# Generic Schema Matching with Cupid

**Jayant Madhavan**[1]
University of Washington
jayant@cs.washington.edu

**Philip A. Bernstein**
Microsoft Corporation
philbe@microsoft.com

**Erhard Rahm**[1]
University of Leipzig
rahm@informatik.uni-leipzig.de

## Abstract

Schema matching is a critical step in many applications, such as XML message mapping, data warehouse loading, and schema integration. In this paper, we investigate algorithms for generic schema matching, outside of any particular data model or application. We first present a taxonomy for past solutions, showing that a rich range of techniques is available. We then propose a new algorithm, Cupid, that discovers mappings between schema elements based on their names, data types, constraints, and schema structure, using a broader set of techniques than past approaches. Some of our innovations are the integrated use of linguistic and structural matching, context-dependent matching of shared types, and a bias toward leaf structure where much of the schema content resides. After describing our algorithm, we present experimental results that compare Cupid to two other schema matching systems.

## 1 Introduction

*Match* is a schema manipulation operation that takes two schemas as input and returns a mapping that identifies corresponding elements in the two schemas. Schema matching is a critical step in many applications: in E-business, to help map messages between different XML formats; in data warehouses, to map data sources into warehouse schemas; and in mediators, to identify points of integration between heterogeneous databases.

Schema matching is primarily studied as a piece of these other applications. For example, schema integration uses matching to find similar structures in heterogeneous schemas, which are then used as integration points [1,3,12]. Data translation uses matching to find simple data transformations [10]. Given the importance of XML message mapping, we expect to see match solutions to appear next in this context.

Schema matching is challenging for many reasons. Most importantly, even schemas for identical concepts may have structural and naming differences. Schemas may model similar but non-identical content. They may be expressed in different data models. They may use similar words to have different meanings. And so on.

Today, schema matching is done manually by domain experts, sometimes using a graphical tool [8]. At best, some tools can detect exact matches automatically – even minor name and structure variations lead them astray.

Like [4], we believe that Match is such a pervasive, important and difficult problem that it should be studied independently. Moreover, since it is critical to such a wide variety of tools, we believe it should be built as an independent component and be *generic*, meaning that it can apply to many different data models and application domains. To support these positions, in this paper we offer the following contributions: a *taxonomy* of approaches used by different applications, to show the complexity of the solution space; a *new match algorithm* that uses more powerful techniques than past approaches and is generic across data models and application areas; and *experimental comparisons* of our implementation with others, to show the benefits of our approach and a way of evaluating other implementations in the future.

Ultimately, we see Match as a key component of a general-purpose system for managing models. By *model*, we mean a complex structure that describes a design artifact such as a database schema, XML schema, UML model, workflow definition, or web-site map. The vision of Model Management is a system that manipulates models generically, to match and merge them, and invert and compose mappings between them [2]. This paper focuses on just one piece of that vision, the Match operation.

The rest of the paper is organized as follows. We define the schema matching problem in Section 2. Section 3 looks at past solutions, presents a taxonomy for schema matching techniques, and reviews systems that use them. Section 4 summarizes our approach in a new match algorithm, Cupid, whose details are described in Sections 5-7. Section 8 reports on experiments comparing Cupid with two other algorithms. Section 9 is the conclusion.

## 2 The Schema Matching Problem

A schema consists of a set of related *elements*, such as tables, columns, classes, or XML elements or attributes. The result of a Match operation is a *mapping*. A mapping consists of a set of *mapping elements*, each of which indicates that certain elements of schema S1 are related to certain elements of schema S2. For example, in Figure 1 a mapping between purchase order schemas *PO* and *POrder* could include a mapping element *m* that relates ele-

*Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment*

**Proceedings of the 27th VLDB Conference,**
**Roma, Italy, 2001**

[1] Work performed while at Microsoft Research.

| PO | POrder |
|---|---|
| Lines | Items |
|    Item |    Item |
|       Line |       ItemNumber |
|       Qty |       Quantity |
|       Uom |       UnitOfMeasure |

**Figure 1: Two schemas to be matched**

ment *Lines.Item.Line* to element *Items.Item.ItemNumber.*

In general, a mapping element may also have an associated expression that specifies its semantics (called a *value correspondence* in [9]). For example, *m*'s expression might be "*Lines.Item.Line=Items.Item.ItemNumber*." We do not treat such expressions in this paper. Rather, we only address *mapping discovery,* which returns mapping elements that identify related elements of the two schemas. Since we are not concerned with mapping expressions, we treat mappings as non-directional.

The related problem of *query discovery* operates on mapping expressions to obtain queries for actual data translation. Both types of discovery are needed. Each is a rich and complex problem that deserves independent study. Query discovery is already recognized as an independent problem, where it is usually assumed that a mapping either is given [9] or is trivial [14].

Schema matching is inherently subjective. Schemas may not completely capture the semantics of the data they describe, and there may be several plausible mappings between two schemas (making the concept of a single best mapping ill-defined). This subjectivity makes it valuable to have user input to guide the match and essential to have user validation of the result. This guidance may come via an initial mapping, a dictionary or thesaurus, a library of known mappings, etc. Thus, the goal of schema matching is: *Given two input schemas in any data model and, optionally, auxiliary information and an input-mapping, compute a mapping between schema elements of the two input schemas that passes user validation.*

## 3 A Taxonomy of Matching Techniques

Schema matchers can be characterized by the following orthogonal criteria (a longer survey based on this taxonomy appears in [13]):

