On Matching Schemas Automatically

Erhard Rahm
University of Leipzig
Leipzig, Germany
rahm@informatik.uni-leipzig.de

Philip A. Bernstein
Microsoft Research
Redmond, WA, U.S.A.
philbe@microsoft.com

February, 2001
Technical Report
MSR-TR-2001-17

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
http://www.research.microsoft.com

On Matching Schemas Automatically

Erhard Rahm, University of Leipzig, Germany[*]

Philip A. Bernstein, Microsoft Research, Redmond, WA

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, in previous research many techniques have been proposed to achieve a partial automation of the Match operation for specific application domains. We present a taxonomy that covers many of the existing approaches, and we describe these approaches in some detail. In particular, we distinguish between schema- and instance-level, element- and structure-level, and language- 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.

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 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. Moreover, the level of effort is linear in the number of matches to be performed, a growing problem given the rapidly increasing number of web data sources and e-businesses to integrate. 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 [BLP00, Be00, BR00].

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.

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 some part of) the solution space. We use the

[*] This work was performed while on leave at Microsoft Research, Redmond, WA.

classification in Sections 6 through 8 to organize our presentation of previously proposed approaches 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.

Schema Integration

Most work on schema match has been motivated by the problem of schema integration: Given a set of independently developed schemas, construct a global view [BLN86]. In an artificial intelligence setting, this is the problem of integrating independently developed ontologies, generating an integrated ontology.

Since the schemas are independently developed, they often have different structure and terminology, even if they model the same real world domain. 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.

Data Warehouses

A variation of the schema integration problem 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 S′ is to reuse an existing source-to-warehouse transformation S⇒W. First, the common elements of S′ and S are found (a Match operation) and then S⇒W is reused for those common elements.

E-Commerce

In E-commerce, 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. The application designer would still need to validate and probably modify the result of the automated match.

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, 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 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.

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# with Customer.CustID according to 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.

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

Table 1: Sample Input Schemas

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. 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 mappings that do not include mapping expressions. We therefore often represent a mapping as a similarity relation, ≅, over the powersets of S1 and S2, where each pair in ≅ represents one mapping element of the mapping. For example, the result of a Match between the schemas of Table 1 could be "Cust.C# ≅ Customer.CustID", "Cust.CName ≅ Customer.Company", and "{Cust.FirstName, Cust.LastName} ≅ Customer.Contact". A complete specification of the result of the 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 ≅.

Match has similarities with the join operation of relational databases. Like Match, Join is a binary operation that determines pairs of corresponding elements from its input operands. On the other hand, there are several differences. First, Match operates on metadata (schemas) while Join deals with data (instances). Second, 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. Finally, the 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 the Match result may have a different mapping expression. Hence, the semantics of Match is much less restricted than for Join and much more difficult to capture in a consistent way.

As introduced in [BR00], OuterMatch operations are useful counterparts to Match in much the same way as 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. 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

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 tool 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 Match system too may use the libraries and other auxiliary information, such as dictionaries and thesauri, to help find matches.

