

Matching large schemas: Approaches and evaluation

Hong-Hai Do^{a,*}, Erhard Rahm^b

^a*Interdisciplinary Center for Bioinformatics, University of Leipzig, Härtelstr. 16-18, 04107 Leipzig, Germany*

^b*Department of Computer Science, University of Leipzig, Augustusplatz 10-11, 04109 Leipzig, Germany*

Received 7 April 2005; received in revised form 18 April 2006; accepted 26 September 2006

Abstract

Current schema matching approaches still have to improve for large and complex Schemas. The large search space increases the likelihood for false matches as well as execution times. Further difficulties for Schema matching are posed by the high expressive power and versatility of modern schema languages, in particular user-defined types and classes, component reuse capabilities, and support for distributed schemas and namespaces. To better assist the user in matching complex schemas, we have developed a new generic schema matching tool, COMA + +, providing a library of individual matchers and a flexible infrastructure to combine the matchers and refine their results. Different match strategies can be applied including a new scalable approach to identify context-dependent correspondences between schemas with shared elements and a fragment-based match approach which decomposes a large match task into smaller tasks. We conducted a comprehensive evaluation of the match strategies using large e-Business standard schemas. Besides providing helpful insights for future match implementations, the evaluation demonstrated the practicability of our system for matching large schemas.

© 2006 Elsevier B.V. All rights reserved.

Keywords: Schema matching; Schema matching evaluation; Data integration; Schema integration; Model management

1. Introduction

Schema matching aims at identifying semantic correspondences between two schemas, such as database schemas, XML message formats, and ontologies. Solving such match problems is of key importance to service interoperability and data integration in numerous application domains. To reduce the manual effort required, many techniques

and prototypes have been developed to semi-automatically solve the match problem [1–5]. They exploit a wide range of information, such as schema characteristics (i.e. metadata, like element names, data types, and structural properties), characteristics of data instances, as well as background knowledge from dictionaries and thesauri.

Proposed match approaches were typically applied to some test schemas for which they could automatically determine most correspondences. As surveyed in [1], most test schemas were structurally rather simple and of small size of 50–100 elements. Unfortunately, the effectiveness of automatic match techniques typically decreases for larger schemas [6]. In particular, matching the complete input schemas

*Corresponding author. Current address: Hong-Hai Do, SAP AG, SAP Research CEC Dresden Chemnitz Straße 48, D-01187 Dresden, Germany. Tel.: +49 351 4811 6115; fax: +49 6227 78 46060.

E-mail addresses: hong-hai.do@sap.com (H.-H. Do), rahm@informatik.uni-leipzig.de (E. Rahm).

may lead not only to long execution times, but also poor quality due to the large search space. Moreover, it is difficult to present the match result to a human engineer in a way that she can easily validate and correct it.

Modern schema languages, e.g. W3C XML Schema and the new object-relational SQL versions (SQL:1999, SQL:2003), support advanced modeling capabilities, such as user-defined types and classes, aggregation and generalization, and component reuse, leading to additional complication for schema matching. In particular, the same complex types or substructures (e.g., for address, contact information, etc.) may occur many times in a schema with a context-dependent semantics. Such shared elements require special treatment to avoid an explosion of the search space and to effectively deal with n:m match cardinalities. Distributed schemas and namespaces, as supported by W3C XML Schema, also have not been considered in current match systems but schemas are assumed to be monolithic.

To better assist the user in matching large schemas, we have developed a new customizable generic matching system, COMA++, which is described in this paper. It is built upon our previous COMA prototype [6] and takes over its composite match approach to combine different match algorithms. COMA++ is fully operational [7]¹ and provides major improvements over previous work:

- We have developed a new approach to import *complex schemas* written in W3C XML Schema definition language and other languages with similar modeling capabilities. Structural conflicts due to alternative modeling methods in input schemas are detected and unified in our internal directed graph representation. Our approach is able to deal with large schemas distributed over many documents and namespaces.
- COMA++ extends COMA with a flexible infrastructure to *construct more powerful matchers* by combining existing ones and to refine previously determined match results. We are able to construct match strategies as workflows to divide and solve complex match tasks in multiple stages of successive refinement. The implementation of matchers has been optimized to achieve fast execution times for large match problems.

