The VLDB Journal 10: 334–350 (2001) / Digital Object Identifier (DOI) 10.1007/s007780100057

# A survey of approaches to automatic schema matching

**Erhard Rahm**[1], **Philip A. Bernstein**[2]

[1] Universität Leipzig, Institut für Informatik, 04109 Leipzig, Germany; (e-mail: rahm@informatik.uni-leipzig.de)
[2] Microsoft Research, Redmond, WA 98052-6399, USA; (e-mail: philbe@microsoft.com)

Edited by P. Scheuermann. Received: 5 February 2001 / Accepted: 6 September 2001
Published online: 21 November 2001 – © Springer-Verlag 2001

**Abstract.** Schema matching is a basic problem in many database application domains, such as data integration, E-business, data warehousing, and semantic query processing. In current implementations, schema matching is typically performed manually, which has significant limitations. On the other hand, previous research papers have proposed many techniques to achieve a partial automation of the match operation for specific application domains. We present a taxonomy that covers many of these existing approaches, and we describe the approaches in some detail. In particular, we distinguish between schema-level and instance-level, element-level and structure-level, and language-based and constraint-based matchers. Based on our classification we review some previous match implementations thereby indicating which part of the solution space they cover. We intend our taxonomy and review of past work to be useful when comparing different approaches to schema matching, when developing a new match algorithm, and when implementing a schema matching component.

**Keywords:** Schema matching – Schema integration – Graph matching – Model management – Machine learning

## 1. Introduction

A fundamental operation in the manipulation of schema information is *Match*, which takes two schemas as input and produces a mapping between elements of the two schemas that correspond semantically to each other [LC94, MIR94, MZ98, PSU98, MWJ99, DDL00]. Match plays a central role in numerous applications, such as web-oriented data integration, electronic commerce, schema integration, schema evolution and migration, application evolution, data warehousing, database design, web site creation and management, and component-based development.

Currently, schema matching is typically performed manually, perhaps supported by a graphical user interface. Obviously, manually specifying schema matches is a tedious, time-consuming, error-prone, and therefore expensive process. This is a growing problem given the rapidly increasing number of web data sources and E-businesses to integrate. Moreover, as systems become able to handle more complex databases and applications, their schemas become larger, further increasing the number of matches to be performed. The level of effort is at least linear in the number of matches to be performed, maybe worse than linear if one needs to evaluate each match in the context of other possible matches of the same elements. A faster and less labor-intensive integration approach is needed. This requires automated support for schema matching.

To provide this automated support, we would like to see a generic, customizable implementation of Match that is usable across application areas. This would make it easier to build application-specific tools that include automatic schema match. Such a generic implementation can also be a key component within a more comprehensive model management approach, such as the one proposed in [BHP00, Be00, BR00], where the mapping returned by a match operation may be used as input to operations to merge schemas and compose mappings.

Fortunately, there is a lot of previous work on schema matching developed in the context of schema translation and integration, knowledge representation, machine learning, and information retrieval. The main goals of this paper are to survey these past approaches and to present a taxonomy that explains their common features. We expect the survey to be helpful both to designers of new approaches and to users who need to select from a library of approaches.

In the next section, we summarize some example applications of schema matching. Section 3 defines the match operator, and Section 4 describes a high-level architecture for implementing it. Section 5 provides a classification of different ways to perform Match automatically. This section illustrates both the complexity of the problem and (at least part of) the solution space. We use the classification in Sects. 6 through 8 to organize our presentation of previously proposed techniques and to explain how they may be applied in the overall architecture. Section 9 is a literature review, which describes some integrated solutions and how they fit in our classification. Section 10 is the conclusion.

## 2. Application domains

To motivate the importance of schema matching, we summarize its use in several database application domains.

### *2.1. Schema integration*

Most work on schema match has been motivated by schema integration, a problem that has been investigated since the early 1980s: Given a set of independently developed schemas, construct a global view [BLN86, EP90, SL90, PS98]. In an artificial intelligence setting, this is the problem of integrating independently developed ontologies into a single ontology.

Since the schemas are independently developed, they often have different structure and terminology. This can obviously occur when the schemas are from different domains, such as a real estate schema and property tax schema. However, it also occurs even if they model the same real world domain, just because they were developed by different people in different real-world contexts. Thus, a first step in integrating the schemas is to identify and characterize these interschema relationships. This is a process of schema matching. Once they are identified, matching elements can be unified under a coherent, integrated schema or view. During this integration, or sometimes as a separate step, programs or queries are created that permit translation of data from the original schemas into the integrated representation.

A variation of the schema integration problem is to integrate an independently developed schema with a given conceptual schema. Again, this requires reconciling the structure and terminology of the two schemas, which involves schema matching.

### *2.2. Data warehouses*

A variation of the schema integration problem that became popular in the 1990s is that of integrating data sources into a data warehouse. A data warehouse is a decision support database that is extracted from a set of data sources. The extraction process requires transforming data from the source format into the warehouse format. As shown in [BR00], the match operation is useful for designing transformations. Given a data source, one approach to creating appropriate transformations is to start by finding those elements of the source that are also present in the warehouse. This is a match operation. After an initial mapping is created, the data warehouse designer needs to examine the detailed semantics of each source element and create transformations that reconcile those semantics with those of the target.

Another approach to integrating a new data source $\mathsf{S}'$ is to reuse an existing source-to-warehouse transformation $\mathsf{S}\Rightarrow\mathsf{W}$. First, the common elements of $\mathsf{S}'$ and $\mathsf{S}$ are found (a match operation) and then $\mathsf{S}\Rightarrow\mathsf{W}$ is reused for those common elements.

### *2.3. E-commerce*

In the current decade, E-commerce has led to a new motivation for schema matching: message translation. Trading partners frequently exchange messages that describe business transactions. Usually, each trading partner uses its own message format. Message formats may differ in their syntax, such as EDI (electronic data interchange) structures, XML, or custom data structures. They may also use different message schemas. To enable systems to exchange messages, application developers need to convert messages between the formats required by different trading partners.

Part of the message translation problem is translating between different message schemas. Message schemas may use different names, somewhat different data types, and different ranges of allowable values. Fields are grouped into structures that also may differ between the two formats. For example, one may be a flat structure that simply lists fields while another may group related fields. Or both formats may use nested structures but may group fields in different combinations.

Translating between different message schemas is, in part, a schema matching problem. Today, application designers need to specify manually how message formats are related. A match operation would reduce the amount of manual work by generating a draft mapping between the two message schemas, which an application designer can subsequently validate and modify as needed.

Schema match may also be helpful to applications being considered for the semantic web [BHL01], such as mapping messages between autonomous agents or matching declarative mediator definitions.

### *2.4. Semantic query processing*

Schema integration, data warehousing, and E-commerce are all similar in that they involve the design-time analysis of schemas to produce mappings and, possibly an integrated schema. A somewhat different scenario is semantic query processing – a run-time scenario where a user specifies the output of a query (e.g., the SELECT clause in SQL), and the system figures out how to produce that output (e.g., by determining the FROM and WHERE clauses in SQL). The user's specification is stated in terms of concepts familiar to her, which may not be the same as the names of elements specified in the database schema. Therefore, in the first phase of processing the query, the system must map the user-specified concepts in the query output to schema elements. This too is a natural application of the match operation.

After mapping the query output to the schema elements, the system must derive a qualification (e.g., a WHERE clause) that gives the semantics of the mapping. Techniques for deriving this qualification have been developed over the past 20 years [MRSS82, KKFG84, WS90, RYAC00]. We expect that these techniques can be generalized to specify the semantics of a mapping produced by the match operation. However, an investigation of this hypothesis is beyond the scope of this paper.

## 3. The match operator

To define the match operator, Match, we need to choose a representation for its input schemas and output mapping. We want to explore many approaches to Match. These approaches depend a lot on the kinds of schema information they use and how they interpret it. However, they depend hardly at all on that information's internal representation, except to the extent that it is expressive enough to represent the information of interest. Therefore, for the purposes of this paper, we define

**Table 1.** Sample input schemas

| S1 elements | S2 elements |
|---|---|
| Cust | Customer |
| C# | CustID |
| CName | Company |
| FirstName | Contact |
| LastName | Phone |

a *schema* to be simply a *set of elements* connected by some *structure*.

In practice, a particular representation must be chosen, such as an entity-relationship (ER) model, an object-oriented (OO) model, XML, or directed graphs. In each case, there is a natural correspondence between the building blocks of the representation and the notions of elements and structure: entities and relationships in ER models; objects and relationships in OO models; elements, subelements, and IDREFs in XML; and nodes and edges in graphs.

We define a mapping to be a set of *mapping elements*, each of which indicates that certain elements of schema S1 are mapped to certain elements in S2. Furthermore, each mapping element can have a *mapping expression* which specifies how the S1 and S2 elements are related. The mapping expression may be directional, for example, a certain function from the S1 elements referenced by the mapping element to the S2 elements referenced by the mapping element, or it may be non-directional, that is, a relation between a combination of elements of S1 and S2. It may use simple relations over scalars (e.g., =, $\leq$), functions (e.g., addition or concatenation), ER-style relationships (e.g., is-a, part-of), set-oriented relationships (e.g., overlaps, contains [LNE89]), or any other terms that are defined in the expression language being used.

For example, Table 1 shows two schemas S1 and S2 representing customer information. A mapping between S1 and S2 could contain a mapping element relating Cust.C# to Customer.CustID with the mapping expression "Cust.C# = Customer.CustID". A mapping element with the expression "Concatenate(Cust.FirstName, Cust.LastName) = Customer.Contact" describes a mapping between two S1 elements and one S2 element.

