An Experiment on the Matching and Reuse of XML Schemas

Jianguo Lu[1], Shengrui Wang[2], Ju Wang[1]

[1]School of Computer Science, University of Windsor
jlu@cs.uwindsor.ca, ju_wang@yahoo.com
[2] Department of Computer Science, University of Sherbrooke
Shengrui.wang@usherbrooke.ca

Abstract XML Schema is becoming an indispensable component in developing web applications. With its widespread adoption and its web accessibility, XML Schema reuse is becoming imperative. To support XML Schema reuse, the first step is to develop mechanism to search for relevant XML Schemas over the web. This paper describes a XML Schema matching system that compares two XML Schemas. Our matching system can find accurate matches and scales to large XML Schemas with hundreds of elements. In this system, XML Schemas are modelled as labeled, unordered and rooted trees, and a new tree matching algorithm is developed. Compared with the tree edit-distance algorithm and other schema matching systems, it is faster and more suitable for XML Schema matching.

Keywords: *XML Schema, Schema match, Software component search*

1 Introduction

XML Schema has become an indispensable component in web application development. Schemas are used to represent all kinds of data structure in programming, and are often mapped to classes. To some extent, we can think XML Schemas are similar to data types or classes in traditional programming language. What makes XML Schema different from traditional software components is that it is widely available on the web, encoded in XML and programming language independent, and adopted by all the major software vendors. All these features make XML Schema reuse not only imperative, but also have the potential to succeed beyond traditional software component reuse. We can envision that almost any data structure that you can think of will be available on the web. Programmers need a search tool to find the relevant schema instead of developing the schema from scratch.

Schema matching has its root in software component search and software agent search. Both have a long history. [17] provides a good survey in component search, and [22] is the seminal paper on software agent matching, which also inspired numerous works on web service searching. Schema matching is also widely studied in the area of database area [6] [14], with the aim to integrate relational and semi-structured data. [18] surveys the works in this area.

This paper describes a schema matching system that generates element mappings between two schemas. One design rationale of the system is that it should work effectively and efficiently – generating good results in acceptable time, such that it is

capable of matching real life schemas with several hundreds of elements. We model an XML Schema as an unordered, labeled and rooted tree. In general, an XML schema corresponds to a directed graph in which recursive definitions are represented by loops and reference definitions are represented by cross edges. The graph representation is not adopted in our work for two reasons. First, intuitively the directed graph representation of an XML Schema still encompasses a hierarchical structure similar to a tree, with a few "loop" exceptions. Secondly and more importantly, approximate graph matching [3] is too computationally costly as we have investigated in [11]. Our recent algorithm in graph matching employed strong heuristics to reduce search space, but still can only deal with graphs with dozens of node [11]. Obviously, graph matching algorithms would be difficult to match XML Schemas with hundreds of nodes.

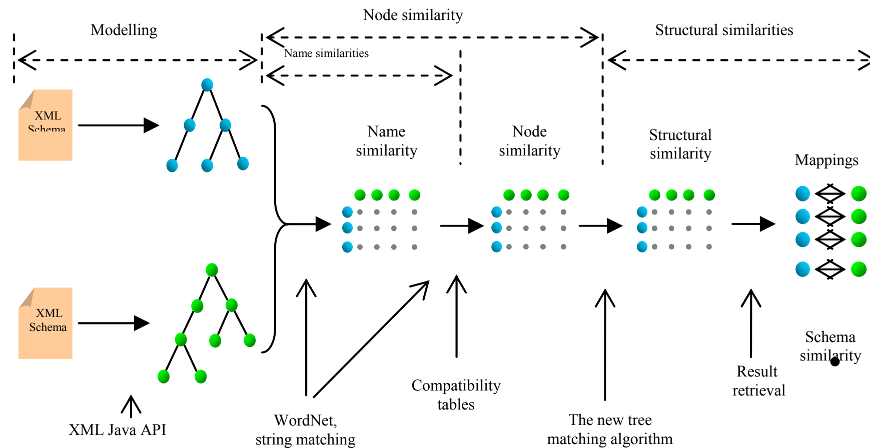

Figure 1: Matching process

Our new tree matching algorithm identifies the structural relations by extracting approximate common substructures in two trees. Our observation on the properties of XML Schemas shows that similar schemas are made up of similar elements and these elements are connected similarly, that is, similar schemas (or similar portions of schemas) have similar ancestor-descendent and sibling relations. Based on this, the algorithm uses heuristics to reduce the searching space dramatically, and achieves a trade-off between matching optimality and time complexity.

