Minimizing User Effort in XML Grammar Matching

Joe Tekli[1] and Richard Chbeir[2*]

[1] University of Milan, Department of Information Technology,
Via Bramante, 65 – 26013 Crema, Italy

[2] University of Bourgogne, LE2I Laboratory UMR-CNRS,
9 Alain Savary – 21000 Dijon, France

Abstract. XML grammar matching has found considerable interest recently, due to the growing number of heterogeneous XML documents on the Web, and the need to integrate, search and retrieve XML documents originated from different data sources. In this study, 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. This requires i) considering the various characteristics and constraints of XML grammars (in comparison with 'grammar simplifying' approaches), ii) allowing a flexible combination of different matching criteria (in comparison with static approaches), and iii) effectively considering the semi-structured nature of XML (in contrast with heuristic methods). To achieve this, we propose an extensible framework based on the concept of tree edit distance as an optimal technique to consider XML structure, integrating different matching criteria to capture all basic XML grammar characteristics, ranging over element semantic and syntactic similarities, cardinality and alternativeness constraints, as well as data-type correspondences and relative ordering. In addition, our framework is flexible, enabling the user to choose mapping cardinality (i.e., *1:1*, *1:n*, *n:1*, *n:n*), in comparison with exiting static methods (usually constrained to *1:1*). User constraints and feedback are equally considered in order to adjust matching results to the user's perception of correct matches. Experiments on real and synthetic XML grammars demonstrate the effectiveness and efficiency of our matching strategy in identifying mappings, in comparison with alternative methods.

Keywords: XML, Semi-structured data, XML Grammar, Schema matching, Structural Similarity, Tree edit distance, Semantic similarity, Vector space model.

1. Introduction

With the growing amount of heterogeneous XML information on the Web, i.e., documents originated from different data-sources and not conforming to the same grammars, there has been an overwhelming need to automatically process those documents and grammars for data and schema integration, and consequently information extraction, retrieval and search functions. All these applications require, in one 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 (finding, scoring and browsing changes between different versions of a document) [12, 13], XML structural querying (finding and ranking results according to their similarity) [57, 72], as well as structural clustering of XML documents gathered from the Web [14, 48]. 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 [18, 41]. XML grammar comparison is mainly exploited in data warehousing (mapping data sources to warehouse schemas) [51], message translation (central in B2B applications) [51], as well as XML data maintenance and schema evolution (detecting differences/updates between different versions of a given grammar to consequently revalidate corresponding XML documents) [34].

* Corresponding author. Tel.: +33 3 80 39 36 55; Fax: +33 3 80 39 68 69; e-mail: richard.chbeir@u-bourgogne.fr

In this study, we address the XML grammar matching/comparison problem, i.e., the comparison of DTDs [7] and/or XML Schemas [49] based on their most common characteristics. In fact, the effectiveness of grammar matching systems is assessed w.r.t.[1] the amount of manual work required to perform the matching task [17], which depends on: i) the level of simplification in the representation of the grammars, and ii) the combination of various matching techniques [15]. In general, most XML-related grammar matching methods in the literature are developed for generic schemas and are consequently adapted to XML grammars such as [15, 18, 37, 41]. On one hand, they often induce various 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 [15, 30]. On the other hand, they usually exploit individual matching criteria to identify similarities [41, 60] (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 criteria (semantic similarity, data-type similarity…) usually utilize machine learning techniques [18, 30] or basic mathematical formulations (e.g., *max*, *average*, etc.) [15, 52] which are not always adapted to the semi-structured nature of XML grammars in combining the results of different matchers.

Here, 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 various characteristics and constraints of the XML grammars being matched, in comparison with existing 'grammar simplifying' approaches, e.g., [15, 30], ii) allowing a flexible and extensible combination of different matching criteria, adaptable to various application scenarios, in comparison with existing static methods, e.g., [41, 60], and iii) effectively considering the semi-structured nature of XML, as the most prominent and distinctive feature of an XML grammar [2, 46], in comparison with existing heuristic or generic approaches, e.g., [18, 37], in order to produce more accurate results.

Hence, the contributions of our study can be summarized as follows. First, we devise a tree representation model for XML grammars that considers the hierarchical aspect and various constraints of XML elements/attributes, thus handling the expressive power of XML grammars without being constrained to a specific grammar language (e.g., DTD [7] or XSD [49]). Second, as XML grammars are effectively represented as trees, we put forward a method for XML grammar matching based on the concept of tree edit distance. We make use of tree edit distance as an effective and efficient means for structural similarity evaluation and identifying the structural matches between two XML grammars. In addition, we exploit tree edit distance as an open framework for the integration of different match criteria in evaluating XML grammar tree similarity. Dedicated matchers are utilized to capture element semantic and syntactic similarities, as well as constraint and data-type correspondences. User constraints and feedback are also considered in our method. Note that to our knowledge, this is the first attempt to exploit tree edit distance in XML grammar matching. Experimental results reflect our method's high matching quality and performance in comparison with existing approaches.

The remainder of the paper is organized as follows. Section 2 reviews the background and state of the art methods related to XML grammar matching. In Section 3, we present some preliminary notions and definitions. Section 4 depicts our XML grammar tree representation model. Section 5 develops our XML grammar matching framework. In Section 6, we present the experimental results obtained when evaluating our approach. Section 7 concludes the paper and discusses future directions.

2. Background and Related Works

2.1. XML Grammar and Constraint Operators

An XML grammar (e.g., DTD [7] or XSD [49]) is a structure made of a set of XML elements, sub-elements and attributes, linked together via the containment relation. It identifies element/attribute structural positions, data-types (e.g., *ID*, *IDREF*, *Decimal*, *Integer*, *String*, etc.), and the rules they adhere to in the XML document. These rules are defined via dedicated grammar constraint operators, which specify constraints on the existence and repeatability of elements/attributes.

[1] with respect to

XML grammar constraints consist of two main groups: *cardinality constraints (cc)* and *alternativeness constraints (ac)*.

2.1.1. Cardinality Constraint Operators

Cardinality constraint operators specify the number of occurrences and repetitions allowed for a given element in the document instances corresponding to the grammar at hand. Five cardinality operators exist in the DTD language [7]: i) the *'?'* operator indicates that the corresponding element is optional, ii) The '*' operator specifies that an element is repeatable, and that it may occur 0 or many times, iii) the '+' operator specifies that an element is repeatable, and that it must appear at least once. It is also possible to specify cardinality constraints on attributes: iv) the *Implied* operator specifies that an attribute is optional, v) the *Required* operator indicates that the attribute's occurrence is mandatory.

In XSD [49], cardinality operators amount to *MinOccurs* and *MaxOccurs*, specifying respectively the minimum and maximum number of times an element/attribute can appear in the corresponding XML document. These operators are obviously more expressive than their DTD counterparts and can exactly simulate the behaviors of the latter (e.g., *MinOccurs=0* is equivalent to '?' in DTDs, *MaxOccurs='unbounded'* is equivalent to '+', whereas *MinOccurs=0* and *MaxOccurs='unbounded'* is equivalent to '*'). Note that an element/attribute with no cardinality constraint (*null*) is mandatory.

2.1.2. Alternativeness Constraint Operators

Alternativeness constraint operators specify an element's disposition w.r.t. its siblings. Two main operators are allowed, in both DTD/XSD languages: *And* and *Or*. The *And* operator (represented as ',' in DTDs, and *sequence* in XML Schemas) represents a sequence of elements, such as each element should appear in the document. The *Or* operator (represented as '|' in DTDs, and, *choice* in XML Schemas) represents an alternative of elements, such as one and only one of the concerned elements should appear in the document. An additional hybrid operator, *All*, is introduced in XSD [49]. It allows all connected siblings to appear in any order, such as all appear at once, or not at all.

2.2. Overview on Grammar Matching and Comparison

Identifying similarities among grammars, otherwise known as *schema matching*, is usually viewed as the task of finding correspondences between elements of two schemas [17]. It has been investigated in various fields, mainly in the context of data integration [17, 51], and recently in the contexts of schema clustering [33, 46] and change detection [34, 59].

In general, a grammar/schema consists of a set of related elements (entities and relationships in the ER model, objects and relationships in the OO model, etc.). In particular, as described in the previous section, an XML grammar is made of a set of elements and attributes, linked together via the containment relation. Thus, the schema matching operator can be defined as a function that takes two schemas, S_1 and S_2, as input and returns a mapping between those schemas as output [51]. The mapping between two schemas indicates which elements of schema S_1 are related to elements of schema S_2 and vice-versa.

Criteria used to match the elements of two schemas are usually based on heuristics that approximate the user's understanding of a good match [51]. These heuristics usually consider the linguistic similarity between schema element names (e.g., string edit distance, synonyms, hyponyms, etc.), the similarity between element constraints (e.g., '?', '*', '+' in DTDs, or *MinOccurs* and *MaxOccurs* in XML Schemas), in addition to the similarity between element structures (matching combinations of elements that appear together). Some matching approaches also consider the data content of schema elements (e.g., element/attribute values, if available) when identifying mappings [18]. In most approaches, scores (similarity values) in the *[0, 1]* interval are assigned to the identified matches so as to reflect their relevance. These values can be normalized to produce an overall score underlining the similarity between the two schemas being matched. Such overall similarity scores are utilized in [33, 46], for instance, to identify clusters of similar grammars prior to conducting the integration task.

2.3. State of the Art in XML Grammar Matching

While schema matching is mostly studied in the relational and Entity-Relationship models [10, 32, 44], research in XML grammar comparison has been gaining increasing importance in the past few years due to the unprecedented abundant use of XML, especially on the Web. As discussed in the introduction, the main criterion generally utilized to assess the effectiveness of automatic schema matching methods is the amount of manual work required to perform the matching task [17, 46]. This criterion depends on two main factors: i) the level of *simplification in the representation* of the schema, and ii) the *combination of various matching techniques* to perform the match task [15].

2.3.1. Simplifying Grammar Representations

While most existing methods to XML grammar matching consider, in different ways, the linguistic as well as the structural aspects of XML grammars, they generally differ in the internal representations of the grammars. In general, various simplifications are required by different approaches inducing adapted schema representations upon which the matching process is run.

Early approaches: Methods in [37, 41, 60] tend to transform schemas into simplified graph representations, more similar to data-guides [26] for semi-structured data than to XML grammars. In general, various simplifications targeting the hierarchical structure of XML grammars, as well as repeatability and alternativeness constraints (e.g., '+', '*', *MinOccurs*, etc.), are usually undertaken in performing the match task. In [60], the authors simplify DTD constraints and merge sub-elements having the same label to abridge DTD graph representations. Then, a bottom-up procedure starts by matching leaf nodes, based on node label equality, and subsequently identifies inner-node matches based on the latter. In [41], the authors propose to construct a *pair-wise connectivity graph* (PCG) made of node pairs, one from each of the input schema graph. An initial similarity score, using classic string matching (i.e., string edit distance) between node labels, is computed for each node pair in the PCG, and then refined by propagating the scores to their adjacent nodes. A similar approach is provided in [37], where the authors consider the linguistic features of grammars nodes (making use of an external thesaurus), as well as node data-types (making use of an auxiliary data-type compatibility table), instead of only node label equality, such as in [41, 60].

Emphasis on XML structure: More recent approaches to XML grammar comparison, e.g., [6, 25, 59, 61, 63, 69], put more emphasis on the hierarchical structure of XML, and consider (to different extents) XML grammar cardinality and alternativeness constraints. The authors in [59] provide a DTD matching approach geared toward document transformation, and exploit heuristic functions based on the concept of relative information capacity [29] for choosing transformation operations among multiple alternatives. The authors however state that the heuristics used might produce unpredictable matching results, due the existence of ambiguous data capacity cases [59]. In [33], the authors develop *XClust*, an integration strategy that involves the clustering of DTD trees. The proposed algorithm is based on the semantic similarities between element names (making use of a thesaurus or ontology), corresponding immediate descendents, as well as sub-tree leaf nodes (i.e., leaf nodes corresponding to the sub-trees rooted at the elements being compared). It also considers recursive element declarations. While the internal representation of DTDs, with *XClust*, is more sophisticated than the methods presented above, it does not consider DTDs that specify alternative elements (i.e., the *Or* alternativeness operator is disregarded, replaced by an *And* operator). Studies similar to *XClust* are proposed in [6] and [63]. They both target XML Schemas (XSD) instead of DTDs. The various XSD base types (e.g., decimal, string, etc.) are considered in comparing elements. The authors in [6] mention the need to compare complex and derived (user defined) types, using dedicated type hierarchies, without, however, detailing the corresponding process. Both studies disregard XML grammar alternativeness operators (i.e., *Or* and *All*) in their similarity computations. In [61], the authors propose *QMatch* for matching XML schemas as collections of tree paths. The proposed algorithm is built on top of a bunch of classifiers for i) comparing schema element/attribute labels (semantic comparison w.r.t. WordNet [42]), ii) comparing element/attribute properties (order, cardinality constraints, and data-types), and iii) comparing paths structures (the latter is a composite classifier built upon the basic label and property classifiers). However, the proposed approach does not consider schemas with alternative declarations (i.e., the *Or* and *All* operators are not considered).

Emphasis on Structure and Grammar Constraints: The only approach we know of to effectively consider alternative elements (i.e., elements connected via the *Or* and *All* operators) is developed in [69]. The method is based on the relaxation labeling technique to solve the problem of assigning labels to elements w.r.t. a set of constraints[1]. Schemas are first compared at the element level considering their basic properties: element name, parent node, set of children, set of attributes, and set of brothers. Consequently, the pair-wise similarity matrix between all pairs of elements in both schemas being compared is iteratively optimized using label relaxation, taking into account the contextual characteristics of elements (i.e., their structural positions in the schemas being compared). The approach is iterative in that different variations of the pair-wise element similarity matrix are computed until similarities converge within a predefined threshold (or until the maximum number of iterations in reached). This might prove to be computationally cumbersome in practical applications (note that the authors do not discuss the efficiency of their approach). In addition, while it seems more sophisticated than its predecessors, the approach in [69] only allows restricted XML grammar declarations. For instance, operator concatenations, and thus composite declarations are not allowed (e.g., declaration *root*(*a, b,* (*c*|*d*)) is not allowed, only single operator declarations such as *root*(*a, b, c*) or *root*(*a* | *b* | *c*)).

In short, a correspondence between the level of simplification in the grammar representations and the amount of manual work required for the matching task can be identified: *the more schemas are simplified, the more manual work is required to update the matching results by considering the simplified constraints*. For instance, if the *Or* operator is replaced by the *And* operator in the simplified representation of a grammar (e.g., (*a* | *b*) is replaced by (*a* , *b*) in a given DTD element declaration), the user has to analyze the results produced by the matching approach, and manually evaluate and update the matches corresponding to elements that were initially linked by the *Or* operator (i.e., alternatives) prior to the simplification phase. In this context, *XClust* [33], *QMatch* [61], and *Relaxation Labeling* [69] seem more sophisticated than alternative matching approaches. They induce the least simplifications to the grammars being compared. *XClust* and *QMatch* only disregard the *Or* operator (and the XSD-specific *All* operator), whereas *Relaxation Relabeling* considers XML grammar repeatability and alternativeness constraints with restrictive declarations.

2.3.2. Combination of Several Matchers

The amount of user effort required to effectively perform the matching task can also be alleviated by the combination of several matchers [15], i.e., the execution of several matching techniques that capture the correspondences between schema elements from different perspectives. In other words, the schemas are assessed from different angles, via multiple matching criteria, the results being combined to obtain the best possible matches. In this context, existing approaches can be classified in two major groups: *hybrid* method and *composite* methods [51]. A hybrid approach is such as various matching criteria are used within a single algorithm. In general, these criteria (e.g., element name, data type, etc.) are fixed and used in a specific way. In contrast, a composite matching approach combines the results of several independently executed matching algorithms (which can be simple or hybrid).

Hybrid approaches: Methods of this category usually provide better match candidates and better performance than the separate execution of multiple matchers [51]. Superior matching quality is normally achieved since hybrid approaches are developed in specific contexts and target specific features which could be overlooked by more generic composite methods. In addition, they usually

[1] Relaxation labeling has been exploited in computer vision, mainly for edge detection and scene interpretation on the basis of labeled scene components. It relies on the idea that objects labeled the same way should be similar to each other [68].

provide better performance by reducing the number of passes over the schemas. Instead of going over the schema elements multiple times to test each matching criterion, such as with composite approaches, hybrid methods allow multiple criteria to be evaluated simultaneously on each element before continuing with the next one. In fact, most methods in the literature are *hybrid*, e.g., those discussed in the previous sub-section [6, 25, 37, 41, 59-61, 63, 69], in that various matching criteria (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, called *matchers*, in order to be flexible in choosing the criteria to evaluate, and extensible in adding additional ones.

Composite Approaches: To our knowledge, three composite matching approaches have been developed in the context of XML grammar matching: *LSD* [18], *NNPLS* [30], and *Coma* [15]. *LSD* [18] employs machine learning techniques to semi-automatically find mappings between two schemas. It requires a *training phase* and a *mapping phase*. For the training phase, the system asks the user to provide the mappings for a small set of data sources, and then uses these mappings to train its set of learners. Different types of learners can be integrated in the approach to detect different types of similarities (e.g., label syntactic matcher, semantic matcher, etc.). A special learner is introduced to take into account the structure of XML data: the *XML learner*. An approach similar to *LSD* is developed in [30]. It exploits supervised learning, via a neural network-based classifier, *NNPLS* (Neural Network-based Partial Least Squares), to synthesize the results of various lexical measures (targeting XML grammar node labels, i.e., *n-gram*, string edit distance, etc.) and structural similarity measures (e.g., edge matching, path matching [9], etc.) into an overall similarity between two XML grammars. Similarly to *LSD*, the neural network classifier has to be trained prior to conducting the comparison process. Since both methods in [18] and [30] are based on supervised learning, their main drawback is the *training phase* which could require substantial manual effort prior to launching the matching process. A simple platform for combining multiple matchers in a flexible way, entitled *COMA*, is provided in [15]. The authors in [15] propose different mathematical methods to combine matching scores (e.g., max, average, weighted average, etc.) rather than training and using a meta-learner such as in [18], or a neural network classifier in [30]. This avoids considerable manual effort. *COMA* could be completely automatic or iterative with user feedback. Aggregation functions, similar to those utilized to combine matching scores from individual matchers (e.g., max, average, weighted average, etc.), are employed to obtain an overall similarity score between the two schemas being compared. *COMA* is extended in [16] to efficiently process large scale schemas, by decomposing the latter into several smaller fragments [52], and only matching the fragment pairs with high similarity.

All three methods discussed above consider simplified XML grammar representations, disregarding cardinality and alternativeness constraints in the schemas being compared.

2.3.3. Large Scale XML Grammar Comparison

A few methods have been proposed to extend or adapt existing approaches in processing large scale schemas [2, 20, 47, 55]. These exploit indexing techniques (e.g., B-tree coupled with the vector space model) [20], sequence encoding methods (e.g., Prufer sequences) [2], node clustering or schema fragmentation (to limit the target node search space) [52, 55], so as to improve matching performance. Another strategy to improve grammar comparison efficiency in the context of XML grammar clustering, consists in providing a global similarity evaluation function comparing a single grammar to a set (cluster) of grammars [47]. This avoids the need to compute pair-wise similarities between two individual grammars, and hence saves a huge amount of computing effort in performing grammar. Nonetheless, recall that our main concern, in this study, lies within matching quality rather than performance. Hence, we report the discussion around techniques to improve XML grammar matching performance to a dedicated upcoming study.

2.3.4. Discussion

To sum up, most XML grammar comparison and matching approaches in the literature require various simplifications in the grammars being matched, eliminating certain element/attribute constraints and/or operators [37, 41, 60, 61], simplifying data-types [18, 30, 33], or restricting the expressiveness of element declarations [69]. This produces simplified schema representations upon which the matching processes are executed. In this context, *XClust* [33], *QMatch* [61], and *Relaxation Labeling* [69] seem more sophisticated than alternative matching approaches, as they induce the least simplifications to the grammars being compared.

On the other hand, most methods in the literature are *hybrid* (e.g., [6, 37, 41, 61, 63]) in that a predefined number of matching criteria are simultaneously evaluated within a single algorithm. Few approaches follow the alternative *composite* matching logic (e.g., [15, 18, 30]), 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. Few methods have been recently proposed to extend and/or adapt existing approaches in processing large scale schemas (e.g. [2, 20, 55]).

3. Preliminaries

3.1. Overview

Our main goal in this study is to develop an effective XML grammar matching method minimizing the amount of manual work needed to perform the match task. As mentioned previously, this requires:

– Considering the various characteristics and constraints of the XML grammars being matched, in comparison with existing 'grammar simplifying' approaches.
– Allowing a flexible and extensible combination of different matching criteria, adaptable to different application scenarios, in comparison with existing static *hybrid* methods.
– Effectively considering the semi-structured nature of XML, as the most prominent and distinctive feature for an XML grammar, in comparison with usually heuristic *hybrid* approaches or generic *composite* methods.

First, we devise a tree representation model for XML grammars that considers the hierarchical aspect and various constraints of XML elements/attributes, thus handling the expressive power of XML grammars. Second, as XML grammars are effectively represented as trees, we put forward an approach for XML grammar matching founded on the concept of tree edit distance. We make use of tree edit distance as an effective and efficient means for structural similarity evaluation and identifying the structural matches between two XML grammars. In addition, we exploit tree edit distance as an open framework for the integration of different match criteria in evaluating XML grammar similarity. Dedicated matchers are utilized to capture element semantic and syntactic similarities, as well as constraint and data-type correspondences. User constraints and user feedback are also considered in our method.

In the following, we present some preliminary definitions and basic notions prior to describing our XML grammar tree representation model and matching framework.

3.2. Basic Definitions

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

In the remainder of this paper, the term *tree* denotes *ordered labeled tree*

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

Definition 3 - Edit Script: It is a sequence of edit operations $ES = \prec op_1, op_2, \ldots, op_k \succ$. When an edit script ES is applied to a tree T, the resulting tree T' is obtained by applying each of the individual edit operations in ES to T, following their order of appearance in the script. By associating a cost, $Cost_{Op}$, to each edit operation in ES, the cost of ES can be computed as the sum of the costs of its component operations: $Cost_{ES} = \sum_{i=1}^{|ES|} Cost_{Op_i}$ [5, 11] ●

Definition 4 – Tree Edit Distance: The edit distance between two trees A and B is defined as the minimum cost of all edit scripts that transforms A to B: $TED(A, B) = Min\{Cost_{ES}\}$. Thus, the problem of comparing two trees A and B, i.e. evaluating the structural similarity between A and B, is defined as the problem of computing tree edit distance [11, 70] (usually, $Sim(A,B) = \frac{1}{1+TED(A,B)}$) ●

Definition 5 - Edit Distance Mapping: It is a graphical representation of the minimum cost edit script, depicting which edit operations apply to which nodes in the two trees being compared. Formally, it is defined as a triple (M, T, T') from tree T to tree T' such as $M \subseteq V_T \times V_{T'}$, where V_T and $V_{T'}$ designate the sets of nodes of trees T and T' respectively [5, 70] (cf. example in Figure 14). A mapping element $m \in (M, T, T')$ is represented in the form $\{x_i\} \leftrightarrow \{y_j\}$, where $\{x_i\} \subseteq V_T$ and $\{y_j\} \subseteq V_{T'}$, such as $\{x_i\}$ and $\{y_j\}$ underline single nodes and/or sets of nodes (sub-trees), following the edit operation at hand ●

Definition 6 – Mapping Cardinality: It consists of two orthogonal types: *local* and *global* cardinality [17].