- *Schema* vs. *Instance based* – Schema-based matchers consider only schema information, not instance data [1,12]. Schema information includes names, descriptions, relationships, constraints, etc. Instance-based matchers either use meta-data and statistics collected from data instances to annotate the schema [9], or directly find correlated schema elements, e.g. using machine learning [5].
- *Element* vs. *Structure granularity* – An element-level matcher computes a mapping between individual schema elements, e.g. an attribute matcher [6]. A structure-level matcher compares combinations of elements that appear together in a schema, e.g. classes or tables whose attribute sets only match approximately [1].
- *Linguistic based* – A *linguistic* matcher uses names of schema elements and other textual descriptions. Name matching involves: putting the name into a canonical form by stemming and tokenization; comparing equality of names; comparing synonyms and hypernyms using generic and domain-specific thesauri; and matching sub-strings. Information retrieval (IR) techniques can be used to compare descriptions that annotate some schema elements.
- *Constraint based* – A *constraint-based* matcher uses schema constraints, such as data types and value ranges, uniqueness, required-ness, cardinalities, etc. It might also use intraschema relationships such as referential integrity.
- *Matching Cardinality* – Schema matchers differ in the cardinality of the mappings they compute. Some only produce 1:1 mappings between schema elements. Others produce n:1 mappings, e.g. one that maps the combination of *DailyWages* and *WorkingDays* in the source schema to *MonthlyPay* in the target.
- *Auxiliary information* – Schema matchers differ in their use of auxiliary information sources such as dictionaries, thesauri, and input match-mismatch information. Reusing past match information can also help, for example, to compute a mapping that is the composition of mappings that were performed earlier.
- *Individual* vs. *Combinational* – An individual matcher uses a single algorithm to perform the match. Combinational matchers can be one of two types: *Hybrid* matchers use multiple criteria to perform the matching [1,6,10]. *Multiple matchers* run independent match algorithms on the two schemas and combine the results [5].

We now look at some published implementations in light of the above taxonomy.

The SEMINT system is an instance-based matcher that associates attributes in the two schemas with *match signatures* [6]. These consist of 15 constraint-based and 5 content-based criteria derived from instance values and normalized to the [0,1] interval, so each attribute is a point in 20-dimensional space. Attributes of one schema are clustered with respect to their Euclidean distance. A neural network is trained on the cluster centers and then is used to obtain the most relevant cluster for each attribute of the second schema. SEMINT is a hybrid element-level matcher. It does not utilize schema structure, as the latter cannot be mapped into a numerical value.

The DELTA system groups all available meta-data about an attribute into a text string and then applies IR techniques to perform matching [4]. Like SEMINT, it does not make much use of schema structure.

The LSD system uses a multi-level learning scheme to perform 1:1 matching of XML DTD tags [5]. A number of base learners that use different instance-level matching schemes are trained to assign tags of a mediated schema to data instances of a source schema. A meta-learner combines the predictions of the base learners. LSD is thus a multi-strategy instance-based matcher.

The SKAT prototype implements schema-based matching following a rule-based approach [11]. Rules are formulated in first-order logic to express match and mismatch relationships and methods are defined to derive new matches. It supports name matching and simple structural matches based on is-a hierarchies.

The TranScm prototype uses schema matching to drive data translation [10]. The schema is translated to an internal graph representation. Multiple handcrafted matching rules are applied in order at each node. The matching is done top-down with the rules at higher-level nodes typically requiring the matching of descendants. This top-down approach performs well only when the top-level structures of the two schemas are quite similar. It represents an element-level and schema-based matcher.

The DIKE system integrates multiple ER schemas by exploiting the principle that the similarity of schema elements depends on the similarity of elements in their vicinity [12]. The relevance of elements is inversely proportional to their distance from the elements being compared, so nearby elements influence a match more than ones farther away. Linguistic matching is based on manual inputs.

ARTEMIS, the schema integration component of the MOMIS mediator system, matches classes based on their name affinity and structure affinity [1,3]. MOMIS has a description logic engine to exploit constraints. The classes of the input schemas are clustered to obtain global classes for the mediated schema. Linguistic matching is based on manual inputs using an interface with WordNet [16].

Both DIKE and ARTEMIS are hybrid schema-based matchers utilizing both element- and structure-level information. We give more details about them in Section 8.

## 4 The Cupid Approach

The prototypes of the previous section illustrate, and in many cases were the original source of, the matching approaches described in our taxonomy. However, each of them is an incomplete solution, exploiting at most a few of the techniques in our taxonomy. This is not really a criticism. Each of them was either a test of one particular approach or was not designed to solve the schema matching problem per se, and therefore made matching compromises in pursuit of its primary mission (usually schema integration). However, the fact remains that none of them provide a complete general-purpose schema matching component. We believe that the problem of schema matching is so hard, and the useful approaches so diverse, that only by combining many approaches can we hope to produce truly robust functionality.

In the rest of this paper, we explain our new schema matching component, Cupid. In addition to being generic, our solution has the following properties:

- It includes automated linguistic-based matching.
- It is both element-based and structure-based.
- It is biased toward similarity of atomic elements (i.e. leaves), where much schema semantics is captured.
- It exploits internal structure, but is not overly misled by variations in that structure.
- It exploits keys, referential constraints and views.
- It makes context-dependent matches of a shared type definition that is used in several larger structures.

Cupid shares some general approaches with past algorithms, though not the algorithms themselves, such as: rating match quality in the [0,1] interval, clustering similar terms (SEMINT), and matching structures based on local vicinity (DIKE, ARTEMIS). The Cupid approach is schema-based and not instance-based.

To explain the algorithm, we first restrict ourselves to hierarchical schemas. Thus, we model the interconnected elements of a schema as a *schema tree*. A simple relational schema is an example of a schema tree; a schema contains tables, which contains columns. An XML schema with no shared elements is another example; elements contain sub-elements, which in turn contain other sub-elements or attributes. Later in the paper, we enrich the model to capture more semantics, making it more generic.

We summarize the overall algorithm below in a running example. We want to match the two XML schemas, *PO* and *Purchase Order*, in Figure 2. The schemas are encoded as graphs, where nodes represent schema elements. Although even a casual observer can see the schemas are very similar, there is much variation in naming and structure that makes algorithmic matching quite challenging.

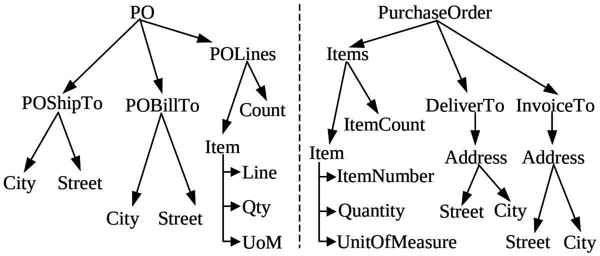

**Figure 2: Purchase order schemas**

Like previous approaches [1,3,5,6,12], we attack the problem by computing similarity coefficients between elements of the two schemas and then deducing a mapping from those coefficients. The coefficients, in the [0,1] range, are calculated in two phases. The first phase, called *linguistic* matching, matches individual schema elements based on their names, data types, domains, etc. We use a thesaurus to help match names by identifying short-forms (*Qty* for *Quantity*), acronyms (*UoM* for *UnitOfMeasure*) and synonyms (*Bill* and *Invoice*). The result is a *linguistic similarity coefficient*, *lsim*, between each pair of elements.