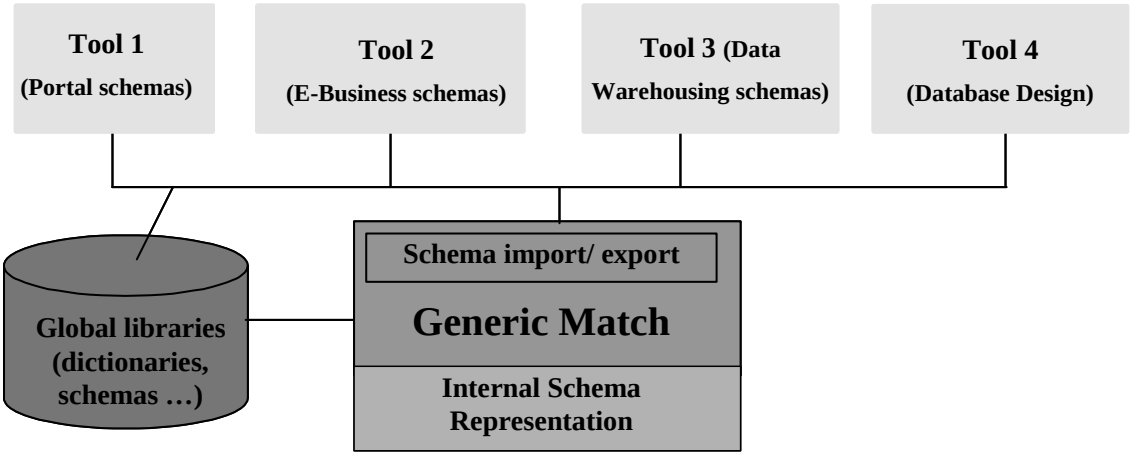

Figure 1: High Level Architecture of Generic Match

We assume that the generic Match implementation 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) models for representing 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 Match implementation from the internal representation into the representation required by each tool. This allows the generic Match implementation 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 Match implementation 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. Figure 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 a hybrid matcher or by combining multiple match results produced by different

match algorithms. 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 constraint-based approach (e.g., based on keys and relationships).
- *Matching cardinality:* Each element of the resulting mapping may match 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, there may be different match cardinalities at the instance level.
- *Auxiliary information:* Most matchers not only rely on the input schemas S1 and S2 but also on auxiliary information, such as dictionaries, global schemas, previous matching decisions, and user input.

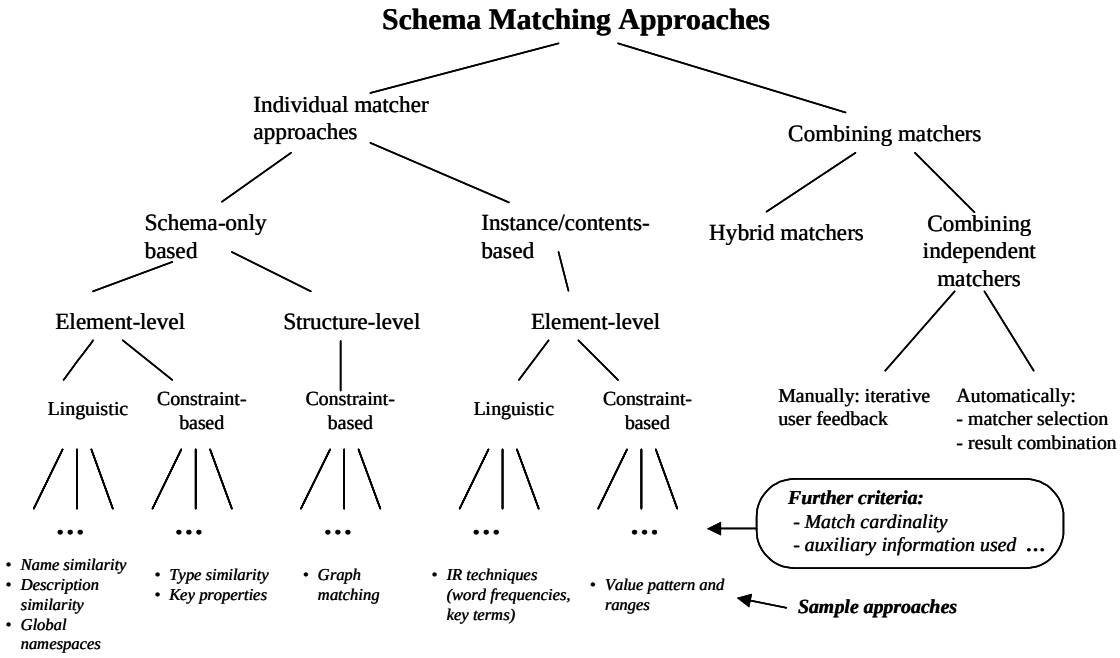

Figure 2: Classification of schema matching approaches

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 Section 6, instance-level matching in Section 7, and combinations of multiple matchers in Section 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, different kinds of relationships (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 to 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 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 1, 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 for the two cases are shown in Table 2. Notice that structure-level matching is used in combination with element matching, for example to match "State" with USState" and "ZIP" with "PostalCode."

