Adaptive Similarity of XML Data

Eva Jílková, Marek Polák, and Irena Holubová

Department of Software Engineering, Charles University in Prague, Czech Republic
{polak,holubova}@ksi.mff.cuni.cz

Abstract. In this work we explore application of XML schema similarity mapping in the area of conceptual modeling of XML schemas. We expand upon our previous efforts to map XML schemas to a common platform-independent schema using similarity evaluation based on exploitation of a decision tree. In particular, in this paper a more versatile method is implemented and the decision tree is trained using a large set of user-annotated mapping decision samples. Several variations of training that could improve the mapping results are proposed. The approach is implemented within a modeling and evolution management framework called *eXolutio* and its variations are evaluated using a wide range of experiments.

Keywords: XML schema matching, PSM-to-PIM mapping, model driven architecture.

1 Introduction

The XML (eXtensible Markup Language) [5] has become one of the leading formats for data representation and data exchange in the recent years. Due to its extensive usage, large amounts of XML data from various sources are available. Since it is common that sooner or later user requirements change, it is very useful to adapt independently created XML schemas that represent the same reality for common processing. However, such schemas may differ in structure or terminology. This leads us to the problem of *XML schema matching* that maps elements of XML schemas that correspond to each other. Schema matching is extensively researched and there exists a large amount of applications, such as data integration, e-business, schema integration, schema evolution and migration, data warehousing, database design and consolidation, biochemistry and bioinformatics, etc.

Matching a schema manually is a tedious, error-prone and expensive work. Therefore, automatic schema matching brings significant savings of manual effort and resources. But it is a difficult task because of the heterogeneity and imprecision of input data, as well as high subjectivity of matching decisions. Sometimes the correct matches have to be marked only by a domain expert. As s compromise, in semi-automatic schema matching the amount of user intervention is significantly minimized. For example, the user can provide information before matching/mapping, during the learning phase. Or, after creation of a

R. Meersman et al. (Eds.): OTM 2014, LNCS 8841, pp. 535–552, 2014.

mapping (s)he can accept or refuse suggested mapping decisions which could be later reused for improvement of further matching.

In this work we apply the task of schema matching to a specific application of conceptual modeling – *MDA* (Model-Driven Architecture) [19]. MDA models the application domain at several levels of abstraction. *PSM* (Platform-Specific Model) schemas are integrated using a common conceptual schema – *PIM* (Platform-Independent Model) schema. In the optimal case, the PIM schema for a given domain is first designed and then various PSM schemas are derived from it for specific applications. In reality, the PIM schema has to be designed to describe a common domain in a situation when various PSM schemas for specific applications already exist. Or, a new PSM schema may need to be integrated with an existing hierarchy of PIM and PSM schemas. In both the cases independent PSM schemas may come from different sources, they may be of various types and they may use different naming conventions.

Schema matching is used as the key step during this integration process. In particular, we match elements from independent PSM schemas against elements in the common PIM schema to establish the respective PSM-to-PIM mapping. In particular, this work uses a semi-automatic approach to schema matching. We explore the applicability of decision trees for this specific use case. A decision tree is constructed from a large set of training samples and it is used for identification of correct mapping. For our target application various modifications of the training process are proposed and experimentally evaluated on the basis of several common hypotheses. The proposed approach extends our previous work [11] and it was implemented and experimentally tested in the modeling and evolution management tool called *eXolutio* [18] which is based on the idea of MDA.

The paper is structured as follows: In Section 2 we briefly describe the relation of schema matching and similarity and its usage in our target application. In Section 3 related work and existing implementations of schema matching are discussed. The proposed solution is described in Section 4. In Section 5 respective experiments are presented. Finally, results and possible future improvements are briefly resumed in Section 6.

2 Schema Matching

The semi-automatic or automatic process of finding correspondences between elements of two schemas is called *schema matching*. In this paper the term schema matching is used for simplicity in a general way, but there are various specific types. *Schema-to-schema matching* has as an input two XML schemas. *Instance-to-instance matching* has as an input two XML documents. And *schema-to-instance matching* has as an input an XML document and an XML schema. *Similarity* is a measure that expresses the level of correspondence. Its value is from interval $[0, 1]$, where 0 means no similarity and 1 means that the compared items are equal in the selected similarity meaning. *Matcher* is an algorithm that evaluates similarity of schemas according to particular criteria.

2.1 Usage of Schema Matching in MDA

Assume that an XSD[1] was created and we would like to integrate it now with a set of PSM schemas having a common PIM schema. Elements of the XSD are first converted to their corresponding PSM schema representatives. This conversion is straightforward, as it is proven in [20]. For the full integration we need to find the interpretation of its elements against the PIM elements. This could be done either manually or using schema matching. We explore usage of schema matching for this task in our work. A PSM element – PIM element pair is thus identified as an interpretation of PSM element against PIM element if it is suggested as a match by schema matching.

3 Related Work

In this section existing schema matching approaches are described. As we have mentioned, since the number of the approaches is high, we have selected only the key classical and most popular representatives.

3.1 COMA

COMA matcher [1] is an example of a *composite* approach. Individual matchers are selected from an extensible library of match algorithms. The process of matching is *interactive* and *iterative*. A match iteration has the following three phases: (1) *User feedback and selection of the match strategy*, (2) *Execution of individual matchers*, and (3) *Combination of the individual match results*.

Interactive Mode. The first step in the iteration is optional. The user is able to provide *feedback* (to confirm or reject previously proposed match candidates or to add new matches) and to define a *match strategy* (selection of matchers, strategies to combine individual match results). In *automatic mode* there is only one iteration and the match strategy is specified by input parameters.

Reuse of Match Results. Since many schemas to be matched are very similar to the previously matched schemas, match results (intermediate similarity results of individual matchers and user-confirmed results) are stored for later reuse.