The second phase is the *structural matching* of schema elements based on the similarity of their contexts or vicinities. For example, *Line* is mapped to *ItemNumber* because their parents, *Item*, match and the other two children of *Item* already match. The structural match depends in part on linguistic matches calculated in phase one. For example, *City* and *Street* under *POBillTo* match *City* and *Street* under *InvoiceTo*, rather than under *DeliverTo*, because *Bill* is a synonym of *Invoice* but not of *Deliver*. The result is a *structural similarity coefficient*, *ssim*, for each pair of elements.

The *weighted similarity* (*wsim*) is a mean of *lsim* and *ssim*: $wsim = w_{struct} \times ssim + (1-w_{struct}) \times lsim$, where the constant $w_{struct}$ is in the range 0 to1. A mapping is created by choosing pairs of schema elements with maximal weighted similarity.

In the next two sections, we describe the linguistic and structural matching phases in more detail. We then extend the algorithm beyond tree structures in Section 7.

## 5 Linguistic Matching

The first phase of schema matching is based primarily on schema element names. In the absence of data instances, such names are probably the most useful source of information for matching. We also make modest use of data types and schema structure in this phase. This section outlines the process. More details are presented in [7].

Linguistic matching proceeds in three steps: *normalization*, *categorization* and *comparison*.

1. *Normalization* – Similar schema elements in different schemas often have names that differ due to the use of abbreviations, acronyms, punctuations, etc. So, as part of our normalization step, we perform *tokenization* (parsing names into tokens based on punctuation, case, etc.), *expansion* (identifying abbreviations and acronyms) and *elimination* (discarding prepositions, articles, etc.). In each of these steps we use a thesaurus that can have both common language and domain-specific references.
2. *Categorization* – Schema elements in each schema are separately clustered into *categories*. This is based on their data types, schema hierarchy and linguistic content (from their names). For example, there might be categories for real-valued elements and another one for money-related elements. A schema element can belong to multiple categories.
3. *Comparison – Linguistic similarity coefficients* (*lsim)* are computed between schema elements by comparing the tokens extracted from their names. We use a thesaurus that has synonymy and hypernymy relationships for this purpose. We also perform sub-string matching. The purpose of the earlier categorization is to reduce the number of one-one comparisons of elements in the two schemas, by only considering schema elements that belong to similar categories in the two schemas. See [7] for details.

The result of this phase is a table of *lsim* coefficients between elements in the two schemas. The computed *lsim* values are in the range [0,1], with 1 indicating a perfect linguistic match.

## 6 Structure Matching

In this section we present a structure matching algorithm for hierarchical schemas, i.e. tree structures. For each pair of schema elements the algorithm computes a *structural similarity, ssim,* which is a measure of the similarity of the contexts in which the elements occur in the two schemas. From *ssim* and *lsim*, the weighted similarity *wsim* is computed, as described in Section 4.

### 6.1 Matching Schema Trees

The *TreeMatch* algorithm in Figure 3 is based on the following intuitions:

(a) Atomic elements (leaves) in the two trees are similar if they are individually (linguistic and data type) similar, and if elements in their respective vicinities (ancestors and siblings) are similar.

(b) Two non-leaf elements are similar if they are linguistically similar, and the subtrees rooted at the two elements are similar.

(c) Two non-leaf schema elements are structurally similar if their leaf sets are highly similar, even if their immediate children are not. This is because the leaves represent the atomic data that the schema ultimately describes.

Figure 3 describes the basic tree-matching algorithm that exploits the above intuition.

```
TreeMatch(SourceTree S, TargetTree T)
  for each $s \in S$, $t \in T$ where s,t are leaves
      set ssim (s,t) = datatype-compatibility(s,t)
  S' = post-order(S), T' = post-order(T)
  for each s in S'
     for each t in T'
        compute ssim(s,t) = structural-similarity(s,t)
        wsim(s,t) = $w_{struct}$.ssim(s,t) + (1-$w_{struct}$).lsim (s,t)
        if wsim(s,t) > $th_{high}$
          increase-struct-similarity(leaves(s),leaves(t),$c_{inc}$)
        if wsim(s,t) < $th_{low}$
          decrease-struct-similarity(leaves(s),leaves(t),$c_{dec}$)
```

**Figure 3: The TreeMatch algorithm**

The structural similarity of two leaves is initialized to the type compatibility of their corresponding data types. This value ([0,0.5]) is a lookup in a compatibility table. Identical data types have a compatibility of 0.5. (A max of 0.5 allows for later increases in structural similarity.)

The elements in the two trees are then enumerated in post-order, which is uniquely defined for a given tree. Both the inner and outer loops are executed in this order.

The first step in the loop computes the *structural similarity* of two elements. For leaves, this is just the value of *ssim* that was initialized in the earlier loop. When one of the elements is not a leaf, the structural similarity is computed as a measure of the number of leaf level matches in the subtrees rooted at the elements that are being compared (intuition (c)). We say that a leaf in one schema has a *strong link* to a leaf in the other schema if their weighted similarity exceeds a threshold $th_{accept}$. This indicates a potentially acceptable mapping. We estimate the structural similarity as the fraction of leaves in the two subtrees that have at least one strong link (and are hence mappable) to some leaf in the other subtree, i.e.:

$$ssim(s,t) = \frac{\left| \begin{array}{l} \{x \mid x \in leaves(s) \wedge \exists y \in leaves(t), stronglink(x,y)\} \\ \cup \{x \mid x \in leaves(t) \wedge \exists y \in leaves(s), stronglink(y,x)\} \end{array} \right|}{| leaves(s) \cup leaves(t) |}$$

where *leaves(s)* = set of leaves in the subtree rooted at *s*. We chose not to compute a 1-1 bipartite matching (used in [12]) as it is computationally expensive and would preclude m:n mappings (that often make sense).