- We address the problem of *context-dependent matching*, which is important for schemas with shared elements. Our new context-dependent match approach can also scale to very large schemas.
- To better support user interaction and to improve performance in matching large schemas, we have implemented the *fragment-based match approach* proposed in [8]. Following the divide-and-conquer idea, it decomposes a large match problem into smaller sub-problems by matching at the level of schema fragments. As a first step, we support three static fragment types.
- Due to the flexibility to customize matchers and match strategies, our system can also be used as a platform for comparative evaluation of different match approaches. We performed a *comprehensive evaluation* based on large real-world schemas to demonstrate the practicability of our system. We analyze the quality and execution time of different match strategies and the impact of many factors, such as schema size, the choice of matchers, and of combination strategies. The resulting insights should be helpful for the development and evaluation of future match systems.

The rest of the paper is organized as follows: The next section gives an overview of our system. Section 3 presents the approach to import external Schemas and unify conflicts due to alternative modeling methods. Section 4 describes how individual match algorithms can be combined and which matchers are currently supported. Section 5 presents our approach to refine previously determined match results and to construct match strategies, especially for context-dependent and fragment-based matching. The evaluation is described in Section 6. Section 7 reviews recent related work. Finally, we conclude and discuss future work in Section 8.

2. System overview

Fig. 1a shows the architecture of COMA++. It consists of five components, the *Repository* to persistently store match-related data, the *Schema* and *Mapping Pools* to manage schemas and mappings in memory, the *Match Customizer* to configure matchers and match strategies, and the *Execution Engine* to execute match operations. All components are managed and used through an integrated graphical user interface (GUI). Given

¹The latest implementation of COMA++ can be downloaded from <http://dbs.uni-leipzig.de>.

