Equivalence of XSD Constructs and its Exploitation in Similarity Evaluation

Irena Mlynkova

irena.mlynkova@mff.cuni.cz

Charles University
Faculty of Mathematics and Physics
Department of Software Engineering
Prague, Czech Republic

Introduction

- XML = a standard for data representation and manipulation
  - ⇒ used in most areas of IT
- Classical optimization approach: similarity
  - Clustering, dissemination-based applications, data/schema integration systems, data warehousing, e-commerce, semantic query processing, …
- Our focus: similarity of XML schemas
  - XML-to-relational mapping strategies
  - Quantitative = the degree of difference of the schemas

Goals of the Paper

- **Disadvantages to be solved:**
  - **Current approaches focus on**
    - **Semantic similarity**
    - **Similarity of DTDs**
  - **Structural similarity is analyzed trivially**
    - **Comparison of leaf nodes / direct child nodes**
- **Our aims:**
  - **Focus on XML Schema constructs**
  - **Emphasis on structural similarity**
    - **Utilized edit distance**
  - **Preservation of exploitation of semantic similarity**

Equivalence of XSD Constructs

- **XML Schema constructs: lot of "syntactic sugar"**

Definition. **Let S_x and S_y be XSD fragments. Let I(S) = {*D* s.t. *D* is an XML document fragment valid against S}.**

- **S_x and S_y are structurally equivalent, $S_x \sim S_y$, if I(S_x) = I(S_y).**
- **S_x and S_y are semantically equivalent, $S_x \approx S_y$, if they abstract the same reality.**
  - **A vague definition**

⇒ **Having a set *X* of all XSD constructs:**

  - **Quotient sets $X/\sim$ and $X/\approx$, respective equivalence classes, canonical representatives**

Equivalence Classes of ~

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 $M_1, M_2, ..., M_k, M$

```
<xs:attribute name="holiday">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="yes"/>
      <xs:enumeration value="no"/>
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
```

```
<xs:attribute name="holiday" type="typeHoliday"/>

<xs:simpleType name="typeHoliday">
  <xs:restriction base="xs:string">
    <xs:enumeration value="yes"/>
    <xs:enumeration value="no"/>
  </xs:restriction>
</xs:simpleType>
```

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

Examples

```
<xs:complexType name="typeName">
  <xs:choice>
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="surname" type="xs:string"/>
    </xs:sequence>
    <xs:sequence>
      <xs:element name="surname" type="xs:string"/>
      <xs:element name="first" type="xs:string"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Equivalence Classes of ≈

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

```
<xs:element name="person">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:ID"/>
  </xs:complexType>
</xs:element>