If the two elements being compared are highly similar, i.e. if their weighted similarity exceeds the threshold $th_{high}$, we increase the structural similarity (*ssim*) of each pair of leaves in the two subtrees (one from each schema) by the factor $c_{inc}$ (*ssim* not to exceed 1). The rationale is that leaves with highly similar ancestors occur in similar contexts. So the presence of such ancestors should reinforce their structural similarity. For example, in Figure 2, if *POBillTo* is highly similar to *InvoiceTo*, then the structural similarity of their leaves *City-Street* would be increased, to bind them more tightly than to other *City-Street* pairs. For similar reasons, if the weighted similarity is less than the threshold $th_{low}$, we decrease the structural similarities of leaves in the subtrees by the factor $c_{dec}$. The linguistic similarity, however, remains unchanged.

The similarity computation has a mutually recursive flavor. Two elements are similar if their leaf sets are similar. The similarity of the leaves is increased if they have ancestors that are similar. The similarity of intermediate substructure also influences leaf similarity: if the subtree structures of two elements are highly similar, then multiple element pairs in the subtrees will be highly similar, which leads to higher structural similarity of the leaves (due to multiple similarity increases). The post-order traversals ensure that before two elements $e_1$ and $e_2$ are compared, all the elements in their subtrees have already been compared. This ensures that $e_1$'s and $e_2$'s leaves capture the similarity of $e_1$'s and $e_2$'s intermediate subtree structure before $e_1$ and $e_2$ are compared.

The structural similarity of two nodes with a large difference in the number of leaves is unlikely to be very good. Such comparisons lead to a large number of element similarities that are below the threshold $th_{low.}$ We prevent this by only comparing elements that have a similar number of leaves in their subtrees (say within a factor of 2). In addition to only comparing relevant elements, such a pruning step decreases the number of element pairs that need to be compared.

Instead of using leaves, we could consider only the immediate descendants of the elements being compared. Using the leaves for measuring structural similarity identifies most matches that this alternative scheme would. In addition, using the leaves ensures that schemas that have a moderately different sub-structure (e.g. nesting of elements) but essentially the same data content (similar leaves) are correctly matched.

The post-order traversal results in a *bottom-up* matching of the two schemas. Such an approach is more expensive than *top-down* matching [10]. But, a bottom-up approach is more conservative and is able to match moderately varied schema structures. A top-down approach is optimistic and will perform poorly if the two schemas differ considerably at the top level.

### 6.2 Mappings

The output of schema matching is a set of mapping elements, which were described in Section 2. Mapping elements are generated using the computed linguistic and structural similarities. In the simplest case we might just need leaf-level mapping elements. For each leaf element *t* in the target schema, if the leaf element *s* in the source schema with highest weighted similarity to *t* is acceptable ($wsim(s, t) \geq th_{accept}$), then a mapping element from *s* to *t* is returned. This resulting mapping may be 1:n, since a source element may map to many target elements.

The exact nature of a mapping is often dependent on requirements of the module that accepts these mappings. For example, Query Discovery might require a 1:1 mapping instead of the 1:n mapping returned by the naïve scheme above. Such requirements need to be captured by a data-model- or tool-specific mapping-generator that takes the computed similarities as input.

To generate non-leaf mappings, we need a second post-order traversal of the two schemas to re-compute the similarities of non-leaf elements. This is because the updating of leaf similarities during tree-match may affect the structural similarity of non-leaf nodes since they were first calculated. After this, a scheme similar to leaf-level mapping generation can be used.

## 7 Extending to General Schemas

### 7.1 Schema Graphs

The schemas we have looked at so far are trees. Real-world schemas are rarely trees, since they share substructure and have referential constraints. To extend our techniques to these cases, we first present a generic schema model that captures more semantics, leading to non-tree schemas. We then extend our match algorithm to use it by handling shared types and referential constraints.

In our generic schema model, a schema is a rooted graph whose nodes are elements. We will use the terms nodes and elements interchangeably. In a relational schema, the elements are tables, columns, user-defined types, keys, etc. In an XML schema the elements are XML elements and attributes (and simpleTypes, complexTypes, and keys/keyrefs in XML Schema (XSD) [17]).

Elements are interconnected by three types of relationships, which together lead to non-tree schema graphs. The first is *containment*, which models physical containment in the sense that each element (except the root) is contained by exactly one other element. (Containment also has *delete propagation* semantics, though we do not use that property here.) E.g. a table contains its columns, and is contained by its relational schema. An XML attribute is contained by an XML element. The schema trees we have used so far are essentially containment hierarchies.

A second type of relationship is *aggregation*. Like containment, it groups elements, but is weaker (allows multiple parents and has no delete propagation). E.g. a compound key aggregates columns of a table. Thus, a schema graph need not be a tree (a column can have two parents: a table and a compound key).

The third type of relationship is *IsDerivedFrom*, which abstracts *IsA* and *IsTypeOf* relationships to model shared type information. Schemas that use them can be arbitrary graphs (e.g. cycles due to recursive types). In XSD, an IsDerivedFrom relationship connects an XML element to its *complex type*. In OO models, IsDerivedFrom connects a subtype to its supertype. IsDerivedFrom shortcuts containment: if an element *e* IsDerivedFrom a type *t*, then *t*'s members are implicitly members of *e*. E.g. if *USAddress* specializes *Address*, then an element *Street* contained by *Address* is implicitly contained by *USAddress* too.

## 7.2 Matching Shared Types

When matching schemas expressed in the above model, the linguistic matching process that we described earlier is unaffected. We may, however, choose not to linguistically match certain elements, e.g. those with no significant name, such as keys. Structure matching *is* affected. Before this step, we convert the schema to a tree, for two reasons: to reuse the structure matching algorithm for schema trees and to cope with *context-dependent mappings*.

