# Extensible User-based XML Grammar Matching

Joe Tekli[1], Richard Chbeir[1] and Kokou Yetongnon[1]

[1] LE2I Laboratory UMR-CNRS, University of Bourgogne
21078 Dijon Cedex France
{joe.tekli, richard.chbeir, kokou.yetongnon}@u-bourgogne.fr

**Abstract.** XML grammar matching has found considerable interest recently due to the growing number of heterogeneous XML documents on the web and the increasing need to integrate, and consequently search and retrieve XML data originated from different data sources. In this paper, we provide an approach for automatic XML grammar matching and comparison aiming to minimize the amount of user effort required to perform the match task. We propose an open framework based on the concept of tree edit distance, integrating different matching criterions so as to capture XML grammar element semantic and syntactic similarities, cardinality and alternativeness constraints, as well as data-type correspondences and relative ordering. It is flexible, enabling the user to chose mapping cardinality (*1:1*, *1:n*, *n:1*, *n:n*), in comparison with existing static methods (constrained to *1:1*), and considers user feedback to adjust matching results to the user's perception of correct matches. Conducted experiments demonstrate the efficiency of our approach, in comparison with alternative methods.

**Keywords:** XML and Semi-structured data, XML grammar, schema matching, structural similarity, tree edit distance, vector space model.

## 1 Introduction

With the growing amount of heterogeneous XML information on the Web, i.e., documents originated from different data-sources, there is an increasing need to perform XML data integration, data warehousing and consequently XML information extraction, search and retrieval. All these applications require, in some way or another, XML document and grammar similarity evaluation. In this area, most work has focused on estimating similarity between XML documents, which is relevant in several scenarios such as change management and data warehousing [6][7], structural querying [27][35], and document clustering [8][24]. Yet, few efforts have been dedicated to comparing XML grammars, useful for data integration purposes, in particular the integration of DTDs/XML schemas that contain nearly or exactly the same information but are constructed using different structures [11][21]. XML grammar comparison is mainly exploited in data warehousing [26] (mapping data sources to warehouse schemas), message translation [26] as well as XML data maintenance and schema evolution [17].

In this study, we address the XML grammar comparison problem, i.e., the comparison of DTDs [4] and/or XML Schemas [25] based on their most common characteristics. In fact, the effectiveness of grammar matching systems is assessed w.r.t. (with respect to) the amount of manual work required to perform the matching task [10], which depends on: i) the level of simplification in the representation of the grammars, and ii) the combination of various matching techniques [9]. In general, most XML-related grammar matching methods in the literature are developed for generic schemas and are consequently adapted to XML grammars. On one hand, they usually induce certain simplifications to XML grammars in order to perform the match task. In particular, constraints on the existence, repeatability and alternativeness of XML elements (e.g., '?', '+' and '*' in DTDs, or *minoccurs* and *maxoccurs* in XML Schemas) are disregarded. On the other hand, they usually exploit individual matching criterions to identify similarities (evaluating for instance the syntactic similarity between element labels, disregarding semantic meaning) and thus do not capture all element resemblances. Methods that do consider several criterions (semantic similarity, data-type similarity, …) utilize machine learning techniques [11] or basic mathematical formulations (e.g., *max*, *ave*, …) [9] which are usually not adapted to XML-based data in combining the results of different matchers.

Thus, our main goal is to develop an effective XML grammar matching method minimizing the amount of manual work needed to perform the match task. This requires i) considering the characteristics and constraints of the XML grammars being matched (in comparison with existing 'grammar simplifying' approaches), and ii) providing a flexible and extensible framework for combining different matching criterions (in comparison with existing static methods) that is adapted to the semi-structured nature of XML grammars (in comparison with relatively generic approaches).

Hence, the contributions of our paper can be summarized as follows: i) introducing a generic tree representation model, that copes with the expressive power of common XML grammars, without being constrained to a specific grammar language (e.g., DTD[4] or XSD[25]) ii) providing an open framework, founded on the well known concept of tree edit distance, for integrating different matching criterions to evaluate the similarity between XML grammar trees, and iii) developing a prototype to evaluate and validate our approach. Note that to our knowledge, this is the first attempt to exploit tree edit distance in an XML grammar matching context. The remainder of the paper is organized as follows. In Section 2, we depict our XML grammar tree representation model. Section 3 develops our tree edit distance based XML grammar matching framework. Section 4 presents our prototype and experimental results. Section 5 briefly reviews background in XML schema matching. Section 6 concludes the paper.

## 2 XML Grammar Tree Representation

We first provide some basic definitions, prior to developing our XML grammar tree model.

**Def. 1 – Ordered Labeled Tree:** It is a rooted tree in which the nodes are labeled and ordered. We denote by $T[i]$ the $i^{th}$ node of $T$ in preorder traversal, and by $R(T)=T[0]$ its root ●

**Def. 2 - First Level Sub-tree:** Given a tree $T$ with root $p$ of degree $k$, the first level sub-trees, $FL\text{-}SbTree_T = \{T_1, \ldots, T_k\}$ of $T$ are the sub-trees rooted at the children of $p$: $p_1, \ldots, p_k$ ●

**Def. 3 –XML Grammar Constraint Operators:** These are operators utilized to specify constraints on the existence and repeatability of elements/attributes. They consist of two main groups: *cardinality constraints* (*cc*) and *alternativeness constraints* (*ac*).

With cardinality constraint operators, it is possible to specify whether an element is optional ('?' in DTD – equivalent to *minoccurs=0* in XSD) or whether it may occur several times (i.e., '*' in DTD for 0 or more times – equivalent to *minoccurs=0* and *maxoccurs='unbounded'* in XSD – and '+' in DTD for 1 or more times, equivalent to *maxoccurs='unbounded'* in XSD). It is also possible to specify whether an attribute is optional (*Implied*) or mandatory (*Required*). An element/attribute with no constraints is mandatory.

Alternativeness constraint operators specify whether some sub-elements are alternative w.r.t. each other (the *Or* operator, represented by '|' in DTD – *choice* in XSD) or are grouped in a sequence (*And* operator, represented by ',' in DTD – *sequence* in XSD). An additional hybrid operator, *All*, is introduced in XSD [25], which allows its sub-elements to appear in any order, such as all of them appear at once, or not at all ●

Here, we define the notions of *composite alternativeness constraint* and *alternativeness constraint vector*, central to preserving the structural levels of XML grammar elements/attributes.

**Def. 4 - Composite alternativeness constraint:** It is an alternativeness operator, i.e., *And*, *Or* or *All* (cf. Definition 3), to which we associate a cardinality constraint, e.g., ?, *, … (cf. Definition 3), in order to underline the repeatability of groups of elements. Formally, it can be represented as a doublet *cac* = *(sac, cc)* where *sac* is a simple alternativeness constraint and *cc* the corresponding cardinality constraint. For instance, XSD declaration *<All MinOccurs=0><element name='a'><element name='b'></All>* corresponds to an *(All, MinOcc=0)* composite constraint associated to both elements *a* and *b* ●

**Def. 5 - Alternativeness constraint vector:** It is a vector $\overline{ac}$ of simple and/or composite alternativeness constraints, underlining the disposition of a grammar element w.r.t. its siblings and parent element in the grammar. For instance, in DTD declaration *((a | b)?, c)*, vector <*And, (Or, ?)*> would be associated to elements *a* and *b*, while vector <*And*> is associated to *c* ●

With most existing XML grammar matching methods, grammars are represented as simplified XML-like trees or graph structures[1]. We provide here a tree model that i) captures the structural properties of XML grammars, ii) and accurately considers their most common characteristics.