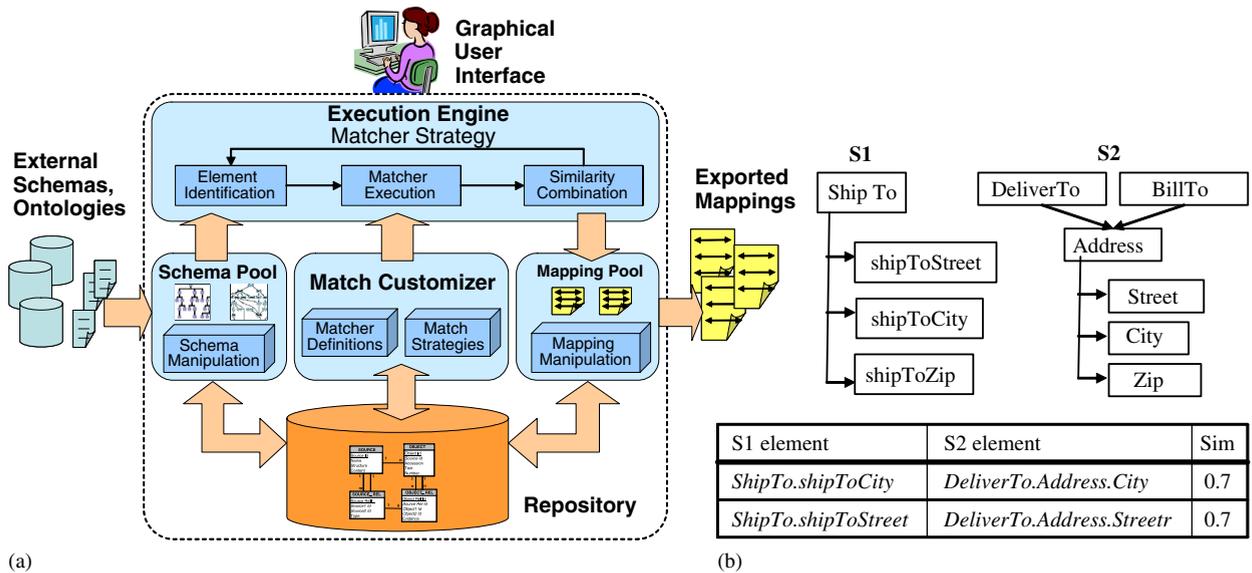


Fig. 1. System architecture and sample input/output (Schemas/mappings) of a match operation. (a) System architecture and (b) sample schemas and mapping.

that no fully automatic solution is possible, we allow the match process to be interactively influenced in many ways, i.e. before match to configure a match strategy, during match for iterative refinement, as well as after match to manipulate the obtained match results.

The repository centrally stores various types of data related with match processing, in particular: (a) imported schemas; (b) produced mappings; (c) auxiliary information such as domain-specific taxonomies and synonym tables; and (d) the definition and configuration of matchers and match strategies. As illustrated in Fig. 1b, schemas are uniformly represented by directed acyclic graphs as the internal format for matching. The Schema Pool provides different functions to import external schemas and to load and save them from/to the repository. Currently, we support XML Schema definition (XSD), XML data reduced (XDR), relational schemas via open database connectivity (ODBC), and Web ontology language (OWL). From the Schema Pool, two arbitrary schemas can be selected to start a match operation.

The match operation is performed in the Execution Engine according to a *match strategy* configured in the Match Customizer. As indicated in Fig. 1a, it is based on iterating three steps, *element identification* to determine the relevant schema elements for matching, *matcher execution* applying multiple matchers to compute the element similarities, and *similarity combination* to combine

matcher-specific similarities and derive a mapping with the best correspondences between the elements. The obtained mapping can in turn be used as input in the next iteration for further refinement. Each iteration can be individually configured by the Match Customizer, in particular, w.r.t. the types of elements to be considered, the choice of matchers, and the strategies for similarity combination (see Section 4).

Using this infrastructure, match processing is supported as a workflow of several steps of matcher combination and mapping refinement. For large schemas, we implemented specific workflows (i.e. strategies) for context-dependent and fragment-based matching. We shortly introduce these match strategies, which will be discussed in detail in Section 5:

- *Context-dependent matching*: Shared schema elements exhibit multiple contexts which should be differentiated for a correct matching. In Fig. 1b, *Address* of S2 is a shared element with two contexts, *DeliverTo.Address* and *BillTo.Address*. In addition to a simple *NoContext* strategy, i.e. no consideration of element contexts, we support two context-sensitive strategies, *AllContext* and *FilteredContext*. *AllContext* identifies and matches all contexts by considering for a shared element all *paths* (sequences of nodes) from the root to the element. Unfortunately, this strategy turns out to be expensive and impractical for

large schemas with many shared elements due to the explosion of the search space. Therefore, we devised the FilteredContext strategy, which performs matching in two steps and restricts context evaluation to the most similar nodes.

- *Fragment-based matching*: In a match task with large schemas, it is likely that large portions of one or both input schemas have no matching counterparts. Hence, we have implemented the fragment-based matching idea proposed in [8]. It decomposes schemas into several smaller fragments and only matches the fragment pairs with a high similarity. In addition to user-selected fragments, we currently support three static fragment types, *Schema*, *Subschema*, *Shared*, considering the complete schema, single subschemas (e.g. message formats in an XML schemas), and shared subgraphs (e.g. *Address* of S2 in Fig. 1b), respectively.