An element, such as a shared type, can be the target of many IsDerivedFrom relationships. Such an element *e* might map to different elements relative to each of e's parents. For example, reconsider the XML schemas in Figure 2. Suppose we change the *PurchaseOrder* schema so that *Address* is a shared element, referenced by both *DeliverTo* and *InvoiceTo*. *POShipTo.Street* and *POBillTo.Street* now both map to *Address.Street* in *PurchaseOrder,* but for each of them the mapping needs to qualify *Address.Street* to be in the *context* of either *DeliverTo* or *InvoiceTo*. Including both of the mappings without their contexts is ambiguous, e.g. complicating query discovery. Thus, context-dependent mappings are needed. We achieve this by expanding the schema into a *schema tree*.

There can be many paths of IsDerivedFrom and containment relationships from the root of a schema to an element *e*. Each path defines a context, and thus is a candidate for a different mapping for *e*. By converting a schema to a tree, we can materialize all such paths. To do this, the algorithm, shown in Figure 4, does a pre-order traversal of the schema, creating a private copy of the subschema rooted at the target *t* of each IsDerivedFrom for each of *t*'s parents — essentially type substitution.

```
schema_tree  = construct_schema_tree(schema.root, NULL)
construct_schema_tree(Schema Element current_se,
                      Schema Tree Node current_stn)
  If current_se is the root or current_se was reached
                 through a containment relationship
    If current_se is not_instantiated then return current_stn
    new_stn = new schema tree node corresponding to current_se
    set new_stn as a child of current_stn
    current_stn = new_stn
  for each outgoing containment or isDerivedFrom relation
    new_se = schem element that is the target of the relationship
    construct_schema_tree(new_se, current_stn)
  return current_stn
```

**Figure 4: Schema tree construction**

For each element we add a schema tree node whose successors are the nodes corresponding to elements reachable via any number of IsDerivedFrom relationships followed by a single containment. Some elements are tagged *not-instantiated* (e.g. keys) during the schema tree construction and are ignored during this process.

We now have a representation on which we can run the TreeMatch algorithm of Section 6.

The similarities computed are now in terms of schema tree nodes. The resulting output mappings identify similar elements, qualified by contexts. This results in more expressive and less ambiguous mappings.

Schema tree construction fails if a cycle of containment and IsDerivedFrom relationships is present. Such cycles are the result of recursive type definitions. We do not have a complete solution for this case and defer treatment of cyclic schemas for future work.

In Section 7.4, we describe optimizations to mitigate the increased computation costs due to the expanded tree.

## 7.3 Matching Referential Constraints

Referential integrity constraints are supported in most data models. A foreign key in a relational schema is a referential integrity constraint. So are ID/IDREF pairs in DTDs, and key-keyref pairs in XSD.

Referential constraints are represented by *RefInt* elements in our model. Referential constraints are directed from a source (e.g. foreign key column) to a target (e.g. primary key that the foreign key refers to). Such RefInt elements aggregate the source, and *reference* (a new relationship) the target of such relationship. E.g. the modeling of a foreign key is as shown in Figure 5.

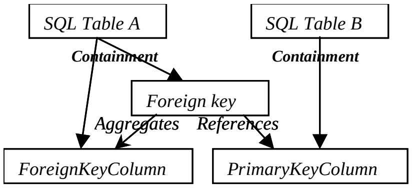

**Figure 5: RefInts in SQL schemas and XML DTDs**

The aggregates relationship is 1:n. For example, a compound foreign key aggregates its constituent columns. The foreign key references the single compound primary key element of the target table (which aggregates the key columns of that table). The 1:n nature of the reference relationship allows a single IDREF attribute to reference multiple IDs in an XML DTD.

We augment the schema tree with nodes that model referential constraints. The description below is for relational schemas, but a similar approach applies elsewhere.

We interpret referential constraints as potential *join views*. For each foreign key, we introduce a node that represents the join of the participating tables (see Figure 6). This reifies the referential constraint as a node that can be matched. Intuitively, it makes sense since the referential constraint implies that the join is meaningful. Notice that the join view node has as its children the columns from both the tables. The common ancestor of the two tables is made the parent of the new join view node.

These augmented nodes have two benefits. First, if two pairs of tables in the two schemas are related by similar referential constraints, then when the join views

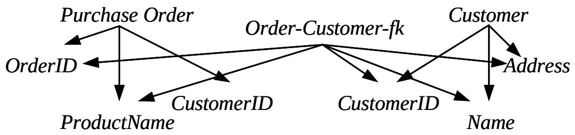

**Figure 6: Augmenting the schema tree**

for the constraints are matched, the structural similarities of those tables' columns are increased. This improves the structural match. Second, this enables the discovery of mappings between a join view in one schema and, a single table or other join views in the second schema.

The additional join view nodes create a directed acyclic graph (DAG) of schema paths. Since the inverse-topological ordering of a DAG (equivalent to post-order for a tree) is not unique, the algorithm is not Church-Rosser, i.e. the final similarities depend on the order in which nodes are compared. To make it Church-Rosser, we could add more ordering constraints. E.g. we could compare the RefInt nodes after the table nodes. However, determining which ordering would be best is still an open problem.

If a table has multiple foreign keys, we add one node for each of them. We also have the option of adding a node for each combination of these foreign keys (valid join views). However, we choose not to, in the interest of maintaining tractability. Similarly, the join view node that is added may also have a foreign key column (of the target table). We could expand these further thus escalating expansion of referential constraints, but choose not to, both for computation reasons and due to the lower relevance of tables at further distances.

## 7.4 Other Features

We now discuss some other features of Cupid.