**Def. 6 – XML Grammar Tree:** Formally, we model an XML Grammar as a rooted ordered labeled tree $D = (N_D, E_D, L_D, CC_D, \overline{AC_D}, T_D, g_D)$ where: $N_D$ is the set of nodes in $D$, $E_D \subseteq N_D \times N_D$ is the set of edges (element/attribute containment relation), $L_D$ is the set of labels corresponding to the nodes of $D$ ($L_D = El_D \cup A_D$ such as $El_D$ and $A_D$ designate respectively the labels of the elements and attributes of $D$), $CC_D$ is the set of cardinality constraints associated to the elements and attributes of $D$ (i.e., '?', '*', '+', *MinOccurs*, *MaxOccurs*, '*Required*', '*Implied*' and *null*, cf. Definition 3), $\overline{AC_D}$ is the set of alternativeness constraint vectors associated to the elements and attributes of $D$ (central to preserving the structural levels of XML grammar nodes, cf. Figures 2 and 3), $T_D$ is the set of data-types ($T_D = ET \cup AT$, includes the basic XML element data-types $ET$ = {'#PCDATA', 'ANY', *'String'*, *'Decimal'*, …, *Composite*} and attribute data-types $AT$ = {'CDATA', 'ID', 'IDREF', …}), and $g_D$ is a function $g_D : N_D \rightarrow L_D, CC_D, \overline{AC_D}, T_D$ that associates a label $l \in L_D$, a cardinality constraint $cc \in CC_D$, an alternative constraint vector $\overline{ac} \in \overline{AC_D}$ and a data-type $t \in T_D$ to each node $v \in V_D$ ●

**Def. 7 – XML Grammar Tree Node:** A node $n \in N_D$ of XML grammar tree $D = (N_D, E_D, L_D, CC_D, \overline{AC_D}, T_D, g_D)$ is represented by a quintuplet $n = (l, cc, \overline{ac}, t, Ord)$ where $l \in L_D$, $cc \in CC_D$, $\overline{ac} \in \overline{AC_D}$ and $t \in T_D$ are respectively its label, cardinality constraint, alternativeness constraint vector and node data-type. The additional *Ord* component underlines the DTD node's order w.r.t. its siblings. It is detailed in the following section ●

In XML documents, attributes are usually treated as unordered nodes[2]. In other words, the order, left-to-right, of attribute nodes corresponding to a given element is not relevant (e.g., <*Paper title="…" Genre="…"*> is equivalent to <*Paper Genre="…" Title="…"*>). Consequently, the same is true for attributes in XML grammars. In addition, grammar element nodes connected via the *Or* and *All* operators are unordered [25] (e.g., DTD declaration *Paper (Author | Publisher)* is equivalent to *Paper (Publisher | Author)*). Thus, the XML grammar tree would encompass ordered parts, i.e., elements connected via the *And* operator, and unordered ones, i.e., elements connected via the *Or*/*All* operators as well as attribute nodes.

However, algorithms for computing the edit distance between unordered trees are generally *NP-complete* whereas those for comparing ordered trees are of polynomial complexity [2]. Thus, transforming the XML grammar tree into a fully ordered tree would help amend the time efficiency of the edit distance based match operation. This can be done by representing attribute nodes as children of their encompassing element nodes appearing before all sub-element node siblings, and consequently sorting all node siblings, left-to-right by node label. This can be achieved using efficient sorting algorithms such as *Quicksort*, *MergeSort*, *Bucketsort* [15]. An ordering score *Ord*, will be associated to each node, underlining the reordering magnitude of the node. The *Ord* score will be exploited in the matching framework

[1] Graphs are considered when recursive definitions come to play, which we do not treat in our current study

[2] The Document Object Model, http://www.w3.org/DOM

so as to increase/decrease the plausibility of a given match: nodes closer to their initial positions, i.e., with lesser *Ord* scores, would constitute better match candidates. For $n \in N_D$:

$$n.Ord = \frac{NbHops(InitPosition(n), FinalPosition(n))}{(Number\ of\ siblings\ under\ parent\ of\ n)\ \text{-}\ 1} \quad \in [\text{-}1, 1] \qquad \textbf{(1)}$$

Note that the ordering score *Ord* is not modified when sorting attribute nodes and/or element nodes connected via the *Or*/*All* operators since they are initially unordered.

Consider the XML grammars in Figure 1. Corresponding tree representations are depicted in Figures 2 and 3 (note that elements of the same structural level are represented in a stair-like manner to fit in page margins). Now since XML grammars are represented as special ordered labeled trees (cf. Definition 6), the problem of matching two grammars comes down to matching the corresponding trees.

```
<!ELEMENT Paper ((Publisher | Author+), PaperLength?, url*)>
<!ATTLIST Paper Title CDATA #IMPLIED>
<!ATTLIST Paper  Genre CDATA>
<!ELEMENT Publisher (#PCDATA)>
<!ELEMENT Author (FirstName, MiddleName?, LastName)>
<!ELEMENT Length (#PCDATA)>
<!ELEMENT url (Homepage, Download+)?>
<!ELEMENT Homepage (#PCDATA)>
<!ELEMENT Download (#PCDATA)>
<!ELEMENT FisrtName (#PCDATA)>
<!ELEMENT MiddleName (#PCDATA)>
<!ELEMENT LastName (#PCDATA)>
```

**a.** Paper.dtd

```
<element name= “Publication”>
    <sequence>
        <element name= “Title” type="String"/>
        <element name= “Year” type= “Date”/>
        <choice>
            <element name= “Author” maxoccurs= “unbounded”>
                <sequence>
                    <element name= “First” type= “String”>
                    <element name= “Last” type= “String”>
                </sequence>
            </element>
            <element name= “Editor” maxoccurs= “unbounded”>
                <all>
                    <element name= “Name” type= “String”>
                    <element name= “Country” type= “String”>
                </all>
            </element>
        </choice>
        <element name= “Publisher” type= “String” minoccurs= “0” />
        <element name= “Length” type= “Decimal”/>
        <element name=“Link” type=“String” minoccurs= “0”/>
    </sequence>
</element>
```

**b.** Publication.xsd

**Fig. 1.** Sample XML grammars.

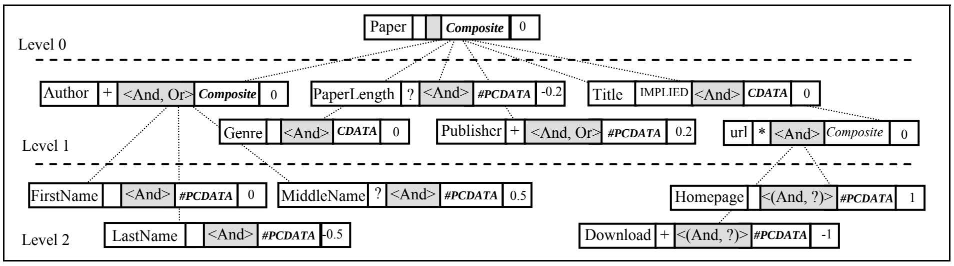


**Fig. 2.** Tree representation *P* of grammar *Paper.dtd* in Figure 1.

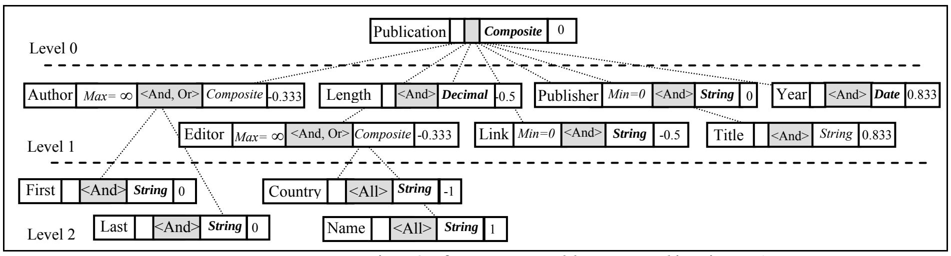


**Fig. 3.** Tree representation *Q* of grammar *Publication.xsd* in Figure 1.

# 3 XML Grammar Matching Framework

Tree edit distance methods have been widely utilized to compare XML documents, represented as Ordered Labeled Trees, and have been proven optimal w.r.t. less accurate structural comparison methods [5]. A great advantage of edit distance is that along the similarity value, a mapping between the nodes in the compared trees is provided in terms of the edit script (i.e., sequence of edit operations transforming one tree into another). This is crucial for schema matching, and would constitute the output of the match operation. Our matching framework consists of four main components: i) the *Edit Distance* component for computing the distance (similarity) between DTD trees, ii) the extensible *Matchers* component, encompassing several matching algorithms, exploited via *Edit Distance* to capture XML grammar node resemblances, iii) the *Mapping Identification* component, interacting with *Edit Distance* to identify the edit script (*ES_Extraction*), and consequently the edit distance mappings, and iv) the *UserFeed* component to consider user mappings and feedback in producing matching results.
The overall architecture of our grammar matching approach is depicted in Figure 4, and will be detailed in the following sections.