Aggregation of Individual Matcher Results. Similarity values from individual matchers are aggregated to a combined similarity value. Several aggregate functions are available, for example *Min*, *Max* or *Average*.

Selection of Match Candidates. For each schema element its best match candidate from another schema is selected, i.e. the ones with the highest similarity value according to criteria like *MaxN* (n elements from schema S with maximal similarity are selected as match candidates), *MaxDelta* (an element from schema S with maximal similarity is determined as match candidate plus all S elements with a similarity differing at most by a tolerance value d which can

[1] XML Schema Definition.

be specified either as an absolute or relative value), or *Threshold* (all S elements showing a similarity exceeding a given threshold value t are selected).

The COMA++ [2], an extension of COMA, supports a number of other features like merging, saving and aggregating match results of two schemas.

3.2 Similarity Flooding

Similarity Flooding [4] can be used to match various data structures – data schemas, data instances or a combination of both. The algorithm is based on the idea that the similarity of an element is propagated to its neighbors. The input data is converted into *directed labeled graphs*. Every edge in the graphs is represented as a triple (s, l, t), where s is a source node, t is a target node, and l is a label of the edge. The algorithm has the following steps: (1) *Conversion of input schemas to internal graph representation*, (2) *Creation of auxiliary data structures*, (3) *Computation of initial mapping*, (4) *Iterative fix-point computation* and (5) *Selection of relevant match candidates*. The accuracy of the algorithm is calculated as the number of needed adjustments. Output mapping of elements is checked and if necessary, corrected by the user.

Matcher. The main matcher is structural and is used in a *hybrid* combination with a simple name matcher that compares common affixes for initial mapping. The matcher is iterative and based on *fixpoint computation* with *initial mapping* as a starting point.

Fixpoint Computation. The similarity flooding algorithm is based on an iterative computation of σ-values. The computation of the σ-values for a map pair (x, y) is performed iteratively until the Euclidean length of the residual vector $\Delta(\sigma^n, \sigma^n - 1)$ becomes less than ϵ for some $n > 0$ (i.e. the similarities stabilize):

$$\sigma^{i+1}(x,y) = \sigma^i(x,y) + \sum_{\substack{(a,l,x)\in E_A \\ (b,l,y)\in E_B}} \sigma^i(a,b) w((a,b),(x,y)) + \sum_{\substack{(x,l,c)\in E_A \\ (y,l,d)\in E_B}} \sigma^i(c,d) w((x,y),(c,d)) \quad (1)$$

where $\sigma^i(x, y)$ is the similarity value in i-th iteration of nodes x and y and σ^0 is the value computed in the initial mapping.

Similarity Flooding can be further improved for example by usage of another matcher for initial mapping or auxiliary source of information – e.g. dictionary.

3.3 Decision Tree

In [3] a new method of combining independent matchers was introduced. It is based on the term *decision tree*.

Definition 1. *A* decision tree *is a tree $G = (V, E)$, where V_i is the set of internal nodes (independent match algorithms), V_l is the set of leaf nodes (output decision whether elements do or do not match), $V = V_i \cup V_l$ is the set of all nodes, E is the set of edges (conditions that decide to which child node the computation will continue).*

The decision tree approach does not have the following disadvantages of aggregation of result of independent matchers used, e.g., in COMA:

- **Performance:** In the composite approach with an aggregate function, all of the match algorithms have to run. The time required is worse than with a decision tree.
- **Quality:** Aggregation can lower the match quality, e.g., if we give higher weights to several matchers of the same type that falsely return a high similarity value.
- **Extendability** is worse, because adding a new matcher means updating the aggregation function.
- **Flexibility** is limited, because an aggregation function needs manual tuning of weights and thresholds.
- **Common Threshold:** Each match algorithm has its own value distribution, thus it should have own threshold.

The main disadvantage of Decision Trees is a need of a set of training data.

3.4 Advantages and Disadvantages

A general comparison of the previously discussed methods is introduced in Table 1. The decision tree approach seems to be the most promising for our application – it is dynamic and versatile. Furthermore it has desirable values of compared properties – it is highly extensible, quick and has a low level of user intervention and a low level of required auxiliary information.

Table 1. A comparison of the selected existing solutions

	Extensibility	Speed	User intervention	Auxiliary info
COMA	Low	Low	High	Low
Similarity Flooding	None	High	None	None
Decision Tree	High	High	Low	Low

4 Proposed Solution

First, we will briefly describe the algorithm for construction of decision tree proposed in [11] which we used for PSM-to-PIM mapping in our preliminary implementation called *eXolutio* [18] (as described later in Section 5). Then we will follow with a description of *C5.0* algorithm [16] that we utilized in our work for training of the decision tree. As we will show, it solves several problems of the original algorithm.

4.1 Original Decision Tree Construction

The decision tree in [11] is constructed as follows: The matchers are split into three groups (called *feature groups*) according to the main feature that they compare: *class name* (if the matcher compares names of the model classes), *data*

type (if the matcher compares similarity of data types of the given elements) and *structural similarity* (if the similarity is measured by the analysis of the models structures – relations of the nodes). In each feature group the matchers are assigned with a priority according to their efficiency. Then the matchers are sorted in ascending order according to importance of group (where for example in our case the class name group is the most important one) and their priority inside the group. Finally, the decision tree is built. The first matcher is selected as the root of the tree and other matchers are taken in sequence and added to the tree. If we want to add matcher M to the actual node n (i.e. use function $addMatcherToTree(M, n)$), there are the following possible situations:

- If node n has no child, method M is added as a child of n.
- If node n has children $c_1, ..., c_n$ from the same feature group that M belongs to and it has the same priority, then matcher M is added as the next child of node n.
- If node n has children $c_1, ..., c_n$ from the different feature group that M belongs to or it has a different priority, then for each node $c_i; i \in (1, n)$ we call $addMatcherToTree(M, c_i)$.