- ***Optionality:*** Elements of semi-structured schemas may be marked as *optional*, e.g. non-required attributes of XML-elements. To exploit this knowledge, the leaves reachable from a schema tree node *n* are divided into two classes: *optional* and *required*. A leaf is optional if it has at least one optional node on each path from *n* to the leaf. The structural similarity coefficient expression is changed to reduce the weight of optional leaves that have no strong links (they are not considered in both the numerator and denominator of *ssim*). Therefore, nodes are penalized less for unmappable optional leaves than unmappable required leaves, so the matching is more tolerant to the former.
- ***Views:*** View definitions are treated like referential constraints. A schema tree node is added whose children are the elements specified in the view. This represents a common context for these elements and can be matched with views or tables of the other schema.
- ***Initial mappings:*** The matcher uses a user-supplied initial mapping to help initialize leaf similarities prior to structural matching (cf. Section 2). The linguistic similarity of elements marked as similar in the initial map is initialized to a predefined maximum value. Such a hint can lead to higher structural similarity of ancestors of the two leaves, and hence a better overall match. The user can modify a generated result map, make corrections, and then re-run the match with the corrected input map, thereby generating an improved map. Thus, initial maps are a way to incorporate user interaction in the matching process.
- ***Lazy expansion:*** Recall that schema tree construction expands elements into each possible context, much like type substitution. This expansion duplicates elements, leading to repeated comparisons of identical subtrees, e.g. the *Address* element is duplicated in multiple purchase order contexts and each is compared separately. We can avoid these duplicate comparisons by a *lazy schema tree expansion*, which compares elements of the schema graph before converting it to a tree. The elements are enumerated in inverse topological order of containment and IsDerivedFrom relationships. After comparing an element that is the target *t* of multiple IsDerivedFrom and containment relationships, multiple copies of the subtree rooted at *t* are made, including the structural similarities computed so far. This works because when two nodes are compared for the first time, their similarity depends only on that of their subtrees. We thus avoid identical recomputation for the context-dependent copies of the subtree.
- ***Pruning leaves:*** In a deeply nested schema tree with a large number of elements, an element *e* high in the tree has a large number of leaves. These leaves increase the computation time, even though many of them are irrelevant for matching *e*. Therefore, it may be better to consider only nodes in a subtree of depth *k* rooted at node *e* (pruning the leaves).

While comparing nearly identical schemas, it seems wasteful to compare leaves. To avoid this, first compare the immediate children of the nodes. If a very good match is detected, then skip the leaf level similarity computation.

# 8 Comparative Study

In this section we compare the performance of Cupid with two other schema matching prototypes, DIKE [12] and MOMIS [1], using simple canonical examples and real world schemas. The only prior published evaluation we know of is a comparison of the SEMINT and DELTA systems on US Air Force database schemas [4].

The three systems – Cupid, DIKE and MOMIS – are roughly comparable, in that they are purely schema-based and do element- and structure-level matching. Cupid and MOMIS also have a linguistics-based matching-component, which are significantly different. The three systems differ in their structure matching algorithms. A quantitative comparison of these systems is not possible for two reasons: (i) matching is an inherently subjective operation, and (ii) DIKE and MOMIS were designed with a primary goal of schema integration, so some of their features are biased towards integration, e.g. the type conflict resolution in DIKE, and the class level matching in MOMIS. Still, we believe experimental evaluation is essential to make progress on this hard problem.

The Cupid prototype, presented in Sections 4-7, currently operates on XML and relational schemas. The

| | Description | Cupid | DIKE | MOMIS-ARTEMIS$^{\beta}$ |
|---|---|---|---|---|
| 1 | Identical schemas | Y | Y | Y |
| 2 | Atomic elements with same names, but different data types$^{\chi}$ | Y | Y | Y |
| 3 | Atomic elements with same data types, but different names (a prefix or suffix is added) | Y | Y$^{\alpha}$ | Y |
| 4 | Different class names, but atomic elements same names and data types | Y | Y | Y |
| 5 | Different Nesting of the data – similar schemas with nested and flat structures | Y | Y | N |
| 6 | Type Substitution or Context dependent mapping | Y | N | N |

α - LSPD entries have to be added to identify corresponding elements

β - for each name the corresponding matching entry in the WordNet dictionary has to be chosen to ensure correct mappings

χ - data type compatibility tables are used by each tool

**Table 1: Comparison based on canonical example**

output mappings are displayed by BizTalk Mapper [8], which then compiles them into XSL translation scripts. In [7] we present some typical values of the thresholds used in the matching algorithm for this application.

The DIKE system [12] operates on ER models. The input includes a Lexical Synonymy Property Dictionary (LSPD) that contains linguistic similarity coefficients between elements in the two schemas. The schemas are interpreted as graphs with entities, relationships and attributes as nodes. The similarity coefficient of two nodes is initialized to a combination of their LSPD entry, data domains and keyness. This coefficient is re-evaluated based on the similarity of nodes in their corresponding vicinities — nodes further away contribute less. Conflict resolution is also performed on the schemas, e.g. an attribute might be converted to an entity to get a better integrated schema. The output is an integrated schema, and an abstracted schema (a simplification of the former).

The MOMIS mediator system [1] accepts schemas as class definitions. The WordNet system [16] is used to obtain *name affinities* among schema elements. For each element name, the user chooses an appropriate *word form* in WordNet, and narrows down its possible meanings to the most relevant ones. The description-logic-based ODB-Tools [1] is used to infer name affinities from inter-class relationships in the schema. ARTEMIS [3], the schema-mapping component of MOMIS, computes the *structural affinity* for all pairs of classes based on their name affinity and their respective class attributes. The classes of the input schemas are clustered into *global classes* of the mediated schema, based on their name and structural affinities. The attributes of clustered classes are fused, if possible, to determine the exact global class definitions.

## 8.1 Canonical Examples

We compared the matching performance of the three tools on canonical examples that try to isolate their matching properties. The test schemas used were object-oriented schemas with a small number of class definitions. The results are summarized in Table 1. A detailed analysis of the examples that were used and the results is presented in [7]. We make a few observations based on these:

**1.** Cupid is able to overcome some differences in schema element names due to the normalization performed as part of the linguistic matching. This requires user effort in the case of the other tools.

**2.** Cupid is robust to different nesting of schema elements due to its reliance of leaves rather than intermediate structure. DIKE is able to perform the same due to its entity merging operation.

**3.** Cupid is the only tool that is able to disambiguate context dependent mappings. The results in the case of DIKE are much dependent on the user feedback.

## 8.2 Real world example

We used two XML purchase orders, CIDX and Excel, from www.BizTalk.org (see Figure 7). We chose these particular schemas because, while somewhat similar, they also have XML elements with differences in nesting, some missing elements, non-matching data types and slightly different names. For DIKE, we had to remodel the schemas as an appropriate ER model.