We define the match operation to be a function that takes two schemas S1 and S2 as input and returns a mapping between those two schemas as output, called the *match result*. Each mapping element of the match result specifies that certain elements of schema S1 logically correspond to, i.e., match, certain elements of S2, where the semantics of this correspondence is expressed by the mapping element's mapping expression.

Unfortunately, the criteria used to match elements of S1 and S2 are based on heuristics that are not easily captured in a precise mathematical way that can guide us in the implementation of Match. Thus, we are left with the practical, though mathematically unsatisfying, goal of producing a mapping that is consistent with heuristics that approximate our understanding of what users consider to be a good match.

Similar to previous work we focus mostly on match algorithms that return a mapping that does not include mapping expressions. We therefore often represent a mapping as a similarity relation, $\cong$, over the powersets of S1 and S2, where each pair in $\cong$ represents one mapping element of the mapping. For example, the result of calling Match on the schemas of Table 1 could be "Cust.C# $\cong$ Customer.CustID", "Cust.CName $\cong$ Customer.Company", and "{Cust.FirstName, Cust.LastName} $\cong$ Customer.Contact". A complete specification of the result of the invocation of Match would also include the mapping expression of each element, that is "Cust.C# = Customer.CustID", "Cust.CName = Customer. Company", and "Concatenate (Cust.FirstName, Cust.LastName) = Customer.Contact". In what follows, when mapping expressions are involved, we will explicitly mention them. Otherwise, we will simply use $\cong$.

As we will see, some implementations of Match are similar to join processing in relational databases, in that both Match and Join are binary operations that determine pairs of corresponding elements from their input operands. There are many differences, of course. Match operates on metadata (schema elements) and Join on data (rows of tables). Moreover, Match is more complex than Join. Each element in the Join result combines only one element of the first with one matching element of the second input, while an element in a match result can relate multiple elements from both inputs. Furthermore, Join semantics is specified by a single comparison expression (e.g., an equality condition for natural join) that must hold for all matching input elements. By contrast, each element in a match result may have a different mapping expression. Hence, the semantics of Match is less restricted than that of Join and is more difficult to capture in a consistent way.

The similarity of Match and Join extends to OuterMatch operations, which are useful counterparts to Match in much the same way that OuterJoin is a counterpart to Join. A right (or left) OuterMatch ensures that every element of S2 (or S1) is referenced by the mapping. A full OuterMatch ensures every element of both S1 and S2 are referenced by the mapping. By ensuring that every element of a schema S is referenced in the mapping returned by Match, the mapping can be more easily composed with other mappings that refer to S. Examples of such compositions appear in [BR00], which introduced the OuterMatch operation. Although the usage of OuterMatch involves some subtlety, its implementation is a straightforward extension of Match: given an algorithm for the match operation, OuterMatch can easily be computed by adding elements to the match result that reference the otherwise non-referenced elements of S1 or S2. We therefore do not consider OuterMatch further in this paper.

## 4. Architecture for generic match

When reviewing and comparing approaches to Match, it helps to have an implementation architecture in mind. We therefore describe a high-level architecture for a generic, customizable implementation of Match.

Figure 1 illustrates the overall architecture. The clients are schema-related applications and tools from different domains, such as E-business, portals, and data warehousing. Each client uses the generic implementation of Match to automatically determine matches between two input schemas. XML schema editors, portal development kits, database modeling tools and the like may access libraries to select existing schemas, shown in the lower left of Fig. 1. The implementation of Match may

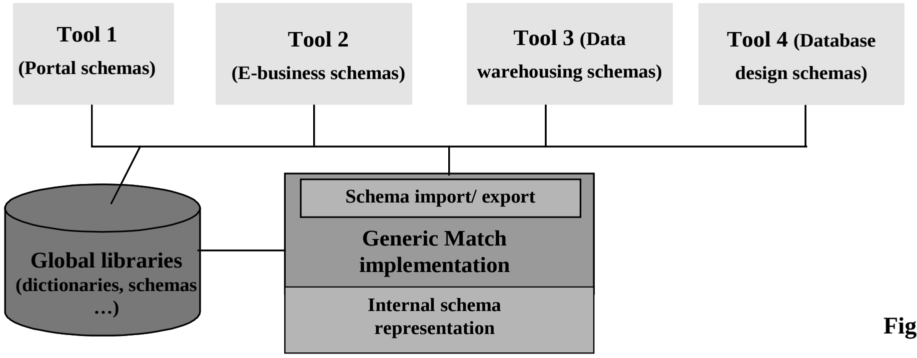
Tool 1
(Portal schemas)
Tool 2
(E-business schemas)
Tool 3 (Data warehousing schemas)
Tool 4 (Database design schemas)
Global libraries
(dictionaries, schemas …)
Schema import/ export
Generic Match implementation
Internal schema representation

**Fig. 1.** High-level architecture of generic Match

**Table 2.** Full vs partial structural match (example)

| S1 elements | S2 elements | |
|---|---|---|
| Address<br>Street<br>City<br>State<br>ZIP | CustomerAddress<br>Street<br>City<br>USState<br>PostalCode | full structural match of Address and CustomerAddress |
| AccountOwner<br>Name<br>Address<br>Birthdate<br>TaxExempt | Customer<br>Cname<br>CAddress<br>CPhone | partial structural match of AccountOwner and Customer |

also use the libraries and other auxiliary information, such as dictionaries and thesauri, to help find matches.

We assume that the generic implementation of Match represents the schemas to be matched in a uniform internal representation. This uniform representation significantly reduces the complexity of Match by not having to deal with the large number of different (heterogeneous) representations of schemas. Tools that are tightly integrated with the framework can work directly on the internal representation. Other tools need import/export programs to translate between their native schema representation (such as XML, SQL, or UML) and the internal representation. A semantics-preserving importer translates input schemas from their native representation into the internal representation. Similarly, an exporter translates mappings produced by the generic implementation of Match from the internal representation into the representation required by each tool. This allows the generic implementation of Match to operate exclusively on the internal representation.

In general, it is not possible to determine fully automatically all matches between two schemas, primarily because most schemas have some semantics that affects the matching criteria but is not formally expressed or often even documented. The implementation of Match should therefore only determine *match candidates*, which the user can accept, reject or change. Furthermore, the user should be able to specify matches for elements for which the system was unable to find satisfactory match candidates.

## 5. Classification of schema matching approaches

In this section we classify the major approaches to schema matching. Fig. 2 shows part of our classification scheme together with some sample approaches.

An implementation of Match may use multiple match algorithms or *matchers*. This allows us to select the matchers depending on the application domain and schema types. Given that we want to use multiple matchers we distinguish two subproblems. First, there is the realization of individual matchers, each of which computes a mapping based on a single matching criterion. Second, there is the combination of individual matchers, either by using multiple matching criteria (e.g., name and type equality) within an integrated *hybrid matcher* or by combining multiple match results produced by different match algorithms within a *composite matcher*. For individual matchers, we consider the following largely-orthogonal classification criteria:

- *Instance vs schema:* matching approaches can consider instance data (i.e., data contents) or only schema-level information.
- *Element vs structure matching:* match can be performed for individual schema elements, such as attributes, or for combinations of elements, such as complex schema structures.
- *Language vs constraint:* a matcher can use a linguistic-based approach (e.g., based on names and textual descriptions of schema elements) or a constraint-based approach (e.g., based on keys and relationships).
- *Matching cardinality:* the overall match result may relate one or more elements of one schema to one or more elements of the other, yielding four cases: 1:1, 1:n, n:1, n:m. In addition, each mapping element may interrelate one or more elements of the two schemas. Furthermore, there may be different match cardinalities at the instance level.
- *Auxiliary information:* most matchers rely not only on the input schemas S1 and S2 but also on auxiliary information, such as dictionaries, global schemas, previous matching decisions, and user input.