Figure 1 depicts the structure of the matching system. We compute three types of similarities for every node pairs, i.e., *name similarity*, *node similarity* and *structural similarity*. Name similarity is related to calculating the relationship between two names. It is the main entity used for computing similarity between two nodes, other entities include data types and cardinalities information. The structural similarity shows the relation between two sub-trees rooted at these two nodes.

The system is tested extensively using about 600 XML Schemas in total. We evaluated both matching accuracy and computational efficiency of our system. Comparisons were made with the traditional edit distance tree matching algorithm [24]

and a popular XML Schema matching system COMA [6]. The results show that our new tree matching algorithm outperforms these two methods, and can be used to match larger schemas that contain hundreds of elements.

2 Modelling XML Schemas as Trees

An XML Schema is modeled as a *labeled unordered rooted tree*. Each element or attribute of the schema is translated into a *node*. Attributes and elements that reside inside an element are translated as children of the element node. The names of elements and attributes, along with some optional information such as data types and cardinalities, are the labels of the nodes.

The modelled tree does not include every detail of an XML Schema. Excluded information falls into two categories. One is related to elements or attributes such as default value and value range. The other is relevant to structure, such as element order indicators. Although by XML Schema standard the order of the elements matters, it is ignored in our tree model based on the assumption that the order does not make differences as big as changing the labels.

Modelling XML Schema is a tedious task due to the complexity of XML Schema. During the modelling, we need to take care of the following constructs in XML Schema, to insure that a schema is modelled as a tree.

Reference Definition

Reference definition is a mechanism to simplify schema through the sharing of common segments. To transform his structure into a tree, we duplicate the shared segment under the node that refers to it. By doing this, we increased the number of nodes. In implementation of the modelling, we create an array which contains the distinct node labels and establish connections from each node to this array. In subsequent processes, the node labels are handled based the array instead of the nodes themselves.

There are two types of references in XML Schema specification: data type reference and name reference. Data type reference is created by the clause '*type=dataTypeName*' (where '*dataTypeName*' is not a built-in data type), and the referred segment is a <*complexType*> or <*simpleType*>; while name reference is created by '*ref=elementName*', and referred segment must be a <*element*>. All the referred types or elements must be *top level* such that they are nested in <*schema*> only. Therefore, our solution is that: build two lists called '*referred*' and '*referring*', list 'referred' contained all the top level elements and types (both complex and simple), and list 'referring' contain the elements having 'type' or 'ref' reference; then after scanning the schema file, for every element in 'referring', we physically duplicate the segment which they refer. Solving those segments which are from outside of the schema file follows the same method as *importing* and *inclusion*.

Recursive Definition

Recursive definition happens when a leaf element refers to one of its ancestors. This definition also breaks the tree structure, and it has to be solved differently from the way of solving reference definition, otherwise it falls into infinite loop.

Matching recursively defined node is equivalent to matching the inner node being referred. So we utilize a detecting procedure, which scans the path from a node up to

the root of the tree to find out whether this node refers to its ancestor or not. Once a node which has recursive definition is found, we cut the connection and mark the node with recursive property to distinguish it from its referred ancestor.

Namespace

Namespace is a way to avoid name ambiguity, such as two same data type names in one schema file, by assigning them to different vocabularies. This is accomplished by adding unique *URI*s and giving them aliases. The aliases serve as prefixes, such as 'xsd:' in the example, to associate the terms with certain vocabularies – namespaces. In our implementation, namespace affects reference definitions in three ways: built-in data type, user-defined data type, and element reference. To support this feature, our program tracks every prefix and its corresponding URI, takes them and the term right after the prefix as one unit, then put this unit into the reference solving.

Importing and Including

Importing and *including* are mechanisms of reusing elements and attributes defined in other schema files. Including limits the sharing within the same namespace, and importing can cross different namespaces. When being imported, the imported schema file's information is provided in the <import> tag, including the file name, location and the imported namespace. Our program also parses and models this schema, then together with its namespace, brings its top level elements and types into the 'referred' list. If any of them are referred by the components in the original schema file, they will be handled by the reference solving process. For including, the included file's information is kept in <include> tag, and the same method is applied to solve including with the difference of namespace. The namespace for including is as the same as the original schema file.