- *Local* (also known as *element-level*) mapping cardinality, underlines the number of nodes participating in a single mapping element. Formally, given a mapping (M, T, T'), and a mapping element $m \in (M, T, T')$, such as $m = \{x_i\} \leftrightarrow \{y_j\}$ (cf. Definition 5), the local cardinality of m amounts to the number of nodes participating in m, that is: $|\{x_i\}| : |\{y_j\}|$. Local cardinality can be *1:1* (one-to-one), *1:n* (one-to-many), *n:1* (many-to-one), or *n:n* (many-to-many).
- *Global* (also known as *structure-level*) mapping cardinality, underlines the number of mappings to which a given node participates. Formally, given a mapping (M, T, T'), and a node $x \in T$ such as x participates in a set of mapping elements $\{m_r\} \in (M, T, T')$, then the global cardinality corresponding to node x, w.r.t. (M, T, T'), amounts to 1: $|\{m_r\}|$. A node can participate in zero (no mapping), one, or several mapping elements, underlining a global cardinality of *1:1*, *1:n*, *n:1*, and/or *n:n* ●

For instance, mapping element *Name* ↔ *Surname* is of *1:1* local cardinality, whereas *Name* ↔ (*FirstName, LastName*) is of *1:2* (generally designated as *1:n*) local cardinality. As for global cardinality, if the previous mappings are considered in the same matching result, then node *Name* will be participating in two mapping elements. Hence, its global mapping cardinality is *1:2* (i.e., *1:n*). Otherwise, if considered separately, *Name* would be assigned global *1:1* cardinality in both cases.

In the following, we describe the edit operations utilized in our approach (adapted from [11, 48]).

Definition 7 - Insert Node: Let T be an (ordered labeled) tree with an internal node p, and let $T_1, \ldots, T_m$ be the first level sub-trees corresponding to node p (i.e., sub-trees rooted at the children of node p, Definition 2). Given a node x not belonging to T, $Ins(x, i, p, \ell)$ is a node insertion operation applied to T, inserting x as the i^{th} child of p. In the transformed tree T', node p will have $T_1, \ldots, T_{i-1}, x, T_{i+1}, \ldots, T_{m+1}$ as its first level sub-trees, with ℓ the label of inserted leaf node x ●

Definition 8 – Delete Node: Let T be a tree with an internal node p, having a leaf node x as the i^{th} child of p, $Del(x, p)$ is a node deletion operation applied to T, yielding T' where node p will have first level sub-trees $T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_m$ ●

Definition 9 – Update Node: Given a node x in tree T, and a label ℓ, $Upd(x, \ell)$ is a node update operation applied to x resulting in T' which is identical to T except that in T', x bears ℓ as its label. The update operation could be also formulated as follows: $Upd(x, y)$ where $y.\ell$ denotes the new label to be assumed by x ●

Definition 10 – Insert Tree: Let T be a tree, with an internal node p, and let $T_1, \ldots, T_m$ be the first level sub-trees of node p. Given a tree A not belonging to T, $InsTree(A, i, p)$ is a tree insertion operation applied to T, inserting A as the i^{th} sub-tree of p. In the transformed tree T', node p will have $T_1, \ldots, T_{i-1}, A, T_{i+1}, \ldots, T_{m+1}$ as its first level sub-trees ●

Definition 11 – Delete Tree: Let T be a tree with an internal node p, having a tree A as the i^{th} first level sub-tree of p, $DelTree(A, p)$ is a tree deletion operation applied to T, yielding T' where node p will have first level sub-trees $T_1, \ldots, T_{i-1}, T_{i+1}, \ldots, T_m$ ●

In addition to the structural properties of XML documents, XML grammars define element and attribute labels, which generally bear semantic meaning. Hence, the need to integrate semantic similarity evaluation in our XML grammar matching framework becomes obvious. In the fields of Natural Language Processing (NLP) and Information Retrieval (IR), manually built knowledge bases, (e.g., *WordNet* [42], *Roget*'s thesaurus [68], *ODP* [38], …) provide references for understanding semantic relations between concepts, and evaluating semantic similarity.

Definition 12 – Knowledge base: It is a semantic network made of a set of concepts representing groups of words/expressions (or URLs such as with ODP), and a set of links connecting the concepts, representing semantic relations. It provides a framework for organizing words (expressions) into a semantic space [38, 42] ●

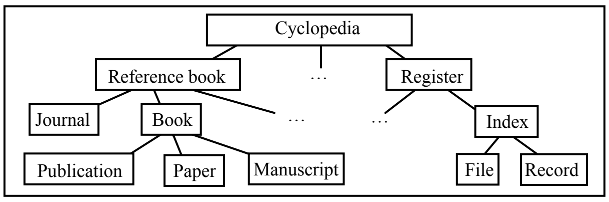


Figure 1. Extract of the WordNet[1] taxonomy.

Consequently, several methods have been proposed to determine the semantic similarity between concepts in a knowledge base, e.g. [35, 53, 54]. These can be classified as edge-based and node-based [54]. Methods of the former group generally estimate similarity as the shortest path (in edges, edge weights, or number of nodes) between the two concepts being compared. Nonetheless, with node-based approaches, semantic similarity between concepts is estimated as the maximum amount of information content they share in common, and makes use of the statistical distribution of corresponding words/expressions in a given text corpus (e.g., the Brown Corpus of American English [24]). Please refer to [8] for a detailed review on semantic similarity evaluation methods.

4. XML Grammar Tree Representation

With most existing XML grammar comparison approaches (cf. Section 2.3), grammars are represented as XML-like trees. Simplified graph structures are considered when recursive definitions come to play [33, 60], and are usually turned acyclic to obtain trees, which are naturally easier to process. As mentioned previously, most existing methods simplify grammars, disregarding element/attribute data-types and/or constraints. Hence, we provide here a tree representation that i) accurately captures the structural properties of XML grammars, and ii) considers their most common characteristics.

[1] WordNet is an online lexical reference system (taxonomy), developed by a group of researchers at Princeton University NJ USA, where nouns, verbs, adjectives and adverbs are organized into synonym sets, each representing a lexical concept [42].

4.1. XML Grammar Tree Model

First, we define the notions of *composite alternativeness constraint* and *alternativeness constraint vector* which are central to preserving the structural levels of XML grammar elements/attributes following our tree representation model.

Definition 13 - Composite alternativeness constraint: It is an alternativeness operator, e.g., *And*, *Or,* or *All*, to which we associate a cardinality constraint, e.g., ?, *, etc. (cf. Section 2.1), 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 the sake of generality, a simple alternativeness constraint can be represented as a doublet *sac* = (*sac, null*) ●

For instance, XSD declaration *<All MinOccurs=0><element name='a'><element name='b'> </All>* corresponds to an (*All, MinOcc=0*) composite constraint associated with both elements *a* and *b*. Declaration (*a, b, c*)+ corresponds to an (*And,* +) composite constraint associated with *a*, *b*, and *c*. Likewise for the *Or* operator.

Definition 14 - Alternativeness constraint vector: It is a vector $\overrightarrow{ac}$ of simple and/or composite alternativeness constraints, underlining the disposition of an XML grammar element w.r.t. its siblings and parent element in the grammar ●

For instance, in DTD declaration ((*a* | *b*)*?, c*), vector ⟨*And,* (*Or, ?*)⟩ is associated with both elements *a* and *b*, while vector ⟨*And*⟩ is associated with element *c*.

Note that indices are associated with alternativeness operators in the $\overrightarrow{ac}$ vector to distinguish different operators occurring at the same encapsulation level. For instance, in DTD declaration ((*a,b*) | (*c,d*)), vector ⟨*Or,* And_1⟩ would be associated with nodes *a* and *b* whereas vector ⟨*Or,* And_2⟩ would be associated with *c* and *d*, underlining the fact that elements *a/b* and *c/d* are connected via different *And* operators. Likewise with declaration ((*a*|*b*|*c*) , (*c*|*d*|*f*)), vectors ⟨*And,* Or_1⟩ and ⟨*And,* Or_2⟩ would distinguish the *Or* operators connecting nodes *a/b/c* and *d/e/f* respectively.

A basic XML grammar consists of element, attribute, and entity declarations. Yet, entities are variables used to define shortcuts to common text. Thus, they are disregarded in our XML grammar tree model. Consequently, our XML grammar tree representation is defined as follows.

Definition 15 - XML Grammar Tree: Formally, we model an XML grammar as a rooted ordered labeled tree $G = (N_G, E_G, L_G, CC_G, \overrightarrow{AC_G}, T_G, g_G)$ where:

- N_G is the set of nodes (i.e., vertices) in *G*,
- $E_G \subseteq N_G \times N_G$ is the set of edges (element/attribute containment relations), which reflect the hierarchical structure of the XML grammar tree,
- L_G is the set of labels corresponding to the nodes of *G*. $L_G = El_G \cup A_G$ such as El_G and A_G designate respectively the labels of the elements and attributes of *G*,
- CC_G is the set of cardinality constraints associated with the elements and attributes of *G* ('?', '*', '+', *'MinOccurs'*, *'MaxOccurs'*, '*Required*', '*Implied*' and *null*, cf. Section 2.1.1),
- $\overrightarrow{AC_G}$ is the set of alternativeness constraint vectors associated with the elements and attributes of *G* (central to preserving the structural levels of XML grammar nodes, Figure 5),
- T_G is the set of data-types, $T_G = ET \cup AT$, including the basic XML element data-types *ET* = {'#PCDATA', 'String', 'Decimal', 'Integer', … , *Composite, Recursive*} and attribute data-types *AT* = {'CDATA', 'ID', 'IDREF', 'IDREFS', NMTOKEN', …, *Enumeration*},
- g_G is a function $g_G : N_G \rightarrow L_G, CC_G, \overrightarrow{AC_G}, T_G$ that associates a label $\ell \in L_G$, a cardinality constraint $cc \in CC_G$, an alternativeness constraint vector $\overrightarrow{ac} \in \overrightarrow{AC_G}$ and a data-type $t \in T_G$ to each node $n \in N_G$ ●

Hence, following our tree representation, an XML grammar tree node is modeled as follows:

Definition 16 - XML Grammar Tree Node: A node $n \in N_G$ of XML grammar tree $G = (N_G, E_G, L_G, CC_G, \overrightarrow{AC_G}, T_G, g_G)$ is represented by a quintuplet $n = (\ell, cc, \overrightarrow{ac}, t, Ord)$ where $\ell \in L_G$, $cc \in CC_G$, $\overrightarrow{ac} \in \overrightarrow{AC_G}$ and $t \in T_G$ are respectively its label, cardinality constraint, alternativeness constraint vector and data-type. The *Ord* component underlines the node's order w.r.t. its siblings (detailed in the following). The constituents of node *v* can be referred to as $n.\ell$, $n.cc$, $n.\overrightarrow{ac}$, $n.t$, and $n.Ord$ (Figure 2) ●

$n.\ell$	$n.cc$	$n.\overrightarrow{ac}$	$n.t$	$n.Ord$

Figure 2. Graphical representation of XML grammar tree node *n*.

4.2. Tree Ordering

In XML documents, attributes are usually treated as unordered nodes. In other words, the order, left-to-right, of attribute nodes corresponding to a given element is not relevant [66] (e.g., *<Paper title="…" Genre="…">* is equivalent to *<Paper Genre="…" Title="…">*). Consequently, the same is true for attributes in XML grammars. In addition, XML grammar element nodes connected via the *Or/All* operators are unordered [49] (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 [5, 71]. 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. An ordering score *Ord* will be associated with each grammar tree node, underlining the reordering magnitude of the node. The *Ord* score will be exploited in the matching framework 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 $n \in N_D$:

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

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

The pseudo-code of our node ordering algorithm is provided in Figure 3. It traverses the first-level sub-trees $FL\text{-}SbTree_T = \{T_1, ..., T_k\}$ of a given tree *T* of root *p* and degree *k*, recursively, and orders corresponding root nodes following node labels $p_1.\ell, ..., p_k.\ell$. This can be achieved in average linear time using efficient sorting algorithms such as *Quicksort*, *Mergesort*, *Bucketsort* [31]. It simultaneously computes and associates, to each sub-tree root node p_i, its corresponding *Ord* score.

```
Algorithm SiblingOrdering
Input: T        // XML grammar tree
Output: T       // XML grammar tree T with reordered node siblings
Begin
    MergeSort(FL-SbTree_T)   // Sorts first level sub-trees of T following their root node labels
    M = Degree(A)            //The number of first level sub-trees in T, i.e., |FL-SbT_T|
    For (i=1 ; i ≤ M ; i++)
    {
        SiblingOrdering(T_i)     // Recursive formulation
    }
Return T
End
```

Figure 3. Pseudo-code of our sibling ordering algorithm.

4.3. Sample XML Grammars and Corresponding Tree Representations

Consider, for instance, the XML grammars in Figure 4. Corresponding tree representations, following our tree model, are depicted in Figure 5 (note that elements of the same structural level are represented in a stair-like manner to fit in page margins).

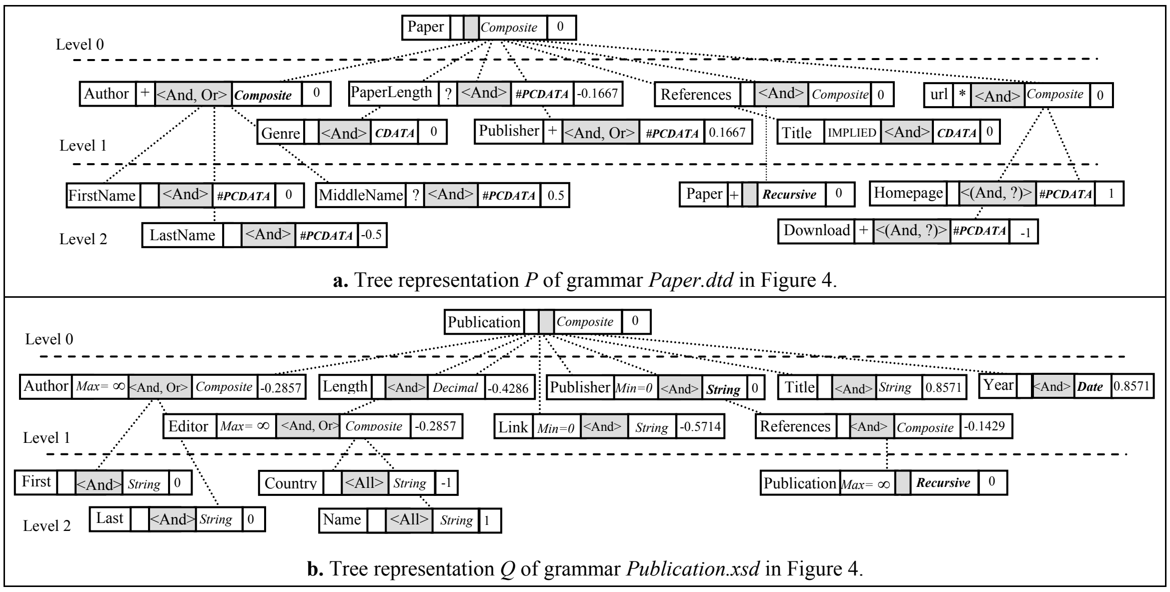
```
<!ELEMENT Paper ((Publisher | Author+), PaperLength?,
                References, url*)>

<!ATTLIST Paper Title CDATA #IMPLIED>
<!ATTLIST Paper  Genre CDATA>

<!ELEMENT Publisher (#PCDATA)>
<!ELEMENT Author (FirstName, MiddleName?, LastName)>
<!ELEMENT Length (#PCDATA)>
<!ELEMENT References (Paper+)>
<!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= "References">
           <element ref= "Publication" MaxOccurs= "unbounded">
      </element>
      <element name="Link" type="String" MinOccurs= "0"/>
   </sequence>
</element>
```

b. Publication.xsd

Figure 4. Sample XML grammars.


Level 0
Paper | | Composite | 0
Level 1
Author | + | <And, Or> | Composite | 0
PaperLength | ? | <And> | #PCDATA | -0.1667
References | | <And> | Composite | 0
url | * | <And> | Composite | 0
Genre | | <And> | CDATA | 0
Publisher | + | <And, Or> | #PCDATA | 0.1667
Title | IMPLIED | <And> | CDATA | 0
Level 2
FirstName | | <And> | #PCDATA | 0
MiddleName | ? | <And> | #PCDATA | 0.5
LastName | | <And> | #PCDATA | -0.5
Paper | + | | Recursive | 0
Homepage | | <(And, ?)> | #PCDATA | 1
Download | + | <(And, ?)> | #PCDATA | -1
Level 0
Publication | | | Composite | 0
Level 1
Author | Max= ∞ | <And, Or> | Composite | -0.2857
Length | | <And> | Decimal | -0.4286
Publisher | Min=0 | <And> | String | 0
Title | | <And> | String | 0.8571
Year | | <And> | Date | 0.8571
Editor | Max= ∞ | <And, Or> | Composite | -0.2857
Link | Min=0 | <And> | String | -0.5714
References | | <And> | Composite | -0.1429
Level 2
First | | <And> | String | 0
Last | | <And> | String | 0
Country | | <All> | String | -1
Name | | <All> | String | 1
Publication | Max= ∞ | | Recursive | 0

a. Tree representation *P* of grammar *Paper.dtd* in Figure 4.

b. Tree representation *Q* of grammar *Publication.xsd* in Figure 4.

Figure 5. XML grammar tree representations.

In the tree representation of grammar *Paper.dtd*, nodes of labels *'Title'* and *'Genre'* are attributes. In other words, the sorting process does not affect their ordering scores (which maintain zero values), nor the scores of their siblings. Similarly, nodes of labels *'Publisher'* and *'Author'* underline elements connected via the *Or* operator. In other words, they are unordered w.r.t. each other and the positioning of node *'Author'* before *'Publisher'* does not affect their ordering scores. The score of node *'Author'* remains equal to zero whereas that of *'Publisher'* is equal to $\frac{1}{6}$ = 0.1667, since *'Publisher'* changed its position w.r.t. node *'PaperLength'* with which it is associated via an *And* operator.

4.4. Special Case of Recursive Elements

To our knowledge, only two existing XML grammar matching approaches consider recursive element declarations, i.e., [33, 60]. In both methods, the authors represent (recursive) XML grammars as acyclic graphs (i.e., trees) by creating for each recursive element node *n* a new leaf node *n'* to which are directed all edges entering *n*. Consequently, the similarity between recursive elements simultaneously i) contributes to the similarity of their reference nodes, ii) and is determined based on the similarity of the latter, if the reference nodes are matched.

We follow the same strategy in our approach. For each recursive grammar node *n*, we create a new leaf node *n'*, such as *n'* has the same components as *n*, to the exception of its data-type, which is set to *Recursive*. For instance, each of the XML grammars *Paper.dtd* and *Publication.xsd* in Figure 4 encompasses a recursive element declaration, referencing root elements *Paper* and *Publication* respectively. Hence, dedicated leaf nodes are introduced in the corresponding tree representations (cf. Figure 5). Note that the similarity between recursive leaf nodes contributes to the similarity of their reference nodes, and is determined based on the latter if they match. The similarity evaluation issue is detailed in the following section.

Since XML grammars are represented as special ordered labeled trees (cf. Definition 15), the problem of matching two grammars comes down to matching the corresponding trees.

5. XML Grammar Matching Framework

Tree edit distance methods have been widely utilized to compare XML documents, represented as Ordered Labeled Trees [66], and have been proven optimal w.r.t. less accurate structural comparison methods [9] (such as *weighted tag* [48] or *Fourier Transform* [22]). In addition, an advantage of using the 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 (cf. Definition 3 and Definition 5). This proves to be crucial in the context of schema matching, as it would basically constitute the output of the match operation (recall that the schema matching operator can be defined as a function that takes two schemas, S_1 and S_2, as input and returns a mapping between the schemas as output [51]). To our knowledge, this study underlines the first attempt to exploit tree edit distance in XML grammar matching. In addition, the edit distance mapping could be utilized as an explanation component, which could help the user adapt weights for the distance measure in order to reflect her individual notion of matching.

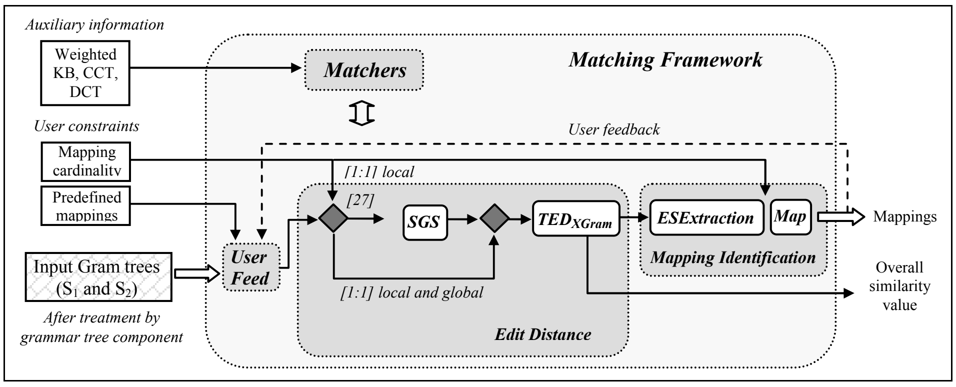


Figure 6. Simplified activity diagram describing our XML grammar matching framework.

Our XML grammar matching and comparison approach (Figure 6) consists of four components:

i. The *XML Grammar Tree Comparison* component for computing the distance (and consequently the similarity) between two XML grammar trees,
ii. The extensible *Matchers* component, encompassing several independent matching algorithms, exploited via the *Edit Distance* component to capture the similarities between XML grammar nodes based on their characteristics (label, data-type, cardinality constraints, alternativeness constraints, and node ordering, cf. Definition 16),

iii. The *Mapping Identification* component, interacting with the *Tree Edit Distance* component to identify the edit script (*ES_Extraction*), and consequently the edit distance mappings, between the compared XML grammar trees,
iv. The *UserFeed* component to consider user predefined mappings and user feedback in producing matching results.

In the remainder, sub-sections 5.1 to 5.4 respectively develop each of the components above.