The obtained match result, or mapping, is a set of correspondences, each of which captures a single pair of matching elements and a similarity score indicating the plausibility of the match relationship. Fig. 1b shows in the lower part a simple mapping example with two correspondences where elements are described by their paths. The Mapping Pool maintains all generated mappings and supports various functions to further manipulate them. Such mapping operations can be utilized in match strategies and include *MatchCompose* [6] to transitively combine mappings sharing a same schema, *diff* (determines the difference between the correspondence sets), *intersect*, and *merge* (union). Furthermore, mappings can be compared with each other according to different quality measures (*compare*), e.g. to determine the quality

of a test mapping w.r.t. a user-confirmed real mapping. Using the GUI, one can also edit each mapping to remove false correspondences and to add missing ones (*edit*). The Mapping Pool offers further functions to load and save mappings from/to the repository, as well as to import and export them via an XML/RDF [9] and a CSV format.

3. Schema import and representation

Advanced modeling capabilities, such as user-defined types/classes, reuse of components, distributed schemas and namespaces, as typically supported by modern schema languages, lead to a significant complication for schema matching [8]. In this section we discuss the problems and our solution to import external schemas into our internal graph representation. While we choose W3C XSD for illustration purposes, our approach is also applicable to other schema languages with similar modeling capabilities.

3.1. Conflicts between alternative designs

For flexibility reasons, current schema languages often support alternative approaches to model the same real-world concepts. This leads to different, yet semantically similar structures to be dealt with in schema matching. In particular, we can observe the following common design conflicts:

3.1.1. Schema designs—monolithic vs. distributed schemas

The traditional way to construct a schema is to put all components in a single monolithic schema, which is quite handy for simple applications (see

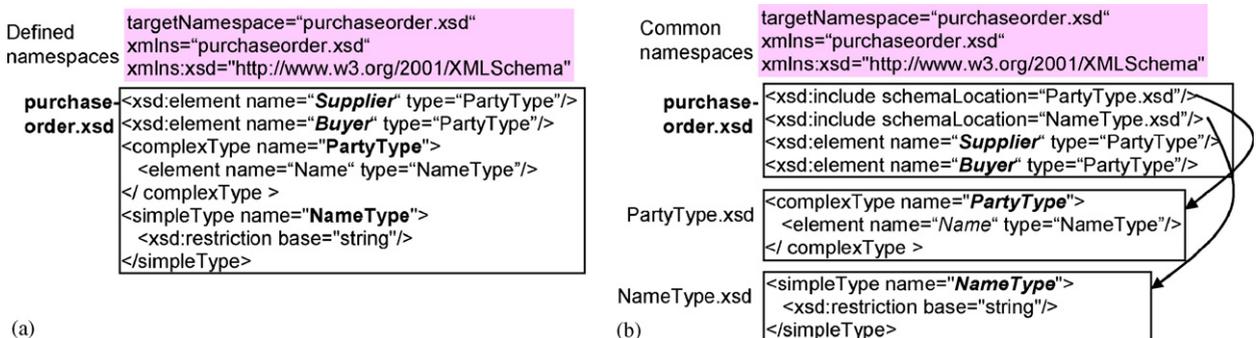


Fig. 2. Monolithic vs. distributed representation of a Schema. (a) Monolithic schema and (b) distributed schema.