Though we have used the algorithm as the preliminary approach in our implementation, it has several drawbacks. First, it does not propose a method for automatic determination of conditions on edges and thresholds for continuous matchers. They have to be either set by the user or the default values are used. Furthermore, the decision tree does not suggest mapping results automatically. It computes an aggregated similarity score. During the traversal of the decision tree for each of the feature groups the maximal similarity value returned by the matcher from this group is stored. Then the aggregated similarity score is computed as an average of the maximal similarity value for each of the feature groups. For each PSM element it returns possible match candidates – PIM elements evaluated by the aggregate similarity score sorted in the descending order. This helps to find matches, but it is not done automatically – the user has to evaluate each mapping.

In this work we decided to generate the decision tree using machine learning techniques. This approach solves the above mentioned problems, as we would like to use the advantages of the decision tree approach and minimize the previous disadvantages.

4.2 Decision Tree Training via C5.0

Currently, there are several algorithms for the induction of a decision tree from training data, such as *ID3* [6], *CLS* [9], *CART* [8], *C4.5* [7], or *SLIQ* [10], to name just a few. The *C5.0* [16] algorithm is exploited and utilized in this paper for training of the decision tree. We decided to utilize for our purposes the *C5.0* because this algorithm and its predecessors are widely used and implemented in various tools, e.g. Weka [22]. First, we introduce a notation that is used in the rest of the section.

S – a set of training samples. (An example of training samples is depicted in Figure 1.)
$S(v, M)$ – a set of examples from S that have value v for matcher M.
$S((i_1, i_2), M)$ – a set of examples from S that have value from interval (i_1, i_2) for matcher M.
$C = \{C_1, C_2\}$ – the decision tree algorithm classifies S into two subsets with possible outcomes $C_1 = match$ and $C_2 = mismatch$.
$Info(S)$ – entropy of the set S.
$freq(C_i, S)$ – the number of examples in S that belong to class C_i.
$|S|$ – the number of samples in the set S.
$Gain(M, S)$ – the value of information gain for matcher M and set of samples S.
$Info_M(S)$ – entropy for matcher M.

The entropy of the set of training samples S is computed as follows:

$$Info(S) = -\sum_{i=1}^{2} \left(\frac{freq(C_i, S)}{|S|} \log_2 \left(\frac{freq(C_i, S)}{|S|} \right) \right) \tag{2}$$

The set S has to be partitioned in accordance with the outcome of matcher M. There are two possibilities:

1. Matcher M has n discrete values. In that case the entropy for matcher M and set S is computed as follows (using the above defined notation):

$$Info_M(S) = \sum_{i=1}^{n} \left(\frac{|S(i, M)|}{|S|} Info(S(i, M)) \right) \tag{3}$$

2. Matcher M has values from continuous interval $[a, b]$, that is why threshold $t \in [a, b]$ that brings the most information gain has to be selected by Algorithm 1. Entropy for matcher M and set S is then computed according to the following formulae:

$$Info_M(S) = \left(\frac{|S([a, t], M)|}{|S|} Info([a, t], M) \right) + \left(\frac{|S((t, b], M)|}{|S|} Info((t, b], M) \right) \tag{4}$$

The gain value for a set of samples S and matcher M is computed as follows:

$$Gain(M, S) = Info(S) - Info_M(S) \tag{5}$$

Then the decision tree is constructed by Algorithm 2 (which differs from the algorithm described in Section 4.1). There are the following possibilities for the content of the set of training samples S in the given node *parent* of the decision tree:

1. If S is empty, then the decision tree is a leaf identifying class C_i – the most frequent class at the parent of the given node *parent*. This leaf is added as a child to node *parent*.

2. If S contains only examples from one class C_i, then the decision tree is a leaf identifying class C_i. This leaf is added as a child to node *parent*.
3. If S contains examples from different classes, then S has to be divided into subsets. Matcher M with the highest value of information gain is selected. There are two possibilities:
   (a) If matcher M has n discrete mutually exclusive values $v_1, ..., v_n$, then set S is partitioned into subsets S_i where S_i contains samples with value v_i for matcher M.
   (b) If matcher M has values $(v_1, ..., v_n)$ from continuous interval $[a, b]$, then threshold $t \in [a, b]$ has to be determined. Subsets S_1, S_2 contain samples with values from interval $[a, t]$, $[t, b]$, respectively, for matcher M.

   Matcher M is added as a child to node *parent*. For all the subsets S_i subtrees are constructed and added to node M as children.

The threshold for matcher M with values $(v_1, ..., v_n)$ from continuous interval $[a, b]$ is selected as follows:

- Values are sorted in the ascending order, duplicates are removed. Let us denote them $u_1, ..., u_m$.
- All possible thresholds $A_i \in [u_i, u_{i+1}]$ have to be explored.
- For each interval $[u_i, u_{i+1}]$ the midpoint A_i is chosen as a split to two subsets $[u_1, A_i]$ and $(A_i, u_m]$.
- For each midpoint the information gain is computed and the midpoint A_{max} with the highest value of information gain is selected.
- The threshold is then returned as a lower bound of interval $[u_{max}, u_{max+1}]$.