5.1. XML Grammar Tree Comparison Component

Several algorithms have been developed to compute a distance, as the sum of a sequence of elemental edit operations that can transform one tree structure into another (cf. [5] for a detailed survey on Tree Edit Distance). In the context of XML, the most recent and efficient proposals, e.g., [48, 62], 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 method, *SGS*, to compute the *Similarity* between XML *Grammar Sub-trees*, based on the vector space model in information retrieval [40]. XML grammar sub-tree similarities are consequently exploited as tree edit operations' costs in a dynamic programming *Tree Edit Distance* algorithm ($TED_{XGram,}$ cf. system architecture in Figure 6).

Note that our grammar comparison method can be viewed as an extension of [62], one of the most recent tree edit distance based methods for comparing XML document structures.

5.1.1. Similarity between XML Grammar Sub-trees (SGS)

When evaluating XML grammar sub-tree similarity, one should consider all grammar node characteristics (element names, depth and relative order, cardinality constraints, alternativeness constraint vectors, data-types, and ordering scores) so as to produce accurate results. To do so, we exploit the vector space model in information retrieval [40]. When comparing two grammar sub-trees SbT_i and SbT_j, each is represented as a vector, $\vec{V_i}$ and $\vec{V_j}$ respectively, with weights underlining the similarities between their nodes.

Definition 17 – XML Grammar Sub-tree Vector Space: Given two sub-trees SbT_i and SbT_j, we define corresponding sub-tree vectors $\vec{V_i}$ and $\vec{V_j}$ in a space which dimensions represent, each, a single node $n_r \in SbT_i \cup SbT_j$, such as $1 < r < n$ where n is the number of distinct nodes in both SbT_i and SbT_j, grammar nodes being distinguished by their components (i.e., label, cardinality and alternativeness constraints, data-type and ordering score). The coordinate of a given sub-tree vector $\vec{V_i}$ on dimension n_r is noted $w_{\vec{V_i}}(n_r)$, and stands for the weight of n_r in sub-tree SbT_i

When node $n_r \in SbT_i$, vector coordinate $w_{\vec{V_i}}(n_r)$ underlines corresponding sub-tree node occurrences (which comes down to computing *TF – Term Frequency –* in classic vector space model [40]) ●

Definition 18 – XML Grammar Node Weight: The weight of a node n_r in vector $\vec{V_i}$, representing a sub-tree SbT_i, is composed of two factors: a node/vector similarity $Sim(n_r, \vec{V_i}, Aux)$ factor and a depth $D\text{-}factor(n_r)$ factor, such as $w_{\vec{V_i}}(n_r) = Sim(n_r, \vec{V_i}, Aux) \times D\text{-}factor(n_r) \in [0, 1]$:

- $Sim(n_r, \vec{V_i}, Aux)$ quantifies the similarity between node n_r and sub-tree vector $\vec{V_i}$. It is computed as the maximum similarity between n_r and all nodes of SbT_i considering the various grammar node characteristics (cf. Definition 16). Formally, $Sim(n_r, \vec{V_i}, Aux) = \underset{n \in \vec{V_i}}{Max}(Sim_{GNode}(n_r, n, Aux)) \in [0, 1]$.

– *D-factor*(n_r) considers the hierarchical depth of node n_r when computing its weight in sub-tree vector $\overrightarrow{V_i}$. Generally, information placed near the root node of an XML document and/or grammar is more important than information further down in the hierarchy [4, 72]. Thus, node labels higher in the XML document and/or grammar tree hierarchy should have a greater influence than their lower counterparts. This could be mathematically concretized using *Formula (2)*, adapted from [72]:

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

Definition 19 - Similarity between XML Grammar Nodes: It quantities the similarity between two grammar nodes, considering their various characteristics:

$$\begin{aligned}
&\textit{if} \;\; ((n.t = Recursive \wedge m.t = Recursive) \vee (n.t \quad Recursive \wedge m.\,t \quad Recursive))\\
&\qquad \mathrm{Sim}_{\mathrm{GNode}}(n, m, Aux) = f_{Agg}\,(\,\mathrm{Sim}_{\mathrm{Label}}(n.\ell, m.\ell, \overline{SN}),\\
&\qquad\qquad \mathrm{Sim}_{\mathrm{CConstraint}}(n.cc, m.cc, CCT),\\
&\qquad\qquad \mathrm{Sim}_{\mathrm{AConstraint}}\,(n.\overrightarrow{ac}, m.\overrightarrow{ac}, CCT),\\
&\qquad\qquad \mathrm{Sim}_{\mathrm{Data\text{-}Type}}(n.t, m.t, DTCT),\\
&\qquad\qquad \mathrm{Sim}_{\mathrm{OrdScore}}(n.Ord, m.Ord)\,)\\
&\textit{Otherwise}\\
&\qquad \mathrm{Sim}_{\mathrm{GNode}}(n, m, Aux) = 0
\end{aligned} \qquad (3)$$

where *f* is an aggregation function for combining the similarity values between XML grammar node characteristics, *Aux*={*SN, CCT, DTCT*} designates the auxiliary data sources required by the matchers to compute similarity: $\overline{SN}$ (weighted semantic network), *CCT* (constraint compatibility table) and *DTCT* (data-type compatibility table, cf. Section 5.2) ●

Formula (3) considers the special case of recursive leaf nodes. Similarity is null whenever a recursive leaf node is compared to a non-recursive leaf node, regardless of the remaining node characteristics (i.e., label, cardinality/alternativeness constraints and ordering). It thus allows a recursive leaf node to be compared (and hence possibly matched) to only another recursive leaf node. This is in accordance with both studies in [33, 60], which explicitly consider the case of leaf node declarations in XML grammars.

As for the aggregation function, various mathematical formulations for combining matcher results have been investigated in [15, 50], among which the *maximum*, *minimum*, *average* and *weighted sum* functions. Here, we exploit the latter as it provides flexibility in performing the match operation, adapting the process w.r.t. the user's perception of XML grammar element similarity:

$$\begin{aligned}
&f_{Agg}\,(\mathrm{Sim}_{\mathrm{Label}}(n.\ell, m.\ell, SN), ..., \mathrm{Sim}_{\mathrm{OrdScore}}(n.Ord, m.Ord))\\
&\quad = \; \mathrm{w}_{\mathrm{Label}} \times \mathrm{Sim}_{\mathrm{Label}}(n.\ell, m.\ell, \overline{SN}) +\\
&\qquad\; \mathrm{w}_{\mathrm{CConstraint}} \times \mathrm{Sim}_{\mathrm{CConstraint}}(n.cc, m.cc, CCT) +\\
&\qquad\; \mathrm{w}_{\mathrm{AConstraint}} \times \mathrm{Sim}_{\mathrm{AConstraint}}\,(n.\overrightarrow{ac}, m.\overrightarrow{ac}, CCT) +\\
&\qquad\; \mathrm{w}_{\mathrm{Data\text{-}Type}} \times \mathrm{Sim}_{\mathrm{Data\text{-}Type}}(n.t, m.t, DTCT) +\\
&\qquad\; \mathrm{w}_{\mathrm{OrdScore}} \times \mathrm{Sim}_{\mathrm{OrdScore}}(n.Ord, m.Ord)
\end{aligned} \qquad (4)$$

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. Similarity scores are computed via corresponding matchers (Section 5.2).

Following *Formula (4)*, different weights are assigned to different node component similarities, reflecting the impact of each of the grammar element characteristics in identifying the mappings. The fine-tuning of similarity weights comes down to an optimization problem so as to maximize the overall similarity aggregation function in *Formula (4)*. This can be solved using a number of techniques that exploit machine learning, such as Neural Networks [39], bootstrapping [21], and non-linear combination functions [1], in order to identify the best weights for a given problem class [50]. The main idea with such techniques is to assign a higher (lower) weight with higher (lower) similarity values, acting like contrast filters in image processing by increasing the contrast on input matrixes. Providing such a capability, in addition to manual tuning, would enable the user to parameterize and adapt the matching process following the application scenario and XML grammars at hand. We do not further address the fine-tuning of similarity weights here since it is out of the scope of this paper (and will be addressed in a subsequent empirical study).

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* (*Formula (5)*) widely exploited in information retrieval [56].

$$SGS(SbT_i, SbT_j, Aux) = \text{Cos}(\vec{V_i}, \vec{V_j}) = \frac{\sum_{r=1}^{M} w_{\vec{V_i}}(n_r) \times w_{\vec{V_j}}(n_r)}{\sqrt{\sum_{r=1}^{M} w_{\vec{V_i}}(n_r)^2 \times \sum_{r=1}^{n} w_{\vec{V_j}}(n_r)^2}} \in [0,1] \qquad \textbf{(5)}$$

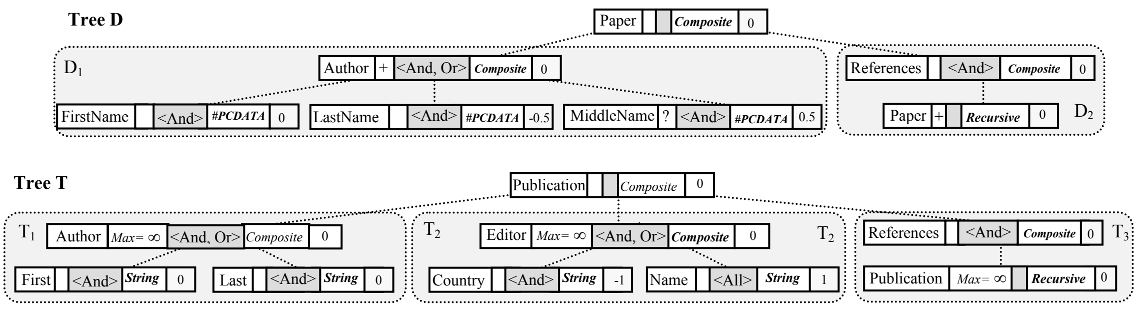
```
Algorithm SGS          // Similarity between XML Grammar Sub-trees

Input: $SbT_i$, $SbT_j$          // XML grammar subtrees
        Aux = { $\overline{SN}$ , CCT, DTCT}   // Auxiliary information required by the different matchers
                                 // including the weighted semantic network $\overline{SN}$,
                                 // the constraint compatibility table CCT,
                                 // and the data-type compatibility table DTCT.
Output: SGS($SbT_i$, $SbT_j$)          // Similarity between $SbT_i$ and $SbT_j$

Begin
VS = Generate_Vector_Space($SbT_i$, $SbT_j$)
$\vec{V_i}$ = Generate_Occurrence_Vector($SbT_i$, VS)   // Node occurrence weights: 0/non null,
$\vec{V_j}$ = Generate_Occurrence_Vector($SbT_j$, VS)   // taking into account node depths

For each node $n_r$ in $\vec{V_i}$          // Computing weights for vector $\vec{V_i}$
{
   If ( $w_{\vec{V_i}}(n_r)$ == 0)
      For each node $n_s$ in $SbT_i$
      {
         Sim($n_r$, $\vec{V_i}$) = Sim($n_r$, $n_s$, Aux) × D-factor($n_r$)          // Node weight
         If (Sim($n_r$, $\vec{V_i}$) > $w_{\vec{V_i}}(n_r)$ ) { $w_{\vec{V_i}}(n_r)$ = Sim($n_r$, $\vec{V_i}$) }   // Max weight
      }
}
For each node $n_s$ in $\vec{V_j}$          // Computing weights for vector $\vec{V_j}$
{
   If ( $w_{\vec{V_j}}(n_s)$ == 0)
      For each node $n_r$ in $SbT_j$
      {
         Sim($n_s$, $\vec{V_j}$) = Sim($n_s$, $n_r$, Aux) × D-factor($n_s$)          // Node weight
         If (Sim($n_s$, $\vec{V_j}$) > $w_{\vec{V_j}}(n_s)$ ) { $w_{\vec{V_j}}(n_s)$ = Sim($n_s$, $\vec{V_j}$) }   // Max weight
      }
}
Return  Cos($\vec{V_i}$, $\vec{V_j}$)          // SGS ($SbT_i$, $SbT_j$, Aux) = Cos($\vec{V_i}$ , $\vec{V_j}$), cf. Formula (5)
End
```

Figure 7. Algorithm SGS for computing the similarity between two XML grammar sub-trees.

Algorithm *SGS* for computing the similarity between grammar sub-trees is developed in Figure 7. It consists in building the vector space corresponding to the grammar sub-trees being compared, and computing the node weights and sub-tree vector similarity as described above. The algorithm's input parameters are the sub-trees SbT_i and SbT_j to be compared, as well as the various kinds of auxiliary information *Aux* required by the different matchers to compute node similarity. It outputs the sub-tree similarity value $SGS(SbT_i, SbT_j, Aux)$.

Computation Example: Consider the simple grammar trees in Figure 8 (truncated from those in Figure 5, to simplify computations). When comparing sub-trees D_1 and T_1 following *SGS*, the corresponding vector space would consist of 6 dimensions, related to each distinct node in both sub-trees: n_1=(*'Author'*, +, ⟨*And, Or*⟩, *Composite*, 0), n_2= (*'FirstName'*, , ⟨*And*⟩, *#PCDATA*, 0), n_3= (*'LastName'*, , ⟨*And*⟩, *#PCDATA*, 0.5), n_4= (*'MiddleName'*, ? , ⟨*And*⟩, *#PCDATA*, 0.5), n_5= (*'First'*, , ⟨*And*⟩, *#PCDATA*, 0), n_6 = (*'Last'*, , ⟨*And*⟩, *#PCDATA*, 0). Note that both sub-tree root nodes are identical, i.e., they bear identical labels, constraints ('+' ≡ 'MaxOccurs=∞')[1], data-types and ordering scores. Hence, they are not distinct and would be represented in one dimension corresponding to n_1.

In this example, and for the sake of simplicity, we only consider node labels in computing similarity (i.e., $w_{Label} = 1$, whereas remaining weights are set to zero) and consider a simple syntactic string comparison metric in evaluating string similarity: $\frac{S_1 \cap S_2}{S_1 \cup S_2}$, which belongs to the *N-Gram* string matching family (*1-Gram* matcher, identifying the number of characters in common between two strings w.r.t. the total number of characters, cf. Section 5.2.1). Hence, in the current example, no auxiliary information is needed (i.e., *Aux=Ø*, the *Aux* input parameter being omitted for simplicity).



Tree D
Paper | | | Composite | 0
D1
Author | + | <And, Or> | Composite | 0
FirstName <And> #PCDATA 0
LastName <And> #PCDATA -0.5
MiddleName ? <And> #PCDATA 0.5
References | | <And> | Composite | 0
Paper | + | | Recursive | 0
D2
Tree T
Publication | | | Composite | 0
T1
Author | Max= ∞ | <And, Or> | Composite | 0
First <And> String 0
Last <And> String 0
T2
Editor | Max= ∞ | <And, Or> | Composite | 0
Country <And> String -1
Name <All> String 1
T2
References <And> Composite 0
T3
Publication | Max= ∞ | | Recursive | 0

Figure 8. Sample XML grammar trees truncated from those in Figure 5.

	n_1	n_2	n_3	n_4	n_5	n_6
V_{D1}	1	1	1	1	0	0
V_{T1}	1	0	0	0	1	1

a. Simple node occurrence vectors.

	n_1	n_2	n_3	n_4	n_5	n_6
V_{D1}	1	1	1	1	0.5556	0.5714
V_{T1}	1	0.5556	0.5714	0.2222	1	1

b. Node vector similarity factor values.

	n_1	n_2	n_3	n_4	n_5	n_6
V_{D1}	0.5	0.3333	0.3333	0.3333	0.1852	0.1905
V_{T1}	0.5	0.1852	0.1905	0.0741	0.3333	0.3333

c. Final weights, i.e., *Sim ×D-factor.*

Figure 9. Sub-tree vectors obtained when comparing sub-trees D_1 and T_1.

For instance, $Sim(n_5, \overrightarrow{V_{D1}}) = 0.5556$ designates the maximum similarity between node n_5 and all nodes of sub-tree vector $\overrightarrow{V_{D1}}$. It comes down to the similarity between nodes n_5 ($n_5.\ell$ = *'FirstName'*) and n_2 ($n_5.\ell$ = *'First'*), sharing similar characteristics (recall that we only consider label syntactic similarity via the *N-Gram* matcher here). Final vector weights are obtained by multiplying both

[1] Note that the '+' DTD constraint is equivalent to *MinOccurs=1* ∧ *MaxOccurs*=∞. Nonetheless, *MinOccurs = 1* is a default XSD constraint, which is why it is usually omitted.

similarity and depth factors *Sim* × *D-factor* as shown in Figure 9.c (cf. Definition 18). As a result, the similarity between grammar sub-trees D_1 and T_1 is computed: $SGS(D_1, T_1) = Cos(\overrightarrow{V_{D1}}, \overrightarrow{V_{T1}}) = 0.8770$.

When comparing D_2 with either T_1 or T_2, the similarity between the recursive leaf node *Paper* and all nodes in T_1 and T_2 is null, since neither sub-tree encompasses recursive elements. Likewise for the recursive leaf node of label *Publication* in sub-tree T_3, when comparing T_3 with D_1. In our current example, the latter recursive nodes can only be compared to each other, i.e., when evaluating the similarity between sub-trees D_2 and T_3. When comparing D_2 and T_3, the vector space consists of *3* dimensions: n_1=(*'References'*, , ⟨*And, Or*⟩, *Composite*, 0), n_2=(*'Paper'*, +, ⟨*And*⟩, *Recursive*, 0), and n_2=(*'Publication'*, +, ⟨*And*⟩, *Recursive*, 0). Vector weights are shown in Figure 10, with $Sim(n_3, \overrightarrow{V_{D2}}) = Sim(n_2, \overrightarrow{V_{T3}}) = Sim(n_2, n_3) = \frac{2}{12} = 0.1667$ (applying the *1-Gram* matcher to compare recursive node labels). Hence, $SGS(D_2, T_3) = 0.7881$.

	n_1	n_2	n_3
V_{D2}	1	1	0
V_{T3}	1	0	1

a. Node occurrences.

	n_1	n_2	n_3
V_{D2}	1	1	0.1667
V_{T3}	1	0.1667	1

b. Node vector similarity factor values.

	n_1	n_2	n_3
V_{D2}	0.5	0.3333	0.0567
V_{T3}	0.5	0.0567	0.3333

c. Final weights, i.e., *Sim ×D-factor.*

Figure 10. Sub-tree vectors obtained when comparing sub-trees D_2 and T_3.

5.1.2. Tree Edit Operations Costs and Tree Edit Distance (TED_{XGram})

The tree edit distance algorithm TED_{XGram}, utilized in our study, is an adaptation of Nierman and Jagadish's main edit distance process [48]. It exploits *SGS* to identify the similarities between each pair of sub-trees (SbT_i and SbT_j) in the two trees T_1 and T_2 being compared, assigning tree insert/delete operation costs accordingly. Tree operations costs will hence vary as follows [62]:

$$\text{Cost}_{\text{DelTree}}(\text{SbT}_i) = \sum_{\text{All nodes } x \text{ of SbT}_i} \text{Cost}_{\text{Del}}(x) \times \frac{1}{1 + \underset{\text{all SbT}_j \in T_2}{Max} \{SGS(\text{SbT}_i, \text{SbT}_j, \text{Aux})\}} \quad \textbf{(6)}$$

$$\text{Cost}_{\text{InsTree}}(\text{SbT}_j) = \sum_{\text{All nodes } x \text{ of SbT}_j} \text{Cost}_{\text{Del}}(x) \times \frac{1}{1 + \underset{\text{all SbT}_i \in T_1}{Max} \{SGS(\text{SbT}_j, \text{SbT}_i, \text{Aux})\}} \quad \textbf{(7)}$$

Given two XML grammar trees T_1 and T_2 being compared, the maximum tree edit operation cost for a given sub-tree $\text{SbT}_i \in T_1$, $Cost_{DelTree/InsTree}(SbT_i)$, is equal to the cost of deleting/inserting each node of SbT_i, $\sum_{\text{All nodes } x \text{ of SbT}_i} \text{Cost}_{\text{Del/Ins}}(x)$, underlining a minimal sub-tree similarity between SbT_i and all sub-trees in T_2, ($SGS(SbT_i, SbT_j, Aux)=0$ for all $SbT_j \in T_2$). Consequently, the operation cost decreases w.r.t. the similarity between SbT_i and its counterparts $SbT_j \in T_2$. Details concerning tree edit operations costs and their mathematical properties, are provided in [62].

In addition to tree insertion/deletion operations costs which vary w.r.t. XML grammar sub-tree similarities (cf. *SGS* developed in the previous section), the TED_{XGram} algorithm (Figure 11) considers XML grammar node similarities in computing update operations costs (cf. Figure 11, line 5). Using the *update* operation, TED_{XGram} compares the roots of sub-trees considered in the recursive process (at startup, these would correspond to the grammar tree roots). With update operations applied to classic ordered labeled trees (e.g., XML document OLTs), element labels are the only information to be assessed. Hence, the cost of the update operation usually varies w.r.t. label equality/difference such as a minimum operation cost is assigned when the compared labels are identical (i.e., $Cost_{Upd}(a, b) = 0$ when $a.\ell = b.\ell$), as opposed to a maximum unit cost otherwise (i.e., $Cost_{Upd}(a, b) = 1$ when $a.\ell \neq b.\ell$). Yet, when XML grammar trees come to play, grammar element constraints and data-types have to be considered. Thus, we redefine the update operation, in the context of XML grammar trees, as follows:

Definition 20 - Update XML Grammar Node: Given a node *n* in XML grammar tree *T*, a label *ℓ*, a cardinality constraint *c*, an alternativeness constraint vector $\overrightarrow{ac}$, a data-type *t* and an ordering score *Ord*, *Upd*(*n, ℓ, cc,* $\overrightarrow{ac}$*, t, Ord*) is a node update operation applied to *n* resulting in *T'* which is identical to *T* except that in *T'*, *n* bears *ℓ* as its label, *cc* as its cardinality constraint, $\overrightarrow{ac}$ as its alternativeness constraint vector, *t* as its data-type and *Ord* as its ordering score. The update operation could also be formulated as follows: *Upd*(*n*, *m*) where *m.ℓ*, *m.cc*,, *m.t*, *m.*$\overrightarrow{ac}$, *m.Ord* denote respectively the new label, cardinality constraint, alternativeness constraint vector, data-type and ordering score ●

Hence, the cost of the update operation varies as follows:

$$\begin{array}{l} \textit{if} \quad (n.t \neq Recursive \wedge m.t \neq Recursuve) \quad \vee \quad (n.t = Recursive \wedge m.t = Recursive) \\ \qquad \text{Cost}_{\text{Upd}}(n, m, Aux) = (1 - \text{Sim}_{\text{GNode}}(n, m, Aux)) \times \text{D-factor}(n) \\ \textit{Otherwise} \\ \qquad \text{Cost}_{\text{Upd}}(n, m, Aux) = \infty \end{array} \qquad \textbf{(8)}$$

Following *Formula (8)*, the more initial and replacing XML grammar nodes are similar, the lesser will be the update operation cost, which will transitively yield a lesser minimum cost edit script (higher similarity value). When nodes are identical (having identical labels, constraints, data-types and ordering scores), similarity is maximal, and thus the cost of the update operation is zero, indicating that there are no changes to be made. The operation is assigned an infinite cost so as to guaranty that recursive leaf nodes can only be replaced (and hence matched) with other recursive nodes. In other words, the update operation cannot be executed on a pair of nodes such as only one of them is recursive, the latter being deemed incomparable [33, 60].