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

```
<xs:element name="relationships">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="personInferior"
                            maxOccurs=“unbounded“>
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
          </xs:sequence>
          <xs:attribute name="id" type="xs:ID"/>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Example

Similarity Evaluation

- **Similarity of XML documents = tree edit distance**
  - **XML documents D_A and D_B = labelled trees T_A and T_B**
  - **Number of operations to transform T_A to T_B**
- **Basic tree edit operations: Relabeling, InsertNode, DeleteNode**
  - **XML data: sharing, repetitions, recursion, …**

  ⇒ **XML tree edit operations: InsertTree, DeleteTree**
- **Algorithm:**
  1. **XSDs are parsed + their trees are constructed**
  2. **Costs for inserting/deleting subtrees are computed**
  3. **Resulting minimal edit distance is evaluated**
     - **Dynamic programming**

XSD Tree Construction (1)

- **XSD content models can be complex**
  - **“Syntactic sugar”, operators, recursion, shared fragments, ….**

1. **Normalization:**
   - **Replace each non-canonical construct with respective canonical representative of ~ and ≈**
   - **For each XSD construct *v* keep the set $v_{eq\sim}$ and $v_{eq\approx}$ of classes it originally belonged to**

⇒ Schema involves elements, attributes, operators choice and sequence, allowed occurrences, simple types and assertions

- **No shared schema fragments**
- ***Note:* We omit solution of recursion for paper length**

XSD Tree Construction (2)

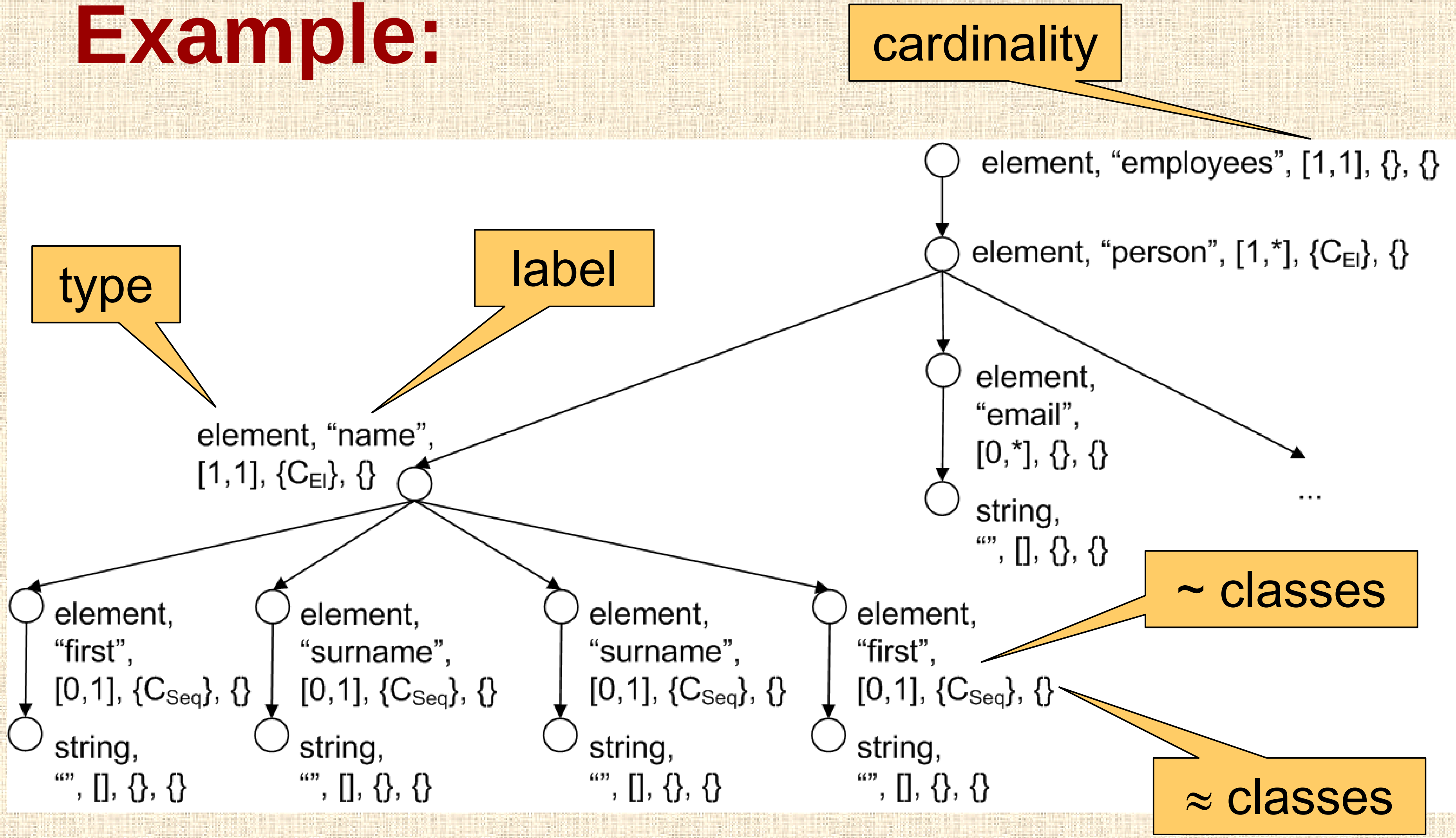
2. **Simplification rules:**

I.a) $(e_1 \mid e_2)^* \to e_1^*, e_2^*$	II.a) $e_1^{++} \to e_1^{+}$	II.b) $e_1^{**} \to e_1^{*}$
I.b) $(e_1, e_2)^* \to e_1^*, e_2^*$	II.c) $e_1^{*}? \to e_1^{*}$	II.d) $e_1?^{*} \to e_1^{*}$
I.c) $(e_1, e_2)? \to e_1?, e_2?$	II.e) $e_1^{+*} \to e_1^{*}$	II.f) $e_1^{*+} \to e_1^{*}$
I.d) $(e_1, e_2)^+ \to e_1^+, e_2^+$	II.g) $e_1?^{+} \to e_1^{*}$	II.h) $e_1^{+}? \to e_1^{*}$
I.e) $(e_1 \mid e_2) \to e_1?, e_2?$	II.i) $e_1?? \to e_1?$	

⇒ **Cardinality constraints are connected to single elements, no usage of | (choice) operator**

- **A slight information loss, but still sufficient for our purpose**

Example:


cardinality
element, “employees”, [1,1], {}, {}
element, “person”, [1,*], {C_El}, {}
type
label
element, “name”, [1,1], {C_El}, {}
element, “email”, [0,*], {}, {}
string, “”, [], {}, {}
...
~ classes
element, “first”, [0,1], {C_Seq}, {}
string, “”, [], {}, {}
element, “surname”, [0,1], {C_Seq}, {}
string, “”, [], {}, {}