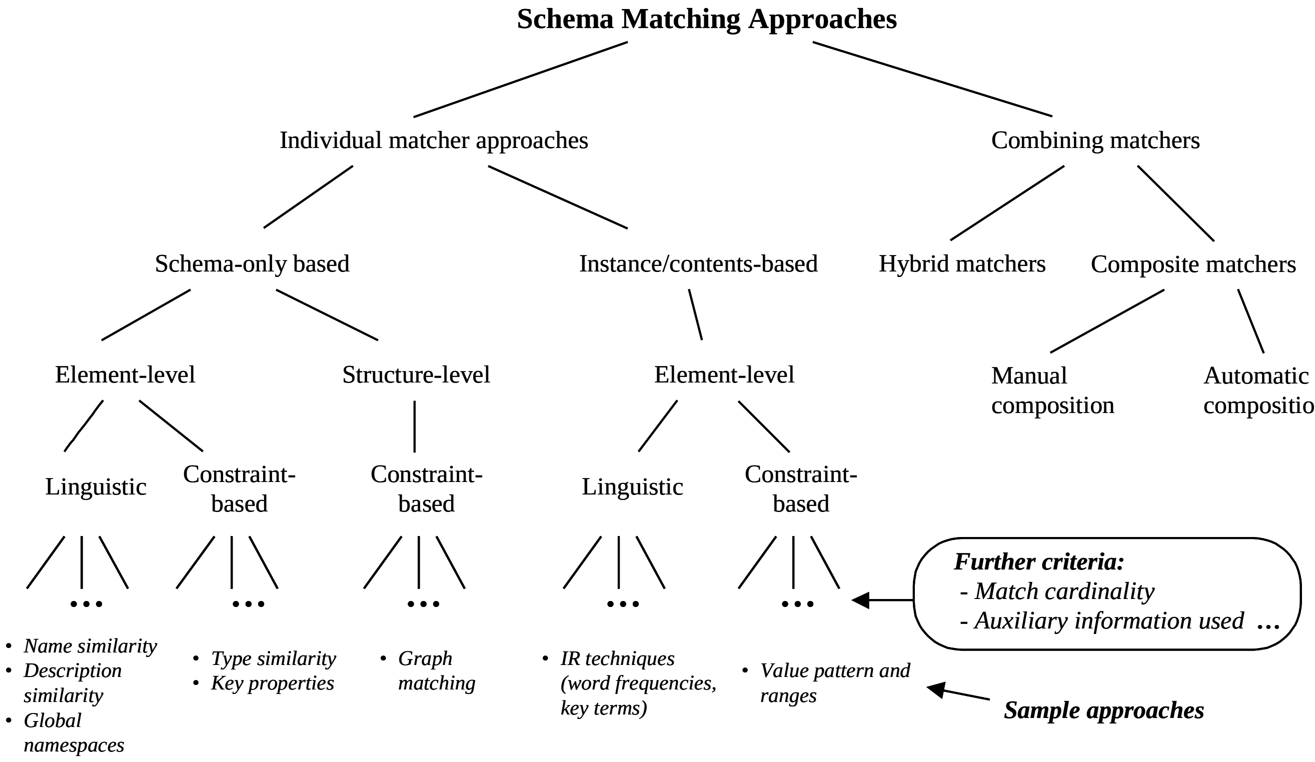

**Fig. 2.** Classification of schema matching approaches

Note that our classification does not distinguish between different types of schemas (relational, XML, object-oriented, etc.) and their internal representation, because algorithms depend mostly on the kind of information they exploit, not on its representation.

In the following three sections, we discuss the main alternatives according to the above classification criteria. We discuss schema-level matching in Sect. 6, instance-level matching in Sect. 7, and combinations of multiple matchers in Sect. 8.

## 6. Schema-level matchers

Schema-level matchers only consider schema information, not instance data. The available information includes the usual properties of schema elements, such as name, description, data type, relationship types (part-of, is-a, etc.), constraints, and schema structure. In general, a matcher will find multiple match candidates. For each candidate, it is customary to estimate the degree of similarity by a normalized numeric value in the range 0–1, in order to identify the best match candidates (as in [PSU98, BCV99, DDL00, CDD01]).

We first discuss the main alternatives for match granularity and match cardinality. Then we cover linguistic and constraint-based matchers. Finally, we outline approaches based on the reuse of auxiliary data, such as previously defined schemas and previous match results.

### *6.1. Granularity of match (element-level vs structure-level)*

We distinguish two main alternatives for the granularity of Match, element-level and structure-level matching. For each element of the first schema, *element-level matching* determines the matching elements in the second input schema. In the simplest case, only elements at the finest level of granularity are considered, which we call the *atomic level*, such as attributes in an XML schema or columns in a relational schema. For the schema fragments shown in Table 2, a sample atomic-level match is "Address.ZIP ≅ CustomerAddress.PostalCode" (recall that "≅" means "matches").

*Structure-level matching*, on the other hand, refers to matching combinations of elements that appear together in a structure. A range of cases is possible, depending on how complete and precise a match of the structure is required. In the ideal case, all components of the structures in the two schemas fully match. Alternatively, only some of the components may be required to match (i.e., a partial structural match). Examples of the two cases are shown in Table 2. The need for partial matches sometimes arises because subschemas of different domains are being compared. For example, in the second row of Table 2, AccountOwner may come from a finance database while Customer comes from a sales database.

For more complex cases, the effectiveness of structure matching can be enhanced by considering known equivalence patterns, which may be kept in a library. One simple pattern is shown in Fig. 3 relating two structures in an is-a hierarchy to a single structure. The subclass of the first schema is represented by a Boolean attribute in the second schema. Another well-known pattern consists of two structures interconnected by a referential relationship being equivalent to a single structure (essentially, the join of the two). We will see an example of this in Sect. 6.4.

Element-level matching is not restricted to the atomic level, but may also be applied to coarser grained, *higher (non-atomic)*

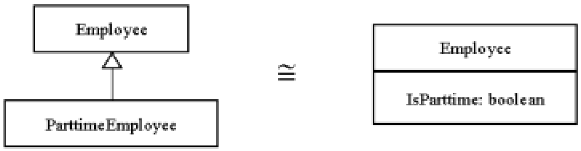

**Fig. 3.** Equivalence pattern

*level* elements. Sample higher-level granularities include file records, entities, classes, relational tables, and XML elements. In contrast to a structure-level matcher, such an element-level approach considers the higher-level element in isolation, ignoring its substructure and components. For instance, the fact that the elements "Address" and "CustomerAddress" in Table 2 are likely to match can be derived by a name-based element-level matching without considering their underlying components.

Element-level matching can be implemented by algorithms similar to relational join processing. Depending on the matcher type, the match comparison can be based on such properties as name, description, or data type of schema element. For each element of a schema S1, all elements of the other schema S2 with the same or similar value for the match property have to be identified. A general implementation, similar to nested-loop join processing, compares each S1 element with each S2 element and determines a similarity metric per pair. Only the combinations with a similarity value above a certain threshold are considered as match candidates. For special cases, more efficient implementations are possible. For example, as for equi-joins, checking for equality of properties can be done using hashing or sort-merge. The join-like implementation is also feasible for hybrid matchers where we consider multiple properties at a time (e.g., name + data type).

### 6.2. Match cardinality

An S1 (or S2) element can participate in zero, one or many mapping elements of the match result between the two input schemas S1 and S2. Moreover, within an individual mapping element, one or more S1 elements can match one or more S2 elements. Thus, we have the usual relationship cardinalities, namely 1:1 and the set-oriented cases 1:n, n:1, and n:m, between matching elements both with respect to different mapping elements (*global cardinality*) and with respect to an individual mapping element (*local cardinality*). Element-level matching is typically restricted to local cardinalities of 1:1, n:1, and 1:n. Obtaining n:m mapping elements usually requires considering the structural embedding of the schema elements and thus requires structure-level matching.

Table 3 shows examples of the four local cardinality cases for individual mapping elements. In row 1, the match is 1:1. Previous work has mostly concentrated on such 1:1 matches because of the difficulty of automatically determining the mapping expressions in the other cases. When matching multiple S1 (or S2) elements at a time, we see that expressions are used to specify how these elements are related. For example, row 3 explains how FirstName and LastName are extracted from Name. Another example is row 4, which uses a SQL expression combining attributes from two tables. It corresponds to an n:m relationship at the attribute level (four S1 attributes match two S2 attributes) and an n:1 relationship at the structure level (two tables, B and P, in S1 match one table, A, in S2). The structure-level match ensures that the two A elements are derived together in order to obtain correct book-publisher combinations.

The global cardinality cases with respect to all mapping elements are largely orthogonal to the cases for individual mapping elements. For instance in the example of row 1, we have a global 1:1 match if no other S1 elements match Amount and no other S2 elements match Price. On the other hand, if Price in S1 also matches other S2 elements (e.g., Cost as in row 2) we obtain a global 1:n match in combination with local 1:1 or 1:n matches.

Note that in addition to the match cardinalities at the schema level, there may be different match cardinalities at the instance level. For the first three examples in Table 3, one S1 instance is matched with one S2 instance (1:1 instance-level match). The example in row 4 corresponds to an n:1 instance-level match, which combines two instances, one of B and one of P, into one of A. An example of n:m instance-level matching is the association of individual sale instances of S1 with different aggregate sale instances (per month, quarter, etc.) of S2.

Most existing approaches map each element of one schema to the element of the other schema with highest similarity. This results in local 1:1 matches and global 1:1 or 1:n mappings. More work is needed to explore more sophisticated criteria for generating local and global n:1 and n:m mappings, which are currently hardly treated at all.

### 6.3. Linguistic approaches

Language-based or linguistic matchers use names and text (i.e., words or sentences) to find semantically similar schema

**Table 3.** Match cardinalities (Examples)

| | Local match cardinalities | S1 element(s) | S2 element(s) | Matching expression |
|---|---|---|---|---|
| 1. | 1:1, element level | Price | Amount | Amount = Price |
| 2. | n:1, element-level | Price, Tax | Cost | Cost = Price*(1+Tax/100) |
| 3. | 1:n, element-level | Name | FirstName,<br>LastName | FirstName, LastName =<br>Extract (Name, . . . ) |
| 4. | n:1 structure-level<br>(n:m element-level) | B.Title,<br>B.PuNo,<br>P.PuNo,<br>P.Name | A.Book,<br>A.Publisher | A.Book, A.Publisher =<br>Select B.Title, P.Name<br>From B, P<br>Where B.PuNo=P.PuNo |

elements. We discuss two schema-level approaches, name matching and description matching.

### Name matching

Name-based matching matches schema elements with equal or similar names. Similarity of names can be defined and measured in various ways, including:

- Equality of names.
  An important subcase is the equality of names from the same XML namespace, since this ensures that the same names indeed bear the same semantics.
- Equality of canonical name representations after stemming and other preprocessing.
  This is important to deal with special prefix/suffix symbols (e.g., CName → customer name, and EmpNO → employee number)
- Equality of synonyms.
  (E.g., car ≅ automobile and make ≅ brand)
- Equality of hypernyms.[1]
  (E.g., book *is-a* publication and article *is-a* publication imply book ≅ publication, article ≅ publication, and book ≅ article)
- Similarity of names based on common substrings, edit distance, pronunciation, soundex (an encoding of names based on how they sound rather than how they are spelled), etc. [BS01].
  (E.g., representedBy ≅ representative, ShipTo ≅ Ship2)
- User-provided name matches.
  (E.g., reportsTo ≅ manager, issue ≅ bug)

Exploiting synonyms and hypernyms requires the use of thesauri or dictionaries. General natural language dictionaries may be useful, perhaps even multi-language dictionaries (e.g., English-German) to deal with input schemas of different languages. In addition, name matching can use domain- or enterprise-specific dictionaries and is-a taxonomies containing common names, synonyms and descriptions of schema elements, abbreviations, etc. These specific dictionaries require a substantial effort to be built up in a consistent way. The effort is well worth the investment, especially for schemas with relatively flat structure where dictionaries provide the most valuable matching hints. Furthermore, tools are needed to enable names to be accessed and (re-)used, such as within a schema editor when defining new schemas.