Computation Example: Consider, for instance, the grammar trees in Figure 8. Similarly to the example in the previous section, for the sake of simplicity, we only consider the node label criterion in computing similarity (i.e., w_{Label} = 1, whereas remaining weights are set to zero) and use the same syntactic string comparison metric utilized previously: $\frac{S_1 \cap S_2}{S_1 \cup S_2}$, (*1-Gram* matcher, which does not require any auxiliary input, i.e., *Aux=Ø*). In order to compare trees *D* and *T*, we start by computing tree edit operations costs (costs of leaf node sub-trees are omitted here for the sake of simplicity):

$$\text{Cost}_{\text{DelTree}}(D_1) = \sum_{\text{All nodes } x \text{ of } D_1} \text{Cost}_{\text{Del}}(x) \times \frac{1}{1 + SGS(D_1, T_1)} = 4 \times \frac{1}{1+0.8770} = 2.1311$$

$$\text{Cost}_{\text{DelTree}}(D_2) = \sum_{\text{All nodes } x \text{ of } D_2} \text{Cost}_{\text{Del}}(x) \times \frac{1}{1 + SGS(D_2, T_3)} = 2 \times \frac{1}{1+0.7881} = 1.1185 = \text{Cost}_{\text{InsTree}}(T_3)\text{, related SGS}$$

computations are detailed in the computation example developed in previous section.
Likewise, $\text{Cost}_{\text{InsTree}}(T_1)$ = 1.5983 and $\text{Cost}_{\text{InsTree}}(T_2)$ = 1.7574. Thus, the TED_{XGram} algorithm yields $\text{TED}_{\text{XGram}}(D, T)$ = 3.4189, having: $\text{Cost}_{\text{Upd}}(\text{R}(D), \text{R}(T)) = 1 - \text{Sim}_{\text{GNode}}(\text{R}(D), \text{R}(T)) = 1 - \frac{2}{12} = 0.8333$.

Table 1. Computing edit distance for XML trees *D* and *T*.

a. First recurrence matrix.

	R(T)	T_1	T_2	T_3
R(D)	0.8333	2.4316	4.1889	5.3074
D_1	2.9644	1.6615	3.4189	4.5374
D_2	4.0829	2.78	4.5374	3.6967

b. Second recurrence matrix, transforming D_1 to T_1.

	R(T_1)	T[2]	T[3]
R(D_1)	0	0.5408	1.0780
D[2]	0.5408	0.1481	0.6853
D[3]	1.0780	0.6853	0.2910
D[4]	1.6152	1.2225	0.8282

c. Last recurrence matrix, transforming D_2 to T_3.

	R(T_3)	T[8]
R(D_2)	0	0.8571
D[6]	0.8571	0.2778

Computational details for the first recurrence matrix, i.e., Table 1.a:

- Dist[0] = 0.8333, cost of updating *R*(*D*) to *R*(*T*),
- Dist[1][1] =Dist[0]+$TED_{XGram}(D_1, T_1)$ = 0.8333+0.8282, cost of transforming D_1 to T_1,
- Dist[1][2] = Dist[1][1] + $Cost_{InsTree}(T_2)$ = 1.6615 +1.7574, cost of inserting T_2 into *D*,
- Dist[2][3] = Dist[1][2] + $TED_{XGram}(D_2, T_3)$ = 3.4189 + 0.2778, where $TED_{XGram}(D_2, T_3)$ comes down to $Cost_{Upd}$(D[6], T[8]).

Likewise for Table 1.b and Table 1.c. Note that nodes *D[6]* and *T[8]* in Table 1.c (of labels *Paper* and *Publication* respectively) are both recursive (referencing root nodes *R*(*D*) and *R*(*T*) respectively, cf. Figure 8) which is why they are comparable. Thus, the similarity between XML grammar trees *D* and *T*, w.r.t. the *1-Gram* node label similarity criterion, is $Sim_{XGram}(D, T) = \frac{1}{1+TED_{XGram}(D, T)}$ = 0.7150.

```
Algorithm ES_Extraction()

Input: XML grammar trees A and B, {Dist[][]} the set of distance
       matrixes computed by $TED_{XGram}$ 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 ($A_i \neq B_j$)      //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 $R(A_i) \neq R(B_j)$)
    {
        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
```

Figure 12. Edit script extraction algorithm.

```
Algorithm $TED_{XGram}$()

Input: XML grammar trees A and B, operations costs
       $Cost_{DelTree}$/$Cost_{InsTree}$ for all sub-trees
       in A and B, Aux = {$\overline{SN}$ , 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] = $Cost_{Upd}$(R(A), R(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
```

Figure 11. Tree edit distance algorithm.

```
Algorithm UserFeed()

Input: Grammar tree A, user 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($R(A_i) \in$ (preM, A, B))
      { A' = A' - $A_i$' }
      Else
      { $A_i$' = UserFeed($A_i$ , (preM, A, B)) }
  }
End
```

Figure 13. User feed transformation algorithm.

To sum up, the TED_{XGram} algorithm goes through the sub-trees of each of the grammar trees being compared. It exploits sub-tree insertion/deletion costs (via *SGS*) and update operations costs which reflect the similarities between each sub-tree in the source/destination trees being compared, in order to compute the overall distance (similarity) value.

5.2. XML Grammar Element Matchers

As mentioned previously, we make use of dedicated matchers to evaluate the similarities between XML grammar node labels, constraints, data-types, and ordering scores, their results being integrated in our XML grammar comparison method (cf. Definition 19). 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 criteria) following the task at hand. Table 2 depicts the matchers considered in our XML grammar matching approach so far, along with the different kinds of auxiliary information they exploit.

Matcher results can be combined in several ways, using for instance the *maximum*, *minimum*, *average* or *weighted sum* functions [15, 50]. Here, we also make use of the *weighted sum* function since it enables the user to choose the weight of each simple matcher in accordance with her notion of similarity (such as when computing the similarity between two XML grammar nodes, cf. Definition 19). Thus, for each of the composite matchers *CM* and its component mathchers $M_{i=1..n}$, similarity is evaluated as follows:

$$\text{Sim}_{\text{CM}} = f_{Agg}(\text{Sim}_{\text{Mi}}) = \sum_{i=1\ldots n} w_i \times Sim_{M_i} \in [0, 1] \quad \textbf{(9)}$$

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

Table 2. XML grammar element matchers.

Matcher			Type	Target	Auxiliary Information
Label			Composite - *Computational*	Labels	Weighted semantic network
	Syntactic		Composite - *Computational*	Labels	---
		String- ED	Simple- *Computational*	Labels	---
		N-Gram	Simple - *Computational*	Labels	---
	Semantic		Composite- *Computational*	Labels	Weighted semantic network
		Lin	Simple - *Computational*	Element labels	Weighed semantic network
		WuPalmer	Simple - *Computational*	Element labels	Semantic network
Data-Type			Simple - *Tabular*	Data-Types	Data-type compatibility table
Cardinality Constraint			Hybrid - *Tabular, Rule-based and Computational*	Cardinality constraints	Constraint compatibility table
Alternativeness Constraint			Hybrid - *Computational*	Alternativeness constraint vectors	Constraint compatibility table
OrdScore			Simple - *Computational*	Ordering scores	---

5.2.1. Label Matcher

It is a composite matcher encompassing, in turn two composite ones: the *Syntactic* and *Semantic* matchers for comparing element labels. While the former combines simple matchers that capture the syntactic resemblances between grammar node labels (e.g., *String-ED* [64] and *N-Gram* [28]), the latter makes use of methods for evaluating their semantic meaning (e.g., *WuPalmer* [67] and *Lin* [35]), by looking up their terminological relationships in a given semantic network [8]. The *Label* matcher computes label similarity as follows:

$$\begin{aligned} \text{Sim}_{\text{Label}}(\ell_1, \ell_2) &= f_{Agg}(\text{Sim}_{\text{Syn}}(\ell_1, \ell_2), \text{Sim}_{\text{Sem}}(\ell_1, \ell_2, \overline{SN})) \\ &= \text{w}_{\text{Syn}} \times \text{Sim}_{\text{Syn}}(\ell_1, \ell_2) + \text{w}_{\text{Sem}} \times \text{Sim}_{\text{Sem}}(\ell_1, \ell_2, \overline{SN}) \end{aligned} \quad \textbf{(10)}$$

$$\text{where } w_{Syntactic} + w_{Semantic} = 1 \text{ and } (w_{Syntactic}, w_{Semantic}) \geq 0.$$

$$\text{Note that } (Sim_{Syntactic}, Sim_{Semanticc}) \in [0, 1].$$

Likewise, for remaining composite matchers:

$$\text{Sim}_{\text{Syntactic}}(\ell_1, \ell_2) = f_{Agg}(\text{Sim}_{\text{String-ED}}(\ell_1, \ell_2), \text{Sim}_{\text{N-Gram}}(\ell_1, \ell_2))$$
$$= w_{\text{String-ED}} \times \text{Sim}_{\text{String-ED}}(\ell_1, \ell_2) + w_{\text{N-Gram}} \times \text{Sim}_{\text{N-Gram}}(\ell_1, \ell_2) \quad \textbf{(11)}$$
where $w_{String\text{-}ED} + w_{M\text{-}Gram} = 1$ and $(w_{String\text{-}ED}, w_{N\text{-}Gram}) \geq 0$.
Note that $(Sim_{String\text{-}ED}, Sim_{N\text{-}Gram}) \in [0, 1]$.

$$\text{Sim}_{\text{Semantic}}(\ell_1, \ell_2, \overline{SN}) = (w_{\text{WuPalmer}} \times \text{Sim}_{\text{WuPalmer}}(\ell_1, \ell_2, SN) + w_{\text{Lin}} \times \text{Sim}_{\text{Lin}}(\ell_1, \ell_2, \overline{SN})) \quad \textbf{(12)}$$
where $w_{WuPalmer} + w_{Lin} = 1$ and $(w_{WuPalmer}, w_{Lin}) \geq 0$.
Note that $(Sim_{WuPalmer}, Sim_{Lin}) \in [0, 1]$

Note that the *Label* (*Syntactic* and *Semantic*) composite matcher is extensible to additional matching techniques and processes [28] (for instance, a *Phonetic* matcher could be integrated so as to identify the similarity between labels based on their soundex codes [15]).

5.2.2. Data-type Matcher

This matcher is *tabular* (in contrast with its *computational* predecessors) and makes use of a dedicated compatibility table to assess the similarity between schema elements. Note that a *tabular* matcher derives its similarity/distance values by retrieval only (i.e., no computations are involved). The similarity/distance between every two values of the domain is stored in a table (here the *DTCT* table), and the matcher simply searches the table to retrieve the values [45].

Table 3. Default DTCT (Data-Type Compatibility Table).

	PCDATA	**String**	**Decimal**	**ANY**	***Composite***	**CDATA**	**ID**	**…**
PCDATA	1	0.95	0.6	0.6	0.5	0.8	0.4	
String	0.95	1	0.7	0.6	0.5	0.7	0.4	
Decimal	0.7	0.7	1	0.6	0.5	0.5	0.4	
ANY	0.6	0.6	0.6	1	0.7	0.4	0.4	
Composite	0.5	0.5	0.5	0.7	1	0.4	0.4	
CDATA	0.8	0.8	0.5	0.4	0.4	1	0.7	
ID	0.4	0.4	0.4	0.4	0.4	0.7	1	
…								

Note that Table 3 comprises sample values which can be utilized as default input. Nonetheless, different data-type compatibility values can be provided by the user. In other words, the user can adapt data-type similarities according to the context at hand, but should do so consistently through the whole *DTCT* table. In Table 3 for example, $Sim_{Data\text{-}Type}$(*'#PCDATA', 'CDATA'*) $= 0.8$ underlines that both data-types allow string data values. Nonetheless, similarity is not maximal (*=1*) since *#PCDATA* corresponds to grammar (DTD) elements whereas *CDATA* describes attribute contents. This is also reflected by $Sim_{Data\text{-}Type}$(*'ID, 'CDATA'*) > $Sim_{Data\text{-}Type}$(*'ID', '#PCDATA'*) which underlines that data-types *ID* and *CDATA* both correspond to attributes whereas *#PCDATA* is an element type.

Similarly to the reference semantic network (exploited in the *Semantic* label matcher), the data-type compatibility table (*DTCT*) is considered as auxiliary information (exploited in the *Data-type* matcher) following our grammar matching approach, and thus is to be provided by the user prior to executing the match task. Corresponding values reflect the user's perception of data-type similarities, such as $Sim_{Data\text{-}type} \in [0, 1]$.

5.2.3. Cardinality Matcher

It is a hybrid matcher, including *tabular* and *rule-based computational* matching features, to assess the similarity between XML grammar element cardinality constraints. It exploits *CCT* (cf. Table 4) in identifying constraint compatibility scores. Yet, in contrast with *DTCT* (Table 3), not all values in *CCT* are pre-computed. That is due to the presence of the *MinOccurs* and *MaxOccurs* operators of the XSD language [49]. Thus, a rule-based feature component is introduced to evaluate the compatibility of *MinOccurs* and *MaxOccurs* values w.r.t. remaining cardinality operators:

- Rule 1: $Sim_{CConstraint}$(‘?’, *minOccurs=0*) = 1
- Rule 2: $Sim_{CConstraint}$(‘+’, *maxOccurs=‘unbounded’*) = 1
- Rule 3: $Sim_{CConstraint}$(‘*’, *minOccurs=0* $\wedge$ *maxoccurs=‘unbounded’*) = 1
- …

In addition, a computational feature component is exploited to evaluation the compatibility between the infinite number of *minoccurs* and *maxoccurs* configurations themselves. It comes down to evaluating the similarity between their corresponding values.

$$Sim_{CConstraint}(MinOccurs = x \wedge MaxOccurs = y, MinOccurs = x' \wedge MaxOccurs = y') = \frac{\left(1 - \frac{|x - x'|}{|x| + |x'|}\right) + \left(1 - \frac{|y - y'|}{|y| + |y'|}\right)}{2} \quad \textbf{(13)}$$

For instance, $Sim_{CConstraint}(minoccurs = 1, minoccurs = 5) = 1 - \frac{|1-5|}{|1|+|5|} = 0.3333$. Likewise for *maxoccurs*.

Table 4. Default CCT (Constraint Compatibility Table).

	?	*	+	Implied	Required	*null*	*minOccurs* ¦ *maxOcurs*
?	1	0.5	0.5	1	0.8	0.8	Rule-based compatibility values
*****	0.5	1	0.8	0.5	0.5	0.5	
+	0.5	0.8	1	0.5	0.5	0.5	
Implied	1	0.5	0.5	1	0.8	0.8	
Required	0.8	0.5	0.5	0.8	1	1	
null	0.8	0.5	0.5	0.8	1	1	
minOccurs / **maxOccurs**	Rule-based compatibility values						Computational values

Similarly to *DTCT* in Table 3, Table 4 comprises of sample values which can be utilized as default *CCT* input. Nonetheless, different input compatibility scores can be provided by the user. The *CCT* table is also required as input auxiliary information to be provided by the user prior to executing the match task. Corresponding values reflect the user’s perception of cardinality constraint similarities, such as $Sim_{CConstraint} \in [0, 1]$.

5.2.4. Alternativeness Constraint Vector Matcher

It is based on the classic Wagner-Fisher string edit distance algorithm [64] and compares alternativeness constraint vectors as series of simple/composite alternativeness constraints underscoring the disposition of an XML grammar element w.r.t. its siblings and parent node in the grammar (cf. Definition 14). In other words, it can be viewed as a special string matcher where characters stand for alternativeness constraints. The main idea is to identify the number of edit operations (simple *insertions*, *deletions* and *updates*) necessary to transform one alternativeness constraint vector ($\overrightarrow{ac}_1$) into another ($\overrightarrow{ac}_2$).

With classic string edit distance, string characters are considered unrelated. However, alternative constraints share similarities due to associated cardinality constraints. Recall that an alternativeness constraint is represented as a doublet (*sac, cc*) where *sac* is the simple alternativeness operator (*And*, *Or*, or *All*) and *cc* is the associated cardinality constraint (e.g., ‘*?*’, ‘+’, *Minoccurs*, cf. Definition 13). For instance, $\overrightarrow{ac}_1 = \langle(And, +)\rangle$ and $\overrightarrow{ac}_2 = \langle(And, *)\rangle$ are similar having $\overrightarrow{ac}_1.sac = \overrightarrow{ac}_2.sac = And$. Their similarity comes down to that of their corresponding cardinality constraints, i.e., $\overrightarrow{ac}_1.cc = +$ and $\overrightarrow{ac}_2.cc = *$. Such similarities are considered in our matcher, by varying operations costs accordingly. Thus, in order to compare alternativeness constraint operators, we vary the cost of the update operation, in the Wagner-Fisher algorithm [64], as follows (*insertion* and *deletion* operations maintaining unit costs):

$$Cost_{Upd}(\overrightarrow{ac}_1, \overrightarrow{ac}_2) = \begin{bmatrix} 1 & \textit{if } \overrightarrow{ac}_1.sac \neq \overrightarrow{ac}_2.sac \\ 1 - Sim_{CConstraint}(\overrightarrow{ac}_1.cc, \overrightarrow{ac}_2.cc) & \textit{otherwise} \end{bmatrix} \quad \textbf{(14)}$$

For instance, comparing alternativeness constraint operators corresponding to the elements of label *'Publisher'* in grammar trees *P* (*Paper.dtd*) and *Q* (*Publication.xsd*) in Figure 5, such as $\overrightarrow{ac}_1 = \langle Or, And \rangle$ and $\overrightarrow{ac}_2 = \langle And \rangle$, yields:

– $\text{Sim}_{\text{AConstraint}}(\overrightarrow{ac}_1, \overrightarrow{ac}_2) = \dfrac{1}{1 + ED_{WagnerFisher}(\overrightarrow{\text{ac}}_1, \overrightarrow{\text{ac}}_2)} = 0.5$, having $ED_{Wagner\text{-}Fisher} = 1$ the cost of deleting constraint *Or* transforming $\overrightarrow{ac}_1$ to $\overrightarrow{ac}_2$.

If $\overrightarrow{ac}_2$ was equal to $\langle(And, ?)\rangle$, the similarity would be computed as follows:

– $\text{Sim}_{\text{AConstraint}}(\overrightarrow{ac}_1, \overrightarrow{ac}_2) = \dfrac{1}{1 + ED_{WagnerFisher}(\overrightarrow{\text{ac}}_1, \overrightarrow{\text{ac}}_2)} = 0.4546$, having $ED_{Wagner\text{-}Fisher} = 1.2$, which is the sum of the costs of deleting *Or* and updating *And* transforming it to *And?* ($\text{Sim}_{\text{CConstraint}}(null, ?) = 0.8$ following Table 4).

Note that our *Alternativeness Constraint Vector* comparison process is viewed as a hybrid matcher that interacts with its *Cardinality Constraint* counterpart (computing $Sim_{CConstraint}$ values) when comparing alternativeness constraint vectors, such as $Sim_{AConstraint} \in [0, 1]$.

5.2.5. Ordering Score Matcher

It compares the ordering scores of two XML grammar nodes, such as those having similar scores (similar initial positions) would constitute better match candidates. Recall that in our tree representation, XML grammar nodes are sorted left-to-right by node label, assigned each an ordering score *Ord* in the *[-1, 1]* interval, reflecting the reordering magnitude and direction of the node at hand. This allows considering both ordered and unordered parts of the XML grammar in the comparison process, and producing more meaningful mappings (cf. Section 4.2). Hence, our *OrdScore* matcher handles the task of comparing two *Ord* scores as follows:

$$\text{Sim}_{\text{OrdScore}}(Ord_1, Ord_2) = 1 - \frac{|Ord_1 - Ord_2|}{2} \in [0, 1] \quad \text{where } Max(|Ord_1| + |Ord_2|) = 2 \qquad \textbf{(15)}$$

Note that we normalize ordering similarity by the maximum sum of the ordering scores (i.e., *2*) instead of the actual sum itself (i.e., $|Ord_1| + |Ord_2|$) since the latter would unanimously yield zero values whenever one of the scores involved in the comparison is equal to zero (regardless of the other score) which is not accurate.

For instance, the ordering score similarity between nodes of labels *'Author'* in grammar trees *P* (*Paper.dtd*) and *Q* (*Publication.xsd*) is $Sim_{OrdScore} = 1 - \frac{0.2857}{2} = 0.8572$. This designates the difference in ordering positions between the node in *P* (which retained a zero score after the ordering phase, since it is connected to its sibling *'Publisher'* via an *Or* alternativeness constraint, i.e., it is unordered w.r.t. '*Publisher*') and that of *Q* (which changed positions). By normalizing via the actual sum of the ordering scores, we would have achieved $Sim_{OrdScore}=0$ which is least accurate. On the other hand, nodes of labels *'FirstName'* and *'First'* in trees *P* and *Q* respectively have identical ordering positions. Hence, corresponding $Sim_{OrdScore}$ is maximal (=1). Recall that the ordering score is essential in our tree model, enabling the distinction between ordered and unordered siblings (attributes, as well as elements connected via the *Or/All* alternativeness constraints), and the comparison of sibling positions in the case of ordered nodes, such as $Sim_{OrdScore} \in [0, 1]$.

5.3. Edit Script Extraction and Mapping Identification

Identifying the similarity between two XML grammars is useful in applications such as grammar clustering [3, 33], and can be exploited as a pre-processing to the schema integration phase [51]. Yet, the grammar matching operation itself requires identifying element correspondences, which is where edit distance mappings come to play. The *XML Grammar Tree Comparison* component returns the edit distance between two XML grammar trees, in other words the overall similarity value between the grammars. However, identifying the mappings (cf. Definition 5) requires a post-processing of the tree edit distance result. This comes down to identifying/extracting the edit script.

5.3.1. 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, (identified as {*Dist[][]*} in the TED_{XGram} algorithm, cf. Figure 11) tracing the edit script operations costs. Our algorithm for identifying the minimum cost tree edit script is provided in Figure 12. It considers as input the XML grammar trees being compared as well as the related edit distance matrixes computed in TED_{XGram}. It outputs the corresponding edit script (simplified tree operation syntaxes are shown in Figure 12 for ease of algorithm presentation) yielding the minimum amount of modifications to the source grammar tree. As it traverses the edit distance matrixes, the algorithm identifies corresponding tree insertion/deletion and node update operations, gradually building the edit script. Note that while different minimum edit scripts might exist, we designed our algorithm to identify the one which prioritizes deletion operations (i.e., tree deletions - lines 8 and 27 - are treated prior to tree insertion operations - lines 13 and 32), so as to reduce the number of node match candidates, by reducing the number of nodes in the source grammar tree involved in the edit distance mapping. Reducing matching candidates would help reduce the number of erroneous results, and thus amend match quality (as we will show in Section 5.4). The edit script operations' order is inversed (cf. Figure 12, line 38), due to the backward processing of the edit distance matrixes. Note that the operations' order is insignificant regarding the mappings, but is relevant w.r.t. the edit distance result (e.g., operations applied on inner nodes should appear before those applied on leaf nodes in the edit script).