Algorithm 1. Selection of threshold for continuous values $v_1, ..., v_n$ for matcher M and set of samples S

```
1: function ComputeThreshold((v_1, ..., v_n), S, M)
2:     /* sorts values in the ascending order, duplicate values are removed */
3:     (u_1, ..., u_m) ← SortAscDistinct((v_1, ..., v_n))
4:     for i ← 1, m − 1 do
5:         /* average of values U[i] and U[i+1] */
6:         A[i] ← Avg(U[i], U[i + 1])
7:         L[i] ← U[i]
8:         H[i] ← U[i + 1]
9:     end for
10:    /* u_m the highest value from the continuous interval, u_1 the lowest value from the continuous
       interval */
11:    for i ← 1, m − 1 do
12:        gain_i ← Gain(M, S([u_1, A[I]], (A[I], u_m], M))
13:    end for
14:    maxGain ← max_{i=1}^{m−1} gain_i
15:    max ← i|gain_i = maxGain
16:    t ← L[max]
17:    threshCost ← cost of splitting interval into two subintervals [u_1, t] and (t, u_m]
18:    result.gain ← maxGain − −threshCost
19:    result.threshold ← t
20:    return result
21: end function
```

Algorithm 2. Construction of a decision tree T from a set S of user-evaluated training samples

```
1: function BuildTree(S, parent, condition)
2:     T empty tree
3:     if S is empty then
4:         c ← the most frequent class at the parent of the given node parent
5:         /* adds node c as a child to node parent with condition condition on edge */
6:         AddLeaf(parent, c, condition)
7:     else if S contains only results from one class C_i then
8:         c ← C_i
9:         /* adds node c as a child to node parent with condition condition on edge */
10:        AddLeaf(parent, c, condition)
11:    else
12:        M ← matcher with the highest value of information gain Gain(M, S)
13:        /* adds node M as a child to node parent with condition condition on edge */
14:        AddNode(parent, M, condition)
15:        if M has n discrete mutually exclusive values v_1, ..., v_n then
16:            S' ← {S_1, ..., S_n}|S_i = S(v_i, M)
17:            c_i ← v_i
18:        else if M has values (v_1, ..., v_n) from continuous interval [a, b] then
19:            t ← ComputeThreshold((v_1, ..., v_n), M, S)
20:            /* samples with values from interval [a, t] for matcher M */
21:            S_1 ← S([a, t], M)
22:            c_1 ← [a, t]
23:            /* samples with values from interval (t, b] for matcher M */
24:            S_2 ← S((t, b], M)
25:            c_2 ← (t, b]
26:            S' ← {S_1, S_2}
27:        end if
28:        for all S_i ∈ S' do
29:            /* constructs subtree T_i from subset S_i and adds it as a child to node M with
       condition c_i on edge */
30:            T_i ← BuildTree(S_i, M, c_i)
31:        end for
32:    end if
33:    return T
34: end function
```

Example. To conclude, let us provide a simple illustrative example. For simplicity only three matchers are used: `Matched Thesauri` (which uses previous confirmed matching results for the evaluation), `Levenshtein Distance` (which computes the shortest edit distance from one string to another for operations insert, update and delete of a character) and `N-gram` (which computes the number of the same N-grams in two string where an *N-gram* is a sequence of N characters in a given string). The C5.0 algorithm works in the following steps:

- In the beginning, the training set S contains 14 samples. `Matched Thesauri` has discrete values 0 and 1. `Levenshtein Distance` and `N-gram` have values from continuous interval $[0, 1]$. Gain values are computed for all matchers. `Matched Thesauri` has the highest gain value of 0.371, that is why `Matched Thesauri` is selected as the root of the constructed decision tree. Set S is divided into two parts $S(0,$ `Matched Thesauri`$)$ and $S(1,$ `Matched Thesauri`$)$.
- Set $S(1,$ `Matched Thesauri`$)$ contains samples that have value 1 for matcher `Matched Thesauri` and it contains only samples from the same match class. New leaf **match** is added as a child to node `Matched Thesauri`.

- Set $S_{MT0} = S(0,$ `Matched Thesauri`$)$ consists of samples with value 0 for matcher `Matched Thesauri` and results from various classes, so this set has to be further divided. Gain values are computed and matcher with the highest gain value, i.e. `N-gram`, is added as a child of node `Matched Thesauri`. The threshold value for `N-gram` matcher with continuous range is 0.071 and set S_{MT0} is divided into two subsets $S_{N1} = S([0, 0.071],$ `N-gram`$)$ and $S_{N2} = S((0.071, 1],$ `N-gram`$)$.
- There are only mismatch results in set S_{N1}, so leaf **mismatch** is added as a child to node `N-gram`.
- Set S_{N2} also contains results from one class – match. Another leaf **match** is added to node `N-gram`.

The final trained decision tree is displayed in Figure 2.

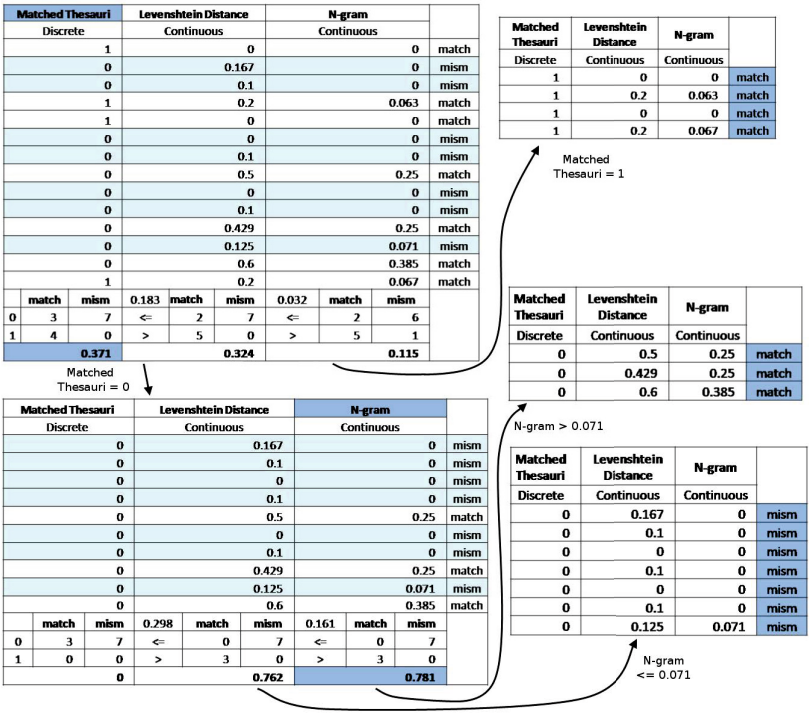

Fig. 1. Data used for training of decision tree in example

5 Experiments

For the purpose of evaluation of the described approach we have performed an extensive set of experiments. Due to space limitations and complexity of the experiments, this section contains description of only one of the experiments and a discussion of its results. The complete set of experiments can be found

