semantic similarity, and on problems related to reasonable setting of involved weights. We will also deal with more elaborate experimental testing. In particular, we will focus on implementing a simulator that would provide distinct XSDs.

## 7 Acknowledgement

This work was supported in part by the National Programme of Research (Information Society Project 1ET100300419).

## References

1. E. Bertino, G. Guerrini, and M. Mesiti. A Matching Algorithm for Measuring the Structural Similarity between an XML Document and a DTD and its Applications. *Inf. Syst.*, 29(1):23–46, 2004.
2. P. V. Biron and A. Malhotra. *XML Schema Part 2: Datatypes (Second Edition)*. W3C, 2004.
3. T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fourth Edition)*. W3C, 2006.
4. H. H. Do and E. Rahm. COMA – A System for Flexible Combination of Schema Matching Approaches. In *VLDB'02: Proc. of the 28th Int. Conf. on Very Large Data Bases*, pages 610–621, Hong Kong, China, 2002. Morgan Kaufmann.
5. M. L. Lee, L. H. Yang, W. Hsu, and X. Yang. XClust: Clustering XML Schemas for Effective Integration. In *CIKM'02: Proc. of the 11th Int. Conf. on Information and Knowledge Management*, pages 292–299, New York, NY, USA, 2002. ACM.
6. V. I. Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.
7. I. Mlynkova, K. Toman, and J. Pokorny. Statistical Analysis of Real XML Data Collections. In *COMAD'06: Proc. of the 13th Int. Conf. on Management of Data*, pages 20–31, New Delhi, India, 2006. Tata McGraw-Hill Publishing.
8. P. K.L. Ng and V. T.Y. Ng. Structural Similarity between XML Documents and DTDs. In *ICCS'03: Proc. of the Int. Conf. on Computational Science*, pages 412–421. Springer-Verlag, 2003.
9. A. Nierman and H. V. Jagadish. Evaluating Structural Similarity in XML Documents. In *WebDB'02: Proc. of the 5th Int. Workshop on the Web and Databases*, pages 61–66, Madison, Wisconsin, USA, 2002.
10. J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational Databases for Querying XML Documents: Limitations and Opportunities. In *VLDB'99: Proc. of 25th Int. Conf. on Very Large Data Bases*, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann.
11. H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. *XML Schema Part 1: Structures (Second Edition)*. W3C, 2004.
12. Z. Zhang, R. Li, S. Cao, and Y. Zhu. Similarity Metric for XML Documents. In *FGWM'03: Proc. of Workshop on Knowledge and Experience Management*, Karlsruhe, Germany, 2003.