Consider trees *D* and *T* in Figure 8 and the corresponding edit distance computations developed in Table 1. Based on the distance matrixes, *ESDiscovery* generates the following edit script: *ES*(*D, T*) = *Upd*(*D[0], T[0]*), *Upd*(*D[2], T[2]*), *Upd*(*D[3], T[3]*), *DelTree*(*D[4]*), *InsTree*(T_2), *Upd*(*D[6], T[8]*), identifying the edit operations to be applied to XML grammar tree *D* in order to transform it into *T*. Then, XML grammar tree mappings are deduced from the corresponding minimum cost edit script, depicting which edit operations apply to which nodes in the two grammar trees being compared.

5.3.2. 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. These mappings indicate which elements of S_1 are related (i.e., similar) to those of S_2 and vice-versa. With tree edit distance, mappings are inferred from the edit script (Definition 5). Thus, producing the mappings between two trees basically comes down to generating the edit script transforming the source tree into the destination one.

Tree edit distance mappings depend on the edit distance operations that are allowed and how they are used. Recall that in our tree edit distance component, we utilize five edit operations: *insert node*, *delete node*, *update node*, *insert tree* and *delete tree*. Hence, the mapping between two XML grammar trees S_1 and S_2 is constructed as follow.

- Simple *1:1* mappings connect:
    - Initially matching nodes. Two nodes of S_1 and S_2 initially match if they are identical (i.e., nodes with identical labels, constraints, data-types, relative order and hierarchical depth).
    - Nodes related by the update operation.

– Complex *1:1*, *1:n*, *n:1* or *n:n* mappings connect:
  - Sub-trees of S_1 that are affected by the *tree deletion* operation, to similar sub-trees in S_2. Such edges are identified when computing the similarity between sub-trees of S_1 and S_2 (cf. mapping example below). 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 sub-tree being inserted 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 *insert/delete tree* operations and are not directly employed in the main tree edit distance algorithm (cf. algorithm TED_{XGram} in Figure 11). For instance, *Del*(*D[4]*) ≡ *DelTree*(*D[4]*) in our running example.

Figure 14 depicts the mapping results corresponding to the edit distance computations (developed previously) between grammar trees *D* and *T* of Figure 8. Recall the edit script transforming tree *D* into *T*:

– *ES*(*D*, *T*) = ≺*Upd*(*D[0]*, *T[0]*), *Upd*(*D[2]*, *T[2]*), *Upd*(*D[3]*, *T[3]*), *DelTree*(*D[4]*), *InsTree*(T_2), *Upd*(*D[6]*, *T[8]*)≻

We only show node labels in Figure 14 for ease of presentation. Mappings are shown in Table 5 (Mapping scores underline node/sub-tree similarity values and will be addressed in the following).

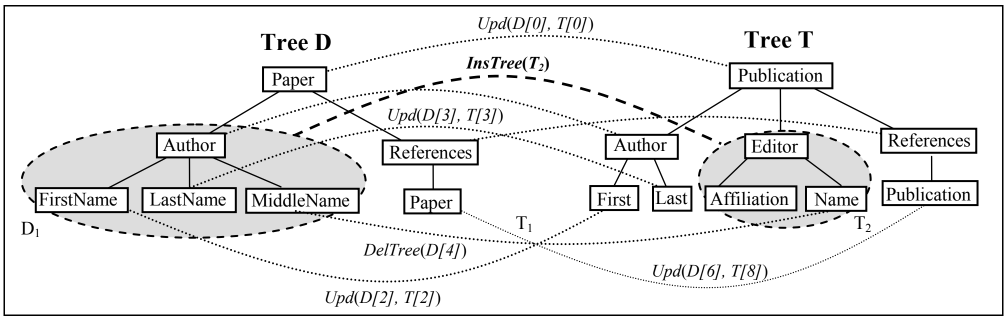


Figure 14. XML grammar tree mappings.

Table 5. Matching elements corresponding to grammar trees *D* and *T* of Figure 8.

Local match cardinality	**Elements of tree D**	**Elements of tree T**	**Mapping Scores**
1:1	D[0] (D[0].ℓ = '*Paper*')	T[0] (B[0].ℓ ='*Publication*')	0.1667
	D[1] (D[1].ℓ = '*Author*')	T[1] (T[1].ℓ ='*Author*')	1
	D[2] (D[2].ℓ = '*FirstName*')	T[2] (T[2].ℓ ='*First'*)	0.8519
	D[3] (D[3].ℓ = '*LastName*')	T[3] (T[3].ℓ ='*Last*')	0.8571
	D[4] (D[4].ℓ = '*MiddleName*')	T[6] (T[6].ℓ = '*Name*')	0.4628
	D[5] (D[5].ℓ = *'References'*)	T[7] (T[7].ℓ = '*References*')	1
	D[6] (D[6].ℓ = *'Paper'*)	T[8] (T[8].ℓ = *'Publication'*)	0.1667
n:n	D[1], D[2], D[3], D[4] (sub-tree D_1)	T[4], T[5], T[6] (sub-tree T_2)	0.4142

Each of the nodes *D[0]*, *D[1]*, *D[2]*, *D[3]*, *D[4]*, *D[6]* and *T[0]*, *T[1]*, *T[2]*, *T[3]*, *T[7]*, *T[8]* in grammar trees *D* and *T* participates in an individual *1:1* local mapping. In addition, *D[1]*, *D[2]*, *D[3]*, *D[4]*, and *T[4]*, *T[5]*, *T[6]* participate in a local *n:n* mapping. Thus, the matching result between trees *D* and *T* yields *1:1* and *n:n* local cardinalities, as well as a *1:n* global mapping, each of the nodes *D[1]*, *D[2]*, *D[3]* in tree *D* participating in more than one individual mapping (two mappings to be exact, one *1:1* and one *n:n*).

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 those of the module that accepts the mapping results. In general, existing matching approaches tend to focus on *1:1* local (and global) mappings [18]. 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 [6] and schema mediation [55]. Thus, we provide the user with a flexible framework able to produce either:

- *1:1* local and global mappings (most restricted case, especially useful for query discovery-related applications [43]),
- *1:1* local mappings (simplifying complex *1:n*, *n:1* and *n:n* mappings, with no specific restrictions on global cardinality),
- All kinds of mappings (no cardinality restrictions).

On one hand, restricting mapping cardinalities to both local and global *1:1* means disregarding all kinds of sub-tree similarities and repetitions when comparing the XML grammar trees. To achieve this, we only disable algorithm *SGS* and make use of the main tree edit distance algorithm TED_{XGram} (cf. overall method architecture in Figure 6). 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 those corresponding to matched nodes (*1:1*) and update operations (strictly producing *1:1* mappings). On the other hand, restricting mappings to *1:1* local cardinality is undertaken by decomposing the global sub-tree related mappings (e.g., mapping connecting sub-trees D_1 and T_2 in our running example). This is achieved by recursively running the *XML Grammar Tree Comparison* process on the concerned sub-trees (D_1 and T_2 in our running example), until single node *1:1* mappings are obtained. Duplicate mappings, i.e., mappings that already exist prior to the execution of the *Edit Distance* recurrence, are disregarded.

Table 6.a and Table 6.b show the mapping results obtained when restricting mapping cardinalities to *1:1* local/global and *1:1* local respectively (note that mapping scores hereunder underline node similarity values, cf. Section 5.3.3). One can see that while mappings are of local *1:1* cardinality in Table 6.b, they are obviously of global *1:n*. Additional mappings in Table 6.b underline the decomposition of the *n:n* local mapping between sub-trees D_1 and T_2 (cf. Table 5), and represent their corresponding edit script: $ES(D_1, T_2)$ = *Upd*(*D[1], T[4]*), *Upd*(*D[2], T[5]*), *Upd*(*D[3], T[6]*), *Ins*(*D[4]*). The last mapping in Table 6.b corresponds to operation *Ins*(*D[4]*) of $ES(D_1, T_2)$ and is automatically disregarded since it already exists in the set of mappings.

Table 6. One-to-one *1:1* cardinality mappings.

a. Local and global *1:1*

Elements of tree D	Elements of tree T
D[0] (ℓ =’*Paper*’)	T[0] (ℓ =‘*Publication*’)
D[1] (ℓ =’*Author*’)	T[1] (ℓ =‘*Author*’)
D[2] (ℓ =’*FirstName*’)	T[2] (ℓ =‘*First’*)
D[3] (ℓ =’*LastName*’)	T[3] (ℓ =‘*Last*’)
D[4] (ℓ =‘*MiddleName*’)	T[6] (ℓ = ‘*Name*’)
D[5] (ℓ = *‘References’*)	T[7] (ℓ = ‘*References’*)
D[6] (ℓ = *‘Paper’*)	T[8] (ℓ = *‘Publication’*)

b. Local *1:1* mappings (no restriction on global cardinality)

Elements of tree D	Elements of tree T	Mapping cores
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’*)	0.8519
D[3] (ℓ =’*LastName*’)	T[3] (ℓ =‘*Last*’)	0.8571
D[4] (ℓ =‘*MiddleName*’)	T[6] (ℓ = ‘*Name*’)	0.4628
D[5] (ℓ = *‘References’*)	T[7] (ℓ = ‘*References’*)	1
D[6] (ℓ = *‘Paper’*)	T[8] (ℓ = *‘Publication’*)	0.1667
D[1] (ℓ =*‘Author’*)	T[4] (ℓ= *‘Editor’*)	0.3333
D[2] (ℓ =*‘FirstName’*)	T[5] (ℓ = *‘Affiliation’*)	0.6153
D[3] (ℓ = *‘LastName’*)	T[6] (ℓ = *‘Name’*)	0.7857
D[4] (ℓ =*‘MiddleName’*)	T[6] (ℓ = *‘Name’*)	0.4628

To sum up, our approach allows the user to choose whether to produce *1:1* local and global mappings (disregarding *SGS*), *1:1* local mappings (dissecting composite sub-tree related mappings by recursively running the *Edit Distance* component), or more complex *1:n*, *n:1* and *n:n* mappings (applying our matching approach without any restrictions).

5.3.3. 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 matches (*0* for strong dissimilarity, *1* for strong similarity, and values in between). In addition, these values can be normalized to produce an overall score underlining the similarity between the two schemas being matched.

With respect to edit distance, mapping scores denote, in a roundabout way, the costs of the edit operations inducing the corresponding mappings:

– Mappings linking nodes that initially match (nodes that are identical) are assigned a maximum similarity value, i.e., *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 allowable update operation cost (*Formula (8)*). In other words, the mapping score designates the similarity value between the concerned nodes, i.e., $Sim_{GNode}(n, m, Aux) \in [0, 1]$.
– Following the same logic, mappings corresponding to tree insertion/deletion operations are assigned scores as follows:

$$\text{MapScore} = \frac{\sum\limits_{\text{All nodes } x \in \text{S}} \text{Cost}_{\text{Ins/Del}}(x) - \text{Cost}_{\text{InsTree/DelTree}}(\text{S})}{\sum\limits_{\text{All nodes } x \in \text{S}} \text{Cost}_{\text{Ins/Del}}(x)} \in [0,1] \qquad \textbf{(16)}$$

having $\sum\limits_{\text{All nodes } x \in \text{S}} \text{Cost}_{\text{Ins/Del}}(x)$ the maximum tree insertion/deletion operation cost for the sub-tree at hand. Hence, as the similarities between inserted/delete sub-trees and the source/destination XML grammar trees increase/decrease, the mapping scores will follow accordingly.

Note that for the special case of recursive leaf nodes, the mapping score between two recursive nodes being matched comes down to that of their referenced nodes, if the latter are matched. Otherwise, the recursive nodes retain the score computed based on their corresponding edit operation. For instance, in our running example, recursive leaf nodes *D[6]* (*ℓ='Paper')* and *T[8]* (*ℓ='Publication')* are linked via an update operation, of cost *0.2778* (cf. Section 5.1.2 for computational details). Nonetheless, they are assigned the mapping score of their referenced nodes, since the latter are matched (cf. Table 5). This process is in line with [33, 60] (i.e., the main XML grammar matching studies to consider the case of recursive declarations) allowing for the recursive nodes to both: i) contribute to the similarity of their reference nodes (recursive elements being represented as descendents of their referenced nodes, and thus contributing to the latter's edit distance computation result), ii) and obtain a mapping score based on the similarity of the reference nodes, if the latter are matched (cf. Section 4.4).

The overall similarity score between the grammar trees being compared is computed based on the obtained global edit distance value. Similarity measures based on edit distance are generally computed as follows:

$$\text{Sim}_{\text{XGram}}(\text{T}_1, \text{T}_2) = \frac{1}{1 + \text{TED}_{\text{XGram}}(\text{T}_1, \text{T}_2, \text{Aux})} \qquad \textbf{(17)}$$

Computation Example: Table 5 and Table 6 show the mappings generated in our running example and related mapping scores. For instance, $MapScore(D[0], T[0]) = 1 - Cost_{Upd}(D[0], T[0]) = Sim(D[0], T[0]) = \frac{'Paper' \cap 'Publication'}{'Paper' \cup 'Publication'} = 0.1667$ (using the *1-Gram* string matching criterion, with *Aux=Ø*). Likewise for remaining *1:1* matches based on the update operation. Nodes that initially match (e.g., *D[1]* and *T[1]*, having *D[1].ℓ = T[1].ℓ = 'Author'*) are assigned a unit mapping score (maximum similarity). As for *1:n*, *n:1* or *n:n* mappings, such as the one linking sub-trees D_1 and T_2, $MapScore = \frac{\sum\limits_{\text{All nodes } x \text{ of } T_2} Cost_{Ins}(x) - Cost_{InsTree}(T_2)}{\sum\limits_{\text{All nodes } x \text{ of } T_2} Cost_{Ins}(x)} = \frac{3 - 1.7203}{3} = 0.4142$, $Cost_{InsTree}(T_2)$ being identified w.r.t. T_2's similarity with D_1 (cf. Section 5.1.2 for $Cost_{InsTree}(T_2)$ computation details).

Hence, the overall similarity score between trees D and T of our running example (using the *1-Gram* label syntactic matcher, with $Aux=\emptyset$) is $Sim_{XGram}(D, T) = 0.2263$, having $TED_{XGram}(D, T) = 3.4189$ (following *Formula (6)*).

In addition to the *XML Grammar Tree Comparison* and *Mapping Identification* components, our matching framework encompasses a *UserFeed* component which enables the users to manually match a few hard-to-match XML grammar nodes.

5.4. User Constraints and User Feedback

Considering user input constraints and user feedback in the XML grammar matching process could further improve matching accuracy. Predefined user mappings happen to be particularly useful in matching ambiguous schema elements [18].

Consider for instance elements of labels '*url*' and '*Link*' in grammars *Paper.dtd* and *Publication.xsd* of Figure 4 respectively (cf. Figure 5 for corresponding tree representations). These elements encompass labels which are neither syntactically nor semantically similar (namely when using a generic WordNet-based semantic network where the word '*url*' does not exist). 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*', nodes being identified by their labels here for simplicity), which is where user constraints and feedback 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 (cf. overall method architecture in Figure 6). In essence, we consider user input constraints and user feedback in our matching framework by updating input XML grammar trees following the constraints at hand, and consequently comparing the updated trees. To do so, we define the *UserFeed* transformation operation as follows.

Definition 21 – 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 XML grammar trees being compared,
- $(preM, A, B)$ 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 corresponding grammar tree roots,
- A' is the transformed tree, such as $A' = A - \{\text{the set of sub-trees } A_i \,/\, R(A_i) \in (preM, A, B)\}$ ●

Sub-trees rooted at the manually matched nodes are eliminated from the grammar trees since structural matching, particularly tree edit distance, is sibling and ancestor order preserving [58]. In other words, given a node i_1 participating in mapping $i_1 \leftrightarrow j_1$, a sibling of i_1 occurring after i_1 in the source grammar cannot match a node occurring before j_1 in the destination grammar (otherwise, the matching would not be order preserving). In addition, the descendent of a given node i_1, i_1 participating in mapping $i_1 \leftrightarrow j_1$, cannot match a node outside j_1's sub-tree (otherwise, it would not be a structurally sound mapping, disregarding the ancestor/descendent relationship). Following the user, these sub-trees can be henceforth independently evaluated for mapping identification, depending on the mapping cardinality of choice (Section 5.3.2).

Consider for instance the XML grammar trees D and T and corresponding mappings in Figure 14 (Table 5). Here, the user might want to specify (before, or after the first execution of the edit distance matching process) that nodes entitled *'Author'* in both grammars actually match, or she might prefer to indicate that node *'Editor'* in grammar tree T does not match any node in D (i.e., $Null \leftrightarrow T[4]$). In the first case, sub-trees D_1 and T_1 would be truncated from D and T respectively, and could be processed independently for mapping evaluation. In the second case, sub-tree T_2 would be truncated from tree D prior to the matching process. Both cases would lead to the elimination of the final mapping in Table 5, i.e., the one linking sub-trees D_1 and T_2, preserving remaining matches (which would be what the user intended to obtain).

Thus, our *Edit Distance* component compares the transformed grammar trees, where nodes corresponding to predefined matches are eliminated, along with their corresponding sub-trees (preserving sibling and ancestor order [29]). 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 13. User mappings are consequently added to those produced by the system: $(M, A, B) = (SystemM, A, B) \cup (preM, A, B)$.

5.5. Complexity Analysis

The overall time complexity of our XML grammar matching and comparison approach simplifies to $O(|T_1| \times |T_2| \times |SN| \times Depth(SN))$, where $|T_1|$ and $|T_2|$ denote the cardinalities of the compared trees, $|SN|$ the cardinality of the weighted semantic network exploited for semantic similarity assessment, and *Depth(SN)* its depth. Overall complexity is evaluated as the sum of the complexities of the various components constituting our XML grammar comparison method, and comes down to $XGramTreeRepresentation_{Complexity}$ + $XGramTreeComparison_{Complexity}$ + $MappingIdentification_{Complexity}$ + $UserFeed_{Complexity}$, the $Matchers_{Complexity}$ factor being encompassed in that of the *Tree Comparison* component.

- Transforming an XML grammar into its tree representation is undertaken in average linear time w.r.t. the number of elements/attributes in the grammar, and comes down to the complexity of the *SiblingOrdering* algorithm in Figure 3, i.e., $O(|T| \times log(|T|))$.
- The complexity of our XML grammar tree comparison component is of $O(|T_1| \times |T_2|) \times Matchers_{Complexity}$, where $O(|T_1| \times |T_2|)$ underlines the complexity of the TED_{XGram} algorithm (Figure 11). It comes down to $O(|T_1| \times |T_2| \times |SN| \times Depth(SN))$, the complexity of the *matchers* component simplifying to that of the label semantic matchers (i.e., *WuPalmer* [67] and *Lin* [35]), which come down to $O(|SN| \times Depth(SN))$.
- The complexity of the mapping identification component comes down to that of the *ES_Extraction* algorithm (cf. Figure 12), which complexity simplifies to $O(|T_1|+|T_2|)$.
- The *User Feed* component allows a seamless integration of user-predefined mappings, and thus simplifies to $O(|T_1| \times |T_2|)$.

To sum up, note that the time complexity of our matching approach simplifies from to $O(|T_1| \times |T_2| \times |SN| \times Depth(SN))$ to $O(|T_1| \times |T_2|)$ (i.e., complexity of the edit distance process) when the composite *Semantic* matcher is disregarded, remaining matchers having marginal complexity levels with respect to the overall approach (as will be demonstrated in our timing experiments).

As for memory usage, our method requires RAM space to store the grammar trees being compared, as well as the distance matrixes and weighted vectors being computed. It simplifies to $O(|T_1| \times |T_2|)$.

6. Experimental Evaluation

We conducted various experiments to test and evaluate our XML grammar matching framework. In the following, we start by briefly describing our experimental prototype. Section 6.2 presents the test methodology and evaluation metrics adopted in our experimental evaluation process. Section 6.3 details our matching experiments. Performance analysis is presented in Section 6.4.

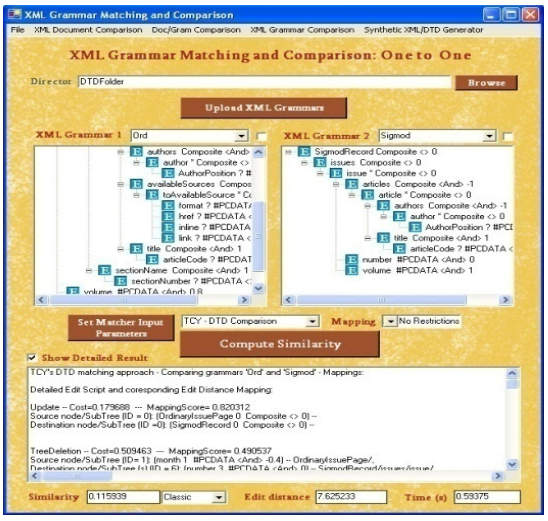
6.1. Prototype

We have implemented our XML grammar matching framework in the experimental *XS3* prototype (*XML Structural and Semantic Similarity*). The system includes four main grammar comparison modules: *One to One*, *One to Many*, *Many to Many* (consequently allowing the clustering of similar XML grammars) and *Set comparison* (computing average inter-set and intra-set similarities, and therefore allowing XML grammar classification).

For each of the modules above, the user starts by providing her matcher weights and corresponding auxiliary information if available (i.e., reference semantic network as well as constraint and data-type compatibility tables).

Note that an extract of the *WordNet* taxonomy and predefined compatibility tables are provided as default inputs by the system. A prototype snapshot is shown in Figure 15.

In addition to our approach, we have implemented three of its most prominent alternative methods proposed in the literature: *COMA* [15], *XClust* [33] and *Relaxation Labeling* [69], so as to compare our approach w.r.t. existing XML grammar matching and comparison solutions. The *XS3* prototype system is available online[1].