Extension

Extension allows new elements and attributes being added. For this situation, we first need to solve the type reference, so we treat the base clause as the same as type reference. After getting the base type being duplicated, we process the newly added components, converting them to nodes and join them as siblings to the duplicated ones.

Grouping

Grouping is similar to complex type definition, providing a way of reusing predefined components. The most often used grouping is attribute grouping, which is specified by <*attributeGroup*> tag. We use the same way as type reference to solve this situation, i.e., add the <*attributeGroup*> definition and reference element to the 'referred' list, then duplicate the referred group.

3 Node Similarity

Since a label of a node consists of name, datatype, and cardinality information, the node similarity is computed based on these entities. Among them the name similarity is the most complex one.

3.1 Name Similarity

Name similarity is a score that reflects the relation between the meanings of two names, such as tag name or attribute name, which usually comprised of multiple words or acronyms. The steps of computing name similarity include tokenization, computing the semantic similarities of words by WordNet, determing the relations of tokens by a string matching algorithm if they can not be solved by WordNet, and calculating the similarity between two token lists.

Tokenization
Quite often a tag name consists of a few words. It is necessary to split up the name into tokens before computing the semantic similarity with another one. This operation is called *tokenization*. A *token* could be a word, or an abbreviation. Although there are no strict rules of combining tokens together, conventionally, we have some clues to separate them from each other such as case switching, hyphen, under line, and number. For instance: 'clientName' is tokenized into 'client' and name, and 'ship2Addr' to 'ship', '2', and 'Addr'.

Computing Semantic Similarity Using WordNet

Once a name is tokenized into a list of words, we use WordNet [25] to compute the similarity between the words. WordNet builds connections between four types of POS (Part of Speech), i.e., noun, verb, adjective, and adverb. The smallest unit in WordNet is *synset*, which represents a specific meaning of a word. It includes the word, its explanation, and the synonyms of this meaning. A specific meaning of one word under one type of POS is called a *sense*. Each sense of a word is in a different synset. For one word, one type of POS, if there are more than one sense, WordNet organizes them in the order from the most frequently used to the least frequently used.

Based on WordNet and its API, we use synonym and hypernym relations to capture the semantic similarities of tokens. Given a pair of words, once a path that connects the two words is found, we determine their similarity according to two factors: the length of the path and the order of the sense involved in this path.

Searching the connection between two words in WordNet is an expensive operation due to the huge searching space. We impose two restrictions in order to reduce the computational cost. The first one is that only synonym and hypernym relations are considered, since exhausting all the relations is too costly. This restriction is also adopted in some related works [1]. Another restriction is to limit the length of the searching path. If a path has not been connected within a length limit, we stop further searching and report no path found.

In our implementation, we use the following formula to calculate the semantic similarity: $wordSim(s,t) = senseWeight(s) * senseWeight(t) / pathLength$

Where s and t denote the source and target words being compared. $senseWeight$ denotes a weight calculated according to the order of this sense and the count of total senses.

We performed a comparison with seven other approaches on the set of word pairs in [12]. In terms of correlation, ours exceeds four approaches and falls behind three of them. Considering that the method we use is simpler and scalable, our similarity measure is acceptable.

Similarity between Words outside Vocabulary

Words outside English vocabulary are often used in schemas definition, such as abbreviations ("qty") and acronyms ('purchase order' as PO). In this case WordNet is no longer applicable, and we use edit-distance string matching algorithm. By doing this, the measurement reflects the relations between the patterns of the two strings, rather than the meaning of the words.

Similarity between Token Lists

After breaking names into token lists, we determine the similarity between two names by computing the similarity of those two token lists, which is reduced to the bipartite graph matching problem [13]. It can be described as follows: the node set of a graph G can be partitioned into two subsets of disjoint nodes X and Y such that every edge connects a node in X with a node in Y, and each edge has a non-negative weight. The task is to find a subset of node-disjoint edges that has the maximum total weight.