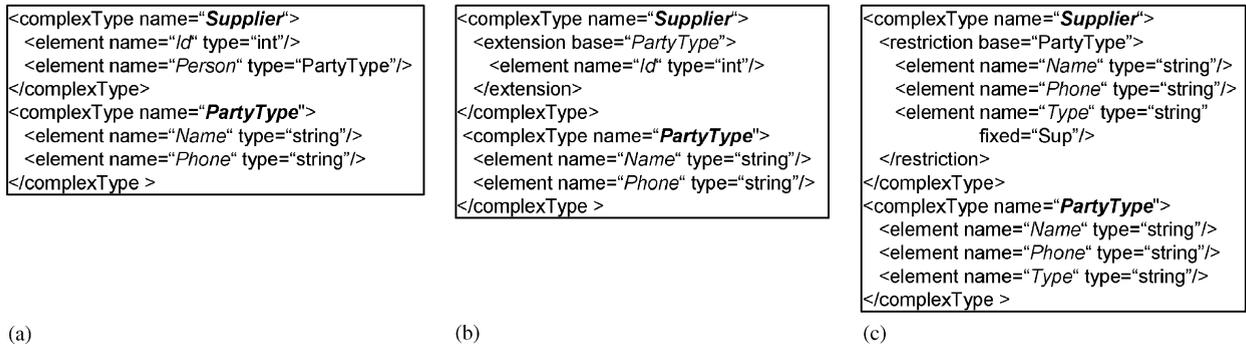


Fig. 3. Structurally different, yet semantically similar type designs. (a) Composition, (b) derivation by extension and (c) derivation by restriction.

Fig. 2a). To better deal with large schemas, XSD, Web OWL, and other XML schema languages [10] allow a schema to be distributed over multiple documents and namespaces, as illustrated in Fig. 2b. In particular, each XSD document can be assigned to a so-called target namespace and XSD provides different directives (*include*, *redefine* and *import*) to incorporate component definitions from one document into another. Orthogonally to the schema designs, a schema may contain multiple *subschemas*, e.g. different message formats, which can be individually instantiated. In both Fig. 2a and b, the *Supplier* and *Buyer* elements are the instantiable roots of *purchaseorder.xsd* and thus represent two different subschemas.

3.1.2. Type designs—composition vs. derivation

Several languages, including XSD and the object-relational SQL extensions SQL:1999 and SQL:2003, support a versatile system of user-defined types for element and attribute declarations. New types can be constructed using either the *composition* or *derivation* approach. Fig. 3 illustrates the type designs supported by XSD. In the composition approach (Fig. 3a), a new type (*Supplier*) is composed of elements/attributes of existing types (*int* and *PartyType*). Using derivation, a new type is derived from a base type and automatically inherits all its components. XSD supports two derivation mechanisms, namely extension and restriction. In Fig. 3b, type *Supplier* extends type *PartyType* and inherits the elements *Name* and *Phone* of *PartyType*. In Fig. 3c, type *Supplier* restricts type *PartyType* and needs to keep all elements *Name*, *Phone*, and *Type* of *PartyType*. Usually, composition and derivation can be recursively applied so that arbitrarily nested type hierarchies are possible.

3.1.3. Reuse designs—inlined vs. shared components

The basic approach of element and attribute declaration is to anonymously specify types inline. It results in tree-like schemas and may be sufficient for smaller schemas with few elements. To avoid redundant or unnecessarily diverse specifications, previously defined components can be reused at different places, resulting in a graph structure. A simple approach is *element reuse* supported by DTD and XSD. In particular, XSD allows global components, i.e. direct children of the $\langle schema \rangle$ element of an XSD document, to be referenced in other type definitions. The more versatile approach is *type reuse*, supported by XSD, SQL:1999, and SQL:2003. Here, the types can be referenced within element or attribute declarations as well as (recursively) within other type definitions. The high flexibility of type reuse makes it a well-suited approach for large business applications. The three alternatives are illustrated in Fig. 4 by means of XSD examples.

The importance of these issues can be illustrated by examining large real-life XSD schemas. Table 1 shows some statistics for two standardized e-Business message schemas, namely *OpenTrans* and *Xcbl OrderManagement* (*XcblOrder* for short),² which are also considered later in our evaluation (see Section 6). These schemas are distributed across many XSD files and have a size of several hundreds to almost 1500 components (XML types, elements and attributes). *Xcbl* follows the type reuse approach and only has few global elements as possible roots for instance documents (*Order*, *ChangeOrder*, *OrderRequest*, *OrderResponse*, etc.). In contrast, *OpenTrans* employs the element reuse approach, resulting in a large number

²OpenTrans: www.opentrans.org, Xcbl: www.xcbl.org.