XML Grammar Matching and Comparison
File
XML Document Comparison
Doc/Gram Comparison
XML Grammar Comparison
Synthetic XML/DTD Generator
XML Grammar Matching and Comparison: One to One
Browse
Upload XML Grammars
Set Matcher Input Parameters
Compute Similarity

Director: DTDFolder

XML Grammar 1: Ord

```
authors Composite <And>
  author * Composite <>
    AuthorPosition ? #
availableSources Compos
  toAvailableSource * Cc
    format ? #PCDAT/
    href ? #PCDATA <
    inline ? #PCDATA
    link ? #PCDATA <
title Composite <And> 1
  articleCode ? #PCDAT
sectionName Composite <And> 1
  sectionNumber ? #PCDATA <
volume #PCDATA <And> 0.8
```

XML Grammar 2: Sigmod

```
SigmodRecord Composite <> 0
  issues Composite <> 0
    issue * Composite <> 0
      articles Composite <And> -1
        article * Composite <> 0
          authors Composite <And> -1
            author * Composite <> 0
              AuthorPosition ? #PCI
          title Composite <And> 1
            articleCode ? #PCDATA <
      number #PCDATA <And> 0
      volume #PCDATA <And> 1
```

TCY - DTD Comparison; Mapping: No Restrictions

[✓] Show Detailed Result

```
TCY's DTD matching approach - Comparing grammars 'Ord' and 'Sigmod' - Mappings:

Detailed Edit Script and coresponding Edit Distance Mapping:

Update -- Cost=0.179688 --- MappingScore= 0.820312
Source node/SubTree (ID = 0): (OrdinaryIssuePage 0 Composite <> 0) --
Destination node/SubTree (ID =0): (SigmodRecord 0 Composite <> 0) --


TreeDeletion -- Cost=0.509463 --- MappingScore= 0.490537
Source node/SubTree (ID= 1): (month 1 #PCDATA <And> -0.4) -- OrdinaryIssuePage/,
Destination node/SubTree (s) (ID = 6): (number 3 #PCDATA <And> 0) -- SigmodRecord/issues/issue/
```

Similarity: 0.115939 (Classic); Edit distance: 7.625233; Time (s): 0.59375

Figure 15. Snapshot of the XS3 *One to one* comparison interface.

6.2. Test Methodology and Evaluation Metrics

As stated previously, the main criterion used to assess the effectiveness of automatic schema matching methods is the amount of manual work and user effort required to perform the matching task. In this context, most existing approaches propose to first manually solve the match task, in order to exploit the obtained results as a reference to evaluate the quality of the matches produced by the system [17]. Thus, similarly to information retrieval, the *Precision* and *Recall* metrics can be utilized in comparing *'real'* and system generated matches.

Precision (*PR*) identifies the number of correctly generated matches, w.r.t. the total number of matches (correct and false) produced by the system. *Recall* (*R*) underlines the number of correctly identified matches, w.r.t. the total number of correct matches, including those not identified by the system. Having:

– *A* the number of correctly identified matches (true positives)
– *B* the number of wrongly identified matches (false positives)
– *C* the number of real matches not identified by the system (false negatives)

Precision and *recall* are computed as follows:

$$PR = \frac{A}{A+B} \in [0,1] \quad \text{and} \quad R = \frac{A}{A+C} \in [0,1] \tag{18}$$

High *precision* denotes that the matching task achieved high accuracy in identifying correct matches, whereas high *recall* means that very few correct matches where missed by the system.

In addition to comparing one approach's precision improvement to another's recall improvement, it is a common practice to consider a combined measure. The *F-value* measure was originally introduced in information retrieval, and consequently used in XML grammar matching [17]. It represents the harmonic mean of *precision* and *recall*. High *precision* and *recall*, and thus high *F-value* (indicating in our case high matching quality) characterize a good grammar matching and comparison method.

$$F\text{-}Value = \frac{2 \times PR \times R}{PR + R} \in [0,1] \tag{19}$$

[1] http://www.u-bourgogne.fr/DbConf/XS3

Another combined measure, named *Overall*, dedicated to schema matching, was introduced in [41]. Unlike classical information retrieval metrics, *Overall* was designed in such a way to attain negative values when the number of false positives (*B*) exceeds the number of true ones (*A*), i.e., *Precision<0.5*. A negative *Overall* underlines that half the matches generated by the system are wrong, and that it might (hypothetically) take the user more effort to remove the false positives (*B*) and add the false negatives (*C*) than to perform the whole matching by hand. *Overall* is maximized (*=1*) with maximum *Precision* and *Recall* (*PR=R=1*), indicating excellent matching quality.

$$Overall = R \times (2 - \frac{1}{PR}) \quad \in \ [-\infty, 1] \qquad \textbf{(20)}$$

The behavior of both *F-Value* and *Overall* measures is studied in detail in [17].

6.3. Matching Experiments

Recent efforts to building a common benchmark for evaluating the quality of XML grammar matching methods have been underlined in [19]. The *XBenchMatch* system described in [19] takes as input: i) the results of a grammar matching and integration algorithm and ii) the intended user output, and generates statistics about the quality of the input and the performance of the matching tool. While it measures matching quality (via *Precision* and *Recall* metrics, similarly to *XS3*), nonetheless, the *XBenchMatch* system is particularly geared toward grammar integration (i.e., constructing a global grammar encompassing the concepts contained in a set of corresponding grammars), and provides evaluation measures mainly dedicated to comparing the quality of integrated grammars (e.g., *structural overlap* measuring the number of nodes shared by each of the input grammars and the integrated grammar, the *backbone measure* identifying the size of the largest common sub-tree between the input and integrated grammars, etc.). In addition, *XBenchMatch* merely consists of a statistical computation platform, and does not provide predefined mapping results in order to conduct comparative matching tests. To our knowledge, *gold standard* mappings, for evaluating the quality of XML grammar matching methods, do not exist to date.

Hence, in order to evaluate the performance of our approach, we conducted a set of matching experiments using a select collection of real and synthetic XML grammars (including those exploited in our running example). Real grammars (DTDs and XML Schemas) were acquired from various online sources[1], including grammars exploited in previous evaluation studies, e.g., [15, 37, 63]. For each matching task, the grammars were first manually evaluated, identifying the set of user-relevant matching elements (three different test subjects, two doctoral students and two post-doctoral researcher, were involved in the experiment). Manual answers were consequently mapped to the system generated ones so as to compute *PR*, *R*, *F-Value* and *Overall* accordingly. Section 6.3.1 details the mapping results for the matching task considered in our running example. Section 6.3.2 provides *PR*, *R*, *F-Value* and *Overall* results for all matching experiments. Section 6.3.3 discusses the impact of user feedback of matching quality. Section 6.3.4 compares the results obtained using our method and three of its recent alternatives (i.e., *COMA* [15], *XClust* [33] and *Relaxation Labeling* [69]).

Details concerning all experimental results are provided in the technical report[2].

6.3.1. Evaluation of our Running Example

When matching grammars *Paper.dtd* and *Publication.xsd* (cf. Figure 4 and Figure 5), our method identified *6* correct mappings, disregarded *2*, and generated *2* incorrect ones (Table 7). 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 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).

[1] http://www.acm.org/sigmod/xml, http://www.cs.wisc.edu/niagara/, http://www.BizTalk.org, http://www.xmlfiles.com, etc.

[2] Available online at http://www.u-bourgogne.fr/Dbconf/XGM.

Despite some of the inconsistencies obtained 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 reduces the amount of user effort and is clearly easier than manually performing the matching. In all our matching tests, all basic matchers were considered with identical weights ($w_{Label} = w_{Cardinality} = w_{Data\text{-}Type} = w_{Alternativeness} = w_{Ord} = 0.2$ whereas $w_{String\text{-}ED} = w_{N\text{-}Gram} = w_{Lin} = w_{WuPalmer} = 0.5$). Extracts of *WordNet* were adopted as reference semantic networks, covering the grammars at hand, and default *DTCT* and *CCT* were exploited (cf. Table 3 and Table 4).

Table 7. Matching *Paper.dtd* and *Publication.xsd* of Figure 4.

Manual Mappings	
paper.dtd	publication.xsd
Paper	Publication
Author	Author
FirstName	First
LastName	Last
PaperLength	Length
Publisher	Publisher
Title	Title
Reference	Reference
$Paper_{Rec}$	$Publication_{Rec}$
Download	Link

System Mappings		
paper.dtd	publication.xsd	Scores
Paper	Publication	0.8863
Author	Author	0.9714
FirstName	First	0.8378
LastName	Last	0.7886
Publisher	Publisher	0.8433
Title	Title	0.8343
Reference	Reference	0.9857
$Paper_{Rec}$	$Publication_{Rec}$	0.8863
PaperLength	Link	0.7736
Genre	Length	0.7486

PR= 0.8   R = 0.8   F-Value=0.8   Overall = 0.6

Note that in this study, we do not address the issue of assigning different matcher weights, which could help the user fine-tune her input parameters to obtain optimal performance. This comes down to an optimization problem requiring a thorough analysis of the relative effect of each individual matcher and criterion on matching quality (similarly to [15]), as well as the evaluation of different optimization techniques to semi-automatically adapt and combine matcher weights (similarly to [50]). We report the latter to a dedicated empirical study.

6.3.2. Evaluation on Real World and Synthetic XML Grammars

The characteristics of the various test grammars[1] exploited in our experimental analysis are summarized in Table 8 (many of which have been exploited in previous evaluations [15, 37, 63]).

Table 8. Characteristics of test grammars.

Grammars	N# of nodes	Max depth	Ave depth
OrdinaryIssuePage.dtd [2]	23	7	3.913
SigmodRecord.dtd [2]	12	7	3.75
bookstore.dtd [5]	12	3	2.0833
bib.dtd [6]	14	3	1.8571
paper.dtd [7]	12	2	1.3334
publication.xsd [6]	12	2	1.25

Grammars	N# of nodes	Max depth	Ave depth
BizTalk_PO.xsd [3]	25	3	1.96
Oreilly_PO.xsd [4]	22	3	1.9091
BizTalk_CIDX_PO.xsd [2]	35	2	1.8
BizTalk_Excel_PO.xsd [2]	53	3	2.5283
MS_PO_Excel.xsd [8]	37	3	1.9456
Syn_PO.xsd [1]	47	3	2.2766

Hereunder, we present the results of 18 match tasks (including our running example, i.e., task# 3), each matching two different grammars (Table 9). To give an impression of the problem size in each match task, we estimate the actual *similarity ratio* between the compared schemas, as the number of user matches to be identified, w.r.t. the maximum number of elements in the grammars at hand. Results show that the tasks are almost equally partitioned between relatively disparate grammars (with similarity around *50%*) and more similar ones (similarity higher than *60%*). This is an indicator of the

[1] http://www.u-bourgogne.fr/Dbconf/XGM
[2] http://www.acm.org/sigmod/xml
[3] http://www.BizTalk.org
[4] http://www.xml.com/pub/a/1999/07/schemas/
[5] http://www.xmlfiles.com
[6] http://www.cs.wisc.edu/niagara/
[7] Running example
[8] http://office.microsoft.com/en-us/templates/

balanced context in which our matching experiments were conducted. Note that the *similarity ratio* has nothing to do with matching quality (the latter depending on the conformance between the user and system generated mappings, no matter how similar/different the grammars are). *PR*, *R*, *F-Value* and *Overall* results are shown in Table 9, and depicted in Figure 16.

In *12* of the *18* match tasks, our approach effectively identified most user mappings, while disregarding some, and generating a few false ones. In task # 2, our method achieved *PR=R=Overall=1* due to the high resemblance between the grammars being matched (*bookstore.dtd*[3] *and bib.dtd* [4]). 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 (following sibling order and ancestor/descendent relations) but which do not correspond to actual user mappings, the latter not necessarily verifying structural integrity. Recall that structural integrity in the context of semi-structured data (and tree edit distance) underlines that the descendent of a given node i_1, given that i_1 participates in mapping $i_1 \leftrightarrow j_1$, cannot match a node outside j_1's sub-tree (otherwise, it wouldn't be a structurally sound mapping), and that matching has to be undertaken w.r.t. sibling order.

Table 9. Our approach's *PR*, *R*, *F-Value* and *Overall* results.

Match Task	Similarity ratio (%)	Grammars		Precision	Recall	F-Value	Overall
1	52.17	*OrdinaryIssuePage.dtd*	*SigmodRecord.dtd*	0.9231	1	0.9600	0.9167
2	64.29	*bookstore.dtd*	*bib.dtd*	1	1	1	1
3	66.67	*paper.dtd* (running example)	*publication.xsd*	0.75	0.75	0.75	0.5
4	80	*BizTalk_PO.xsd*	*Oreilly_PO.xsd*	0.8261	0.95	0.8837	0.75
5	68.57	*BizTalk_CIDX_PO.xsd*	*Oreilly_PO.xsd*	0.6818	0.625	0.6522	0.3333
6	52.83	*BizTalk_Excel_PO.xsd*	*Oreilly_PO.xsd*	0.3409	0.5357	0.4167	-0.5000
7	52.83	*BizTalk_PO.xsd*	*BizTalk_Excel_PO.xsd*	0.2195	0.3214	0.2647	-0.8214
8	62.26	*BizTalk_CIDX_PO.xsd*	*BizTalk_Excel_PO.xsd*	0.625	0.7575	0.6849	0.303
9	77.14	*BizTalk_PO.xsd*	*BizTalk_CIDX_PO.xsd*	0.7586	0.8462	0.8000	0.5769
10	83.78	*MS_PO_Excel.xsd*	*BizTalk_PO.xsd*	0.7083	0.8947	0.7907	0.5262
11	58.49	*MS_PO_Excel.xsd*	*BizTalk_Excel_PO.xsd*	0.3818	0.7	0.4999	-0.4334
12	64.86	*MS_PO_Excel.xsd*	*BizTalk_CIDX_PO.xsd*	0.4595	0.7083	0.5574	-0.1248
13	54.05	*MS_PO_Excel.xsd*	*Oreilly_PO.xsd*	0.8696	1	0.9303	0.8500
14	44.68	*Syn_PO.xsd*	*Oreilly_PO.xsd*	0.5927	0.7619	0.6667	0.2383
15	48.94	*Syn_PO.xsd*	*BizTalk_PO.xsd*	0.4571	0.7619	0.5714	-0.1430
16	61.70	*Syn_PO.xsd*	*BizTalk_CIDX_PO.xsd*	0.6316	0.8276	0.7164	0.3449
17	67.92	*Syn_PO.xsd*	*BizTalk_Excel_PO.xsd*	0.2326	0.2778	0.2598	-0.6387
18	51.06	*Syn_PO.xsd*	*MS_PO_Excel.xsd*	0.5152	0.7083	0.5965	0.0418

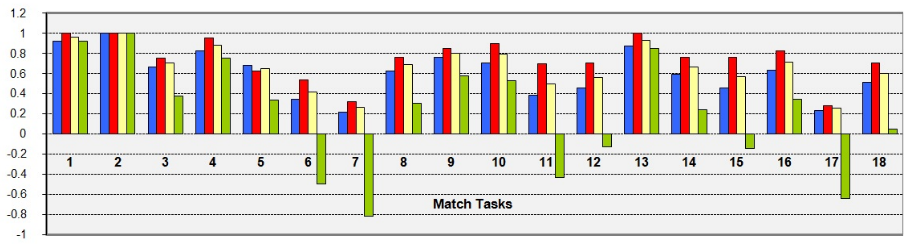


Figure 16. Graphical representation of our approach's *PR*, *R*, *F-Value* and *Overall* results.

Note that in cases where *Overall* is negative, *PR* is lesser than *0.5*, indicating that it might 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 high *PR* and *R* values.

6.3.3. Improvements via 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. Hereunder, we consider the six matching tasks where negative *Overall* was achieved in the initial matching phase (i.e., tasks n# *6*, *7*, *11*, *12*, *15* and *17*). For each task, we carried out three runs, providing an additional user input mapping at each run (note that the same three subjects who defined the reference user mappings, were asked to provide the user feedback). Results in Figure 17 show that user feedback positively affects matching accuracy, amending *Precision*, *Recall*, *F-Value* and *Overall* levels for all six matching tasks. Note that in tasks n# *11*, *12*, *15* and *17*, one can see that *Recall* gradually increases with user feedback (as excepted), but without surpassing the levels obtained in the initial (pre-feedback) phase (to the exception of task n#*12* - third run). In fact, in tasks *11*, *12*, *15* and 17, the system performs well in reducing the amount of incorrect mappings while producing matches, reflected by the higher *Precision* levels. At the same time, it identifies, in tasks *11*, *12*, *15* and *17*, less correct mappings in comparison with the initial (pre-feedback) phase. Yet, the consistently increasing *F-Value* levels (*F-Value* measuring the system's performance in both correctly disregarded (*PR*) and produced (*R*) mappings) indicate that the slight decrease in *Recall*, w.r.t. the initial matching phase, is compensated by a higher gain in *Precision*.

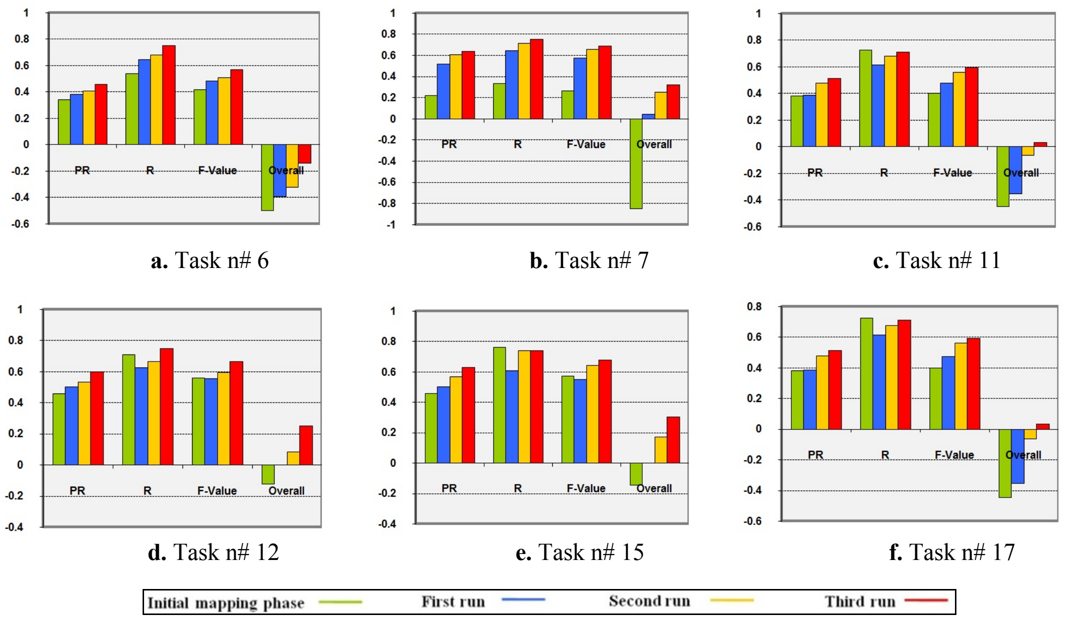


Figure 17. Comparing *PR*, *R*, *F-value* and *Overall* results for matching tasks n# 6, 7, 11, 12, 15 and 17 to evaluate the effectiveness of our approach in incorporating user feedback.

With respect to *Overall*, the system obtains positive values with three out of six tasks (tasks n# *7*, *12* and *15*), right after the first run (i.e., with only one user input mapping). In other words, in each of the tasks n# *7*, *12* and *15*, manually resolving one mapping has eliminated enough ambiguity for the system to produce more than half of the correct mappings. The system achieves positive *Overall* with all tasks, to the exception of n# *6* (Figure 17.a), after the third run (i.e., with three user input mappings). The *Overall* levels of task n# 6 were gradually amended by user feedback, but obviously require more user input mappings so as to cross the zero barrier (i.e., $PR > 0.5$). Note that the number of user input mappings required to amend *Overall* reflects the amount of structural heterogeneity and element mapping ambiguity amongst the grammars being compared: the more grammars are structurally similar and the lesser the schema element ambiguities, the lesser the number of input mappings needed to obtain positive *Overall*.

6.3.4. 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. On one hand, our method is i) dedicated to XML grammars, ii) considers the various kinds of XML grammar characteristics, iii) while being extensible to different matchers, which are crucial criteria required to minimizing user effort in undertaking the match task. On the other hand, most existing methods are either i) too generic, thus not adapted to the structured nature of XML grammars, ii) too restrictive, simplifying grammar constraints, or iii) too specific, i.e., not flexible and extensible to additional matching criteria. Table 10 summarizes the main differences between our method and its alternatives.

We experimentally compared our method's effectiveness to three of its most recent and efficient alternatives, i.e., *COMA* [15], *XClust* [33] and *Relaxation Labeling* [69]. Recall that *XClust* and *Relaxation Labeling* seem more sophisticated than alternative matching approaches since they induce the least simplifications to the grammars being compared (*XClust* only disregards the *Or* operator, whereas *Relaxation Labeling* allows restrictive alternative declarations), while *COMA* is one the most efficient methods to follow the *composite* matching logic, i.e., combining the results of several matching algorithms using simple mathematical formulations (in comparison with expensive learning-based methods, cf. background in Section 2.3).

Table 10. Comparing our method to alternative solutions.

Approaches	Considers cardinality constraints	Considers alternativeness constraints	Considers data-types	Considers recursive declarations	Extensible to several Matchers	Flexible w.r.t. mapping cardinalities	Dedicated to XML grammars
Cupid [37]	×	×	✓	×	×	× (1:1, 1:n)	×
Similarity Flooding [41]	×	×	✓	×	×	× (1:1)	×
LSD [18]	×	×	×	×	✓	× (1:1)	✓ (DTD)
NNPLS [30]	×	×	×	×	✓	× (undefined)	✓ (XSD)
Syntactic Similarity [60]	×	×	✓	✓	×	× (1:1)	✓ (DTD)
Porsche [55]	×	×	×	×	×	✓ (1:1, 1:n, n:1)	✓ (XSD)
XPruM [2]	×	×	✓	×	×	✓ (1:1, 1:n, n:1)	✓ (XSD)
COMA [15]	×	×	✓	×	✓	× (1:1)	×
XClust [33]	✓	×	×	✓	×	× (1:1)	✓ (DTD)
Relaxation Labeling [69]	×	✓ (restrictive)	✓ (restrictive)	×	×	× (1:1)	✓ (XSD)
Our Approach	✓	✓	✓	✓	✓	✓	✓

We ran each of the algorithms on the same 18 matching tasks described in Table 9. Note that with each of the alternative matching approaches, optimal input parameters, as indicated in their corresponding studies, were provided. In addition, the same *WordNet* extracts utilized with our approach were provided as reference semantic networks to *COMA* and *XClust*, both methods encompassing semantic-based label comparison measures. *PR*, *R*, *F-Value* and *Overall* results, are depicted in Figures 18-21 and Table 11. Our method's *PR*, *R*, *F-Value* and *Overall* results, when integrating user feedback, are also presented in the figures below. Improvements due to user feedback are depicted via a colored area reflecting their span from the results obtained with our fully automated approach, the latter being underlined by the base graph in each figure.

Note that *Precision vs Recall* graphs, usually utilized to describe the answer quality of information retrieval systems, are irrelevant here since the matching tasks are completely unrelated.