Homonyms are equal or similar names that refer to different elements. Clearly, homonyms can mislead a matching algorithm. Homonyms may be part of natural language, such as “stud” meaning a fastener or male horse, or may be specific to a domain, such as “line” meaning a line of business or a line item (i.e., row) of an order. A name matcher can reduce the number of wrong match candidates by exploiting mismatch information supplied by users or dictionaries. At least, the matcher can offer a warning of the potential ambiguity due to multiple meanings of the name. A more automated use of mismatch information may be possible by using context information, for example, to distinguish Order.Line from Business.Line. Such a technique blurs the distinction between linguistic-based and structure-based techniques.

Name-based matching is possible for elements at different levels of granularity. Furthermore, it can be applied across levels, e.g., for a lower-level schema element to also consider the names of the schema elements it belongs to (e.g., to find that author.name ≅ AuthorName). This is similar to context-based disambiguation of homonyms.

Name-based matching is not limited to finding 1:1 matches. That is, it can identify multiple relevant matches for a given schema element. For example, it can match “phone” with both “home phone” and “office phone”.

Name matching can be driven by element-level matching, introduced in Sect. 6.1. In the case of synonyms and hypernyms, the join-like processing involves a dictionary D as a further input. If we think of a relation-like representation with

```
S1 (name, ...)          // one row per S1 schema element
S2 (name, ...)          // one row per S2 schema element
D (name1, name2,
   similarity)          // similarity score for
                           [name1, name2] between 0..1
```

then a list of all match candidates can be generated by the following three-way join operation

```
Select S1.name, S2.name, D.similarity
From S1, S2, D
Where (S1.name = D.name1) and
      (D.name2 = S2.name) and
      (D.similarity > threshold)
```

This assumes that D contains all relevant pairs of the transitive closure over similar names. For instance, if A-B-0.9 and B-C-0.8 are in D, then we would expect D also to contain B-A-0.9, C-B-0.8, and possibly A-C-$\sigma$, C-A-$\sigma$. Intuitively, we would expect the similarity value $\sigma$ to be .9 × .8 = .72, but this depends on the type of similarity, the use of homonyms, and perhaps other factors. For example, we might have deliver-ship-.9 and ship-boat-.9, but not deliver-boat-$\sigma$ for any similarity value $\sigma$. One approach to assigning different weights to different types of similarity relationships is discussed in [BHP94].

### Description matching

Often, schemas contain comments in natural language to express the intended semantics of schema elements. These comments can also be evaluated linguistically to determine the similarity between schema elements. For instance, this would help find that the following elements match, by a linguistic analysis of the comments associated with each schema element:

```
S1: empn // employee name
S2: name // name of employee
```

This linguistic analysis could be as simple as extracting keywords from the description which are used for synonym comparison, much like names. Or it could be as sophisticated as using natural language understanding technology to look for semantically equivalent expressions.

[1] X is a hypernym of Y if Y is a kind of X. For instance, hypernyms of “oak” include “tree” and “plant”.

**Table 4.** Constraint-based matching (example)

| S1 elements | S2 elements |
|---|---|
| Employee | Personnel |
| EmpNo – int, primary key | Pno - int, unique |
| EmpName – varchar (50) | Pname – string |
| DeptNo – int, references Department | Dept - string |
| Salary - dec (15,2) | Born - date |
| Birthdate – date | |
| Department | |
| DeptNo – int, primary key | |
| DeptName – varchar (40) | |

### *6.4. Constraint-based approaches*

Schemas often contain constraints to define data types and value ranges, uniqueness, optionality, relationship types and cardinalities, etc. If both input schemas contain such information, it can be used by a matcher to determine the similarity of schema elements [LNE89]. For example, similarity can be based on the equivalence of data types and domains, of key characteristics (e.g., unique, primary, foreign), of relationship cardinality (e.g., 1:1 relationships), or of is-a relationships.

The implementation can often be performed as described in Sect. 6.1 with a join-like element-level matching, now using the data types, structures, and constraints in the comparisons. Equivalent data types and constraint names (e.g., string ≅ varchar, primary key ≅ unique) can be provided by a special synonym table.

In the example in Table 4, the type and key information suggest that Born matches Birthdate and Pno matches either EmpNo or DeptNo. The remaining S2 elements Pname and Dept are strings and thus likely match EmpName or DeptName.

As the example illustrates, the use of constraint information alone often leads to imperfect n:m matches (match clusters), as there may be several elements in a schema with comparable constraints. Still, the approach helps to limit the number of match candidates and may be combined with other matchers (e.g., name matchers).

Certain structural information can be interpreted as constraints, such as intra-schema references (e.g., foreign keys) and adjacency-related information (e.g., part-of relationships). Such information tells us which elements belong to the same higher-level schema element, transitively through multi-level structures. Such constraints can be interpreted as structures and therefore be exploited using structure matching approaches. Such a matching can consider the topology of structures as well as different element types (e.g., for attributes, tables / elements, or domains) and possibly different types of structural connections (e.g., part-of or usage relationships).

Many schema structures are hierarchical, based on some form of containment relationship. When performing a match based on hierarchical structures, an algorithm can traverse the structure either top-down or bottom-up. A *top-down algorithm* is usually less expensive than bottom-up, because matches at a high level of the schema structure restrict the choices for matching finer grained structure only to those combinations with matching ancestors. However, a top-down algorithm can be misled if top-level schema structures are very different, even if finer grained elements match well. By contrast, a *bottom-up algorithm* compares all combinations of fine grained elements, and therefore finds matches at this level even if intermediate and higher level structures differ considerably.

Referring back to Table 4, the previously identified atomic-level matches are not sufficient to correctly match S1 to S2 because we actually need to join S1.Employee and S1.Department to obtain S2.Personnel. This can be detected automatically by observing that components of S2.Personnel match components of both S1.Employee and S1.Department and that S1.Employee and S1.Department are interconnected by foreign key DeptNo in Employee referencing Department. This allows us to determine the correct n:m SQL-like match mapping

```
S2.Personnel (Pno, Pname, Dept, born) ≅
Select S1.Employee.EmpNo,
       S1.Employee.EmpName,
       S1.Department.DeptName,
       S1.Employee.Birthdate
From S1.Employee, S1.Department
Where (S1.Employee.DeptNo
     = S1.Department.DeptNo)
```

Some inferencing was needed to know that the join should be added. This inferencing can be done by mapping the problem into one of determining the required joins in the universal relation model [KKFG84].

### *6.5. Reusing schema and mapping information*

We have already discussed the use of auxiliary information in addition to the input schemas, such as dictionaries, thesauri, and user-provided match or mismatch information. Another way to use auxiliary information to improve the effectiveness of Match is to support and exploit the reuse of common schema components and previously determined mappings. Reuse-oriented approaches are promising, since we expect that many schemas need to be matched and that schemas often are very similar to each other and to previously matched schemas. For example, in E-commerce, substructures often repeat within different message formats (e.g., address fields and name fields).

The use of names from XML namespaces or specific dictionaries is already reuse-oriented. A more general approach is to reuse not only globally defined names but also entire schema fragments, including such features as data types, keys, and constraints. This is especially rewarding for frequently used entities, such as address, customer, employee, purchase order, and invoice, which should be defined and maintained in a schema library. While it is unlikely that the whole world agrees on such schemas, they can be specified for an enterprise, its trading partners, relevant standards bodies, or similar organizations to reduce the degree of variability. Schema editors should access these libraries to encourage the reuse of predefined schema fragments and defined terms, perhaps with a wizard that observes when a new schema definition is similar but not identical to one in a library. The elements reused in this way should contain the ID of their originating library, e.g., via

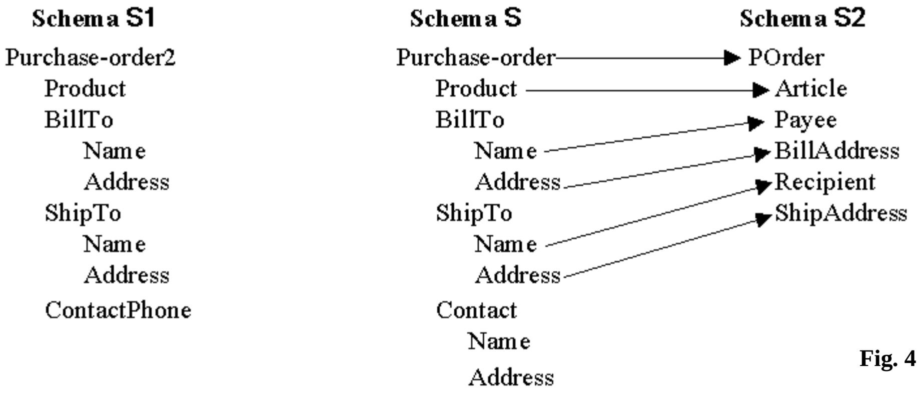

**Fig. 4.** Scenario for reuse of an existing mapping

XML namespaces, so the implementation of Match can easily identify and match schema fragments and names that come from the same library.