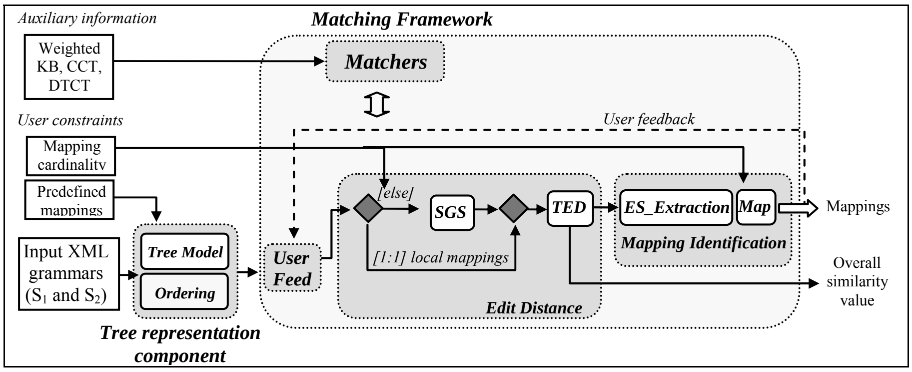


**Fig. 4.** Simplified activity diagram describing our edit distance matching framework.

## 3.1 Edit Distance Component

Several algorithms have been developed to compute a distance, as the sum of a sequence of basic edit operations that can transform one tree structure into another. In the context of XML, the most recent and efficient proposals, e.g., [24] [31], have stressed on the importance of considering XML sub-tree similarities in computing edit distance, as a crucial requirement to obtaining more accurate results. Here, we follow a similar strategy in comparing grammars. We first develop a dedicated method, *SGS*, to compute the *Similarity* between XML *Grammar Sub-trees*, based on the vector space model in information retrieval [20]. XML grammar sub-tree similarities are consequently exploited as tree edit operations' costs in a dynamic programming tree edit distance algorithm (*TED*, cf. system architecture in Figure 4).

### 3.1.1 Similarity between XML Grammar Sub-trees (SGS)

In evaluating XML grammar sub-tree similarity, one should consider all node characteristics (element names, depth and relative order, cardinality constraints, alternativeness constraint vectors, data-types, and ordering scores, cf. Definitions 6 and 7) so as to produce accurate results. To do so, we exploit the vector space model in information retrieval [20].

When comparing two grammar sub-trees $SbT_i$ and $SbT_j$, each would be represented as a vector $\overline{V}$ ($\overline{V_i}$ and $\overline{V_j}$) with weights underlining the similarities between each of their nodes.

**Def. 8 – Sub-tree vector:** For two sub-trees $SbT_i$ and $SbT_j$, vectors $V_i$ and $V_j$ are produced in a space which dimensions represent, each, a distinct indexing unit. An indexing unit stands for a single node $n_r \in SbT_i \cup SbT_j$, such as $1 < r < M$ where $M$ is the number of distinct nodes in both $SbT_i$ and $SbT_j$. The coordinate of a given sub-tree vector $V_i$ on dimension $n_r$ is noted $w_{Vi}(n_r)$ and stands for the weight of $n_r$ in sub-tree $SbT_i$ ●

**Def. 9 – Node weight:** The weight of a node label $n_r$ in vector $\overline{V_i}$ (representing sub-tree $SbT_i$) is composed of two factors, a node/ vector similarity factor $Sim(n_r, \overline{V_i})$ and a depth factor $D\text{-}factor(n_r)$ such as $w_{Vi}(n_r) = Sim(n_r, \overline{V_i}) \times D\text{-}factor(n_r) \in [0, 1]$.

– $Sim(n_r, \overline{V_i})$ quantifies the similarity between node $n_r$ and sub-tree vector $\overline{V_i}$. It is computed as the maximum similarity between $n_r$ and all nodes of sub-tree $SbT_i$ considering the various XML grammar node characteristics (Definition 7). Formally, $Sim(n_r, \vec{V}_i) = \underset{n \in \vec{V}i}{Max}(Sim(n_r, n)) \in [0, 1]$.

– $D\text{-}factor(n_r)$ considers the hierarchical depth of node $v_r$ in assessing its weight w.r.t. sub-tree vector $\overline{V_i}$. Note that node depth is not only a structural characteristic in XML, but is also of semantic relevance. It follows the intuition that information placed near the root node of an XML document is more important than information further down in the hierarchy [1][35]. Thus, higher nodes should have a greater semantic influence.

$$D\text{-}factor(n.l) = \frac{1}{1+n.d} \in [0,1] \quad \text{where } n.d \text{ designates the depth of node } n \text{ ●} \qquad \textbf{(2)}$$

**Def. 10 – Similarity between XML grammar nodes:** It quantities the similarity between two XML grammar nodes, considering their various characteristics. Given two nodes $n$ and $m$:

$$\begin{aligned} \mathrm{Sim}(n, m, Aux) = \; & w_{\mathrm{Label}} \times \mathrm{Sim}_{\mathrm{Label}}(n.l, m.l, KB) + \\ & w_{\mathrm{CConstraint}} \times \mathrm{Sim}_{\mathrm{CConstraint}}(n.cc, m.cc, CCT) + \\ & w_{\mathrm{AConstraint}} \times \mathrm{Sim}_{\mathrm{AConstraint}}(n, m.\overline{ac}, CCT) + \\ & w_{\mathrm{Data\text{-}Type}} \times \mathrm{Sim}_{\mathrm{Data\text{-}Type}}(n.t, m.t, DTCT) + \\ & w_{\mathrm{OrdScore}} \times \mathrm{Sim}_{\mathrm{OrdScore}}(n.Ord, m.Ord) \end{aligned} \qquad \textbf{(3)}$$

where $w_{Label} + w_{CConstraint} + w_{AConstraint} + w_{Data\text{-}Type}\ w_{OrdScore} = 1$ and $(w_{Label}, w_{Constraint}, w_{AConstraint}, w_{Data\text{-}Type}, w_{OrdScore}) \geq 0$, having $Sim_{Label}$, $Sim_{CConstraints}$, $Sim_{AConstraints}$, $Sim_{Data\text{-}Types}$ and $Sim_{OrdScore}$ the similarity scores between corresponding node labels, cardinality constraints, alternative constraint vectors, data-types and ordering scores. Each of those similarity scores is to be computed by the corresponding matcher (Section 4.4). $Aux=\{KB, CCT, DTCT\}$ designates the auxiliary data sources required by the matchers to compute node similarity: $KB$ (knowledge base), $CCT$ (constraint compatibility table) and $DTCT$ (data-type compatibility table [32]) ●

Following *Formula (3)*, different weights are assigned to the different node components, reflecting the impact of each of the element characteristics in identifying the mappings. In fact, several methods for combining matcher results have been investigated in [9], among which the *maximum*, *minimum*, *average* and *weighted sum* functions. Here, we exploit the latter as it provides more flexibility, adapting the process w.r.t. the user's perception of similarity.

Having transformed XML grammar sub-trees into weighted vectors, the similarity between two sub-trees is evaluated using a measure of similarity between vectors such as the *inner product,* the *cosine measure*, the *Jaccard measure*, etc. Here, we adopt the *cosine measure* widely exploited in information retrieval [20].