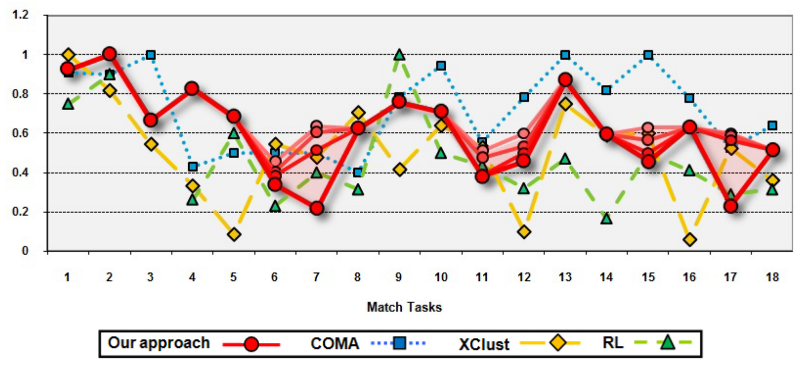


Figure 18. *Precision (PR)* results, comparing our method with *COMA* [15], *XClust* [33] and *RL* [69].

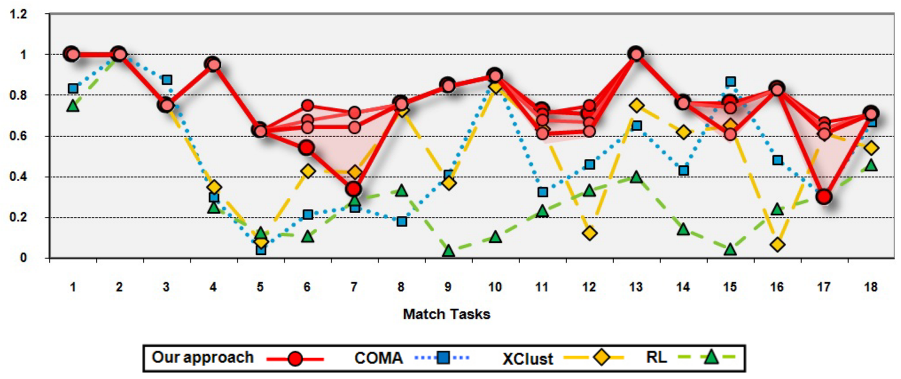


Figure 19. *Recall (PR)* results, comparing our method with *COMA* [15], *XClust* [33] and *RL* [69].

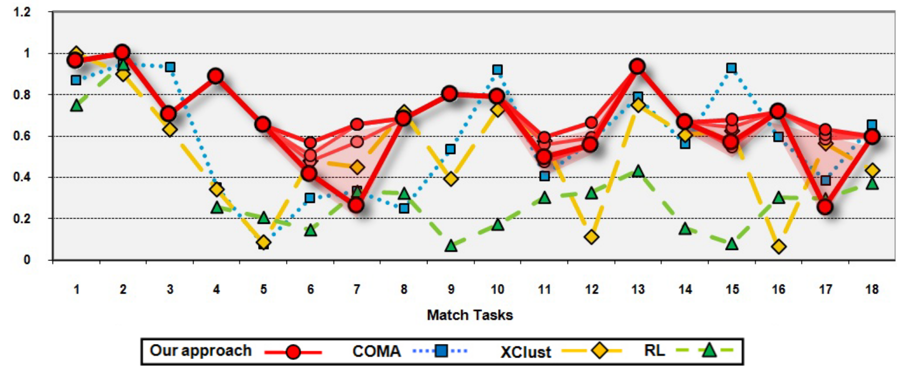


Figure 20. *F-Value* results, comparing our method with *COMA* [15], *XClust* [33] and *RL* [69].

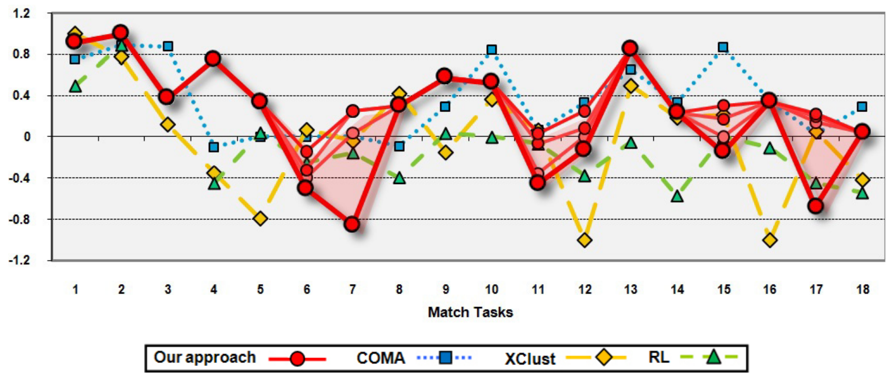


Figure 21. *Overall* results, comparing our method with *COMA* [15], *XClust* [33] and *RL* [69].

Table 11. 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 Labeling		0.4629	0.3030	0.3224	11

Recall that user feedback, with our approach, is only considered with 6 of the 18 match tasks: those which attained negative *Overall* scores in the initial matching phase (cf., Section 6.3.3). Results provided in Table 11 show that our method yields average *Precision* levels higher than those achieved by its predecessors (with and without user feedback), to the exception of *COMA*. That is due to the generic nature of *COMA* considering mappings which do not necessarily verify structurally integrity (i.e., they do not verify sibling order nor do they verify ancestor/descendent relations), but which 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 7, 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 (with and without user feedback). In cases where higher/lower *Precision*/*Recall* levels are obtained simultaneously, the *F-Value* measure is fundamental in assessing the overall 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 Labeling* (with and without user feedback). We omit average *Overall* values since the measure is non-linear in terms of *Precision* and *Recall.* On one hand, its averaging in the presence of negative values is meaningless (when *Precision<0*, *Overall* decreases with the increase of *Recall*, which is counter-intuitive). On the other hand, its averaging with only positive values would yield results proportional to those of *F-Value*, yet less optimistic [17].

Hence, we exploit *Overall* by assessing the number of matching tasks with negative *Overall* values (i.e., where more than half of the produced mappings are incorrect). Recall that in such cases, it might be easier for the user to perform the whole matching task from scratch, instead of analyzing and correcting those produced by the system. Results show that our method, in its initial (pre-feedback) matching phase, produces 6 negatives (negative *Overall* values were obtained 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 of the matching tasks, *XClust* and *RL* producing 7 and 11 negatives respectively. In short, w.r.t. *Overall*, our method outperforms both *XClust* and *RL* without user feedback, and produces the same amount of negative *Overall* values as *COMA* after one user input, consequently outperforming *COMA* with three user inputs. Recall that while it produces more negative *Overalls* than COMA in its initial (automated) phase, our approach always yields higher *F-value* scores. In other words, in most matching tasks where more than half the mappings are correct ($Overall_{OurApp} > 0$), our method generates more accurate mappings than *COMA* ($F\text{-}Value_{OurApp} > F\text{-}Value_{COMA}$). Recall that *COMA* is more generic than our approach (it does not strictly focus on the topological structure of XML grammar nodes, using an XML structure matcher among others) and thus yields average results in most cases. However, our approach is dedicated to structured data; it might induce lower quality with certain matching tasks (usually with structurally disparate grammars), yet generates higher quality mappings in the general case.

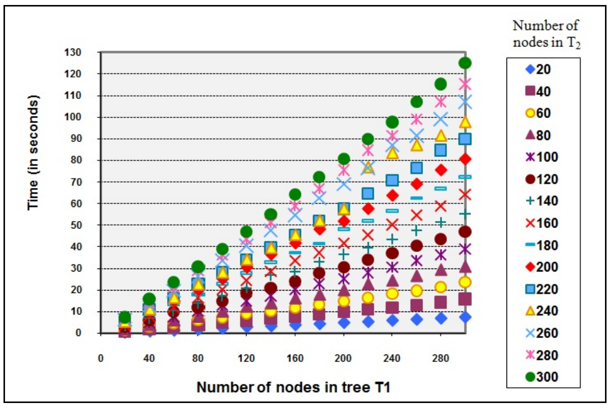
6.4. Performance Evaluation

In addition to testing the effectiveness of our approach in identifying correct mappings, we evaluated its efficiency levels, i.e., its time and space performance. We also compared our method's execution time w.r.t. manual mapping in order to estimate the amount of user (time) savings in performing the match task. In addition, we compared our approach with some of its prominent alternative methods.

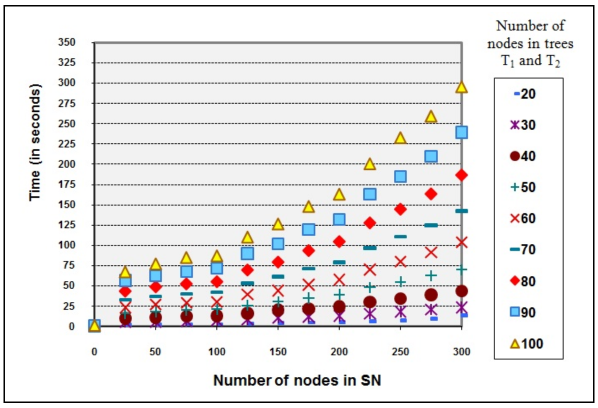
6.4.1. Time and Space Analysis

As shown in Section 5.5, the complexity of our XML grammar comparison method comes down to $O(|T_1|\times|T_2|\times|SN|\times Depth(SN))$ time and $O(|T_1|\times|T_2|)$ space. Time complexity simplifies to $O(|T_1|\times|T_2|)$ when the label *Semantic* similarity matcher is disregarded.

We start by verifying our approach's polynomial (quadratic) time dependency on XML grammar tree size, i.e., $O(|T_1|\times|T_2|)$, which equally underlines a linear dependency on the size of each XML grammar tree being compared (cf. Figure 22.a). Here, all matchers were considered to the exception of the *Semantic* one. Timing experiments were carried out on a PC with an Intel Xeon 2.66 GHz processor with 1GB RAM. Figure 22.a shows that the time to identify the mappings between two XML grammar trees of various sizes grows in an almost perfect linear fashion with tree size.



a. Timing analysis with *Semantic* matcher deactivated.



b. Timing analysis with *Semantic* matcher activated.

Figure 22. Timing results.

When exploiting the *Semantic* matcher in the matching process, the size and depth of the reference semantic network (utilized to evaluate label semantic similarity) come to play. As shown in Section 5.5, the complexity of the *Semantic* matcher is estimated as $O(|SN| \times Depth(SN))$ which is due to traversing the semantic network when searching for the lowest common ancestor between two nodes (concepts) [35, 67]. Thus, in order to reduce our method's complexity, we pre-compute semantic similarity for each pair of nodes in the semantic network considered (which took more than 5 CPU hours for a 600 node semantic network) and store the results in two dedicated indexed tables, one for each semantic measure (*Lin* and *WuPalmer*) (Oracle 9i DB)[1]. In other words, the *Semantic* matcher is no longer *computational*, but becomes *tabular* [45]. Consequently, the system would access the indexed tables to acquire semantic values instead of traversing the semantic network to compute semantic similarity each time it is needed (pair-wise similarity values are computed once, prior to comparing XML grammars). Due to this process, we eliminated the impact of semantic network depth on overall timing complexity. Timing results in Figure 22.b show that our approach becomes linearly dependent on the size on the semantic network considered, complexity simplifying from of $O(|T_1| \times |T_2| \times |SN| \times Depth(SN))$ to $O(|T_1| \times |T_2| \times |SN|)$.

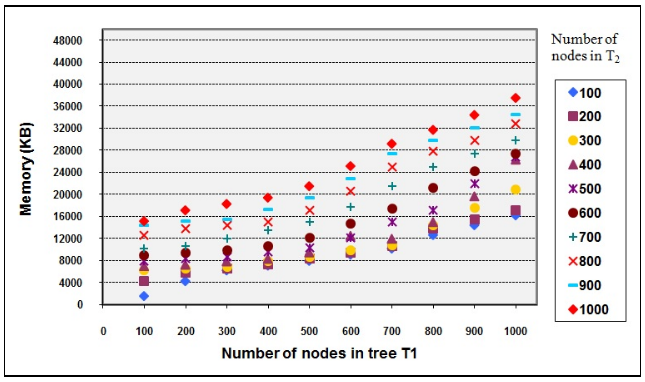


Figure 23. Memory usage.

As for space complexity, memory usage results in Figure 23 show that our approach is quadratic in the combined size of the trees being compares, $O(|T_1| \times |T_2|)$, which underlines a linear dependency on the size of each tree. Note that grammar tree size does not seem to increase the *slope*

[1] Oracle uses the *B-Tree* indexing technique.

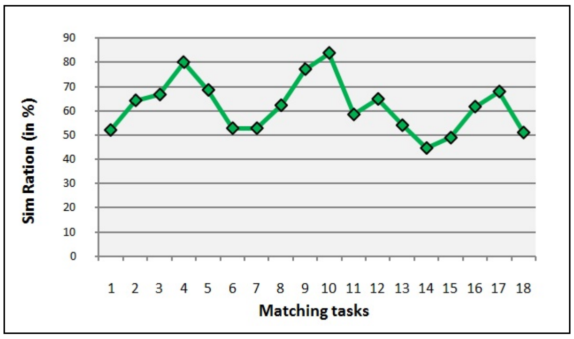
of memory chart lines (only the *y-intersect*). This underlines the possibility of further optimizing our implemented processes, so as to gain in memory, and maybe obtain implementations that run in sub-linear space.

6.4.2. Comparison with User Time

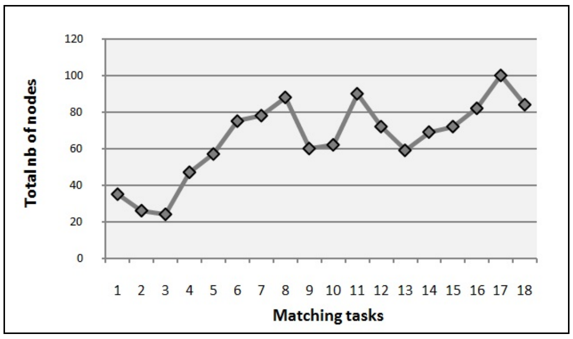
In order to evaluate the efficiency of our tool in minimizing the amount of manual work to perform the match task, we compare our system's timing results to those of manual user mappings. In this experiment, we evaluate the system's semi-automatic time, which comprises of two subsequent intervals: i) the average time required by the system to automatically perform the match task, and ii) the average time required by the user to adjust and correct the resulting system mappings.

First, to give a better impression of the problem size at hand, Figure 24.a depicts the actual *similarity ratio* (i.e., the number of user matches to be identified, w.r.t. the maximum number of elements in the grammars being compared), Figure 24.b presents the total number of nodes in each pair of grammars corresponding to the matching tasks described in Table 9, and Figure 24.c presents the average time it took for each user to manually perform each of the matching tasks (recall that three test subjects, two doctoral students and one post-doctoral researcher, were involved in the experiment). Consequently, Figure 24.d compares average user time and semi-automatic mapping time.

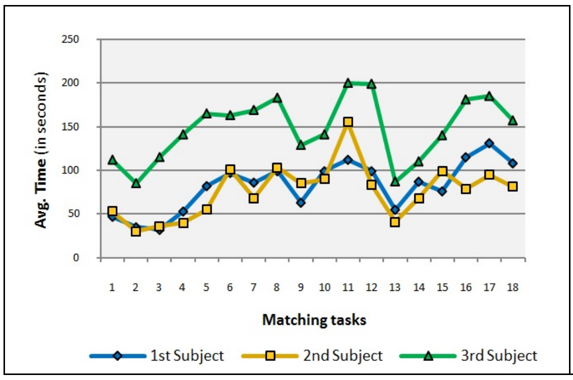
Note that the matching tasks in our experiment are unrelated. Nonetheless, we utilize the graph paradigm to depict the experimental results for ease and clearness of presentation.



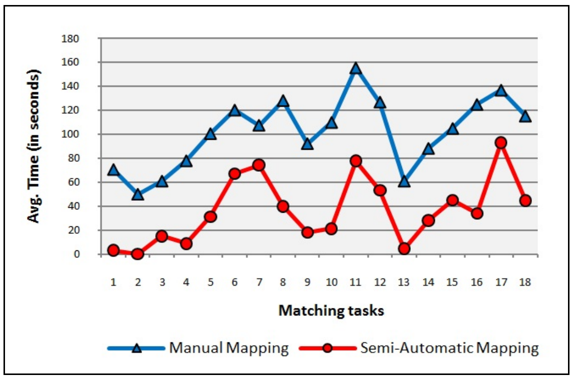
a. Similarity ratio for each of the 18 matching tasks.



b. Total number of nodes in each of the matching tasks.



c. Manual mapping time.



d. Comparing average manual mapping time and semi-automatic system mapping time.

Figure 24. Comparing manual and automatic mapping time.

Results in Figure 24.d show that semi-automatic mapping reduces matching time by a factor of *2* on average (i.e., approximately *100%* reduction in mapping time), in comparison with manual mapping. Results for match tasks *6*, *15* and *17* underline a greater variation (increase) in semi-automatic mapping time, in comparison with manual mapping time. That is due to the high amount of false positives generated by the system (highlighted by negative *Overall* values in our previous experimental results in Section 6.3), which requires the user a greater amount of time to adjust the system mappings (without however surpassing the time levels of full manual mapping). In other words, while negative *Overall* values (hypothetically) suggest that it might be easier for the user to manually perform the whole matching task from scratch, our timing results (w.r.t. all 6 match tasks with negative *Overall* values, cf. Table 9) show that it remains more profitable for the user to run the system, acquire the 'correct' system mappings, and adjust the 'false' ones. Results in Figure 24 also reflect an interesting observation: both manual time and semi-automatic mapping time closely correlate with the number of nodes in the grammars being compared (cf. Figure 24.b): the larger the grammars, the more time it takes for the users and the system to solve the matching task (cf. lower and higher peaks at tasks n# *9*, *10*, *11*, *13* and *17* in each of the graphs in 23.b, c and d). Nonetheless, the *similarity ratio* (cf. Figure 24.a) does not seem to affect mapping time.

In addition, we evaluate the effect of user feedback on mapping performance, estimating the average amount of time required by the user to: i) analyze the system mapping results, ii) provide her feedback input, and iii) adjust the resulting mappings produced by the system following the feedback phase.

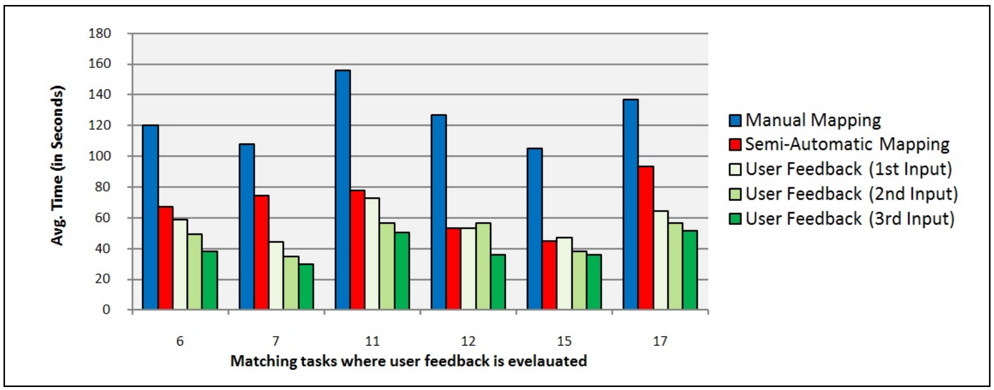


Figure 25. Average user feedback time.

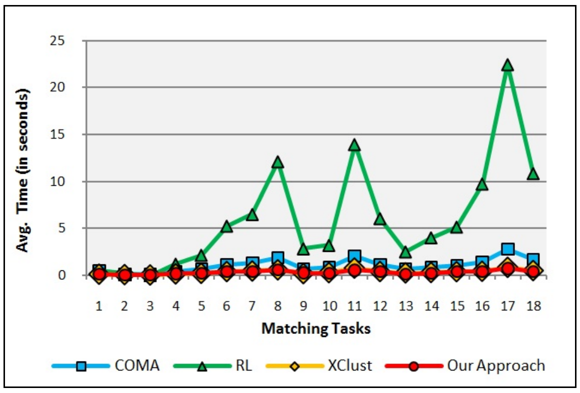
Results in Figure 25 show that user feedback considerably reduces matching time, ranging from an average factor of *2* (i.e., *100%* reduction in mapping time) with one input mapping, an average factor of *2.4* (*120%* time reduction) with the second input mapping, and an average factor of *3* (*200%* time reduction) with the third input mapping. In other words, the more user feedback is provided to the system, the lesser the number of grammar nodes to be automatically matched, and thus: i) lesser time is required by the system to perform the match operation, and ii) lesser time is usually required by the user to adjust the matching results (since input user mappings need not be re-evaluated).

For the sake of discussion, note that user feedback might sometimes induce higher semi-automatic mapping time (cf. Figure 25, mapping task n# 12, 2nd feedback input), which is due to the additional user effort required to i) select the feedback input and/or ii) analyze the mapping results. Yet, such cases remain trivial in comparison with the general trend (reflected in all remaining tasks in Figure 25). In short, results in Figure 25 show that it is often easier (i.e., faster) for the user to select an input mapping, (which only requires a partial analysis of the system generated mappings) than to perform the whole matching task by hand.

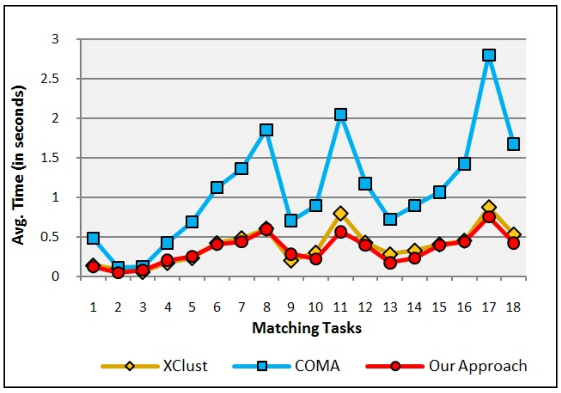
6.4.3. Comparison with Existing Approaches

In addition to verifying the complexity levels of our approach, and highlighting its efficiency w.r.t. manual user mapping, we assess our method's overall time performance w.r.t. some of its most prominent alternatives: *COMA* [15], *XClust* [33] and *Relaxation Labeling* [69].

In this context, we conducted three main experiments, measuring: i) automatic mapping time, ii) semi-automatic mapping time (including the user time to adjust matching results), and iii) time analysis w.r.t. varying grammar sizes. The first experiment measures the time required in order to automatically perform each of the matching tasks described in Table 9. Results in Figure 26 show that our method and *XClust* provide, on average, the best time levels throughout all *18* matching tasks. *COMA* comes second in time consumption, underlining the inherent complexity of its individual matcher algorithms (e.g., path matchers, sibling matchers, leaf node matchers… [15]). *Relaxation labeling* is the most time consuming approach, emphasizing its iterative nature (recall that the *Relaxation Labeling* technique in [69] computes different variations of the similarity matrix, until either a certain convergence threshold or the maximum number of iterations are obtained, as previously described in the state of art, Section 2.3).



a. Comparison results including *Relaxation Labeling*.



b. Comparison with *COMA* and *XClust*.