A further generic approach is to reuse existing mappings. We want to reuse previously determined element-level matches, which may simply be added to the thesaurus. We also want to reuse entire structures, which is useful when matching different but similar schemas to the same destination schema, as may occur when integrating new sources into a data warehouse or digital library. For instance, this is useful if a schema S1 has to be mapped to a schema S2 to which another schema S has already been mapped. If S1 is more similar to S than to S2, this can simplify the automatic generation of match candidates by reusing matches from the existing result of Match(S, S2), although some care is needed since matches are sometimes not transitive. Among other things, this allows the reuse of manually specified matches.

An example for such a re-use is shown in Fig. 4 for purchase order schemas. We already have the match result between S and S2, illustrated by the arrows. The new purchase order schema S1 is very similar to S. Thus, for every element or structure of S1 that has a corresponding element or fully matching structure in S, we can use the existing mapping between S and S2. In this (ideal) case, we can reuse all matches; since S2 is fully covered, no additional match work has to be done.

Such a reuse of previous matches may only be possible for some part of a new schema. Hence a major problem is to determine which part of a new schema is similar to some part of a previously matched one – a match problem in itself. Moreover, similarity values determined for a previous match task may depend on the application domain so that their reuse should be restricted to related applications. For example, Salary and Income may be considered identical in a payroll application but not in a tax reporting application. To our knowledge these reuse issues have not yet been addressed but deserve further work.

## 7. Instance-level approaches

Instance-level data can give important insight into the contents and meaning of schema elements. This is especially true when useful schema information is limited, as is often the case for semistructured data. In the extreme case, no schema is given, but a schema can be constructed from instance data either manually or automatically (e.g., a "data guide" [GW97] or an approximate schema graph [WYW00] may be constructed automatically from XML documents). Even when substantial schema information is available, the use of instance-level matching can be valuable to uncover incorrect interpretations of schema information. For example, it can help disambiguate between equally plausible schema-level matches by choosing to match the elements whose instances are more similar.

Most of the approaches discussed previously for schema-level matching can be applied to instance-level matching. However, some are especially applicable here. For example:

- For text elements a *linguistic characterization* based on information retrieval techniques is the preferred approach, e.g., by extracting keywords and themes based on the relative frequencies of words and combinations of words, etc. For example, in Table 4, looking at the `Dept`, `DeptName` and`EmpName` instances we may conclude that `DeptName` is a better match candidate for `Dept` than `EmpName`.
- For more structured data, such as numerical and string elements, we can apply a *constraint-based characterization*, such as numerical value ranges and averages or character patterns. For instance, this may allow recognizing phone numbers, zip codes, geographical names, addresses, ISBNs, SSNs, date entries, or money-related entries (e.g., based on currency symbols). In Table 4, instance information may help to make `EmpNo` the primary match candidate for `Pno`, e.g., based on similar value ranges as opposed to the value range for `DeptNo`.

The main benefit of evaluating instances is a precise characterization of the actual contents of schema elements. This characterization can be employed in at least two ways. One approach is to use the characterization to enhance schema-level matchers. For instance, a constraint-based matcher can then more accurately determine corresponding data types based, for example, on the discovered value ranges and character pattern, thereby improving the effectiveness of Match. This requires characterizing the content of both input schemas and then matching the schemas with each other.

A second approach is to perform instance-level matching on its own. First, the instances of S1 are evaluated to characterize the content of S1 elements. Then, the S2 instances are matched one-by-one against the characterizations of S1 elements. The per-instance match results need to be merged and abstracted to the schema level, to generate a ranked list of match candidates in S1 for each (schema-level) element in

S2. Various approaches have been proposed to perform such an instance matching or classification, such as rules, neural networks, and machine learning techniques [BM01, DDL00, DDH01, LC94, LC00, LCL00].

Instance-level matching can also be performed by utilizing auxiliary information, e.g., previous mappings obtained from matching different schemas. This approach is especially helpful for matching text elements by providing match candidates for individual keywords. For instance, a previous analysis may have revealed that the keyword "Microsoft" frequently occurs for schema elements "CompanyName", "Manufacturer", etc. For a new match task, if an S2 schema element X frequently contains the term "Microsoft" this can be used to generate "CompanyName" in S1 as a match candidate for X, even if "Microsoft" does not often occur in the instances of S1.

The above approaches for instance-level matching primarily work for finding element-level matches. Finding matches for sets of schema elements or structures would require characterizing the content of these sets. Obviously, the main problem is the explosion of the number of possible combinations of schema elements for which the instances would have to be evaluated.

## 8. Combining different matchers

We have reviewed several types of matchers and many different variations. Each utilizes different information and has thus different applicability and value for a given match task. Therefore, a matcher that uses just one approach is unlikely to achieve as many good match candidates as one that combines several approaches. This can be done in two ways: a hybrid matcher that integrates multiple matching criteria and composite matchers that combine the results of independently executed matchers. Combining multiple matching approaches also opens the possibility to evaluate them simultaneously or in a specific order.

*Hybrid matchers* directly combine several matching approaches to determine match candidates based on multiple criteria or information sources (e.g., by using name matching with namespaces and thesauri combined with data type compatibility). They should provide better match candidates plus better performance than the separate execution of multiple matchers. Effectiveness may be improved because poor match candidates matching only one of several criteria can be filtered out early, and because complex matches requiring the joint consideration of multiple criteria can be solved (e.g., the use of keys, data types and names in Table 4). Structure-level matching also benefits from being combined with other approaches such as name matching. One way to combine structure- with element-level matching is to use one algorithm to generate a partial mapping and the other to complete the mapping.

A hybrid matcher can offer better performance than the execution of multiple matchers by reducing the number of passes over the schema. For instance, with element-level matching hybrid matchers can test multiple criteria at a time on each S2 element before continuing with the next S2 element.

On the other hand, one can use a *composite matcher* that combines the results of several independently executed matchers, including hybrid matchers. This ability to combine matchers is more flexible than hybrid matchers. A hybrid matcher typically uses a hard-wired combination of particular matching techniques that are executed simultaneously or in a fixed order. By contrast, a composite matcher allows us to select from a repertoire of modular matchers based, for example, on application domain or schema languages (e.g., different approaches can be used for structured vs semi-structured schemas). For example, one could use machine learning to combine independent matchers, as in [DDH01] for instance-level matchers and in [EJX01] for a combination of instance-level and schema-level matchers. Moreover, a composite matcher should allow a flexible ordering of matchers so that they are either executed simultaneously or sequentially. In the latter case, the match result of a first matcher is consumed and extended by a second matcher to achieve an iterative improvement of the match result.

Selection of matchers, and determining their execution order and the combination of independently determined match results can be done either automatically by the implementation of Match itself or its clients (e.g., tools), or manually by a human user. An automatic approach can reduce the number of user interactions, but it is difficult to achieve a generic solution that is adaptable to different application domains (although the approach could be controlled by tuning parameters). Alternatively, a user can directly select the matchers to be executed, their execution order and how to combine their results. Such a manual approach is easier to implement and leaves more control to the user. As discussed in Sect. 4, user interaction is necessary in any case because the implementation of Match can only determine match candidates which a user can accept, reject or change.

To deal with complex match tasks, the implementation of Match should support an iterative development of match results with multiple user interactions. With a composite match approach supporting the sequential execution of matchers, user-supplied matches can be considered as a special matcher that provides input for automatic matchers. Still, the matchers should be aware of user-provided match input and not change it but focus on the unmatched parts of the input schemas.

## 9. Sample approaches from the literature

### 9.1. Prototype schema matchers

In Table 5 we show how seven published prototype implementations fit the classification criteria introduced in Sect. 5. The table thus indicates which part of the solution space is covered by which implementations, thereby supporting a comparison of the approaches. It also specifies the supported schema types, the internal metadata representation format, the tasks to be performed manually, and the application domain. We thus indicate the suitability of the approaches with respect to key requirements, in particular degree of automation (dependence on manual input) and genericity with respect to the different application domains and schema languages. The achievable matching accuracy is related to the degree to which the solution spectrum is covered.

The table shows that all systems support multiple matching criteria, six in the form of a hybrid matcher and only one, LSD, by a composite match approach. A flexible ordering of

**Table 5.** Characteristics of proposed schema match approaches

| | | **SemInt [LC94, LC00, LCL00]** | **LSD [DDL00, DDH01]** | **SKAT [MWJ99, MWK00]** | **TranScm [MZ98]** | **DIKE [PSU98a,b, PSTU99]** | **ARTEMIS [CDD01, BCC*00]** | **CUPID [MBR01]** |
|---|---|---|---|---|---|---|---|---|
| Schema types | | relational, files | XML | XML, IDL, text | SGML, OO | ER | relational, OO, ER | XML, relational |
| Metadata representation | | unspecified (attribute-based) | XML schema trees | graph-based OO data model | labeled graph | graph | hybrid relational / OO data model | extended ER |
| Match granularity | | element-level: attributes (attribute clusters) | element and structure-level | element/ structure-level: attributes / classes | element-level | element/ structure-level: entities / relationships / attributes | element/ structure-level: entities / relationships / attributes | element and structure-level |
| Match cardinality | | 1:1 | 1:1 | 1:1 and n:1 | 1:1 | 1:1 | 1:1 | 1:1 and n:1 |
| Schema-level match | Name-based | - | name equality / synonyms | name equality; synonyms; homonyms; hypernyms | name equality; synonyms; homonyms; hypernyms | name equality; synonyms; hypernyms | name equality; synonyms; hypernyms | name equality, synonyms, hypernyms, homonyms, abbreviations |
| | Constraint-based | several criteria: data type, length, key info, . . . | - | is-a (inclusion); relationship cardinalities | is-a (inclusion); relationship cardinalities | domain compatibility | domain compatibility. In MOMIS, uses keys, foreign keys, is-a, aggregation | data type and domain compatibility, referential constraints |
| | Structure matching | - | XML classifier for matching non-leaf elements [DDH01] | similarity w.r.t. "related" elements | similarity w.r.t. "related" elements | matching of neighborhood | matching of neighborhood | matching subtrees, weighted by leaves |
| Instance-level matchers | Text-oriented | - | Whirl [Co98], Bayesian learners | - | - | - | - | - |
| | Constraint-oriented | character / numerical data pattern, value distribution, averages | list of valid domain values | - | - | - | - | - |
| Reuse / auxiliary information used | | - | comparison with training matches; lookup for valid domain values | reuse of general matching rules | - | provision of some synonyms + inclusions with similarity probabilities | thesauri | thesauri, glossaries |
| Combination of matchers | | hybrid | composite matcher with automatic combination of matcher results | hybrid | hybrid matchers; fixed order of matchers | hybrid | hybrid | hybrid |