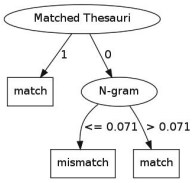

Fig. 2. Sample decision tree

in [21]. Particular experiments differ in used sets of matchers, training sets, etc. - *efficiency of measure methods depending on element types*; *usage of different matchers*; *comparison of domain thesaurus and user-confirmed matches*; *quality of matching of decision trees trained from different sets.*

The proposed approach was implemented in the *eXolutio* [18] tool and replaces the original approach [11] (whose disadvantages were described in Section 4.1). *eXolutio* is based on the MDA approach and models XML schemas at two levels – PIM and PSM. *eXolutio* allows the user to manually design a common PIM schema and multiple PSM schemas with interpretations against the PIM schema. Mapping between the two levels allows to propagate a change to all the related schemas.

All experiments were run on a standard personal computer with the following configuration: `Intel(R) Core(TM) i5-3470 3.20 GHz processor, 8 GB RAM, OS 64-bit Windows 7 Home Premium SP1.`

The following sets of XML schemas have been used for training of the decision tree:

- `BMEcat` is a standard for exchange of electronic product catalogues[2].
- `OpenTransAll` is a standard for business documents[3].
- `OTA` focuses on the creation of electronic message structures for communication between the various systems in the global travel industry[4].

PIM Schemas. A PIM schema used for experiments describes a common interface for planning various types of holidays. It can be found in [21].

XML Schemas for Experiments. For evaluation the following XML schemas were used:

- Artificial XML schema `01_Hotel` designed for the purpose of this work. It describes basic information about hotels.
- Realistic XML schemas: `02_HotelReservation`[5], `03_HotelAvailabilityRQ`[6]

[2] `www.bmecat.org`
[3] `www.opentrans.de`
[4] `www.opentravel.org`
[5] `http://kusakd5am.mff.cuni.cz/hb/schema/reservation`
[6] `http://itins4.madisoncollege.edu/IT/152121advweb/XMLExamples/unit3/schemaSimple/HotelAvailabilityRQ.xsd`

Domain Thesaurus. The domain thesaurus contains sets of words that are related semantically. The thesauri are used during matching by `Dictionary` matcher. The domain thesaurus for the domain of hotels is as follows (related words are marked by ∼): address ∼ location, accommodation ∼ hotel, boarding ∼ meal, count ∼ amount, lengthOfStay ∼ numberOfNights.

5.1 Separate Decision Trees and Common Decision Tree for Classes and Attributes Experiment

The presented experiment is designed from the following observation: Efficiency of methods used to measure similarity between elements depends on the type of elements – if they are **classes** or if they are **attributes**. In this experiment two sets of decision tree are used: *two separate trees for classes and for attributes* and *one common tree for classes and attributes.*

Experiment Setup

- **Used Schemas:** `01_Hotel`, `02_HotelReservation`, `03_HotelAvailabilityRQ`
- **Decision Tree:**
  - Separate decicion tree for attributes (in Figure 3) and for classes (in Figure 4)
  - Common decision tree (in Figure 5)
- **Decision Tree Training Set:**
  - Set of XSD Schemas: `OTA`
  - Sample count:
    - Separate decision tree for attributes: 27,942 match pairs
    - Separate decision tree for classes: 27,793 match pairs
    - Common decision tree: 55,815 match pairs
- **Thesaurus for Dictionary:** None
- **Thesaurus for Matched Thesauri:** None
- **Matchers:** `Children` (which compares the structural similarity of child nodes or neighboring nodes of classes), `DataType` (which compares data types), `Dictionary`, `Length Ratio` (which computes the ratio of lengths of two input strings), `Levenshtein Distance`, `Matched Thesauri`, `Prefix` (which compares whether the string s_1 is a prefix of the string s_2 or the other way around)

In the presented experiment no additional source of information was used. Both sets of decision trees were trained without domain thesauri and without previous user matches. Both sets of decision trees are induced from the same set of training samples `OTA`, particularly match pairs of XML schema `OTA_HotelAvailGetRQ.xsd` and XML schema `OTA_HotelAvailGetRS.xsd`. A separate decision tree for classes and attributes is trained only from match pairs of classes and attributes respectively. The common tree is trained from both sets together. The final decision trees are shown in Figures 3, 4 and 5.