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

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

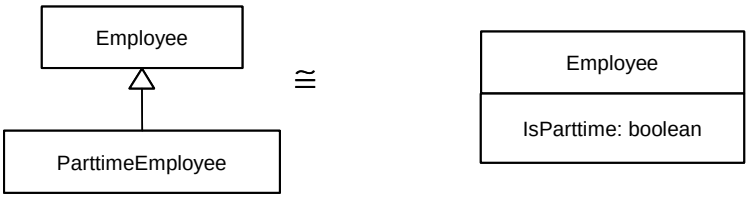

Figure 3: Equivalence pattern

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 Section 6.4.

Element-level matching is not restricted to the atomic level, but may also be applied to coarser grained, *higher (non-atomic) 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 the 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 may match one or multiple S2 (or S1) elements. Thus, we have the usual relationship cardinalities between matching elements of the two input schemas, namely 1:1 and the set-oriented cases 1:n, n:1, and n:m. Element-level matching is typically restricted to 1:1, n:1, and 1:n cardinalities between elements. Obtaining n:m mappings between elements usually requires considering the structural embedding of the elements and is thus covered by structure-level matching.

Table 3 shows examples of the four cases. In row 1, the match is 1:1 assuming that no other S1 elements match Amount and no other S2 elements match Price. Previous work has mostly concentrated on 1:1 matches because of the difficulty to automatically determine 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 4 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.

	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, LastName	FirstName, LastName = Extract (Name, …)
4.	n:1 structure-level (n:m element-level)	B.Title, B.PuNo, P.PuNo, P.Name	A.Book, A.Publisher	A.Book, A.Publisher = Select B.Title, P.Name From B, P Where B.PuNo=P.PuNo

Table 3: Match cardinalities (Examples)

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.

6.3 Linguistic Approaches

Language-based or linguistic matchers use names and text (i.e., words or sentences) to find semantically similar schema 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, pronunciation, soundex (an encoding of names based on how they sound rather than how they are spelled), etc.
  (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 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

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

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. For example, it may be useful 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 Section 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 S2.name, S1.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-σ, C-A-σ. Intuitively, we would expect the similarity value σ 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-σ for any similarity value σ.

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.

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 Section 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.

S1 elements	S2 elements

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

Table 4: Constraint-based matching (example)

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, 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).

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 XML namespaces, so the Match implementation 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. Of course, this is all predicated on being able to detect that the new schema is similar to one that was previously matched — a match problem in itself.

An example for such a re-use is shown in Fig. 4 for purchase order schemas. We already have the Match 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.

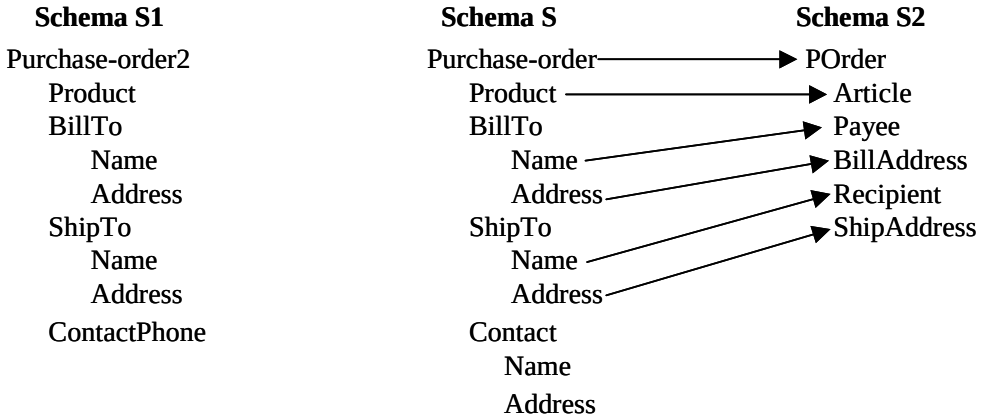

Figure 4: Scenario for reuse of an existing Match mapping

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 name 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 to match them 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 to generate a ranked list of S1 match candidates for each S2 element. Various approaches have been proposed to perform such an instance matching or classification, such as rules, neural networks, and machine learning techniques.

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 this term 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 combinatorial 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 has its strengths and weaknesses. 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: hybrid matchers and the combination of independently executed matchers.