Algorithm *SGS* for computing XML grammar sub-tree similarity consists in building and comparing sub-tree vectors as described above. *SGS* is consequently exploited to identify the similarities between each and every pair of sub-trees $(SbT_i, SbT_j)$ in the two trees $T_1$ and $T_2$ being compared, as well as their similarities with the whole trees $T_1$ and $T_2$ respectively. Tree operations costs would hence vary as follows [31]:

$$\text{Cost}_{\text{InsTree/DelTree}}(\text{Sb}_i)=\sum_{\text{All nodes } n \text{ of SbT}_i} \text{Cost}_{\text{Ins/Del}}(n) \times \frac{1}{1+\text{Max}(SGS(\text{SbT}_i, \text{SbT}_j, \text{Aux}))} \quad \textbf{(4)}$$

#### 3.1.2 Tree Edit Distance (*TED*)

The tree edit distance algorithm *TED*, utilized in our study, is an adaptation of Nierman and Jagadish's main edit distance process [24]. In addition to tree insertion/deletion operations' costs which vary w.r.t. DTD sub-tree similarities (using *SGS*), *TED* (Figure 7) considers XML grammar node similarities in computing update operations costs (cf. Figure 7, line 6). Using the update operation, *TED* compares the roots of sub-trees considered in the recursive process (at startup, these would correspond to the grammar tree roots). The cost of the update operation would vary as:

$$\text{Cost}_{\text{Upd}}(n, m, Aux) = 1 - Sim(n, m, Aux) \quad \in [0, 1] \quad \textbf{(5)}$$

where *Sim(n, m, Aux)* underlines the similarity between tree nodes *n* and *m*, *Aux* standing for the auxiliary information required by the various matchers to assess XML grammar node similarity (knowledge base *KB*, constraint table *CCT* and data-type compatibility table *DTCT*).

Hence, following *Formula (5)*, the more initial and replacing nodes are similar, the lesser should be the update operation cost, which would transitively yield a lesser minimum cost edit script (higher similarity value). In short, the *TED* algorithm goes through all sub-trees of the grammar trees being compared. It exploits sub-tree insertion/deletion costs (via *SGS*) and update operations costs (cf. *Formula (5)*), which reflect the similarities between each sub-tree in the source/destination trees being compared, to produce the overall distance value.

### 3.2 Element Matchers

As mentioned previously, we make use of dedicated matchers to evaluate the similarities between XML grammar tree node labels, constraints, data-types, and ordering scores, their results being integrated in the tree edit distance framework to produce relevant grammar element mappings. Recall that the use of independent matchers provides flexibility in performing the match operation since it is possible to select or disregard different matchers (i.e., different match criterions) following the task at hand. Table 1 presents the matchers we included in our XML grammar matching approach so far, along with the different kinds of auxiliary information they exploit. More details are provided in the technical report [32].

**Tab. 1.** XML grammar element matchers.

| Matcher | | | Type | Target | Auxiliary Information |
|---|---|---|---|---|---|
| Label | | | Composite | Labels | Knowledge base |
| | *Syntactic* | | Composite | Labels | --- |
| | | *String- ED* | Simple | Labels | --- |
| | | *N-Gram* | Simple | Labels | --- |
| | | | | | |
| | *Semantic* | | Composite | Labels | Knowledge base |
| | | *Lin* | Simple | Element labels | Knowledge base |
| | | *WuPalmer* | Simple | Element labels | Knowledge base |
| | | | | | |
| *Cardinality Constraint* | | | Simple | Cardinality constraints | Constraint compatibility table |
| *Data-Type* | | | Simple | Data-Types | Data-type compatibility table |
| *OrdScore* | | | Simple | Ordering scores | --- |
| *Alternativeness Constraint* | | | Hybrid | Alternativeness constraint vectors | Constraint compatibility table |

Similarly to computing XML grammar node similarity (cf. *Formula (3)*), we exploit the *weighted sum* function in combining the results of simple matchers, since it enables the user to choose the weight of each matcher in accordance with her notion of similarity. For each of the composite matchers *CM* and its component ones $M_{i=1..n}$, similarity is evaluated as follows:

$$\mathrm{Sim}_{CM} = \sum_{i=1\ldots n} w_i \times \mathrm{Sim}_{M_i} \quad \in [0, 1] \qquad \textbf{(6)}$$

$$\text{where } \sum_{i=1\ldots n} w_i = 1,\ (w_{i=1\ldots n}) \geq 0 \text{ and } (Sim_{M\,i=1\ldots n}) \in [0, 1]$$

### 3.3 Edit Script Extraction and Mapping Identification

Identifying the similarity between two XML grammars is useful in applications such as grammar clustering [16], and can be exploited as a pre-processing schema integration phase [26]. Yet, the grammar matching operation itself requires identifying element correspondences, where edit distance mappings come to play. The *Edit Distance* component returns the edit distance between XML grammar trees, i.e., similarity (*Sim=1/(1+Dist)*). Identifying mappings requires a post-processing of the edit distance result. This amounts to edit script extraction.

#### 3.3.1 Edit Script Extraction

In fact, edit distance computations are generally undertaken in a dynamic manner, combining and comparing the costs of various edit operations to identify the minimum distance (maximum similarity). Nonetheless, to identify the minimum cost edit script itself, one has to process the intermediary edit distance computations, going throw the edit distance matrixes, (which we identify as *{Dist[][]}*) tracing the edit script operations costs. Our algorithm for identifying the minimum cost tree edit script is provided in Figure 6. It considers as input the grammar trees being compared as well as the related edit distance matrixes computed via tree edit distance component. It outputs the corresponding edit script (simplified tree operation syntaxes are shown in Figure 6 for ease of algorithm presentation). As it traverses the edit distance matrixes, the algorithm identifies corresponding tree insertion/deletion and node update operations, gradually building the edit script. Thus, XML grammar tree mappings are deduced from the edit script, graphically depicting which edit operations apply to nodes in the grammar trees.

#### 3.3.2 Mapping Identification

As stated previously, the schema matching problem comes down to identifying mappings between the elements of two schemas $S_1$ and $S_2$. Edit distance mappings are deduced from the minimum cost edit script between $S_1$ and $S_2$, graphically depicting which edit operations apply to which nodes in the grammar trees. In other words, they depend on the edit distance operations that are allowed and how they are used. In our approach, we make use of five edit operations: *insert node*, *delete node*, *update node*, *insert tree* and *delete tree* [31]. Hence, the mapping between two XML grammar trees $S_1$ and $S_2$ is constructed:

- Simple *1:1* mapping edges are introduced to connect:
  - Nodes that initially match. Two nodes of $S_1$ and $S_2$ initially match if they are identical (nodes with identical labels, constraints, data-types, order and depth).
  - Nodes related by the update operation.
- Simple *1:1* and complex *1:n*, *n:1* or *n:n* mapping edges connect:
  - Sub-trees of $S_1$, affected by *tree deletion*, to similar sub-trees in $S_2$. Such edges are identified when computing the similarity between sub-trees of $S_1$ and $S_2$. No edges are introduced if the sub-tree being deleted from $S_1$ has no similarities in $S_2$.
  - Sub-trees of $S_2$ that are affected by the *tree insertion* operation, to similar sub-trees in $S_1$. No edges are introduced if the inserted sub-tree has no similarities in $S_1$.

*Node insertion/deletion* operations are treated as *tree insertion/deletion* ones. Note that *node insertions/deletions* are utilized to compute the costs of *tree insertion/deletion* operations and are not directly employed in the main edit distance algorithm.

Figure 5 shows the mapping results corresponding to the edit distance computations between two XML grammar trees *D* and *T* extracted from those in Figures 2 and 3. Note that in this figure, we only show node labels for the sake of presentation. The edit script transforming tree *D* into *T*, $ES(D, T) = Upd(D[2], T[2]), Upd(D[3], T[3]), DelTree(D[4]), InsTree(T_2)$.