**Table 5.** (continued)

| | **SemInt [LC94, LC00, LCL00]** | **LSD [DDL00, DDH01]** | **SKAT [MWJ99, MWK00]** | **TranScm [MZ98]** | **DIKE [PSU98a,b, PSTU99]** | **ARTEMIS [CDD01, BCC*00]** | **CUPID [MBR01]** |
|---|---|---|---|---|---|---|---|
| Manual work / user input | selection of match criteria (optional); selection of matching attributes from attribute clusters | user-supplied matches for training sources; user can specify tuning parameters and integrity constraints to guide selection of match candidates [DDH01] | match / mismatch rules + iterative refinement | resolving multiple matches, adding new matching rules | resolving structural conflicts (preprocessing) | user can adjust weights in match calculations | user can adjust threshold weights |
| Application area | data integration | data integration with pre-defined global schema | ontology composition for data integration / interoperability | data translation | schema integration | schema integration | data translation, but intended to be generic |
| Remarks | neural networks; C implementation | | "algorithms" implicitly represented by rules | rules implemented in Java | algorithms to calculate new synonyms, homonyms, similarity metrics | also embedded in the MOMIS mediator, with extensions | |

different matchers, as discussed in Sect. 8, is not yet supported. Most systems provide both structure-level and element-level matching, in particular name and constraint-based matching. However, only two of the seven systems consider instance data and all systems focus on (local) 1:1 matches (two systems support global n:1 matches). Most prototypes have been developed with a specific application domain in mind, mostly data and schema integration, while Cupid strives for general applicability. Most systems support multiple schema types, while LSD is limited to XML and DIKE to ER sources. All systems allow the user to validate generated match results (not shown in the table) and require additional manual work to instrument the system, e.g., by providing prior match knowledge or tuning parameters such as similarity thresholds. The main forms of auxiliary information and reuse support is the provision of thesauri and glossaries and specification of specific match knowledge. Reuse of previous match results is not yet supported.

In this section, we discuss some specific features of the seven approaches. In Sect. 9.2, we briefly highlight some additional schemes. Most of them offer less support with respect to automatic matching and have thus not been included in Table 5.

### SemInt (Northwestern Univ.)

The SemInt match prototype [LC94, LC00, LCL00] creates a mapping between individual attributes of two schemas (i.e., its match cardinality is 1:1). It exploits up to 15 constraint-based and 5 content-based matching criteria. The schema-level constraints use the information available from the catalog of a relational DBMS. Instance data is used to enhance this information by providing actual value distributions, numerical averages, etc. For each criterion, the system uses a function to map each possible value onto the interval [0..1]. Using these functions, *SemInt* determines a *match signature* for each attribute consisting of a value in the interval [0..1] for N matching criteria (either all or a selected subset of the supported criteria). Since signatures correspond to points in the N-dimensional space, the Euclidian distance between signatures can be used as a measure of the degree of similarity and thus for determining an ordered list of match candidates.

In its main approach, SemInt uses neural networks to determine match candidates. This approach requires similar attributes of the first input schema (e.g., foreign and primary keys) to be clustered together. Clustering is automatic by assigning all attributes with a distance below a threshold value to the same cluster. The neural network is trained with the signatures of the cluster centers. The signatures of attributes from the second schema are then fed into the neural network to determine the best matching attribute cluster from the first schema. Based on their experiments the authors found that the straightforward match approach based on Euclidian distance does well on finding almost identical attributes, while the neural network is better at identifying less similar attributes that match[2]. However, the neural network approach has sub-

[2] To evaluate the effectiveness of a match approach, the authors use the IR metrics recall and precision. *Recall* indicates which percentage of all matches in the schemas are correctly determined. *Precision* indicates the fraction of all determined matches that are correct.

stantial performance problems for larger schemas according to [CHR97]. To improve efficiency, the approach identifies a match only to attribute clusters leaving it to the user to select the matching attributes from the cluster.

SemInt represents a powerful and flexible approach to hybrid matching, since multiple match criteria can be selected and evaluated together. This is in contrast to other hybrid matchers using several criteria in a hard-wired fashion[3]. SemInt does not support name-based matching or graph matching for which it may be difficult to determine a useful mapping to the [0..1] interval.

### LSD (Univ. of Washington)

The LSD (Learning Source Descriptions) system uses machine-learning techniques to match a new data source against a previously determined global schema, producing a 1:1 atomic-level mapping [DDL00, DDH01]. It represents a composite match scheme with an automatic combination of match results. In addition to a name matcher they use several instance-level matchers (learners) that are trained during a preprocessing step. Given a user-supplied mapping from a data source to the global schema, the preprocessing step looks at instances from that data source to train the learner, thereby discovering characteristic instance patterns and matching rules. These patterns and rules can then be applied to match other data sources to the global schema. Given a new data source, each matcher determines a list of match candidates.

A global matcher that uses the same machine-learning technology is used to merge the lists into a combined list of match candidates for each schema element. It too is trained on schemas for which a user-supplied mapping is known, thereby learning how much weight to give to each component matcher. New component matchers can be added to improve the global matcher's accuracy.

Although the approach is primarily instance-oriented, it can exploit schema information too. A learner can take self-describing input, such as XML, and make its matching decisions by focusing on the schema tags while ignoring the data instance values. LSD has also been extended to consider user-supplied domain constraints on the global schema to eliminate some of the previously determined match candidates for improving match accuracy [DDH01].

### SKAT (Stanford Univ.)

The SKAT (Semantic Knowledge Articulation Tool) prototype follows a rule-based approach to semi-automatically determine matches between two ontologies (schemas) [MWJ99]. Rules are formulated in first-order logic to express match and mismatch relationships and methods are defined to derive new matches. The user has to initially provide application-specific match and mismatch relationships and then approve or reject generated matches. The description in [MWJ99] deals with name matching and simple structural matches based on is-a hierarchies, but leaves open the details of what has been implemented.

SKAT is used within the ONION architecture for ontology integration [MWK00]. In ONION, ontologies are transformed into a graph-based object-oriented database model. Matching rules between ontologies are used to construct an "articulation ontology" which covers the "intersection" of source ontologies. Matching is based heavily on is-a relationships between the articulation ontology and source ontologies. The articulation ontology is to be used for queries and for adding more sources.

### TransScm (Tel Aviv Univ.)

The TranScm prototype [MZ98] uses schema matching to derive an automatic data translation between schema instances. Input schemas are transformed into labeled graphs, which is the internal schema representation. Edges in the schema graphs represent component relationships. All other schema information (name, optionality, #children, etc.) is represented as properties of the nodes. The matching is performed node by node (element-level, 1:1) starting at the top and presumes a high degree of similarity between the schemas. There are several matchers which are checked in a fixed order. Each matcher is a "rule" implemented in Java. They require that the match is determined by exactly one matcher per node pair. If no match is found or if a matcher determines multiple match candidates, user intervention is required, e.g., to provide a new rule (matcher) or to select a match candidate. The matchers typically consider multiple criteria and can thus represent hybrid approaches. For example, one of the matchers tests the name properties and the number of children. Node matching can be made dependent on a partial or full match of the nodes' descendents.

### DIKE (Univ. of Reggio Calabria, Univ. of Calabria)

In [PSU98a, PSTU99], Palopoli et al. propose algorithms to automatically determine synonym and inclusion (is-a, hypernym) relationships between objects of different entity-relationship schemas. The algorithms are based on a set of user-specified synonym, homonym, and inclusion properties that include a numerical "plausibility factor" (between 0 and 1) about the certainty that the relationship is expected to hold. In order to (probabilistically) derive new synonyms and homonyms and the associated plausibility factors, the authors perform a pairwise comparison of objects in the input schemas by considering the similarity properties of their "related objects" (i.e., their attributes and the is-a and other relationships the objects participate in).

In [PSU98b], the focus is to find pairs of objects in two schemas that are similar, in the sense that they have the same attributes and relationships, but are of different "types," where type $\in$ {entity, attribute, relationship}. The similarity of two objects is a value in the range [0,1]. If the similarity exceeds a given threshold, they regard the objects as matching, and therefore regard a type conflict as significant. Thus, schema matching is the main step of their algorithm. For a given pair of objects $o_1$ and $o_2$ being compared, objects related to $o_1$ and

[3] According to our characterization in Sect. 8, SemInt is not a composite matcher since it does not combine independently calculated match results.

$o_2$ contribute to the degree of similarity of $o_1$ and $o_2$ with a weight that is inversely proportional to their distance from $o_1$ and $o_2$, where distance is the minimum number of many-to-many relationships on any path from $o_1$ to $o_2$. Thus, objects that are closely related to $o_1$ and $o_2$ (e.g., their attributes and objects they directly reference) count more heavily than those that are reachable only via paths of relationships.

The above algorithms are embodied in the DIKE system, described in [PTU00, Ur99]. Related algorithms by the same authors are in [TU00, RTU01].