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 combine the results of several independently executed matchers, including hybrid matchers. This allows selecting the matchers to be applied and combining them in a flexible way based on the application domain and input schemas (e.g., different approaches can be used for structured vs. semi-structured schemas). In the simplest case, a tool or the user invokes only a specific matcher at a time. The user provides feedback on the match candidates that were found (accepts some, rejects others, adds some matches manually), and then invokes another matcher to improve the match result. Subsequently called matchers must consider the already identified matches to reduce the evaluation overhead and focus on the unmatched parts of the schemas.

Alternatively, the Match implementation itself could select and execute multiple matchers and automatically combine their results, as in [DDL00]. Here, the ranked lists of candidate matches of each matcher are combined into a single one. A match candidate is included in the match result only if its combined similarity value is above a certain threshold. The automatic approach reduces the number of user interactions. On the other hand, it is difficult to achieve a generic solution for different application domains.

9. Sample Approaches from the Literature

9.1 Prototype Schema Matchers

In Table 5 we show how six published prototype implementations fit the classification criteria introduced in Section 5. The table thus indicates which part of the solution space is covered in which way, 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 (mostly schema and data integration) of the implementations. The table shows that only two of the six systems consider instance data and all systems focus on 1:1 matches.

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

		SemInt [LC94, LC00, LCL00]	LSD [ADL00]	SKAT [MWJ99, MWK00]	TranScm [MZ98]	DIKE [PSU98a,b, PSTU99]	ARTEMIS [CDD01, BCC*00]
Schema types		Relational, files	XML	XML, IDL, text	SGML, OO	ER	Relational, OO, ER
Metadata representation		unspecified (attribute-based)	XML schema trees	Graph-based object-oriented DB model	Labeled graph	Graph	Hybrid relational / OO data model
Match granularity		Element-level: Attributes (attribute clusters)	Element-level (atomic level only)	Structure-level: classes/attributes	Element-level	Element/structure-level: Entities / relationships / attributes	Element/structure-level: Entities / relationships / attributes
Match cardinality		1:1	1:1	1:1 and n:1	1:1	1:1	1:1
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
	constraint-based	15 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
	structure matching	-	-	Similarity w.r.t. "related" elements	Similarity w.r.t. "related" elements	Matching of neighborhood	Matching of neighborhood
instance-based 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	-	-	Thesauri
Combination of matchers		Hybrid matcher	Automatic: weighted combination of all learners per instance object; combination of instance predictions	-	Hybrid matchers; fixed order of matchers	-	Hybrid of name and structural matchers.
Manual work / user input		Selection of match criteria (optional)	User-supplied matches for training sources	Match / mismatch rules + iterative refinement	Resolving multiple matches, adding new matching rules	Resolving structural conflicts (preprocess-ing); specification of some synonyms + inclu-sions with similarity probabilities; validation	User can adjust weights in match calculations and validate match choices.
Application area		Data integration; 3 test cases	Data integration with pre-defined global schema	Ontology composition to support data integration / interoperability	Data translation	Schema integration	Schema integration
Remarks		neural networks; C implementation		"algorithms" implicitly represented by rules	Rules implemented in Java	Algorithms to calculate new synonyms, homo-nyms, similarity metrics	Also embedded in the MOMIS mediator, with extensions

Table 5: Characteristics of proposed schema match approaches

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] (e.g., data types "char", "numeric", and "date" are mapped to 1, 0.5, and 0, respectively). 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]. On the other hand, the approach only identifies a match to attribute clusters and not individual attributes.

SemInt represents a powerful and flexible approach to hybrid matching, since multiple match criteria can be selected and evaluated together. It 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]. 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.

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

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

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 ONIONS, 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 also for adding additional 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 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. So, 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 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 [DTU00, Ur99].

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 schemata into a virtual global schema on the basis of a reference object-based data model, which is 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.

9.2 Related Prototypes

This section describes three other prototypes that are 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, IBM00]. 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.

Delta (MITRE)

Delta represents a simple approach for determining attribute correspondences utilizing attribute descriptions [BHFW95, CHA97]. 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.

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 [SWZ00].

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 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.

Acknowledgments

We're grateful for many helpful suggestions from Sonia Bergamaschi, Silvana Castano, Chris Clifton, Hai Hong Do, An Hai Doan, Alon Halevy, Jayant Madhavan, Renee Miller, Rachel Pottinger, and Dennis Shasha.