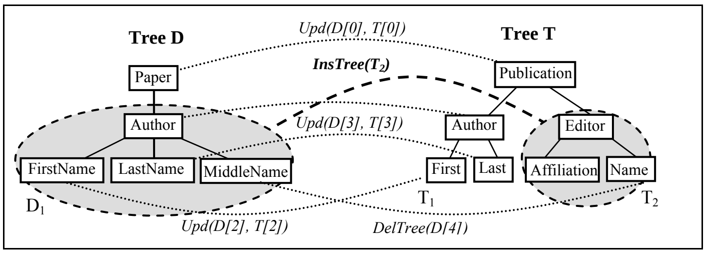


**Fig. 5.** XML grammar tree mappings.

Note that each of the nodes *D[0]*, *D[1]*, *D[2]*, *D[3]* and *T[0]*, *B[1]*, *B[2]*, *B[3]* in grammar trees *D* and *T* participates in an individual *1:1* mapping. In addition, *D[1], D[2], D[3], D[4]*, and *T[5]*, *T[6]*, *T[7]* participate in an *n:n* mapping. In short, our approach produces all kinds of mapping cardinalities, ranging from *1:1* to *n:n*. Nonetheless, the nature of a mapping is often dependent on user requirements or the requirements of the module exploiting the mapping results. In general, existing matching approaches tend to focus on *1:1* mappings [11]. Such mappings are usually easier to comprehend, evaluate and manipulate by users and automated processes alike. Nevertheless, complex *1:n*, *n:1* and *n:n* mappings are required in certain application domains, mainly in automatic document transformation [2]. Thus, we provide the user with a flexible schema matching framework able to produce either:

- *1:1* mappings (easier to assess, and especially useful for query discovery [23]),
- All kinds of mappings (no cardinality restrictions).

Restricting mapping cardinalities to *1:1* cardinality means disregarding all kinds of sub-tree similarities and repetitions when comparing the grammar trees. In other words, we disable algorithm *SGS* and only make use of the main edit distance process *TED* in our edit distance component (cf. Figure 4). In this case, tree insertion/deletion mapping edges (which induce complex *1:n*, *n:1* and *n:n* mappings) are eliminated, and we are left with *1:1* mappings.

Process *Map* (omitted here due to its intuitiveness) coupled with *ES_Extraction* (cf. Figure 4) is dedicated to producing grammar mappings, in the form $(M, S_1, S_2)$ where $M \subseteq N_{S1} \times N_{S2}$. It simply generates mappings following the rules above, and associates related mapping scores.

### 3.3.3 Mapping Scores

Most schema matching approaches associate scores to the identified mappings. These scores underline values, usually in the *[0, 1]* interval, that reflect the plausibility of the corresponding matches (*0* for strong dissimilarity, *1* for strong similarity, and values in between). With respect to edit distance, mapping scores denote, in a roundabout way, the costs of the edit operations inducing the corresponding mappings:

- Mappings linking identical nodes are assigned a maximum similarity, $MapScore = 1$.
- Mappings underlining the update operation between two nodes are assigned scores as follows: $MapScore = 1 - Cost_{Upd}(n, m, Aux) \in [0, 1]$, *1* being the maximum update operation cost (*Formula (5)*). In other words, the mapping score designates node similarity, $Sim(n, m, Aux) \in [0, 1]$.
- Following the same logic, mappings corresponding to tree insertion/deletion operations are assigned scores as follows:

  $$MapScore = \frac{\sum_{\text{All nodes } n \text{ of S}} Cost_{Ins/Del}(n) - Cost_{InsTree/DelTree}(S)}{\sum_{\text{All nodes } n \text{ of S}} Cost_{Ins/Del}(n)} \in [0, 1]$$, having $\sum_{\text{All nodes } x \text{ of S}} Cost_{Ins/Del}(x)$ the maximum tree insertion/deletion operation cost for the sub-tree at hand. Hence, the mapping scores will follow the similarities between inserted/delete grammar sub-trees.

Table 2 shows the mappings generated in our running example, as well as corresponding mapping scores (computational details are omitted for simplicity). In addition to the *Edit Distance* and *Mapping Identification* components, our matching framework encompasses a *UserFeed* component, enabling users to manually match a few hard-to-match elements.

**Tab. 2.** Matching nodes of grammar trees *D* and *T* (Figure 5).

| Match cardinality | Nodes of tree D | Nodes of tree T | Mapping Scores |
|---|---|---|---|
| 1:1 | D[0] (*l*= '*Paper*') | T[0] (*l* ='*Publication*') | 0.1667 |
| | D[1] (*l*= '*Author*') | T[1] (*l* ='*Author*') | 1 |
| | D[2] (*l*= '*FirstName*') | T[2] (*l* ='*First*') | 0.5556 |
| | D[3] (*l*= '*LastName*') | T[3] (*l* ='*Last*') | 0.5714 |
| | D[4] (l = '*MiddleName*') | T[6] (T[6].l = '*Name*') | 0.4628 |
| n:n | D[1],D[2],D[3],D[4] (sub-tree $D_1$) | T[5],T[6],T[7] (sub-tree $T_2$) | 0.4266 |

### 3.4 User Input Constraints and User Feedback

Considering user input constraints and feedback in grammar matching could improve matching accuracy. User mappings are particularly useful in matching ambiguous schema elements [11].

Consider for instance elements of labels '*url*' and '*Link*' in grammars *Paper.dtd* and *Publication.xsd* of Figure 1 respectively. These elements have neither syntactically nor semantically similar labels (that is if using a generic WordNet-based taxonomy as a reference knowledge base). In addition, element '*url*' in *Paper.dtd* encompasses two sub-elements, of labels '*Homepage*' and '*Download*', both of them identifying links. In such situations, the system is left with a set of confusing matching possibilities ('*url*'↔'*Link*', '*Homepage*'↔'*Link*' or '*Download*'↔'*Link*', which is where user constraints come to play.

In our approach, we enable the user to explicitly specify matching elements as input to the match operation, i.e., input user constraints. Likewise, after the execution of the match operation, if the user is still not happy with the produced matches, she can provide new ones i.e., user feedback, then run the edit distance process once again to output new mappings. In essence, we consider user input constraints and feedback in our grammar matching framework by updating input grammar trees following the constraints at hand, and consequently comparing the updated trees. Thus, we define the *UserFeed* grammar transformation operation as follows:

**Def. 11 – UserFeed:** It is an operation that transforms an XML grammar tree *A* into *A'*, such as in the destination tree *A'*, nodes corresponding to predefined matches are eliminated, along with their corresponding sub-trees.

Formally, *UserFeed(A, (preM, A, B)) = A'* where:

- *A* and *B* are the grammar trees being compared by the system.
- *(preM, A, B)* is the set of predefined user matches from *A* to *B* such as $preM \subseteq V_A - \{R(A)\} \times V_B - \{R(B)\}$, where $V_A$ and $V_B$ designate respectively the sets of nodes of trees *A* and *B*, *R(A)* and *R(B)* underlining the corresponding grammar tree roots.
- *A'* is the transformed tree, $A' = A - \{\text{the set of sub-trees } A_i \,/\, R(A_i) \in (preM, A, B)\}$

Thus, w.r.t. user constraints, our *Edit Distance* component will be comparing the transformed grammar trees, where nodes corresponding to predefined matches are eliminated, along with their corresponding sub-trees (structural matching being sibling and ancestor preserving [28]). Note that tree roots (*R(A)* and *R(B)*) are not included in the predefined user matches since their inclusion would indicate that the whole grammar trees actually match, thus eliminating the need to perform the matching task in the first place. Disregarding predefined matches in the edit distance process would i) eliminate the possibility of *automatically* modifying these matches and ii) lessen the risk of attaining confusing matches by reducing the number of match candidates. The *UserFeed* process is shown in Figure 8. User mappings are thus added to those produced by the system: *(M, A, B) = (SystemM, A, B)* ∪ *(preM, A, B)*.

```
Algorithm ES_Extraction()
Input: Trees A and B, {Dist[][]} the set of distance matrixes
       computed by eTED among which the starting matrix
       Dist[][]_A,B
Output: Edit script ES transforming A to B

Begin
    i = Degree(A)          // |FL-SbTree_A|
    j = Degree(B)          // |FL-SbTree_B|

    While (i>0 and j>0)
    {
        If (Dist[i, j]_A,B = Dist[i-1, j]_A,B + Cost_DelTree(A_i)
        {
            ES = ES + DelTree(A_i)
            i = i-1
        }
        Else if (Dist[i,j]_A,B=Dist[i,j-1]_A,B + Cost_InsTree(B_j))
        {
            ES = ES + InsTree(B_j)
            j = j-1
        }
        Else
        {
            If (A_i ≠ B_j)        //Recursive formulation
            {
                ES_Extraction_Core(A_i, B_j, Dist[][]_Ai,Bj )
            }
            i=i-1
            j=j-1
            }
     }

     While (i>0)        // identifying remaining deletions
     {
            ES = ES + DelTree(A_i)
            i = i-1
     }

     While (j>0)        // identifying remaining insertions
     {
            ES = ES + InsTree(B_j)
            j = j-1
     }

     If (i = 0 and j = 0 and R(A_i) ≠ R(B_j))
     {
            ES = ES + Upd(R(A_i), R(B_j))
     }

Reorder(ES)        // Reversing edit operations' order
Return ES          // Edit script transforming tree A to B

End
```