ARTEMIS (Univ. of Milano, Univ. of Brescia) & MOMIS (Univ. of Modena and Reggio Emilia)

ARTEMIS is a schema integration tool [CDD01, CD99]. It first computes "affinities" in the range 0 to 1 between attributes, which is a match-like step. It then completes the schema integration by clustering attributes based on those affinities and then constructing views based on the clusters.

The algorithm operates on a hybrid relational-OO model that includes the name, data types, and cardinalities of attributes and target object types of attributes that refer to other objects. It computes matches by a weighted sum of name and data type affinity and structural affinity. Name affinity is based on generic and domain-specific thesauri, where each association of two names is a synonym, hypernym, or general relationship, with a fixed affinity for each type of association. Data type affinity is based on a generic table of data type compatibilities. Structural affinity of two entities is based on the similarity of relationships emanating from those entities.

ARTEMIS is used as a component of a heterogeneous database mediator, called MOMIS (Mediator envirOment for Multiple Information Sources) [BCV99, BCC*00, BCVB01]. MOMIS integrates independently developed schemas into a virtual global schema on the basis of a reference object-based data model, which it uses to represent relational, object-oriented and semi-structured source schemas. MOMIS relies on ARTEMIS, the lexical system WordNet, and the description-logic-based inference tool ODB-Tools to produce an integrated virtual schema. It also offers a query processor (with optimization) to query the heterogeneous data sources.

Cupid (Microsoft Research)

Cupid is a hybrid matcher based on both element- and structure-level matching [MBR01]. It is intended to be generic across data models and has been applied to XML and relational examples. It uses auxiliary information sources for synonyms, abbreviations, and acronyms. Like DIKE, each entry in these auxiliary sources include a plausibility factor in the [0, 1] range. Unlike DIKE, Cupid can exploit entries that are ordinary words (e.g., Invoice is a synonym of Bill), without requiring them to exactly match compound names of elements (e.g., InvoiceTo or bill_address).

The algorithm has three phases. The first phase does linguistic element-level matching and categorizes elements based on names, data types, and domains (making Cupid hybrid). It parses compound element names into tokens based on delimiters (e.g., Product_ID becomes {Product, ID}), categorizes them based on their data types and linguistic content, and then calculates a linguistic similarity coefficient between data-type- and linguistic-content-compatible pairs of names based on substring matching and auxiliary sources. The second phase transforms the original schema into a tree and then does a bottom-up structure matching, resulting in a structural similarity between pairs of element. This transformation encodes referential constraints into structures that can be matched just like other structures (making Cupid constraint-based). The similarity of two elements at the root of structures is based on their linguistic similarity and the similarity of their leaf sets. If the similarity exceeds a threshold, then their leaf set similarity is incremented. The focus on leaf sets is based on the assumption that much of the information content is represented in leaves and that leaves have less variation between schemas than internal structure. Phase two concludes by calculating a weighted mean of linguistic and structural similarity of pairs of elements. The third phase uses that weighted mean to decide on a mapping. This phase is regarded as application dependent and not emphasized in the algorithm.

Experiments were run to compare Cupid to DIKE and MOMIS on several schema examples. Cupid performed somewhat better overall. However, the more interesting results were in the value of particular features of each algorithm on particular aspects of the examples, which are too detailed to summarize here.

### *9.2. Related prototypes*

This section describes five other prototypes that offer functionality that is related to the schema matching approaches discussed in this paper.

Clio (IBM Almaden and Univ. of Toronto)

The Clio tool under development at IBM Research in Almaden aims at a semi-automatic (user-assisted) creation of match mappings between a given target schema and a new data source schema. It consists of a set of Schema Readers, which read a schema and translate it into an internal representation; a Correspondence Engine (CE), which is used to identify matching parts of the schemas or databases; and a Mapping Generator, which generates view definitions to map data in the source schema into data in the target schema [HMNT99, Mi01]. The correspondence engine makes use of n:m element-level matches obtained from a knowledge-base or entered by a user through a graphical user interface. In [MHH00], Miller et al. present an algorithm for deriving a mapping between the target and source, given a set of element and substructure matches and match expressions. It selects enough of the matches to cover a maximal set of columns of the target schema and uses constraint reasoning to suggest join clauses to tie together components of the source schema. [YMHF01] proposes the use of sample data instances for the input schemas to interactively guide the construction of a mapping query and to verify its correctness.

Similarity flooding (Stanford Univ. and Univ. of Leipzig)

In [MGR02], Melnik et al. present a graph matching algorithm called Similarity Flooding (SF) and explore its usability for schema matching. The approach converts schemas into directed labeled graphs and uses fixpoint computation to determine the matches between corresponding nodes of the graphs. It produces a 1:1 local, m:n global mapping between schema elements. The SF algorithm is implemented as one of the operators in a prototype of a generic schema manipulation tool. In addition to the structural SF matcher the tool supports operators such as a name matcher, schema converters, and filters that can be combined within scripts. A typical match script starts with converting the two input schemas into the internal graph representation. Then a name matcher is used to suggest an initial element-level mapping which is fed to the structural SF matcher. In the last step, various filters are applied to select relevant subsets of match results produced by the structural matcher. The tool accepts several input formats, in particular SQL DDL, XML, and RDF.

Delta (MITRE)

Delta represents a simple approach for determining attribute correspondences utilizing attribute descriptions [BHFW95, CHR97]. All available metadata about an attribute (e.g., text description, attribute name, and type information) is grouped and converted into a simple text string, which is presented as a *document* to a full-text information retrieval tool. The IR tool can interpret such a document as a query. Documents of another schema with matching attributes are determined and ranked. Selection of the matches from the result list is left to the user. The approach is easy to implement but depends on the availability and expressiveness of text descriptions for attributes. [CHR97] compares experimental match results obtained with Delta with those obtained with the SemInt tool and proposes to combine the two approaches, which would result in a composite matcher.

Tess (Univ. of Massachusetts, Amherst)

Tess is a system for helping to cope with schema evolution [Le00]. A schema is a set of types. Tess takes a definition of the old and new type and produces a program to transform data that conforms to the old type into data that conforms to the new type. To accomplish this, it automatically creates a mapping from the old to the new type, using a schema-level matching algorithm. Like TransScm, it presumes a high degree of similarity between the schemas. It identifies pairs of types as match candidates, and then recursively tries to match their substructure in a top-down fashion. Two elements are match candidates if they have the same name, if they have a pair of subelements that match (i.e., that are of the same type), or if they use the same type constructor (in order of preference, where name matching is most preferred). The recursion bottoms out with scalar subelements. As the recursive calls percolate back up, matching decisions on coarser-grained elements are made based on the results of their finer-grained subelements. In this sense, Tess performs both structure-level and element-level matching.

Tree matching (NYU)

Zhang and Shasha developed an algorithm to find a mapping between two labeled trees [ZS89, ZSW92, ZS97], which they later implemented in a system for approximate tree matching [WZJS94]. This is a purely structural match, with no notion of synonym or hypernym. However, it can cope with name mismatches by treating "rename" as one of the transformations that can map one tree into the other. Implementations are available at [ZSW00].

There is, of course, a large literature on graph isomorphism which could be useful. An investigation of its relevance to the more specific problem of schema matching is beyond the scope of this paper.

## 10. Conclusion

Schema matching is a basic problem in many database application domains, such as heterogeneous database integration, E-commerce, data warehousing, and semantic query processing. In this paper, we proposed a taxonomy that covers many of the existing approaches and we described these approaches in some detail. In particular, we distinguished between schema- and instance-level, element- and structure-level, and language- and constraint-based matchers and discussed the combination of multiple matchers. We used the taxonomy to characterize and compare a variety of previous match implementations. We hope that the taxonomy will be useful to programmers who need to implement a match algorithm and to researchers looking to develop more effective and comprehensive schema matching algorithms. For instance, more attention should be given to the utilization of instance-level information and reuse opportunities to perform Match.

Past work on schema matching has mostly been done in the context of a particular application domain. Since the problem is so fundamental, we believe the field would benefit from treating it as an independent problem, as we have begun doing here. In the future, we would like to see quantitative work on the relative performance and accuracy of different approaches. Such results could tell us which of the existing approaches dominate the others and could help identify weaknesses in the existing approaches that suggest opportunities for future research.

*Acknowledgements.* We are grateful for many helpful suggestions from Sonia Bergamaschi, Silvana Castano, Chris Clifton, Hai Hong Do, An Hai Doan, Alon Halevy, Jayant Madhavan, Sergey Melnik, Renée Miller, Rachel Pottinger, Arnie Rosenthal, Dennis Shasha, and the anonymous referees.