References

BBC*00 Beneventano, D., S. Bergamaschi, S. Castano, A. Corni, R. Guidetti, G. Malvezzi, M. Melchiori, M. Vincini: Information Integration: the MOMIS Project Demonstration. Proc. VLDB 2000, 611-614.

BLN86 Batini, C., M. Lenzerini, S. B. Navathe: A Comparative Analysis of Methodologies for Database Schema Integration. Computing Surveys 18(4), 1986, 323-364

BFHW95 Benkley, S., J. Fandozzi, E. Housman, G. Woodhouse: Data Element Tool-Based Analysis (DELTA), MITRE Technical Report MTR´95B147, 1995.

BCV99 Bergamaschi, S., S. Castano, M. Vincini: Semantic Integration of Semistructured and Structured Data Sources. SIGMOD Record 28(1), 1999.

BCVB01 Bergamaschi, S., S. Castano, M. Vincini, D. Beneventano: Retrieving and Integrating Data from Multiple Sources: the MOMIS Approach. Special Issue on Intelligent Information Integration, Data & Knowledge Engineering, Elsevier Science B.V. (to appear).

Be00 Bernstein, P.A.: Is Generic Metadata Management Feasible? Panel Overview. Proc. VLDB 2000, 660-662.

BLP00 Bernstein, P.A., A. Levy, R. A. Pottinger: A Vision for Management of Complex Models. Technical Report MSR-TR-2000-53, http://www.research.microsoft.com/pubs/, June 2000.

BR00 Bernstein, P.A., E. Rahm: Data Warehouse Scenarios for Model Management, Proc. 19th Int. Conf. On Entity-Relationship Modeling, Springer-Verlag, LNCS Vol. 1920, Springer Verlag, 2000, 1-15.

CD99 S. Castano; V. De Antonellis: A Schema Analysis and Reconciliation Tool Environment. Proc. IDEAS '99, IEEE, 1999.

CDD01 Castano, S., V. De Antonellis, S. De Capitani di Vemercati: Semantic Integration of Heterogeneous Data Sources. Trans. On Data and Knowledge Eng., to appear.

CHA97 Clifton, C., E. Housman, A. Rosenthal: Experience with a Combined Approach to Attribute-Matching Across Heterogenenous Databases. Proc. 7. IFIP 2.6 Working Conf. Database Semantics, 1997

Co98 Cohen, W. W.: Integration of Heterogeneous Databases Without Common Domains Using Queries Based on Textual Similarity. Proc. ACM SIGMOD 1998.

DR00 Do, H.H. E. Rahm: On Metadata Interoperability in Data Warehouses. Univ. of Leipzig, Tech Report, March 2000. http://dol.uni-leipzig.de/

DDL00 Doan, A.H., P. Domingos, A. Levy: Learning Source Descriptions for Data Integration. Proc. WebDB 2000, pp. 81-92

GW97 Goldman, R., J. Widom: DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases. Proc. VLDB 1997, 436-445.

HMNT99 Haas, L.M., R.J. Miller, B. Niswonger, M. Tork Roth, P.M. Schwarz, E.L. Wimmers: *Transforming Heterogeneous Data with Database Middleware: Beyond Integration*. IEEE Tech. Bull. Data Engineering 22 (1), 1999:31-36

IBM00 IBM Corp., Clio web page: http://www.almaden.ibm.com/cs/clio/demo.html

KKFG84 Korth, H.F., G.M. Kuper, J. Feigenbaum, A. Van Gelder, J.D. Ullman: System/U: A Database System Based on the Universal Relation Assumption. ACM TODS 9(3), 1984, 331-347.

LNE89 Larson, J.A., S. B. Navathe, R. ElMasri: A Theory of Attribute Equivalence in Databases with Application to Schema Integration. IEEE Trans. On Software Eng. 16, 4 (Apr. 1989), 449 – 463.

LC94 Li, W., C. Clifton: Semantic Integration in Heterogeneous Databases Using Neural Networks. Proc. VLDB 1994, 1-12.

LC00 Li, W., C. Clifton: SemInt: A Tool for Identifying Attribute Correspondences in Heterogeneous Databases Using Neural Network. Data and Knowledge Engineering, 33 (1), 2000.