**Fig. 6.** Edit script extraction algorithm.

```
Algorithm EditDistance()
Input: Trees A and B, operations costs
       Cost_DelTree/Cost_InsTree for all sub-trees
       in A/B, Aux = {KB, CCT, DCT}
Output: Edit distance between A and B

Begin

M = Degree(A)          // |FL-SbTree_A|
N = Degree(B)          // |FL-SbTree_B|

Dist [][] = new [0...M][0…N]
Dist[0][0] = Cost_Upd(λ(A), λ(B), Aux)

For (i = 1 ; i ≤ M ; i++)
{ Dist[i][0] = Dist[i-1][0] + Cost_DelTree(A_i) }

For (j = 1 ; j ≤ N ; j++)
{ Dist[0][j] = Dist[0][j-1] + Cost_InsTree(B_j) }

For (i = 1 ; i ≤ M ; i++)
{
     For (j = 1 ; j ≤ N ; j++)
      {
       Dist[i][j] = min{
              Dist[i-1][j-1] + EditDistance(A_i, B_j),
              Dist[i-1][j] + Cost_DelTree(A_i),
              Dist[i][j-1] + Cost_InsTree(B_j)    }
      }
}

Return Dist[M][N]

End
```

**Fig. 7.** Tree edit distance algorithm.

```
Algorithm UserFeed()
Input: Grammar tree A, user predefined matches
       (preM, A, B)
Output: Transformed grammar tree A'

Begin

A' = A
M = Degree(A')        // |FL-SbTree_A|

For (i = 1 ; i ≤ M ; i++)
  {
        If(R(A_i) ∈ (preM, A, B))
        { A' = A' - A_i' }
        Else
        { A_i' = UserFeed(A_i , (preM, A, B)) }
  }

End
```

**Fig. 8.** User feed transformation algorithm.

## 4 Experimental Evaluation

We have implemented our XML grammar matching framework in the experimental XS3 prototype (*XML Structural and Semantic Similarity*)[1]. Aimed originally at comparing XML-based documents, XS3 has been extended to XML grammar matching and comparison.

### 4.1 Matching Experiments

To our knowledge, a common benchmark with *gold standard* matchings for evaluating the quality of XML grammar matching methods does not exist to date. Hence, we conducted our experiments using a select collection of real and synthetic XML grammars (including those

[1] Available at http://www.u-bourgogne.fr/Dbconf/XS3

exploited in our running example). Real DTDs and XML Schemas were acquired from various online sources[1]. Consequently, we ran our matching approach and compared the generated matches to the manually defined ones. *Precision (PR)*, *Recall (R), F-value* and *Overall* results are shown in Figure 9. Note that the *Overall* metric, introduced in [21], quantifies the amount of user effort needed to perform the match task, i.e., effort needed to transform the match result produced by the system to the user intended one:

$$Overall = R \times (2 - \frac{1}{PR}) \text{ having } PR \neq 0 \qquad (7)$$

In all tests, all basic matchers were considered with equal weights ($w_{Label} = w_{Cardinality} = w_{DataType} = w_{Alternativeness} = w_{Ord} = 0.2$ whereas $w_{String\text{-}ED} = w_{N\text{-}Gram} = w_{Lin} = w_{WuPalmer} = 0.5$). Note that in this study, we do not address the issue of tuning matcher weights. This would require a thorough analysis of the relative effect of each individual matcher and criterion on matching quality (similarly to [9]), which we report to a dedicated study. Extracts of WordNet were adopted as reference knowledge bases, and default *DTCT* and *CCT* tables were exploited. Details concerning all experiments are provided in the technical report [32].

### 4.1.1 Evaluation of our Running Example

When matching grammars *Paper.dtd* and *Publication.xsd* (cf. Figures 3, 4), the system identified *6* correct mappings, disregarded *2*, and generated *2* incorrect ones (Table 3). The mappings which are missed by the system (*'PaperLenght'-'Length'* and *'Download-Link'*) are in fact replaced by others (e.g., *'Genre'-Length'* and *'PaperLength'-'Link'*) which seem more structurally plausible. Recall that the topological structure of grammar nodes (i.e., sibling ordering and ancestor/descendent relations) is crucial in determining the mappings, following our approach, since we focus on semi-structured and structured data (which is not necessarily verified with user mappings). Despite some inconsistencies in the matching results, *PR*, *R, F-Value* and particularly *Overall* show that more than half of the mappings generated by the system are correct, which is obviously easier than manually performing the match task.

**Tab. 3.** Matching *Paper.dtd* and *Publication.xsd* of Figure 5.

| **Manual Mappings** | | **System Mappings** | | |
|---|---|---|---|---|
| paper.dtd | publication.dtd | paper.dtd | publication.dtd | Scores |
| Paper | Publication | Paper | Publication | 0.8849 |
| Author | Author | Author | Author | 0.9667 |
| FirstName | First | FirstName | First | 0.8378 |
| LastName | Last | LastName | Last | 0.7886 |
| PaperLength | Length | | | |
| Publisher | Publisher | Publisher | Publisher | 0.84 |
| Title | Title | Title | Title | 0.8367 |
| Download | Link | | | |
| | | PaperLength | Link | 0.7841 |
| | | Genre | Length | 0.7414 |

*PR= 0.75   R = 0.75   F-Value=0.75   Overall = 0.5*

### 4.1.2 Evaluation on Real World and Synthetic Grammars

Hereunder, we present the results of 18 match tasks, each matching two different grammars (including those of our running example). In 12 of the 18 match tasks, the system effectively identified most user mappings, while disregarding some, and generating a few false ones. In task # 2, the system achieved *PR=R=Overall=1* due to the high resemblance between the grammars being matched (*bib.dtd* and *bookstore.dtd* [2]). Negative *Overall* was obtained in 6 of the 18 matching operations. This is due to the structural heterogeneity between the grammars being matched, the system generating mappings which are structurally coherent (respecting sibling order and ancestor/descendent relations) but which do not correspond to actual user

[1] http://www.acm.org/sigmod/xml, http://www.cs.wisc.edu/niagara/, http://www.BizTalk.org, …
[2] Available at http://www.cs.wisc.edu/niagara/ and http://www.xmlfiles.com respectively.

mappings (user mappings do not necessarily verifying structural integrity). Note that in cases where *Overall* is negative, *PR* is lesser than *0.5*, indicating that it would be easier for the user to carry out the matching by hand, instead of correcting the system generated ones. In short, our system seems efficient in identifying XML grammar mappings since it yielded positive *Overall* results for more than ⅔ of the experiments, while maintaining relatively high *PR* and *R* values.