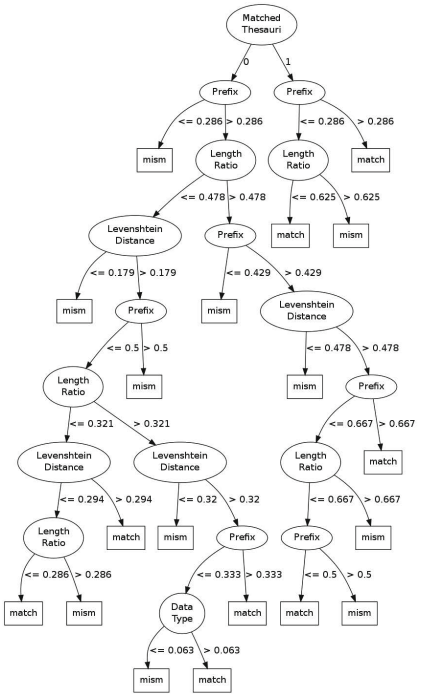

Fig. 3. A separate decision tree for attributes for experiment *SeperateTrees*

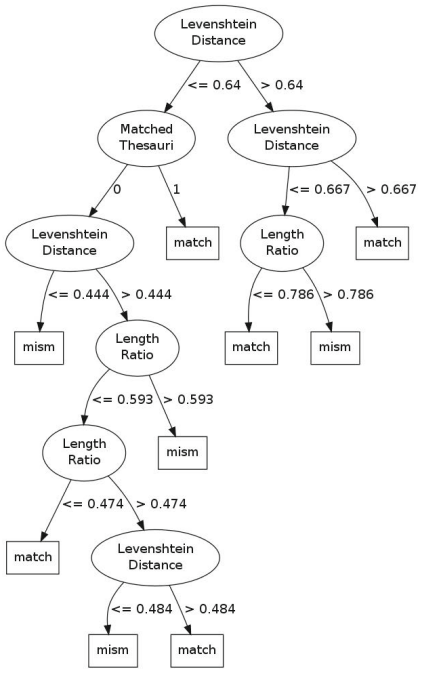

Fig. 4. Separate decision tree for classes for experiment *SeparateTrees*

The root of the separate decision tree for attributes is `Matched Thesauri`, all the other trees in this experiment have `Levenshtein Distance`. The matcher at the second level is the same for both branches and they have the same threshold. Especially the subtree for mapping pairs that are contained in `Matched Thesauri` is interesting. We would assume that this subtree should be smaller or even a leaf with the value 'match'. This could be explained by errors in user annotation of mapping results – the same match pair is annotated with different matching decision than the previous one or some mapping pairs have different meaning in different context.

The separate decision tree for classes is relatively simple. It contains only matchers `Levenshtein Distance`, `Matched Thesauri` and `Length Ratio`, other matchers are not used. It corresponds with the original observation that some methods are more effective for certain types of elements. Matchers whose similarity values do not distinguish mapping pairs enough are not included. Pairs that are contained in the thesaurus are directly suggested as matches.

The threshold value for `Matched Thesauri` matcher in the root of the common tree and the separate tree for classes is nearly similar. The common decision tree

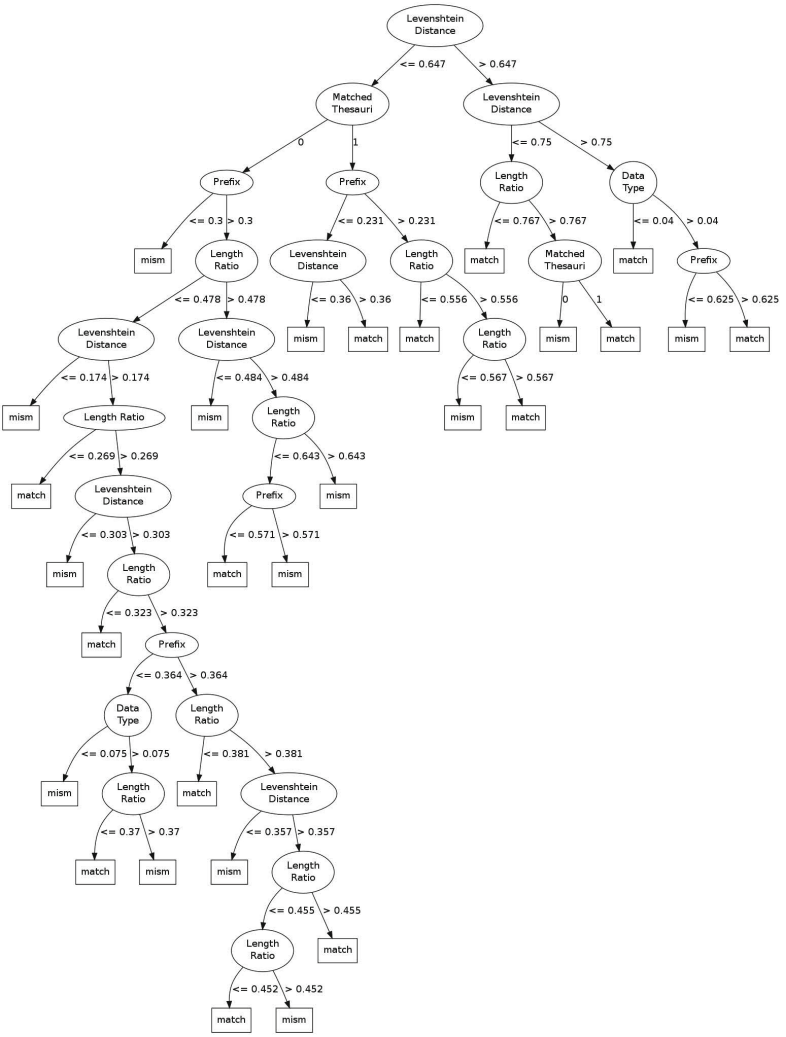

Fig. 5. Common decision tree for experiment *SeparateTrees*

is the most complex from the above mentioned. The common tree also contains two subtrees for `Matched Thesauri`. The first one is at the second level and it contains two full subtrees for both the values. The right subtree for pairs that are contained in thesauri is more complex than the tree in the separate tree for attributes. This could be caused by a larger number of training samples that allows for more detail resolution. The second one, i.e. `Matched Thesauri`, is directly a parent of the leaves.