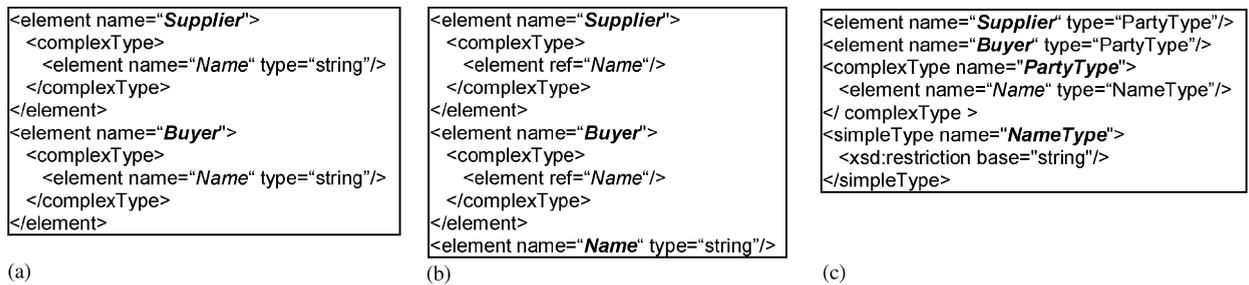


Fig. 4. Examples for different reuse designs. (a) Inlined (no reuse), (b) element reuse and (c) type reuse.

Table 1
Statistics of some e-business XSD standard schemas

Schema	#Files	Size/#components	#Elements (all/global)	#Types (all/global)	#Shared components
OpenTrans	15	614	589/194	25/11	61
XcblOrder	63	1451	1088/8	358/358	91

of global elements, although only few are supposed to represent roots of relevant instance documents. There are a substantial number of shared XML types and elements in both schemas.

3.2. Design unification

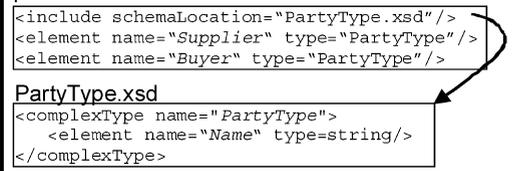
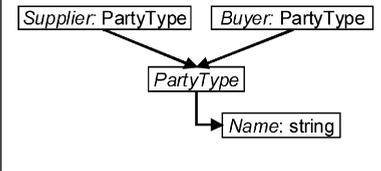
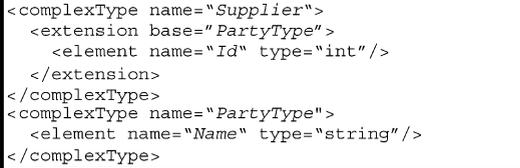
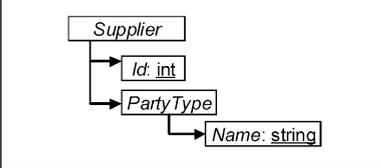
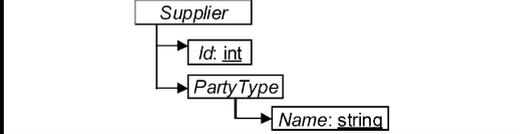
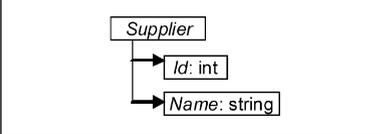
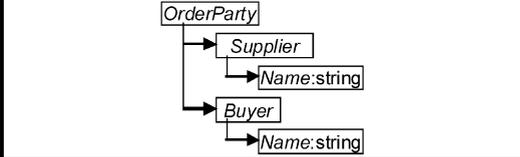
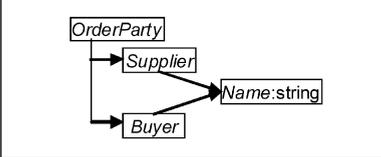
The success of the match operation essentially depends on the detection and unification of the alternative designs discussed above during schema import. Our approach is to decide for a particular design and transform the other designs into it. For this purpose, we first parse all documents of a schema and capture different kinds of component metadata, among others *name*, *namespace*, *type*, *typespace* (the namespace of the type), which help to determine the (type and reuse) designs employed. While each component is uniquely identified by its name and namespace, we assume here that the attributes *name* and *type* are already namespace-qualified for better readability. As illustrated in Table 2, our schema transformation process encompasses four following steps:

1. *Unifying schema designs*: For ease of handling, a distributed schema is transformed to a monolithic one by parsing all schema documents and capturing relevant components and their cross-references. In the example shown in Table 2, Row 1, two XSD documents are parsed and their components are stored together in a single graph. Using name and namespace information, we are

able to identify the type *PartyType* from *PartyType.xsd* and associate it with the elements *Supplier* and *Buyer* as declared in *po.xsd*.

2. *Unifying type designs*. As composition is the common way for type construction, derivation is transformed to composition by propagating the components of the base type to the derived type. As shown in Table 2, Row 2, this is done in the graph representation by linking the base type, *PartyType*, with the derived type, *Supplier*. The components of *PartyType*, e.g. *Name*, thus also become descendants of *Supplier*.
3. *Unifying reuse designs*. We aim at a schema graph of instantiable components, i.e. elements and attributes, as only these components appear in instance documents. Therefore we transform type reuse to element reuse. As shown in Table 2, Row 3, we proceed with the result of Step 2 and eliminate the nodes representing types, such as *PartyType*. The element *Name* in the type *PartyType* now becomes a direct descendant of *Supplier*. At the leaf level we have components of atomic types, such as *string* and *int*.
4. *Reducing inline declarations*. To further compact the schema, we identify and collapse the same components defined inline at multiple places to a single shared one. This is done by a fast search operation for components with the identical metadata, e.g. name and data type. As illustrated in Table 2, Row 4, two different *Name* elements as children of *Supplier* and *Buyer* are transformed to a single shared one.

Table 2
Unification of alternative designs

Transform	XSD Examples / Input Graph	Output Graph (Name: type)
(1) Distributed to monolithic schema	<p>po.xsd</p> <pre><include schemaLocation="PartyType.xsd" /> <element name="Supplier" type="PartyType"/> <element name="Buyer" type="PartyType"/></pre> <p>PartyType.xsd</p> <pre><complexType name="PartyType"> <element name="Name" type="string"/> </complexType></pre> 	
(2) Type derivation to composition	<pre><complexType name="Supplier"> <extension base="PartyType"> <element name="Id" type="int" /> </extension> </complexType> <complexType name="PartyType"> <element name="Name" type="string" /> </complexType></pre> 	
(3) Type reuse to element reuse		
(4) Inlined to reuse declaration		

The result of the import process is a connected graph of instantiable components. By storing shared components only once we are able to keep the complexity of imported schemas at a minimum. For example, for the largest schema of our evaluation, the e-Business standard XcblOrder, the number of nodes and paths is reduced from about 1500 and 40,000 to 800 and 26,000, respectively. (Section 6.1 presents more detailed statistics of the imported schemas, including XcblOrder and OpenTrans.) In general, the import process performs very fast. For XcblOrder, parsing and transforming more than 100 schema documents only takes about 40 s on our test machine. This delay only occurs once when the schema is saved in the repository for later match operations.

4. Matcher combination and matcher library

Starting with a set of *simple* and *hybrid* matchers such as string matching, type matching, dictionary lookups etc., we support constructing more powerful *combined* matchers from existing matchers,

including previously constructed combined matchers. While hybrid matchers combine multiple criteria in a fixed way within a single algorithm, combined matchers are based on a generic combination scheme. COMA++ provides a customizable implementation for such combined matchers, called *CombineMatcher*. In the following, we first present *CombineMatcher* (Section 4.1) and its possible configuration strategies (Section 4.2 and 4.3). In Section 4.4, we describe the simple, hybrid and combined matchers currently available in our system and which were used for the evaluation.