LCL00 Li, W., C. Clifton, S. Liu: Database Integration Using Neural Network: Implementation and Experiences. Knowledge and Information Systems, 2(1), 2000.

MRSS82 Maier, D., D. Rozenshtein, S.C. Salveter, J. Stein, D.S. Warren: Toward Logical Data Independence: A Relational Query Language Without Relations. Proc. ACM SIGMOD 1982, 51-60.

MHH00 Miller, R.J., L. Haas, M.A. Hernández: Schema Mapping as Query Discovery. Proc. VLDB 2000, 77-88.

MIR94 Miller, R., Y. E. Ioannidis, R. Ramakrishnan: Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice. Information Systems 19(1), 3-31, 1994

MZ98 Milo, T., S. Zohar: Using Schema Matching to Simplify Heterogeneous Data Translation. Proc. VLDB 1998.

MWJ99 Mitra, P., G. Wiederhold, J. Jannink: Semi-automatic Integration of Knowledge Sources. *Proc. of Fusion '99*, Sunnyvale, USA, July 1999.

MWK00 Mitra, P., G. Wiederhold, M. Kersten: " A Graph-Oriented Model for Articulation of Ontology Interdependencies"; Proc. Extending DataBase Technologies, EDBT 2000, LNCS Vol. 1777, Springer Verlag, 2000, 86-100..

PSTU99 Palopoli, L., D. Sacca, G. Terracina, D. Ursino: A Unified Graph-Based Framework for Deriving Nominal Interscheme Properties, Type Conflicts and Object Cluster Similarities. Proc.4th CoopIS, IEEE Computer Society, 1999, 34-45.

PSU98a Palopoli, L., D. Sacca, D. Ursino: Semi-Automatic, Semantic Discovery of Properties from Database Schemas. Proc. IDEAS, 1998, 244-253.

PSU98b Palopoli, L., D. Sacca, D. Ursino: An Automatic Technique for Detecting Type Conflicts in Database Schemas. Proc. CIKM, 1998, 306-313.

PTU00 Palopoli, L., G. Terracina, D. Ursino: The System DIKE: Towards the Semi-Automatic Synthesis of Cooperative Information Systems and Data Warehouses. ADBIS-DASFAA 2000, Matfyzpress, 108-117.

RYAC00 Rishe, N. J. Yuan, R. Athauda, S-C. Chen, X. Lu, X. Ma, A. Vaschillo, A. Shaposhnikov, D. Vasilevsky: Semantic Access: Semantic Interface for Querying Databases. VLDB 2000, 591-594.

Ur99 Ursino, D.: Semiautomatic Approaches and Tools for the Extraction and the Exploitation of Intentional Knowledge from Heterogeneous Information Sources. Ph.D. Thesis., 1999 http://www.ing.unirc.it/didattica/inform00/ursino/tesi.zip

WS90 Wald, J.A., P.G. Sorenson: Explaining Ambiguity in a Formal Query Language. ACM TODS 15(2), 1990, 125-161.

WYW00 Wang, Q., J. Yu, K. Wong: Approximate Graph Schema Extraction for Semi-Structured Data. Proc. Extending DataBase Technologies, EDBT 2000, LNCS Vol. 1777, Springer Verlag, 2000, 302-316

WZJS94 Wang, J.T.L., K. Zhang, K. Jeong, D. Shasha: A System for Approximate Tree Matching. IEEE TKDE 6, 4 (Aug. 1994), 559-571.

ZS89 Zhang, K., D. Shasha: Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. SIAM J. on Computing 18, 1989, 1245-1262.

ZS97 Zhang, K., D. Shasha: Approximate Tree Pattern Matching. In Pattern Matching in Strings, Trees, and Arrays, A. Apostolico and Z. Galil (eds.). Oxford University Press. 1997, 341-371.

ZSW92 Zhang, K., D. Shasha., J.T.L. Wang: Fast Serial and Parallel Algorithms for Approximate Tree Matching with VLDC's. Proc Int'l Conf. Combinatorial Pattern Matching, 1992, 148-158.

SWZ00 Zhang, K., D. Shasha, J.T.L. Wang: 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