3.2 Similarity of Built-in Data Type

In XML Schema there are 44 built-in data types, including nineteen primitive ones and twenty-five derived ones. To reduce the number of combinations, we create seven data type categories, i.e., *binary*, *boolean*, *dataTime*, *float*, *idRef*, *integer*, and *string* that cover the 44 data types. The compatibility table is built for the seven categories. After this, when comparing two data types, first we check which category these types belong to, then extract the similarity measure from the category compatibility table.

3.3 Similarity of Cardinalities

XML Schema allows the specification of minimum and maximum occurrences, i.e., cardinality, for elements. The range of cardinality is from 0 to unbounded. It is impossible and unnecessary to compare all the cardinalities in this range. As a result, we apply a threshold. When cardinalities are equal to or bigger than it, we treat the cardinality as this threshold.

4 Approximate Tree Matching

Tree matching is an extensively studied problem. The classical tree edit distance matching algorithm [28] is not an adequate solution for 1) it is not fast enough as is shown in our experiment explained in section 5; 2) it must preserve the tree ancestor structure during the match, hence may miss better matches.

Take the two schemas in Figure 2 for example. In those two schemas, there are two substructures that are very similar. One is about car information, the other one is driver information. Intuitively we would like to match those substructures. However, with the traditional tree edit distance algorithms, that kind of matching is not easy to achieve because shifting two sub-trees (e.g., exchange the position of driver information with car information in Schema 1) requires many edit operations. Based on this observation, we generalized the concept of common substructures between two trees to approximate common substructures (ACS), and developed an efficient tree matching algorithm for extracting a disjoint set of the largest ACSs [25]. This disjoint set of ACSs represents the most likely matches between substructures in the two schemas. In addition, the algorithm provides structural similarity estimate for

each pair of substructures including, of course, the overall similarity between the two schemas. Using our algorithm to match the above car-driver schemas, both driver and car nodes and their components can be matched, even though car is an ancestor of driver in schema one, and it is the other way around in schema two.

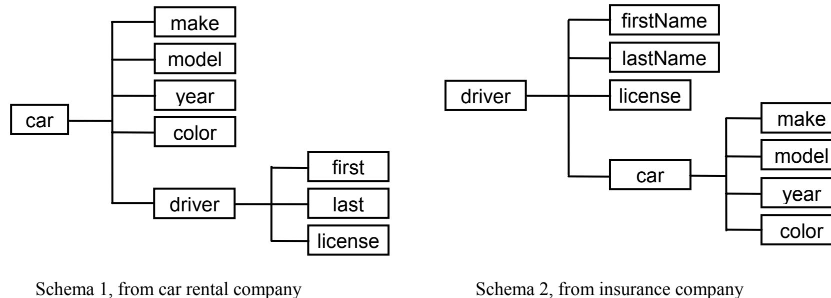

Figure 2: Example schemas from two businesses

5 Experiment

Our system is compared with the traditional edit distance tree matching algorithm for labeled unordered trees [19] that is implemented by us, and the popular schema matching system COMA [6].

5.1 Data

The experiments are performed upon the XML Schemas we collected from various sources. The first group comprises five purchase order schemas which are used in the evaluation of COMA [6]. We choose the same test data to compare with COMA. The second group includes 86 large schemas from www.xml.org. These are large schemas that are proposed by companies and organizations to describe the concepts and standards for particular areas. We use these large schemas to evaluate system efficiency. The third group consists of 95 schemas that are collected from HITIS [10]. These schemas are designed to be the standards of interfaces between hospitality related information systems, such as hotel searching, room reservation, etc. Group four consists of 419 schemas extracted from WSDL files that describe the schemas of the parameters of web service operations. These schemas are small in general. Group three and four are used to test the accuracy of our matching system. Since most of them are relatively small, they are easy to read and judge manually.

5.2 Accuracy

5.2.1 Comparison with Edit-distance Algorithm

Figure 3 compares the precision and recall between our algorithm (method 1) and edit distance algorithm (method 2). The test cases are from data group 1, which consists of 5 purchase orders that are also used in COMA.