In Figures 6, 7, 8 and 9 there are displayed the histograms of the match quality measures – Precision, Recall, F-Measure and Overall respectively. All

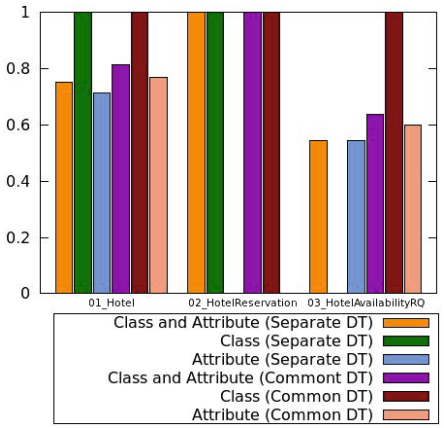

Fig. 6. Precision for experiment *Sepearate-Trees*

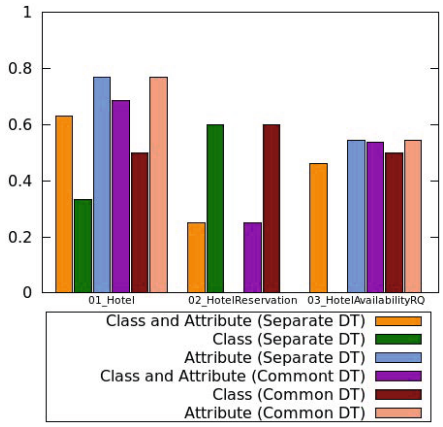

Fig. 7. Recall for experiment *Separate-Trees*

the measures are at first computed for both types of elements together and then separately for attribute and class elements.

In Figure 6 Precision is high for classes in all schemas and for both types of trees. The quality of mapping decision differs significantly with the type of element, but the training set contains a similar number of match pairs for classes (27,793 match pairs) and attributes (27,942 match pairs). The separate tree for classes did not suggest any mapping pair as a match for schema `03_HotelAvailabilityRQ`, just as the separate tree for attributes for schema `02_HotelReservation`. Attributes in schema `03_HotelAvailabilityRQ` are difficult to identify for all the decision trees. All Precision values are from the interval [0.545, 0.769] – they identified almost the same number of relevant results as irrelevant.

Recall is lower than Precision in all the cases except for schema `01_Hotel` and the separate tree for attributes in Figure 7. There were no true positives attributes for schema `02_HotelReservation` for both trees and no true positives classes for schema `03_Hotel- AvailabilityRQ` for separate tree. Values of Recall are lower for attributes than Recall for classes.

In Figure 8 the values for F-Measure are equal for schema `02_HotelReservation` for classes for both trees. Post-match effort for adding false negatives (FN) and removing false positives (FP) is quite high in all cases in Figure 9. The highest value of Overall is 0.6.

The hypothesis was not confirmed, all similarity measures are higher for the common decision tree that is trained from a bigger set of training examples, the quality of the decision tree seems to depend more on the size of the set of training samples. The best score was achieved for Precision. Both trees in this experiment had a larger number of FN than FP. They miss a match suggestion

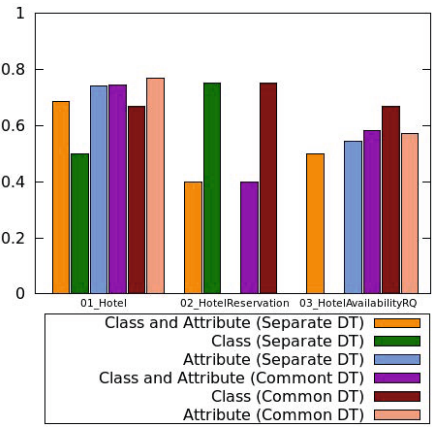

Fig. 8. F-Measure for experiment *SeperateTrees*

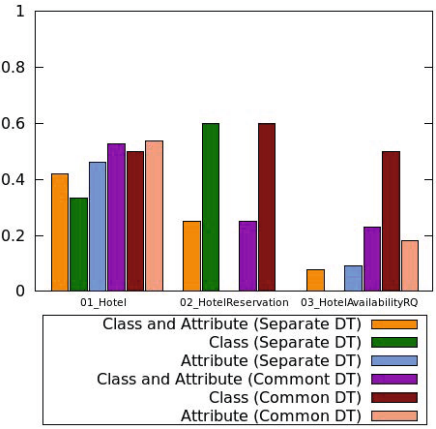

Fig. 9. Overall for experiment *SeparateTrees*

more than they incorrectly suggest it as a match pair. It could be improved by adding an auxiliary source of information or a new matcher.

Examples of matching results from this experiment are shown in Table 2. Match pair `numberOfNights` – `LengthOfStay` is difficult to identify without an auxiliary source of information for both sets of decision tree. Match pairs `CheckOutDate` – `CheckOut`, `ContactInfo` – `Contact` and `BedType` – `ReservationType` were identified correctly by the common tree and incorrectly by separate decision trees.

Table 2. Examples of mapping results for experiment *SeparateTrees*

	XSD	PIM	DT type	DT	User	Result
C	ContactInfo	Contact	Separate	Mismatch	Match	FN
C	ContactInfo	Contact	Common	Match	Match	TP
A	Fax	FaxNumber	Separate	Match	Match	TP
A	Fax	FaxNumber	Common	Match	Match	TP
A	numberOfNights	LengthOfStay	Separate	Mismatch	Match	FN
A	numberOfNights	LengthOfStay	Common	Mismatch	Match	FN
A	CheckOutDate	CheckOut	Separate	Mismatch	Match	FN
A	CheckOutDate	CheckOut	Common	Match	Match	TP
A	BedType	ReservationType	Separate	Match	Mismatch	FP
A	BedType	ReservationType	Common	Mismatch	Mismatch	TN

Further experiments with various hypotheses, e.g. training with sets of different sizes, with different sets of matchers, or with usage of auxiliary information, can be found in [21].