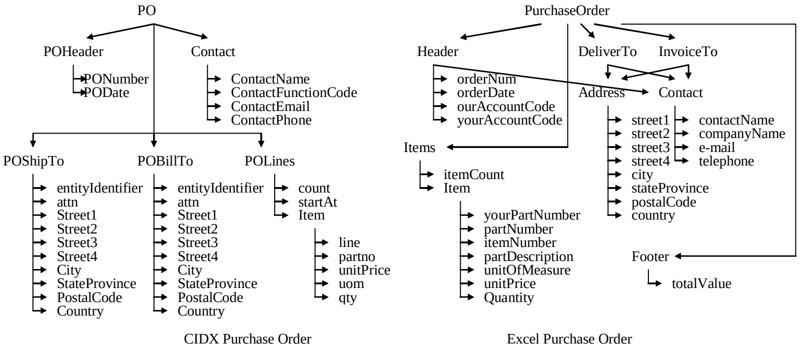

**Figure 7: Purchase order schemas**

The linguistic input to the systems differed as follows. For MOMIS the best possible meanings were chosen for each of the schema elements. For Cupid, the thesauri had a total of 4 abbreviations (*UOM*, *PO*, *Qty*, *Num*) and 2 synonymy entries (*Invoice,Bill; Ship,Deliver*) that were relevant to the example. For DIKE, we added relevant entries needed for matching to the LSPD.

The XML-element level mapping inferred by the three systems is summarized in Table 2. We make the following observations about the mappings:

**1. DIKE:** The abstracted schema depends on the choice of ER model. We first chose to model the root elements, and all XML-elements that had any attributes, as entities.

| CIDX → Excel | Cupid | DIKE | MOMIS – ARTEMIS |
|---|---|---|---|
| POHeader → Header | Yes | Yes | Yes |
| Item → Item | Yes | Yes | The two *Item* elements and the *Items* element are in a single cluster. *POLines* is in its own cluster. |
| POLines → Items | Yes | Yes | |
| POBillTo→InvoiceTo | Yes | No | Clustered together with the Address element |
| POShipTo→DeliverTo | Yes | No | |
| Contact→Contact | Yes | Yes | Yes |
| PO→PurchaseOrder | Yes | Yes | Yes, classes clustered, but corresponding elements not mapped. |

**Table 2: Mapping comparison for CIDX-EXCEL example**

In the abstracted schema that results, entities *POShipTo* and *Address* are merged into a single entity, and so are *PO, POBillTo* and *PurchaseOrder,* and there are three relationships between these two entities (*PO-POShipTo, InvoiceTo* and *DeliverTo*). Hence we believe that some but not all the desired mapping was achieved. The XML-attributes are matched according to the LSPD entries.

To test type-conflict resolution, we then modeled PO-ShipTo, POBillTo and POLines as entities in the CIDX ER model and *DeliverTo*, *InvoiceTo* and *Items* as relationships from *PurchaseOrder* in the Excel ER model. There is one *PO* relationship in the CIDX schema that involves all 5 entities corresponding to the XML-elements that are children of *PO*. In the Excel schema, *PurchaseOrder* is an entity. DIKE correctly identifies mappings *POBillTo→ InvoiceTo* and *POShipTo→DeliverTo,* but not *POLines→ Items*. The entities *POBillTo*, *POShipTo* and *Address* are merged into one entity that has two relationships, *InvoiceTo* and *DeliverTo,* with the *PurchaseOrder* entity.

**2. MOMIS:** Since ARTEMIS clusters the five classes (*POShipTo, POBillTo, InvoiceTo DeliverTo, Address*) together, and the corresponding elements in the *PO* and *PurchaseOrder* cluster are not mapped to each other, we believe that it did not achieve the desired mapping. This might be because, unlike Cupid, MOMIS does not perform context dependent matching. Not all possible attribute level matches are performed: e.g. the *Street*(1…4) attributes in the two schemas are not mapped 1:1 (though their meanings in WordNet are the same, the names themselves are distinct, and hence we would expect them to match correctly). The XML-element *Items* was clustered with the *Item* classes (and not *POLines*). Since attribute matching is done only within global clusters (after the clusters have been decided), the XML-attribute *itemCount* (in *Items*) is matched with *Quantity* (in *Item*).

**3. Cupid:** Cupid identifies all the correct XML-attribute matching pairs (leaves in the example). Cupid is the only one to identify *CIDX.line* to correspond to *Excel.itemNumber* (there were no supporting thesaurus entries). This matching was based purely on the data-type and structural matching. In addition, there are two false positives (e.g. CIDX.*contactName* is mapped to both *Excel.contactName* and *Excel.companyName*). This is due to the naïve mapping-generator; for every XML attribute in the target schema it returns the best matching XML attribute in the source (whether or not the latter was already mapped). The data types and elements in the vicinity of these XML-attributes strongly match and thus these mappings are reported. This demonstrates the need for a more sophisticated scheme to generate mappings from the similarity values. The XML-element mappings in [7] are reported based on their respective structural similarity values.

In [7] we further demonstrate the utility of exploiting referential constraints as join nodes — for a different real-world example, Cupid is able to infer relationships such as the correspondence of a single table in one of the schemas to the join of two tables in the other schema. MOMIS and DIKE are unable to infer similar relationships.

## 8.3 Experimental Conclusions

We draw the following conclusions from our experiments.

**1. Linguistic matching** of schema element names results in useful mappings. Cupid performs simple token manipulation to be tolerant to variations in element names. Unlike Cupid, DIKE and MOMIS expect identical names for matching schema elements in the absence of linguistic input (via LSPD or the user interface to WordNet respectively). MOMIS uses the description logic based ODB tools to infer name affinities within a single schema (by exploiting object hierarchies and referential constraints), and also infers additional name affinities by transitive closure calculations — both are helpful features.

**2.** The **thesaurus** plays a crucial role in linguistic matching. The effect of dropping the thesaurus varies. With Cupid, the resulting mapping is comparatively poor in the CIDX-Excel example, but it is unchanged in other examples [7]. The WordNet interface of MOMIS provides a useful tool for the user to pick from alternative meanings in a thesaurus, but can be a bit restrictive (only one applicable word form). The sense of a word is often domain-specific; e.g. the correct sense of *Header* does not exist in WordNet, and the synonym has to be manually added. The tokenization done by Cupid, followed by stemming, can aid in the automatic selection of possible word meanings during name matching (done by the user in MOMIS) and make it easier to use off-the-shelf thesauri. A robust solution will need a module to incrementally learn synonyms and abbreviations from mappings that are performed over time.