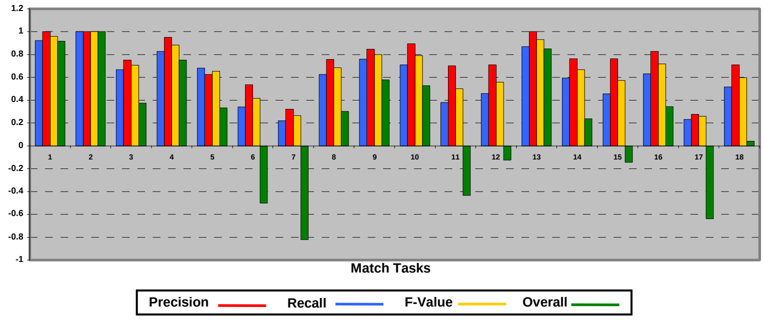


**Fig. 9.** PR, R and Overall results

### 4.1.3 Improvements via User Feedback

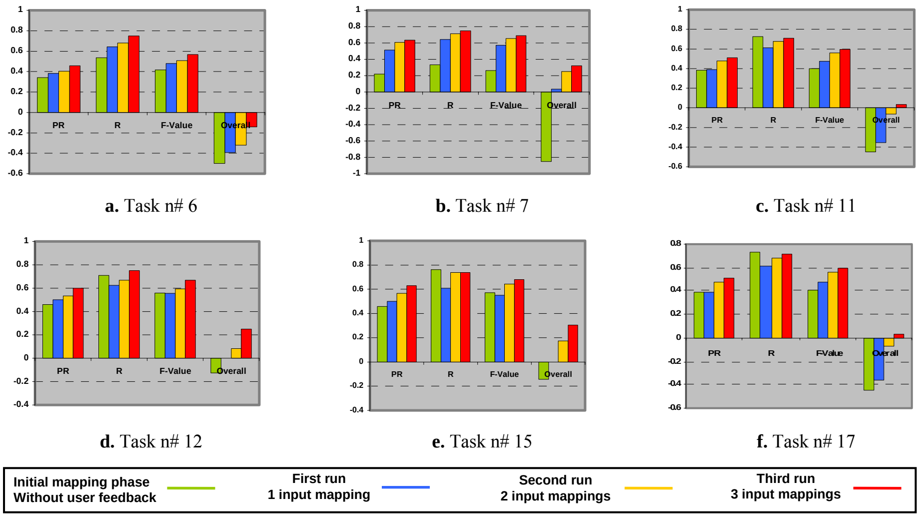


**Fig. 10.** Comparing *PR*, *R*, *F-value* and *Overall* results for matching tasks n# 6, 7, 11, 12, 15 and 17 to evaluate the effectiveness of our approach in incorporating user feedback.

In addition to testing the raw capabilities of the system, we conducted experiments to evaluate the effect of user feedback on matching quality. We considered the six matching tasks where negative *Overall* was achieved in the initial matching phase (tasks n# *6*, *7*, *11*, *12*, *15* and *17*). For each task, we carried out three runs, providing an extra user input mapping at each run. Results in Figure 10 show that user feedback positively affects matching accuracy, amending *Precision*, *Recall*, *F-Value* and *Overall* levels w.r.t. the number of user input mappings: the more input mappings are provided, the lesser the mapping ambiguities, the better the mapping quality. With respect to *Overall* in particular, the system obtained positive values with three out of six tasks (tasks n# *7*, *12* and *15*), right after the first run. In these tasks, manually resolving one mapping has eliminated enough ambiguity for the system to produce more than half of the correct mappings. The *Overall* levels of task n# 6 were gradually amended by feedback, but obviously require more user mappings to cross the zero barrier (i.e., *PR* > *0.5*).

### 4.2 Comparative Study

In order to further evaluate our method, we conducted a comparative study to assess its effectiveness w.r.t. existing XML grammar matching methods. In short, our method is i) dedicated to XML grammars, ii) considers all basic XML grammars characteristics, iii) while being extensible to different matchers (which are crucial to minimizing user effort in undertaking the match task). However, existing methods are either i) too generic (not adapted to the structured nature of XML [9][11]), ii) too restrictive (simplifying grammar constraints) or ii) too specific (not flexible nor extensible to additional matching criterions [16][33]). Table 4 sums up the differences between our method and its alternatives.

**Tab 4.** Comparing our method to alternative solutions.

| Approach | Considers cardinality constraints | Considers alternativeness constraints | Considers data-types | Extensible to several Matchers | Flexible w.r.t. mapping cardinalities | Dedicated to XML grammars |
|---|---|---|---|---|---|---|
| Madhavan, 01 [18] | × | × | ✓ | × | × (1:1, 1:n) | × |
| Melnik et al. 02 [21] | × | × | ✓ | × | × (1:1) | × |
| Doan *et al.*, 01 [11] | × | × | × | ✓ | × (1:1) | ✓ (DTD) |
| Jeong *et al*, 07 [14] | × | × | × | ✓ | × (undefined) | ✓ (XSD) |
| Su *et al.* 01 [30] | × | × | ✓ | × | × (1:1) | ✓ (DTD) |
| Do and Rahm,02 [9] | × | × | ✓ | ✓ | × (1:1) | × |
| Lee *et al.*, 02 [16] | ✓ | × | × | × | × (1:1) | ✓ (DTD) |
| Yi *et al.*, 04 [33] | × | ✓ (restrictive) | ✓ (restrictive) | × | × (1:1) | ✓ (XSD) |
| **Our Approach** | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |

**Tab. 5.** Average *PR*, *R*, *F-Value* and *Overall* values.

| | | *PR* | *R* | *F-Value* | *N# of negative Overalls* |
|---|---|---|---|---|---|
| **Our Approach** | Without user feedback | **0.6096** | **0.7488** | **0.6667** | **6** |
| | User feedback: 1 input mapping | **0.6517** | **0.7703** | **0.7027** | **2** |
| | User feedback: 2 input mappings | **0.6700** | **0.7909** | **0.7221** | **2** |
| | User feedback: 3 input mappings | **0.6842** | **0.8048** | **0.7367** | **1** |
| *COMA* | | 0.7205 | 0.5101 | 0.5790 | 2 |
| *XClust* | | 0.5047 | 0.554 | 0.5251 | 7 |
| *Relaxation Labelling* | | 0.4629 | 0.3030 | 0.3224 | 11 |

Results, in Table 5, show that our method yields average *Precision* levels higher than those achieved by its predecessors, to the exception of *COMA*. That is due to the generic nature of *COMA* (which was not originally designed for XML) considering mappings which are not necessarily structurally coherent (i.e., they do not verify sibling order nor ancestor/descendent relations), and which happen to correspond to user mappings. Such mappings are replaced by structurally valid ones using our approach, but which might not be correct w.r.t. the user (similarly to the *falsely* detected mappings in Table 3, which our system replaced by structurally correct ones). On the other hand, our method consistently maintains *Recall* levels higher than those of all its alternatives. In cases where higher/lower *Precision*/*Recall* levels are obtained simultaneously, the *F-Value* measure is used to assess the general loss and gain in average *Precision*/*Recall*, and thus evaluate result quality. With respect to all 18 matching tests, our method yields higher average *F-Values* in comparison with *COMA, XClust* and *Relaxation Labelling*. Note that the *Overall* measure is non-linear in terms of *Precision* and *Recall*. Thus, its averaging is meaningless here. Hence, we exploit *Overall* by assessing the number of matching tasks with negative *Overall* values (i.e., where more than half the produced mappings are incorrect). Results show that our method, in its initial (pre-feedback) matching phase, produces 6 negatives (negative *Overall* with 6 matching tasks), 2 negatives after the first feedback run (with 1 user mapping for each of the 6 tasks), and only 1 negative after the third run. In comparison, *COMA* produced negative *Overall* values with 2 matching tasks, *XClust* produced 7, and *RL* produced 11 negatives respectively.

## 5 Background and Related Works

The effectiveness of schema matching systems is assessed w.r.t. the amount of manual work required to perform the matching task [10], which depends: i) the level of simplification in the representation of the schema, and ii) the combination of various matching techniques [9].

On one hand, most approaches in the literature, [2][16][18][21][29][30][33] require various simplifications in the grammars being matched, thus inducing adapted schema representations upon which the matching processes are executed. In this context, *XClust* [16] and *Relaxation Labeling* [33] seem more sophisticated than previous matching systems in comparing XML grammars. They induce the least simplifications to the grammars being compared. *XClust* only disregarding the *Or* operator, whereas *Relaxation Relabeling* considers most XML Schema-related repeatability and alternativeness constraints but with restrictive declarations (operator concatenations such as in *root(a, b, (c|d))* are not allowed, only single declarations such as *root(a, b, c)* or *root(a | b | c)*).