6 Conclusion

Schema matching, i.e. the problem of finding correspondences, relations or mappings between elements of two schemas, has been extensively researched and has a lot of different applications. In this paper a particular application of schema matching in MDA is explored. We have implemented our approach within a modeling and evolution management tool called *eXolutio* [18] which is based on the idea of MDA. This mapping is very useful in case of a change, because changes in one place are propagated to all the related schemas. The presented schema matching approach is used to identify mappings between PIM and PSM level of MDA, representing an interpretation of a PSM element against a PIM element.

We have explored various approaches to schema matching and selected the most promising possible approach for our application – schema matching using a decision tree. This solution is dynamic, versatile, highly extensible, quick and has a low level of user intervention and a low level of required auxiliary information. We have extended the previous work [11] by utilization of the C5.0 algorithm for training of decision tree from a large set of user-annotated schema pairs. Our approach is now more versatile, extensible and reusable. Further we evaluated our approach on a wide range of experiments and implemented a module that is easily extensible. We also implemented a user-friendly interface for evaluation of mappings suggested by the decision tree, i.e. a solid background for further experiments.

A straightforward extension of this work is to expand the set of available matchers with more powerful matchers, e.g. with a matcher that uses the *WordNet*[7] thesaurus for synonyms. Further possibilities are for example string matchers[8] and the *Soundex* matcher[9]. Also the user interface leaves a space for improvement. We could add an interface for evaluation of matches during the preparation phase of decision tree training or dynamic editing of trained decision tree – remove, move, add matcher node, change results in leaves or threshold on edges.

Acknowledgment. This work was supported by the project SVV-2014-260100 and the GAUK grant no. 1416213.

References

1. Do, H.H., Rahm, E.: COMA – A system for flexible combination of schema matching approaches. In: Proceedings of the 28th International Conference on Very Large Data Bases, Pages, pp. 610–621. VLDB Endowment, Hong Kong (2002)
2. Aumueller, D., Do, H.H., Massmann, S., Rahm, E.: Schema and Ontology Matching with COMA++. In: Proceeding SIGMOD 2005 Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data, pp. 906–908 (2005) ISBN:1-59593-060-4

[7] `http://wordnet.princeton.edu/`
[8] `http://secondstring.sourceforge.net/`
[9] `http://www.archives.gov/research/census/soundex.html`

3. Duchateau, F., Bellahsene, Z., Coletta, R.: A flexible approach for planning schema matching algorithms. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part I. LNCS, vol. 5331, pp. 249–264. Springer, Heidelberg (2008)
4. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity Flooding: A Versatile Graph Matching Algorithm. In: Proceeding ICDE 2002 Proceedings of the 18th International Conference on Data Engineering, p. 117. IEEE Computer Society, Washington, DC (2002)
5. Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0, 5th edn. W3C Recommendation (November 26, 2008), http://www.w3.org/TR/REC-xml.
6. Quinlan, J.R.: Induction of Decision Trees. Machine Learning 1(1), 81–106 (1986)
7. Quinlan, J.R.: C4.5: programs for machine learning. Morgan Kaufmann Publishers Inc, San Francisco (1993) ISBN:1-55860-238-0
8. Breiman, L., Friedman, J., Olshen, R., Stone, C.: Classification and Regression Trees. Chapman & Hall, New York (1984)
9. Hunt, E. B., Marin, J., Stone, P. T.: Experiments in Induction. Academic Press, New York (1966)
10. Mehta, M., Agrawal, R., Rissanen, J.: SLIQ: A fast scalable classier for data mining. In: Apers, P.M.G., Bouzeghoub, M., Gardarin, G. (eds.) EDBT 1996. LNCS, vol. 1057, pp. 3–540. Springer, Heidelberg (1996)
11. Stárka, J.: Similarity of XML Data. Master's thesis, Charles University in Prague (2010), http://www.ksi.mff.cuni.cz/~holubova/dp/Starka.pdf
12. Shasha, D., Zhang, K.: Approximate Tree Pattern Matching. Pattern Matching Algorithms, pp. 341–371. Oxford University Press (1997)
13. Nierman, A., Jagadish, H.V.: Evaluating Structural Similarity in XML Documents. In: Proceedings of the Fifth International Workshop on the Web and Databases, pp. 61–66 (2002)
14. Li, W., Clifton, C.: SemInt: a tool for identifying attribute correspondences in heterogeneous databases using neural network. Data & Knowledge Engineering 33(1), 169–123 (2000) ISSN 0169-023X
15. Chen, P.: The Entity-Relationship Model – Toward a Unified View of Data. ACM Transactions on Database Systems, 9–36 (March 1976)
16. Quinlan, R.: C5.0, http://www.rulequest.com/see5-unix.html.
17. Stárka, J., Mlýnková, I., Klímek, J., Nečaský, M.: Integration of web service interfaces via decision trees. In: Proceedings of the 7th International Symposium on Innovations in Information Technology, pp. 47–52. IEEE Computer Society, Abu Dhabi (2011) ISBN: 978-1-4577-0311-9
18. Klímek, J., Mlýnková, I., Nečaský, M.: eXolutio: Tool for XML and Data Management. In: CEUR Workshop Proceedings, pp. 1613–1673 (2012) ISSN: 1613-0073
19. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. Object Management Group (2003), http://www.omg.org/docs/omg/03-06-01.pdf
20. Nečaský, M., Mlýnková, I., Klímek, J., Malý, J.: When conceptual model meets grammar: A dual approach to XML data modeling. International Journal on Data & Knowledge Engineering 72, 1–30 (2012) ISBN:3-642-17615-1, 978-3-642-17615-9
21. Jílková, E.: Adaptive Similarity of XML Data. Master's thesis, Charles University in Prague (2013), http://www.ksi.mff.cuni.cz/~holubova/dp/Jilkova.pdf
22. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA data mining software: an update. SIGKDD Explorations 11(1) (2009)