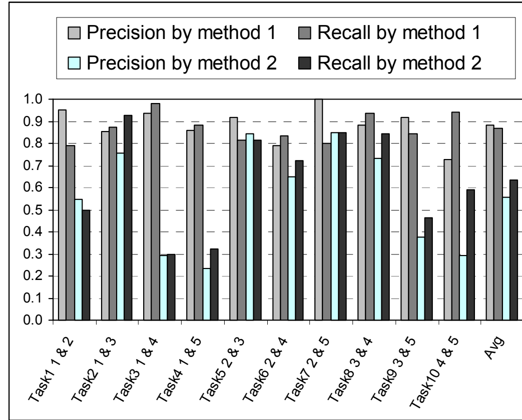

Figure 3: Precision and recall for our method (method 1) and edit-distance algorithm (method 2)

The figure shows that our algorithm outperforms the edit distance tree matching algorithm consistently. Both algorithms adopt node removal operation and use iterative improvement heuristic to search the approximate result. The major difference between these two algorithms is that we deal with two nodes (one for each tree) each time, recursively match two trees from leaves to roots, and the node removal operation is limited to the child level of current nodes only. The edit distance tree matching algorithm always takes two trees, tries to remove some nodes in the range of entire trees each time, compares and keeps the state with smallest distance. Reviewing these five purchase order schemas supports our schema properties observation again – similar concepts described by XML are made up of similar elements, and these elements are constructed in similar ways. Simply speaking, good mappings between two similar schemas could be found by a few node removal operations. Our algorithm takes advantage of this condition and limits the range of node removal. Therefore it removes fewer nodes, but achieves better result. On the other hand, for the edit distance tree matching algorithm, when the input size is large, the wide range of node removal increases the searching space and decreases the chance of getting good mappings.

5.2.2 Comparison with COMA

COMA maintains a library of different matchers (matching methods) and can flexibly combine them to work out the result. It introduced a manual reuse strategy which can improve the results but needs human assistance. Besides precision and recall, COMA adopts the overall measurement that combines precision and recall.

We focus on two matcher combinations in COMA, i.e., 'All' – the best no-reuse combination, and 'All+SchemaM' – the best reuse involved combination. Together with the result of our matching system, the precision, recall and overall measure are compared in Table 1.

	COMA (All)	COMA (All+ SchemaM)	Ours
Precision	0.95	0.93	0.88
Recall	0.78	0.89	0.87
Overall	0.73	0.82	0.75

Table 1: COMA and our algorithm

From this table, we can conclude that in terms of overall accuracy, our matching system outperforms COMA 'All' combination, and falls behind 'All+SchemaM' combination on matching the given five purchase order schemas. Considering the 'All+Schema' needs human assistance, our matching system works well.

5.2.3 Top-k Precision

We use Top-k precision method to assess the schema relations reported by our algorithm and tree edit distance algorithm. *Top-k precision* is defined as $p_{Top-k} = |ReportCorrect_k| / k$, where $ReportCorrect_k$ is the set of correct results in the top-k return ones. The experiment for assessing the schema relations is performed on data group three and four, and is designed as follows: in each group, we randomly pick a schema; compare it with every schema in this group using both algorithms; then we sort the returned schemas. Next, we take the union of top-k schemas from the two lists, subsequently, based on the union set, we manually determine which schema(s) should not be ranked in top-k, and finally compute the top-k precision for each algorithm. In order to get better overall measurement, we compute top-3 and top-5 precisions, repeat above process, and take averages. Figure 4 summarizes the evaluation results which are based on 10 random schemas in group 3 and 20 schemas in group 4.

The result shows that 1) using either algorithm in a schema group, top-3 precision is better than top-5 precision; 2) both algorithms get better precision on schema group 3; and 3) our algorithm gets better overall results than the edit distance algorithm.

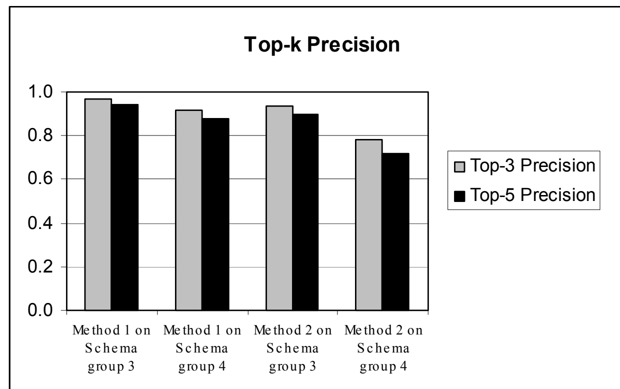

Figure 4: Top-3 and top-5 precision

The reason of better top-3 and top-5 precisions for group 3 is that all the schemas in this group are collected from one domain. Most files have similar piece of information, a few of them are even identical.