**3.** Using **linguistic similarity with no structure similarity**, Cupid cannot distinguish between the instances of a single XML-attribute in multiple contexts (there are 18 such XML attributes in the CIDX-Excel example). So, to make a fair evaluation of the utility of just the linguistic similarity, we compared elements in the two schemas using just their complete path names (from the root) in their schema trees. While in the CIDX-Excel example only 2 of the correct matching XML attribute pairs went undetected, there were as many as 7 false positive mappings. In a relational schema, where the pathnames include only the table and column names, the accuracy is much worse [7].

**4. Granularity of similarity computation.** MOMIS's ultimate goal is a mediated schema, so mappings are performed at a class level granularity. As we have seen, class-level similarity computation, can sometimes lead to non-optimal mappings. Single classes might be nested or normalized differently (with referential constraints) in different schemas.

**5.** Using the **leaves** in the schema tree **for the structural similarity** computation allows the Cupid approach to match similar schemas that have different nesting. Also, reporting **mappings in terms of leaves** allows a sophisticated query discovery module to generate the correct queries for data transformations.

**6.** Incorporating **structure information beyond the immediate vicinity** of a schema element leads to better matching. Thus, in the CIDX-Excel example, Cupid is able to match *POBillTo*, *POShipTo* and *POLines* to *InvoiceTo*, *DeliverTo* and *Items* respectively. For the same reason, DIKE finds many of the matches. ARTEMIS tries to incorporate such information using the ODB-Tools during the name affinity computation.

**7. Context-dependent mappings** generated by constructing schema trees are useful when inferring different mappings for the same element in different contexts.

**8. Performance parameters**. Some of the mapping results for these tools might not be the best achievable by them, in that improvements may be possible by adjusting few of their parameters. Tuning performance parameters in some cases requires expert knowledge of these tools. Thus auto-tuning is an open problem, and a requirement for a robust solution.

## 9 Summary and Future Work

In this paper, we studied schema matching as an independent problem. We provided a survey and taxonomy of past approaches. We presented a new algorithm that improves on past methods in many respects, for example, by including a substantial linguistic matching step and by biasing matches by leaves of a schema. We implemented the algorithm as an independent component. And we compared our implementation to two others. This demonstrated the strengths of our approach and is a possible model for future algorithm comparisons.

While we believe we have made progress on the schema-matching problem, we do not claim to have solved it. A truly robust solution needs to include other techniques, such as machine learning applied to instances, natural language technology, and pattern matching to reuse known matches. Some of the immediate challenges for further work include: integrating Cupid transparently with an off-the-shelf thesaurus; using schema annotations (textual descriptions of schema elements in the data dictionary) for the linguistic matching; and automatic tuning of the control parameters. Scalability analysis and testing are necessary to study the performance on large-sized schemas. And much more comparative analysis of algorithms is needed. Our long-term goal is to make Cupid be a truly general-purpose schema matching component, that can be used in systems for schema integration, data migration, etc. The work reported here is just one step along what we expect will be a very long research path.

## Acknowledgements

We are very grateful to S. Bergamashchi, S. Castano, A. Corni and F. Guerra for providing us with the binaries for MOMIS and for their technical support; and to L. Palopoli, G. Terracina and D. Ursino for providing us the same for DIKE. The analysis in Section 8 would have been impossible without this generosity.

## References

1. Bergamaschi, S., S. Castano, and M. Vincini: Semantic Integration of Semistructured and Structured Data Sources. SIGMOD Record 28(1), 1999, 54-59.
2. Bernstein, P.A., A. Halevy, and R.A. Pottinger: A Vision for Management of Complex Models. SIGMOD Record 29(4), 2000, 55-63.
3. Castano, S. and V. De Antonellis: A Schema Analysis and Reconciliation Tool Environment. IDEAS'99, 53-62.
4. Clifton, C. and E. Hausman, A. Rosenthal: Experience with a Combined Approach to Attribute-Matching Across Heterogeneous Databases. Proc. 7th IFIP Conf. On DB Semantics, 1997.
5. Doan, A., P. Domingos, and A. Halevy: Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. SIGMOD 2001, 509-520.
6. W. Li, C. Clifton: SEMINT: A tool for identifying attribute correspondences in heterogeneous databases using neural networks. Data & Knowledge Engineering, 33(1), 2000, 49-84.
7. Madhavan, J., P.A. Bernstein, and E. Rahm: Generic Schema Matching using Cupid. MSR Tech. Report MSR-TR-2001-58, 2001, http://www.research.microsoft.com/pubs .
8. Microsoft Corp., BizTalk Mapper: http://www.microsoft.com/technet/biztalk/btsdocs .
9. Miller, R., L. Haas, and M.A. Hernandez: Schema Mapping as Query Discovery. VLDB 2000, 77-88.
10. Milo, T. and S. Zohar: Using Schema Matching to Simplify Heterogeneous Data Translation. VLDB 1998, 122-133.
11. Mitra, P., G. Wiederhold, and J. Jannink: Semi-automatic Integration of Knowledge Sources, FUSION 99.
12. Palopoli, L. G. Terracina, and D. Ursino: The System DIKE: Towards the Semi-Automatic Synthesis of Cooperative Information Systems and Data Warehouses. ADBIS-DASFAA 2000, Matfyzpress, 108-117.
13. Rahm, E. and P.A. Bernstein: On Matching Schemas Automatically. MSR Tech. Report MSR-TR-2001-17, 2001, http://www.research.microsoft.com/pubs.
14. Wald, J.A. and P.G. Sorenson: Explaining Ambiguity in a Formal Query Language. ACM TODS 15(2), 1990, 125-161
15. Wang, Q-Y., J.X. Yu, and K-F. Wong: Approximate Graph Schema Extraction for Semi-Structured Data. EDBT 2000, 302-316.
16. WordNet – a lexical database for English: http://www.cogsci.princeton.edu/~wn/.
17. XML Schema: http://www.w3.org/XML/Schema.