## References

[BBC*00] Beneventano D, Bergamaschi S, Castano S, Corni A, Guidetti R, Malvezzi G, Melchiori M, Vincini M (2000) Information integration: the MOMIS project demonstration. In: Proc 26th Int Conf On Very Large Data Bases, pp. 611–614

[BLN86] Batini C, Lenzerini M, Navathe SB (1986) A comparative analysis of methodologies for database schema integration. ACM Comput Surv 18(4):323–364

[BFHW95] Benkley S, Fandozzi J, Housman E, Woodhouse G (1995) Data element tool-based analysis (DELTA). MITRE Technical Report MTR'95 B147

[BM01] Berlin J, Motro M (2001) Autoplex: automated discovery of content for virtual databases. In: Proc 9th Int Conf On Cooperative Information Systems (CoopIS), Lecture Notes in Computer Science, vol. 2172. Springer, Berlin Heidelberg New York, 2001, pp. 108–122

[BCV99] Bergamaschi S, Castano S, Vincini M (1999) Semantic integration of semistructured and structured data sources. ACM SIGMOD Record 28(1):54–59

[BCVB01] Bergamaschi S, Castano S, Vincini M, Beneventano D (2001) Semantic integration of heterogeneous information sources. Data Knowl Eng 36(3):215–249

[BS01] Bell GS, Sethi A (2001) Matching records in a national medical patient index. CACM 44(9):83-88

[BHL01] Berners-Lee T, Hendler J, Lassila O (2001) The semantic web. Sci Am 284(5):34–43

[Be00] Bernstein PA (2000) Is generic metadata management feasible? Panel overview. In: Proc 26th Int Conf On Very Large Data Bases, pp. 660–662

[BHP00] Bernstein PA, Halevy A, Pottinger RA (2000) A vision for management of complex models. A vision for management of complex models. ACM SIGMOD Record 29(4):55–63

[BR00] Bernstein PA, Rahm E (2000) Data warehouse scenarios for model management. In: Proc 19th Int Conf On Entity-Relationship Modeling, Lecture Notes in Computer Science, vol. 1920. Springer, Berlin Heidelberg New York, 2000, pp. 1–15

[BHP94] Bright MW, Hurson AR, Pakzad SH (1994) Automated resolution of semantic heterogeneity in multidatabases. TODS 19(2):212–253

[CD99] Castano S, De Antonellis V (1999) A schema analysis and reconciliation tool environment. In: Proc Int Database Eng Appl Symp (IDEAS), IEEE Computer, New York, pp. 53–62

[CDD01] Castano S, De Antonellis V, De Capitani di Vemercati S (2001) Global viewing of heterogeneous data sources. IEEE Trans Data Knowl Eng 13(2):277–297

[CHR97] Clifton C, Housman E, Rosenthal A (1997) Experience with a combined approach to attribute-matching across heterogenenous databases. In: Proc 7, IFIP 2.6 Working Conf. Database Semantics

[Co98] Cohen WW (1998) Integration of heterogeneous databases without common domains using queries based on textual similarity. In: Proc ACM SIGMOD Conf, pp. 201–212

[DDL00] Doan AH, Domingos P, Levy A (2000) Learning source descriptions for data integration. In: Proc WebDB Workshop, pp. 81–92

[DDH01] Doan AH, Domingos P, Halevy A (2001) Reconciling schemas of disparate data sources: a machine-learning approach. In: Proc ACM SIGMOD Conf, pp. 509–520

[EP90] Elmagarmid AK, Pu C (1990) Guest editors' introduction to the special issue on heterogeneous databases. ACM Comput Surv 22(3):175–178

[EJX01] Embley DW, Jackman D, Xu L (2001) Multifaceted exploitation of metadata for attribute match discovery in information integration. In: Proc Int Workshop on Information Integration on the Web, pp. 110–117

[GW97] Goldman R, Widom J (1997) Dataguides: enabling query formulation and optimization in semistructured databases. In: Proc 23th Int Conf On Very Large Data Bases, pp. 436–445

[HMNT99] Haas LM, Miller RJ, Niswonger B, Tork Roth, Schwarz PM, Wimmers EL (1999) Transforming heterogeneous data with database middleware: beyond integration. IEEE Tech Bull Data Eng 22(1):31–36

[KKFG84] Korth HF, Kuper GM, Feigenbaum J, Van Gelder A, Ullman JD (1984) System/U: a database system based on the universal relation assumption. ACM TODS 9(3):331–347

[LNE89] Larson JA, Navathe SB, ElMasri R (1989) A theory of attribute equivalence in databases with application to schema integration. IEEE Trans Software Eng 16(4):449 –463

[LC94] Li W, Clifton C (1994) Semantic integration in heterogeneous databases using neural networks. In: Proc 20th Int Conf On Very Large Data Bases, pp. 1–12

[LC00] Li W, Clifton C (2000) SemInt: a tool for identifying attribute correspondences in heterogeneous databases using neural network. Data Knowl Eng 33(1):49–84

[LCL00] Li W, Clifton C, Liu S (2000) Database integration using neural network: implementation and experiences. Knowl Inf Syst 2(1):73–96

[Le00] Lerner BS (2000) A model for compound type changes encountered in schema evolution. ACM TODS 25(1):83–127

[MBR01] Madhavan J, Bernstein PA, Rahm E (2001) Generic schema matching with Cupid. In: Proc 27th Int Conf On Very Large Data Bases, pp. 49–58

[MRSS82] Maier D, Rozenshtein D, Salveter SC, Stein J, Warren DS (1982) Toward logical data independence: a relational query language without relations. In: Proc ACM SIGMOD Conf, pp. 51–60

[MGR02] Melnik S, Garcia-Molina H, Rahm E (2002) Similarity flooding - a versatile graph matching algorithm. In: Proc 18th Int Conf Data Eng (to appear)

[MHH00] Miller RJ, Haas L, Hern'andez MA (2000) Schema mapping as query discovery. In: Proc 26th Int Conf On Very Large Data Bases, pp. 77–88

[MIR94] Miller R, YE Ioannidis, Ramakrishnan R (1994) Schema equivalence in heterogeneous systems: bridging theory and practice. Inf Syst 19(1):3–31

[Mi01] Miller R, et al (2001) The Clio project: managing heterogeneity. ACM SIGMOD Record 30(1):78–83

[MZ98] Milo T, Zohar S (1998) Using schema matching to simplify heterogeneous data translation. In: Proc 24th Int Conf On Very Large Data Bases, pp. 122–133

[MWJ99] Mitra P, Wiederhold G, Jannink J (1999) Semi-automatic integration of knowledge sources. In: Proc of Fusion '99, Sunnyvale, USA,

[MWK00] Mitra P, Wiederhold G, Kersten M (2000) A graph-oriented model for articulation of ontology interdependencies. In: Proc Extending DataBase Technologies, Lecture Notes in Computer Science, vol. 1777. Springer, Berlin Heidelberg New York, 2000, pp. 86–100

[PS98] Parent C, Spaccapietra S (1998) Issues and approaches of database integration. CACM 41(5):166–178

[PSTU99] Palopoli L, Sacca D, Terracina G, Ursino D (1999) A unified graph-based framework for deriving nominal interscheme properties, type conflicts and object cluster similarities. In: Proc 4th IFCIS Int Conf On Cooperative Information Systems (CoopIS), IEEE Comput, pp. 34–45

[PSU98a] Palopoli L, Sacca D, Ursino D (1998) Semi-automatic, semantic discovery of properties from database schemas. In: Proc Int. Database Engineering and Applications Symp. (IDEAS), IEEE Comput, pp. 244–253

[PSU98b] Palopoli L, Sacca D, Ursino D (1998) An automatic technique for detecting type conflicts in database schemas. In: Proc $7^{th}$ Int Conf On Information and Knowledge Management (CIKM), pp. 306–313

[PTU00] Palopoli L, Terracina G, Ursino D (2000) The system DIKE: towards the semi-automatic synthesis of cooperative information systems and data warehouses. In: Proc ADBIS-DASFAA Conf, Matfyz, pp. 108–117

[RTU01] Rosaci D, Terracina G, Ursino D (2001) Deriving sub-source similarities from heterogeneous, semi-structured information sources. In: Proc 9th Int Conf On Cooperative Information Systems (CoopIS), Lecture Notes in Computer Science, vol. 2172. Springer, Berlin Heidelberg New York, 2001, pp. 150–162

[RYAC00] Rishe N, Yuan J, Athauda R, Chen SC, Lu X, Ma X, Vaschillo A, Shaposhnikov A, Vasilevsky D (2000) Semantic access: semantic interface for querying databases. In: Proc 26th Int Conf On Very Large Data Bases, pp. 591–594

[SL90] Sheth AP, Larson JA (1990) Federated database systems for managing distributed, heterogeneous, and autonomous databases. ACM Comput Surv 22(3):183–236

[TU00] Terracina G, Ursino D (2000) Deriving synonymies and homonymies of object classes in semi-structured information sources. Advances in data management, Tata McGraw-Hill, pp. 21–32

[Ur99] Ursino D (1999) Semiautomatic approaches and tools for the extraction and the exploitation of intentional knowledge from heterogeneous information sources. Ph.D. Thesis. http://www.ing.unirc.it/didattica/inform00/ursino/tesi.zip

[WS90] Wald JA, Sorenson PG (1990) Explaining ambiguity in a formal query language. ACM TODS 15(2):125–161

[WYW00] Wang Q, Yu J, Wong K (2000) Approximate graph schema extraction for semi-structured data. In: Proc Extending DataBase Technologies, Lecture Notes in Computer Science, vol. 1777. Springer, Berlin Heidelberg New York, 2000, pp. 302–316

[WZJS94] Wang JTL, Zhang K, Jeong K, Shasha D (1994) A system for approximate tree matching. IEEE Trans Data Knowl Eng 6(4):559–571

[YMHF01] Yan L, Miller RJ, Haas LM, Fagin R (2001) Data-driven understanding and refinement of schema mappings. In: Proc ACM SIGMOD Conf, pp. 485–496

[ZS89] Zhang K, Shasha D (1989) Simple fast algorithms for the editing distance between trees and related problems. SIAM J Comput 18:1245–1262

[ZS97] Zhang K, Shasha D (1997) Approximate tree pattern matching. In: Apostolico A, Galil Z (eds) Pattern matching in strings, trees, and arrays. Oxford University, Oxford, pp. 341–371

[ZSW92] Zhang K, Shasha D, Wang JTL (1992) Fast serial and parallel algorithms for approximate tree matching with VLDC's. In: Proc Int Conf Combinatorial Pattern Matching, pp. 148–158

[ZSW00] Zhang K, Shasha D, Wang JTL:
http://cs.nyu.edu/cs/faculty/shasha/papers/agm.html,
http://cs.nyu.edu/cs/faculty/shasha/papers/tree.html,
http://cs.nyu.edu/cs/faculty/shasha/papers/treesearch.html