5.3 Performance

The performance is assessed using group two that consists of 86 large schemas. This experiment is performed on a computer with single Intel Pentium 4 3.0GHz CPU and 1G memory. The operating system is Red Hat Linux release 9. We match every two schemas in this group are matched, so there are 3655 matching tasks in total. Due to the high computation cost of method 2, we bypass this method for schemas that exceed 150 nodes. Therefore, the count of matching tasks that the two algorithms participate is different.

Figure 5 shows the execution times of the three methods. We divide the input size, represented by the multiplication of node count of the two trees, into several intervals, then count the number of matching tasks, and calculate the average execution times for each interval. As we can see, for method 2, there are only six matching tasks when input size is from 16k to 20k, and there is no task when input size is over 20k.

It illustrates the increasing trend for all of the three execution times while the input size gets large. Besides, we can conclude that the preparing part is a heavy job, and the new tree matching algorithm is faster than the edit distance tree matching algorithm.

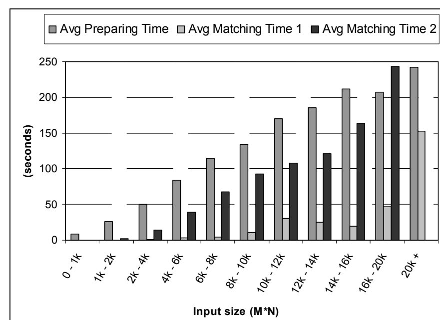

Figure 5: Execution time

There are some tasks in preparing part, including modelling, computing node similarity, and preparing related data structures for later matching. Clearly, the majority cost is spent on computing node similarity, and more specifically, on computing semantic similarities. Computing semantic similarities is a very expensive task: given two words, the program exhausts their relations stored in WordNet, and tries to find the highest ranked connection. Even through we restrict the relation to synonymy and hypernym only, the searching space is still huge. However, we could adopt some alternatives to reduce the dependence of WordNet, such as reuse pre-calculated result and build user-specified similarity tables.

Our tree matching algorithm is faster than the edit distance tree matching algorithm. Due to the same reason describe in previous section, our tree matching algorithm limits node removal operation, therefore it reduces the searching space.

In conclusion, compared with the edit distance tree matching algorithm, our algorithm generates better results in shorter time for most of the matching tasks, especially when input size is large. Therefore it is more applicable in real life schema matching problems.

5.4 Implementation of the Matching System

This matching system is developed using Java. SAX XML parser in Sun JAXP package is used to parse XML schema, and WordNet API JWNL is used to access WordNet's dictionaries. The experiments generate huge amount of result data, therefore, we employ Oracle database to manage the data. In addition, after creating proper indices, we benefit from Oracle database for quick searching and retrieval operations. There are two types of user interfaces, i.e., command line and web-based. Command line interfaces are used to debug the system and conduct experiments, while the Web-based one is used to show the experiment results in a user-friendly way so that the evaluation work is easier.

6 Conclusion and Future Work

This paper presents our first step in creating an XML Schema searching system to support schema reuse. There are already hundreds of thousands of XML Schemas on the web, which need to be collected, classified, indexed, and searched upon. We are developing an XML Schema repository, and providing various search mechanisms ranging from simple keyword search to the sophisticated tree matchings as described in this paper.

To achieve this goal, one salient feature of our system is our exhaustive approach to each step in the matching process, coping with the engineering details in real application scenario, with the ultimate goal for practical application. For example, we considered the details of modelling a XML Schema to a tree, and the practical issues in using WordNet to compute the name similarity. Most existing schema matching systems are prototypes that omitted those engineering details.

The experiment results also show that our new tree matching algorithm can match large trees with hundreds of nodes effectively and efficiently. In a matching task, most executing time is spent on computing node similarities, especially the connection time with WordNet. We are improving this by precalculating and caching the word relationships.