Figure 26. Time comparison with *COMA*, *XClust*, and *Relaxation Labeling*.

The second experiment measures two subsequent time intervals: i) automatic mapping time, plus ii) the average time required by the user to adjust system mappings. Results in Figure 27 show that our method requires, on average, the least amount of time to semi-automatically solve the matching tasks. *Relaxation labeling* underlines the worst time, whereas *XClust* and *COMA* fall in between. Note that timing results in this experiment are also an indicator of each approach's matching quality, since greater user mapping time (to adjust the mapping results) underlines inferior automatic mapping quality. Hence, results in Figure 27 confirm that our method provides, on average, higher matching quality than its predecessors.

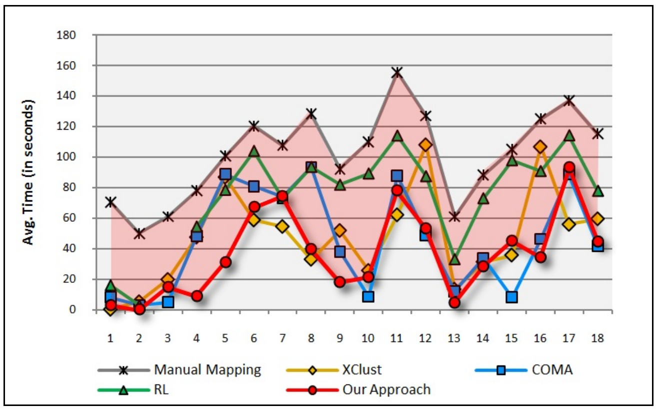
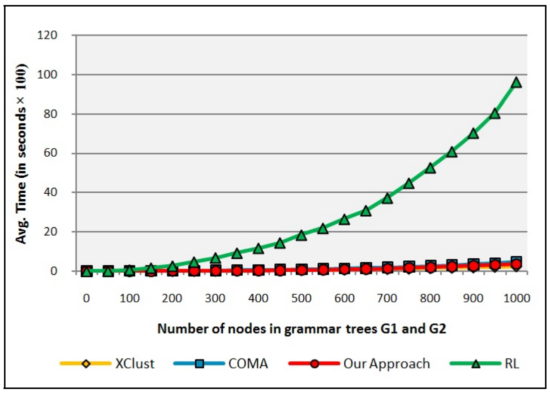
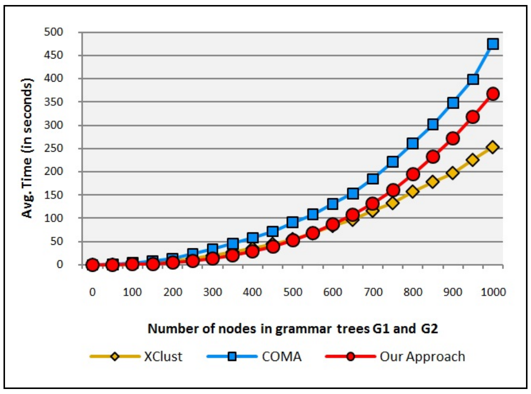


Figure 27. Semi-automatic mapping time.

The third experiment measures time performance for varying grammar sizes (ranging from *50* nodes, up to *1000* nodes per grammar). Results in Figure 28 show that our approach and *XClust* provide the best timings (with almost identical levels) for grammars under *650* nodes. *XClust* gradually outperforms our approach with larger grammars (encompassing more than *700* nodes). Recall that *XClust* simplifies XML grammar representations (disregarding alternativeness element declarations and element/attribute data-types), and is constrained to DTD grammars (disregarding more expressive XSD constraints such as *Minoccurs* and *Maxoccurs*), which might explain its higher execution speed when comparing larger XML grammars. *COMA* is constantly more time-expensive than both our approach and *XClust*. Note that in our experiments, *Relaxation Labeling* provided the worst timing levels (cf. Figure 28.a), exceeding our approach, *XClust* and *COMA* by a factor of *18* on average (for instance, comparing two *50* node schemas with *Relaxation Labeling* requires *19* seconds on average, whereas it is performed in *0.2* seconds using our approach, *0.55* using *XClust*, and *0.9* seconds using *COMA*).



a. Time results, including *Relaxation Labeling.*



b. Time comparison with *COMA* and *XClust*.

Figure 28. Time analysis with varying grammar sizes.

Note that our current experimental results were undertaken on the most basic implementation of our approach, regardless on any special indexing or pointer structures, in order to test the raw capabilities of our method. Nonetheless, we are currently investigating various performance enhancement techniques (such as B-tree indexing [20], Prufer sequence encoding [2], node clustering [55], etc.), in order to improve our method's efficiency in comparing large XML grammars.

In addition to time analysis, we compared the memory consumption of each of the methods mentioned above w.r.t. varying grammar sizes. Results with *COMA*, *XClust*, and *Relaxation Labeling* were similar to the ones obtained using our method (cf. Figure 23). Apparently, memory usage mainly depends on the sizes of the grammars being compared, rather than the storage of local variables and similarity matrixes computed by each approach, which seem to consume relatively negligible memory size.

6.5. Discussion

In order to concisely recap the results of the various experiments described in this section, we portray the differences in performance levels of each of the matching approaches exploited in our experimental study, on a scale from *4* (best performance) to *1* (worst performance) w.r.t. each of the main experiments. *F-Value* and *Overall* underline the results obtained in our matching quality experiments (cf. Section 6.3.4). *Semi-Auto* underlines the average semi-automatic mapping time required to perform the match tasks (cf. Figure 27). *Auto Time* levels underline respectively: the average time to automatically perform the match tasks, and time variation w.r.t. grammar size.

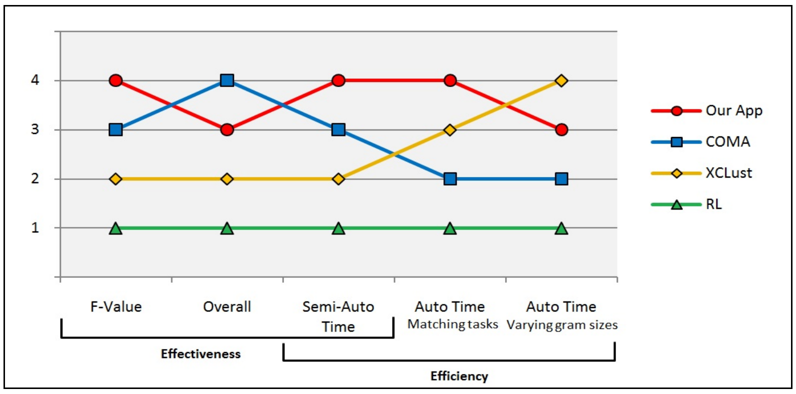


Figure 29. Time analysis with varying grammar sizes.

The graph in Figure 29 provides a simplified overview of our experimental results:

- Our approach provides high quality results in terms of both effectiveness and efficiency levels,
- *COMA* tends to provide better mapping quality (effectiveness) than time performance (efficiency), which emphasizes its composite nature (i.e., execution of several independent matching algorithms, cf. Section 2.3.2) and the inherent complexity of its individual matcher algorithms (e.g., path matchers, sibling matchers, leaf node matchers, etc. [15]),
- *XClust* tends to provide better time performance (than the separate execution of multiple independent matchers such as with *COMA*, or the more fine-grained approach developed in this study), but usually yields less accurate mapping results (probably due to the simplification of the grammars being compared [33]).
- *Relaxation Labeling* underlines the worst performance results in terms of both effectiveness and efficiency. Despite considering most basic XML grammar constraints, the relaxation labeling technique (which iteratively updates the pair-wise element similarity matrix until either a predefined convergence factor or a maximum iteration time is obtained [69]) seems detrimental to both mapping accuracy and execution time.

7. Conclusion

In recent years, the proliferation of distributed and heterogeneous XML data sources on the Web has highlighted the challenge to integrate and interoperate such repositories so as to access, acquire and manipulate more complete information. Hence, solving the XML grammar matching and comparison problem, which is at the core of the integration process, becomes an obvious need.

In this paper, we propose a framework for XML grammar matching and comparison, based on the concept of edit distance. To our knowledge, this is the first attempt to exploit tree edit distance in an XML grammar matching context. Our method aims at minimizing the amount of manual work needed to perform the match task by i) considering all basic XML grammar characteristics and constraints, via a dedicated grammar tree model, in comparison with existing 'grammar simplifying' approaches, ii) allowing a flexible and extensible combination of different matching criteria, adaptable to various application scenarios, in comparison with existing static methods, and iii) effectively considering the semi-structured nature of XML, as the most prominent and distinctive feature of an XML grammar, in comparison with existing heuristic or generic approaches, in order to produce more accurate results. We have implemented our approach and conducted various tests to validate its effectiveness and efficiency in identifying XML grammar mappings, w.r.t. 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 [49] via dedicated data-type restriction and extension operators (which do not exist in DTDs). For instance, a user defined data-type *IntEven* which only allows *even* integer values, is a restriction of the predefined XSD *Integer* data-type, allowing both *even* and *odd* numbers. Likewise, one can imagine much more intricate type extensions and restrictions, involving both simple and complex elements (e.g., extending a complex type by adding new elements specific to the newly defined complex type). In this context, dedicated knowledge bases (i.e., semantic networks) and user-defined semantics [65] would have to be considered to assess the relatedness between the various data-types [23]. We are also currently investigating techniques to performance enhancement (e.g., such as B-tree indexing [20], Prufer sequence encoding [2], node clustering [55]…) in order amend our method's performance levels so as to efficiently compare large scale schemas. In the near future, we plan to study the effect of the different matchers and criteria on matching effectiveness, proposing (if possible) weighting schemes that could help the user tune her input parameters to obtain optimal results. In the long run, we plan to study XML grammar integration, one of the main application domains of grammar matching, toward manipulating materialized XML views [36] and XML schema evolution functions.

Acknowledgments

We would like to thank Phil Bernstein, Sabine Maßmann and Alsayed Algergawy for providing us with their test schemas to conduct our matching experiments. We are also grateful to the reviewers for their valuable comments and suggestions which have allowed us to further improve the manuscript's presentation, organization, and contents. This work was partly funded by *Fondazione Cariplo*.

References

[1] Algergawy A., Nayak R., and Saake G., *Element Similarity Measures in XML Schema Matching.* Elsevier Information Sciences, 2010. 180(24): 4975-4998.

[2] Algergawy A.; Schallehn E. and G. Saake, *Improving XML schema matching using Prufer sequences.* Data and Knowledge Engineering, 2009. 68(8):724–747.

[3] Algergawy A.; Schallehn E. and Saake G., *A schema matching-based approach to XML schema clustering.* Proc. of the 10th International Conference on Information Integration and Web-based Applications & Services, 2008. pp. 131-136.

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

[5] Bille P., *A Survey on Tree Edit Distance and Related Problems.* Theoretical Computer Science, 2005. 337(1-3):217-239.

[6] Boukottaya A. and Vanoirbeek C., *Schema Matching for Transforming Structured Documents.* Proceedings of the ACM Symposium on Document Engineering, 2005. pp. 101-110.

[7] Bray T.; Paoli J.; Sperberg-McQueen C.; Mailer Y.; and Yergeau F. *Extensible Markup Language (XML) 1.0 - 5th Edition. W3C recommendation, 26 Novembre 2008.* http://www.w3.org/TR/REC-xml/ [cited July 2011].

[8] Budanitsky A. and Hirst G., *Evaluating WordNet-based Measures of Lexical Semantic Relatedness.* Computational Linguistics, 2006. 32(1): 13-47.

[9] Buttler D., *A Short Survey of Document Structure Similarity Algorithms.* Proc. of the International Conference on Internet Computing (ICOMP), 2004. pp. 3-9.

[10] Castano S.; De Antonellis V. and Vimercati S., *Global Viewing of Heterogeneous Data Sources.* IEEE Transactions on Knowledge and Data Engineerins, 2001. 13(2).

[11] Chawathe S., *Comparing Hierarchical Data in External Memory.* Proc. of the Inter. VLDB Conf., 1999. pp. 90-101.

[12] Chawathe S.; Rajaraman A.; Garcia-Molina H.; and Widom J., *Change Detection in Hierarchically Structured Information.* Proc. of the ACM International Conference on Management of Data (SIGMOD), 1996. pp. 26-37. Montreal.

[13] Cobéna G. *et al.*, *Detecting Changes in XML Documents.* Proc. of the Inter. ICDE Conf., 2002. pp. 41-52.

[14] Dalamagas T.; Cheng T.; Winkel K.; and Sellis T., *A Methodology for Clustering XML Documents by Structure.* Information Systems, 2006. 31(3):187-228.

[15] Do H. and Rahm E., *COMA: A System for Flexible Combination of Schema Matching Approaches.* Proc. of the Inter. VLDB Conf., 2002. pp. 610-621.

[16] Do H. and Rahm E., *Matching Large Schemas: Approaches and Evaluation.* Information Systems, 2007. 32(6): 857-885.

[17] Do H.; Melnik S.; and Rahm E., *Comparison of Schema Matching Evaluations.* Proceedings of the International Workshop on the Web and Databases (German Informatics Society), 2002. pp. 221-237. Erfurt.

[18] Doan A.; Domingos P.; and Halevy A., *Learning to Match the Schemas of Data Sources: A Multistrategy Approach.* Machine Learning, 2003. 50(3):279-301.

[19] DuChateau F.; Bellahsene Z. and Hunt E., *XBenchMatch: An Benchmark for XML Schema Matching Tools.* Proceedings of the International Conference on Very Large Data Bases (VLDB), 2007. pp. 1318-1321.

[20] DuChateau F.; Bellahsene Z.; Hunt E.; Roantree M., a.R.M., *An Indexing Structure for Automatic Schema Matching.* The 23rd International Conference on Data Engineering (ICDE) - Workshops, 2007. pp. 485-491.

[21] Ehrig M.; Staab S. and Sure Y., *Bootstrapping Ontology Alignment Methods with APFEL.* In Proceedings of the International World Wide Web Conference (WWW'05), 2005. pp. 1148-1149.

[22] Flesca S.; Manco G.; Masciari E.; Pontieri L.; and Pugliese A., *Detecting Structural Similarities Between XML Documents.* Proc. of the International ACM SIGMOD Workshop on The Web and Databases (WebDB), 2002. pp. 55-60.

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

[24] Francis W. N. and Kucera H., *Frequency Analysis of English Usage.* Houghton Mifflin, Boston, 1982.

[25] Giunchiglia F.; Shvaiko P. and Yatskevich M., *S-Match: an Algorithm and an Implementation of Semantic Matching.* European Semantic Web Symposium (ESWS), 2004. pp. 61-75.

[26] Goldman R. and Windom J., *Dataguides: Enabling Query Formulation and Optimization in Semi-structured Databases.* Proc. of the Inter. VLDB Conf., 1997. pp. 436-445.

[27] Gudgin M.; Hadley M.; Mendelsohn N.; Moreau J.-J.; Canon and Nielsen H.F. *Simple Object Access Protocol 1.1* http://www.w3.org/TR/SOAP. June 2003 [cited July 2011].

[28] Hall P. and Dowling G., *Approximate String Matching.* Computing Survey, 1980. 12(4):381-402.

[29] Hull R., *Relative Information Capacity of Simple Relational Database Schemata.* In Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1984. pp. 97-109.

[30] 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, 2008. 34 (3):1651-1658.

[31] Knuth D., *The Art of Computer Programming. Volume 3: Sorting and Search.* 1998. Addison-Wesley, pp. 780.

[32] Larson J.; Navathe S.B. and Elmasri R., *Theory of Attribute Equivalence and its Applications to Schema Integration.* IEEE Transactions on Software Engineering, 1989. 15(4).

[33] Lee M. *et al.*, *XClust: Clustering XML Schemas for Effective Integration.* The Inter. CIKM Conf., 2002. pp. 292-299.
[34] Leonardi E. *et al.*, *DTD-Diff: A Change Detection Algorithm for DTDs.* Proc. of Inter. DASFAA Conf., 2006, 384-402.
[35] Lin D., *An Information-Theoretic Definition of Similarity.* Inter. Conf. on Machine Learning (ICML), 1998. pp. 296-304.
[36] Lo A., Ozyer T., Tahboob R., Kianmehr K., Jida J., and Alhajj R., *XML Materialized Views and Schema Evolution in VIREX.* Elsevier Information Sciences, 2010. 180(24):4940-4957.
[37] Madhavan J. et al., *Generic Schema Matching With Cupid.* Proc. of the Inter VLDB Conf., 2001. pp. 49-58.
[38] Maguitman A. *et al.*, *Algorithmic Detection of Semantic Similarity.* Proc. of the Inter. WWW Conf., 2005. pp. 107-116.
[39] Marie A. and Gal A., *Boosting Schema Matchers.* OTM 2008 Confederated International Conferences, 2008, 283 – 300.
[40] McGill M., *Introduction to Modern Information Retrieval.* 1983. McGraw-Hill, New York.
[41] Melnik S.; Garcia-Molina H.; and Rahm E., *Similarity Flooding: A Versatile Graph Matching Algorithm and its Application to Schema Matching.* Proc. of the IEEE Inter. Conf. on Data Engineering (ICDE), 2002. pp. 117-128.
[42] Miller G., *WordNet: An On-Line Lexical Database.* International Journal of Lexicography, 1990. 3(4).
[43] Miller R.; Hass L. and Hermandez M.A., *Schema Mapping as Query Discovery.* Inter. VLDB Conf., 2000. pp. 77-88.
[44] Milo T. and Zohar S., *Using Schema Matching to simplify Heterogeneous Data Translation.* Proceedings of the International Conference on Very Large Data Bases (VLDB), 1999. pp. 122-133
[45] Motro A., *Vague: A User Interface to Relational Databases that Permits Vague Queries.* ACM Transactions on Office Information Systems, 1988. 6(3):187-214.
[46] Nayak R. and Iryadi W., *XML Schema Clustering with Semantic and Hierarchical Similarity Measures.* Knowledge Based Systems, 2007. 20(4):336-349.
[47] Nayak R. and Tran T., *A Progressive Clustering Algorithm to Group The XML Data By Structural and Semantic Similarity.* International Journal of Pattern Recognition and Artificial Intelligence, 2007. 21(4):723–743.
[48] Nierman A. and Jagadish H. V., *Evaluating structural similarity in XML documents.* Proc. of the ACM SIGMOD International Workshop on the Web and Databases (WebDB), 2002. pp. 61-66.
[49] Peterson D.; Gao S.; Malhotra A.; Sperberg-McQueen C.; and Thompson H. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes.* http://www.w3.org/TR/xmlschema11-2/ [cited July 2011].
[50] Peukert E.; Maßmann S. and König K., *Comparing Similarity Combination Methods for Schema Matching.* GI Jahrestagung (1), 2010. pp. 692-701, Leipzig, Germany.
[51] Rahm E. and Bernstein P., *A Survey of Approaches to Automatic Schema Matching.* The VLDB J., 2001. (10):334-350.
[52] Rahm E.; Do H.H. and Massmann S., *Matching Large XML Schemas.* Sigmod Record, 2004. 33(4): 26-31.
[53] Resnik P., *Using Information Content to Evaluate Semantic Similarity in a Taxonomy.* Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 1995. Vol 1, pp. 448-453.
[54] Richardson R. and Smeaton A., *Using WordNet in a Knowledge-based approach to information retrieval.* Proceedings of the BCS-IRSG Colloquium on Information Retrieval, 1995.
[55] Saleem K.; Bellahsene Z. and Hunt E., *PORSCHE: Performance Oriented Schema mediation.* Information Systems, 2008. 33(7):637-657.
[56] Salton G. and Buckley C., *Term-weighting approaches in automatic text retrieval.* Information Processing and Management, 1988. 24(5):513 -523.
[57] Schlieder T., *Similarity Search in XML Data Using Cost-based Query Transformations.* Proc. of the ACM SIGMOD International Workshop on the Web and Databases (WebDB), 2001. pp. 19-24.
[58] Shasha D. and Zhang K., *Approximate Tree Pattern Matching.* Pattern Matching in Strings, Trees and Arrays, Oxford University Press, 1995. chapter 14.
[59] Su H.; Kuno H. and Rundensteiner E.A., *Automating the Transformation of XML Documents.* Proceedings of the ACM Symposium on Document Engineering, 2001.
[60] Su H.; Padmanabhan S. and Lo M.L., *Identification of Syntactically Similar DTD Elements for Schema Matching.* Proc. of the Inter. Conference on Advances in Web-Age Information Management (WAIM), 2001. pp. 145-159.
[61] Tansalarak N. and Claypool K. T., *QMatch - Using paths to match XML schemas.* Data Knowledge Engineering (DKE), 2007. 60(2):260-282.
[62] Tekli J.; Chbeir R. and Yétongnon K., *A Fine-grained XML Structural Comparison Approach.* Proceedings of the 26th International Conference on Conceptual Modeling (ER), 2007. LNCS 4801, pp. 582-598.
[63] Thang H. Q. and Nam V. S., *XML Schema Automatic Matching Solution.* International Journal of Computer Systems Science and Engineering, 2007. 4(1):68-74.
[64] Wagner J. and Fisher M., *The String-to-String correction problem.* Journal of the ACM, 1974. 21(1):168-173.
[65] Wojnar A., Mlynkova I., and Dokulil J., *Structural and Semantic Aspects of Similarity of Document Type Definitions and XML Schemas.* Special Issue on Intelligent Distributed Information Systems - Elsevier Information Sciences, 2010. 180(10):1817-1836.
[66] World Wide Web Consortium. *The Document Object Model.* http://www.w3.org/DOM [cited 28 May 2009].
[67] Wu Z. and Palmer M., *Verb Semantics and Lexical Selection.* Proc. of the 32nd Annual Meeting of the Associations of Computational Linguistics, 1994. pp. 133-138.
[68] Yaworsky D., *Word-Sense Disambiguation Using Statistical Models of Roget's Categories Trained on Large Corpora.* Proc. of the International Conference on Computational Linguistics (Coling), 1992. Vol 2, pp. 454-460. Nantes.
[69] Yi S.; Huang B. and Chan W.T., *XML Application Schema Matching Using Similarity Measure and Relaxation Labeling.* Information Sciences, 2005. 169(1-2):27-46.
[70] Zhang K. and Shasha D., *Simple Fast Algorithms for the Editing Distance between Trees and Related Problems.* SIAM Journal of Computing, 1989. 18(6):1245-1262.
[71] Zhang K.; Statman R. and Shasha D., *On the editing distance between unordered labeled trees.* Information Processing Letters, 1992. 42(3):133–139.
[72] Zhang Z.; Li R.; Cao S.; and Zhu Y., *Similarity Metric in XML Documents.* Knowledge Management and Experience Management Workshop, 2003.