On the other hand, most methods in the literature are *hybrid*, in that various matching criterions (e.g., the linguistic and structural aspects of XML grammars) are simultaneously assessed in a specific manner within a single algorithm. In contrast, few approaches follow the alternative *composite* matching logic, i.e., combining the results of several independently executed matching algorithms, thus providing more flexibility in performing the matching as is it possible to select, add or remove different matching algorithms following the match task at hand. *LSD* [11] and *NNPLS* [14] and based on supervised learning techniques, and encompass each a training phase which could require substantial manual effort prior to launching the matching process. However, *Coma* [9] underlines a more generic framework for schema matching, providing various mathematical formulations (*max*, *min*, *ave,* …) to combine matching results, and thus is not specifically adapted to XML grammars.

## 6 Conclusion

In this paper, we proposed a framework for XML grammar matching and comparison, based on the concept of tree edit distance. To our knowledge, this is the first attempt to exploit tree edit distance in an XML grammar matching context. Our approach aims at minimizing the amount of manual work needed to perform the match task by i) considering all basic XML grammar characteristics via a dedicated tree model, and ii) combining different matching criterions in a flexible and adapted way to deal with XML. In addition, our method is flexible in producing either *1:1* or all kinds of mapping cardinalities (*1:1*, *1:n*, *n:1* and *n:n*), following user preferences and the application at hand. It also considers user input constraints and user feedback in adjusting mapping results. We have implemented the approach and conducted various tests to validate its efficiency, in comparison with alternative methods.

As continuing work, we are currently investigating the extension of our method to deal with user derived data-types. These are allowed in the XSD language [25] via dedicated data-type restriction and extension operators (which do not exist in DTDs). In this context, dedicated knowledge bases and user defined semantics would have to be considered to assess the relatedness between the various data-types [12]. We also plan to investigate XML grammars with recursive declarations. Here, it would be interesting to extend our XML grammar tree model to a more general graph model (e.g. topic maps), and try to adapt our tree edit distance framework accordingly. We also plan to study the effect of different matchers and criterions on matching quality, proposing (if possible) weighting schemes that could help the user tune her input parameters to obtain optimal results.


## Acknowledgements

We are grateful to Phil Bernstein and Sabine Maßmann for providing us with their test schemas in order to conduct our matching experiments.

## References


[1] Bertino E., Guerrini G., Mesiti M., A Matching Algorithm for Measuring the Structural Similarity between an XML Documents and a DTD and its Applications, *Elsevier Computer Science*, 29 (23-46), 2004.

[2] Bille P., A Survey on Tree Edit Distance and Related Problems, Theoretical Computer Science, vol. 337, n°1-3, pp. 217-239, 2005

[3] Boukottaya A. and Vanoirbeek C., Schema Matching for Transforming Structured Documents. The *International ACM Symposium on Document Engineering*, pp. 101 - 110, 2005.

[4] Bray T., Paoli J., Sperberg-McQueen C.M., Mailer Y., Yergeau F., Extensible Markup Language (XML) 1.0 5th Edition, W3C recommendation, November 2008, http://www.w3.org/TR/REC-xml/.

[5] Buttler D. A Short Survey of Document Structure Similarity Algorithms. In *Proc. of ICOMP*, pp. 3-9, 2004.

[6] Chawathe S., Rajaraman A., Garcia-Molina H., and Widom J., Change Detection in Hierarchically Structured Information. In *ACM SIGMOD Record,* pp. 493-504, 1996.

[7] Cobéna G., Abiteboul S. and Marian A., Detecting Changes in XML Documents. In *ICDE*, pp. 41-52, 2002.

[8] Dalamagas, T., Cheng, T., Winkel, K., and Sellis, T. 2006. A methodology for clustering XML documents by structure. *Inormation. Systems. 31 (3),*187-228, 2006.

[9] Do H.H. and Rahm E, COMA: A System for Flexible Combination of Schema Matching Approaches. In *VLDB Conference*, 610-621, 2002.

[10] Do H.H., Melnik S. and Rahm E., Comparison of Schema Matching Evaluations, In *Proc. of GI-Workshop on the Web and Databases*, pp. 221-237, 2002.

[11] Doan A., Domingos P. and Halevy A.Y., Reconciling Schemas of Disparate Data Sources: A Machine Learning Approach. In *Proc. of the SIGMOD Conference,* 2001.

[12] Formica A., 2008. Similarity of XML-Schema Elements: A Structural and Information content Approach. *The Computer Journal*, 51(2):240-254.

[13] Hall P. and Dowling G., Approximate String Matching. *Computing Surveys 12:4*, pp.381-402, 1980.

[14] Jeong B., Lee D., Cho H. and Lee J., A Novel Method for Measuring Semantic Similarity for XML Schema Matching. *Expert Systems with Applications: An International Journal*, 34 (3):1651-1658, 2008.

[15] Knuth, Donald, Sorting by Merging, *The Art of Computer Programming*. Addison-Wesley, 158-168, 1998.

[16] Lee M., Yang L., Hsu W. and Yang X., XClust: Clustering XML Schemas for Effective Integration. In *Proc. of CIKM*, pp. 292-299, 2002.

[17] Leonardi E. et al., DTD-Diff: A Change Detection Algorithm for DTDs. *DKE,* 61 (2) 384-402, 2007.

[18] Madhavan J., Bernstein P. and Rahm E., Generic Schema Matching With Cupid. In *VLDB*, pp. 49-58, 2001.

[19] Maguitman A. G., Menczer F., Roinestad H. and Vespignani A., Algorithmic Detection of Semantic Similarity. In *Proc. of WWW*, pp. 107-116, 2005.

[20] McGill M. J. *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[21] Melnik S., Garcia-Molina H. and Rahm E., Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching. In *Proceedings of ICDE*, 2002.

[22] Miller G. WordNet: An On-Line Lexical Database. *Journal of Lexicography*, 1990.

[23] Miller R., Hass L. and Hermandez M.A., Schema Mapping as Query Discovery. In *VLDB*, pp. 77-88, 2000.

[24] Nierman A. and Jagadish H. V., Evaluating structural similarity in XML documents. In *Proc. of Int. Workshop on the Web and Databases,* pp. 61-66, 2002.

[25] Peterson D., Gao S., Malhotra A., Sperberg-McQueen C.M. and Thompson H.S., W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes, January 2009, http://www.w3.org/TR/xmlschema11-2/.

[26] Rahm E. and Bernstein P.A., A Survey of Approaches to Automatic Schema Matching. *The VLDB Journal*, 10:334-350, 2001.

[27] Schlieder T., Similarity Search in XML Data Using Cost-based Query Transformations. In *Proc. of SIGMOD WebDB,* pp. 10-24, 2001.

[28] Shasha D. and Zhang K., Approximate Tree Pattern Matching. In *Pattern Matching in Strings, Trees and Arrays*, Oxford Press, 1995.

[29] Su H., Kuno H. and Rundensteiner E.A., Automating the Transformation of XML Documents. In *Proc. of ACM Workshop on Web Information and Data Management*, pp. 68-75, 2001.

[30] Su H., Padmanabhan S. and Lo M.L., Identification of Syntactically Similar DTD Elements for Schema Matching. *Advances in Web-Age Information Management Conf.*, pp. 145-159, 2001.

[31] Tekli J., Chbeir R. and Yetongnon K., A Fine-Grained XML Structural Comparison Approach. In *Proc. of the ER Conference*, pp. 582–598, 2007.

[32] Tekli J., Chbeir R. and Yetongnon K., An XML Grammar Comparison Framework – Technical Report, 2008, http://www.u-bourgogne.fr/DbConf/XMG/.

[33] Yi S., Huang B. and Chan W.T., XML Application Schema Matching Using Similarity Measure and Relaxation Labeling. *Information Sciences*, 169 (1-2):27-46, 2005.

[34] Zhang K. and Shasha D., Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal,* 18(6):1245-1262, 1989.

[35] Zhang Z., Li R., Cao S. and Zhu Y., Similarity Metric in XML Documents. *Knowledge and Experience Management Workshop*, 2003.