We are also applying schema matching system in web service searching, since the major components in web services are XML Schemas which defines the parameters in the operations of a web service.

References

1. S. Banerjee, T. Pedersen. Extended Gloss Overlaps as a Measure of Semantic Relatedness. *IJCAI*, 2003.
2. H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Lett*. 1997 18(8), 689-694.

3. H. Bunke, Recent Developments in Graph Matching, *Proc. 15th Int. Conf. on Pattern Recognition*, Barcelona, 2000, Vol 2, 117 – 124.
4. P. V. Biron, A. Malhotra (ed.), W3C, April 2000, 'XML Schema Part 2: Datatypes', http://www.w3.org/TR/xmlschema-2/
5. A. Budanitsky, G. Hirst. Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures. In *Proceedings of the NAACL 2001 Workshop on WordNet and Other Lexical Resources*, Pittsburgh, June 2001.
6. H. Do, E. Rahm. COMA A system for flexible combination of schema matching approaches. *VLDB* 2002
7. H. Do., S. Melnik, E. Rahm, Comparison of Schema Matching Evaluations, *Proc. GI-Workshop "Web and Databases"*, Erfurt, Oct. 2002.
8. A. Doan, P. Domingos, A. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *proc. SIGMOD Conference*, 2001
9. A. Gupta, N. Nishimura. Finding largest subtrees and smallest supertrees. *Algorithmica*, 21:183-210, 1998
10. HITIS - Hospitality Industry Technology Integration Standard, http://www.hitis.org.
11. A. Hlaoui and S. Wang, "A Node-Mapping-Based Algorithm for Graph Matching", submitted (and revised) to J. Discrete Algorithms 2004.
12. M. Jarmasz, S. Szpakowicz. Roget's Thesaurus and Semantic Similarity. *RANLP* 2003
13. H. W. KUHN. The Hungarian method for the assignment problem. *Naval Research Logistics Quarterly 2* (1955), 83–97.
14. M. L. Lee, L. H. Yang, W. Hsu, X. Yang. XClust: clustering XML schemas for effective integration. In *Proceedings of the eleventh international conference on Information and knowledge management*, Pages: 292 - 299, 2002
15. J. Madhavan, P. A. Bernstein, E. Rahm. Generic schema matching with Cupid. *VLDB*, 2001.
16. S. Melnik, H. Garcia-Molina, E. Rahm. Similarity flooding: a versatile graph matching algorithm and its application to schema matching. *ICDE* 2002.
17. A Mili, R Mili, RT Mittermeir, A survey of software reuse libraries, Annals of Software Engineering, 1998.
18. E. Rahm, P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB J.*, 10(4):334-350, 2001.
19. D. Shasha, J. Wang, K. Zhang, F. Y. Shih. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man, and Cybernetics.* Vol 24, NO.4, April 1994.
20. D. Shasha, J. T. L. Wang, R. Giugno, Algorithmics and Applications of Tree and Graph Searching, In *Proc. PODS*'02, June 3-5 2002.
21. H. Su, S. Padmanabhan, M. Lo. Identification of syntactically similar DTD Elements for schema matching. *WAIM*, 2001.
22. K. Sycara, M. Klusch, S. Widoff, J. Lu, Dynamic Service Matchmaking among Agents in Open Information Environments, Journal of ACM SIGMOD Record, 28(1):47-53, 1999.
23. H. S. Thompson, D. Beech, M. Maloney, N. Mendelsohn (ed.), W3C, April 2000, 'XML Schema Part 1: Structures', http://www.w3.org/TR/xmlschema-1/.
24. J. Wang, B. A. Shapiro, D. Shasha, K. Zhang, K. M. Currey. An algorithm for finding the largest approximately common substructures of two trees. *IEEE Trans*. *PAMI* 20, 1998, 889-895.
25. Shengrui Wang, Jianguo Lu, Ju Wang, Approximate Common Structures in XML Schema Matching. Submitted.
26. WordNet – a lexical database for English. http://www.cogsci.princeton.edu/~wn/
27. K. Zhang, D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245-1263, Dec. 1989.
28. K. Zhang, D. Shasha, J. T. L. Wang, Approximate Tree Matching in the Presence of Variable Length Don't Cares, *Journal of Algorithms*, 16(1):33-66.