element, “surname”, [0,1], {C_Seq}, {}
string, “”, [], {}, {}
element, “first”, [0,1], {C_Seq}, {}
string, “”, [], {}, {}
≈ classes

Tree Edit Operations

- Same as for XML trees: Relabeling, InsertNode, DeleteNode, InsertTree, DeleteTree
- Transformation of T_A to T_B: various sequences of operations
- Optimization: allowable sequences
  - Tree T may be inserted only if tree similar to T occurs in T_B
  - Tree T may be deleted only if tree similar to T occurs in T_A
  - Tree that has been inserted via the InsertTree may not subsequently have additional nodes inserted
  - Tree that has been deleted via the DeleteTree may not previously have had nodes deleted

relaxed

Similarity

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

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

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

- **SemanticSim: distance in thesaurus**
- **SyntacticSim: edit distance**
- **DataTypeSim: type compatibility matrix**

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

Cost of Tree Edit Operations

- **Inserting/deleting tree *T*:**
  - **Single InsertTree/DeleteTree … a combination of InsertTree/DeleteTree and Insert/Delete**
  - **Which is the best?**
- **Idea:**
  - **Pre-computed: $Cost_{Graft}(T)$, $Cost_{Prune}(T)$ for each subtree T**
  - **Dynamic programming: finds the optimal sequence of edit operations**
- **Classical approach for tree edit distance**
  - **See the paper for details…**

Expe-riments

Test		I × II	II × III	III × I
A	$\alpha_3 = \alpha_4 = 0$	1.00	0.82	0.82
B	$\alpha_4 = 0,\ \alpha_3 \neq 0$	0.89	0.70	0.66
C	$\alpha_3 = 0,\ \alpha_4 \neq 0$	1.00	0.80	0.80
D	A without *SemanticSim*	1.00	0.33	0.33
E	B without *SemanticSim*	0.89	0.255	0.24

- **Testing set: 3 synthetic XSDs**
  - **I and II differ within ~, III differs in more aspects**
- **Test A = we ignore the information on original XSD constructs**
- **Test B = similarity is influenced by structural difference between XSD constructs**
  - **More precise results**
- **Test C = structural differences are ignored**
  - **The same trend as in A, more precise**
- **Test D ,E = exploitation of SemanticSim**
  - **Expensive operation**
  - **Provides more precise results**

Conclusion

- Algorithm for evaluating XSD similarity
  - Emphasis on structural level
  - Coping with “syntactic sugar” of XML Schema
  - Preserving exploitation of semantics
- Key idea: Combination of edit distance and semantic similarity
- Future work:
  - More elaborate testing
  - Other edit operations
    - Moving a node or adding/deleting a non-leaf node
  - Setting weights

Thank you