4.1. CombineMatcher

The main idea of *CombineMatcher* is to combine similarity values predicted by multiple matchers to determine correspondences between schema elements. Fig. 5a shows the pseudo-code of *CombineMatcher*, providing two methods, *match* and *sim* (Line 3 and 19, respectively). Given two elements as input, *match* determines the correspondences between their related elements (e.g., ascendants or

```

1 CombineMatcher
2 (oType, defMatchers, agg, dir, sel, combSim)
3
4 match(s1, s2) {
5   //Step 1: Determine elements/constituents to match
6   s1objs = determineRelated(s1, oType)
7   s2objs = determineRelated(s2, oType)
8   //Step 2: Matcher execution
9   allocate simCube[defMatchers][s1objs][s2objs]
10  for each m in defMatchers
11    for each o1 in s1objs
12      for each o2 in s2objs
13        simCube[m][o1][o2] = m.sim(o1, o2)
14  //Step 3,4,5: Similarity combination
15  matchResult = selection(
16    direction(
17      aggregation(simCube, agg), dir), sel)
18 }
19
20 sim(s1, s2) {
21   matchResult = match(s1, s2)
22   //Step 6: Compute combined similarity
23   sim = combinedSim(matchResult, combSim)
24 }

```

(a)

1. *Element identification: DescPaths*

S1 Paths {ShipTo.shipToCity, ShipTo.shipToStreet, ... }

S2 Paths {DeliverTo.Address.City, ... }

2. *Matcher execution: Name, NamePath*

	S1 element	S2 element	Matcher	Sim
Similarity Cube	ShipTo.shipToCity	DeliverTo.Address.City	Name	0.6
	ShipTo.shipToCity	DeliverTo.Address.City	NamePath	0.8
	ShipTo.shipToStreet	DeliverTo.Address.City	Name	0.5
	ShipTo.shipToStreet	DeliverTo.Address.City	NamePath	0.7

3. *Aggregation: Average*

	S1 element	S2 element	Sim
Similarity Matrix	ShipTo.shipToCity	DeliverTo.Address.City	0.7
	ShipTo.shipToStreet	DeliverTo.Address.City	0.6

4. *Direction: SmallLarge* Rank S1 elements for each element of larger schema S2

	S1 element	S2 element	Sim
Ranking	ShipTo.shipToCity	DeliverTo.Address.City	0.7
	ShipTo.shipToStreet	DeliverTo.Address.City	0.6

5. *Selection: Max1*

	S1 element	S2 element	Sim
Match Result	ShipTo.shipToCity	DeliverTo.Address.City	0.7

(b)

Fig. 5. CombineMatcher and processing example. (a) Pseudo-code for CombineMatcher and (b) example for *match*.

descendants) or constituents (e.g., name tokens), while *sim* derives a single value to indicate the similarity between the input elements. As shown in Line 2 of Fig. 5a, a combined matcher needs to be configured with (1) the type of the objects to match (*oType*), (2) a set of default matchers to compute similarity between the identified objects (*defMatchers*), and (3) a combination scheme, (*agg*, *dir*, *sel*, and *combSim*), consisting of strategies for aggregation, direction, selection and combined similarity, to combine matcher-specific similarities.

The *match* method performs five steps, which are illustrated in Fig. 5b matching the sample schemas from Fig. 1b. In the first step, *element/constituent identification*, the objects related with the input elements *s1* and *s2*, respectively, are identified according to the *oType* parameter (Lines 5 and 6 in Fig. 5a). With the object type *DescPaths* as in the example of Fig. 5b, we obtain all paths in S1 and S2 as the elements to match. In the second step, *matcher execution*, the default matchers, e.g. *Name*, *NamePath*, are applied, by calling their *sim* methods (Line 12 in Fig. 5a), to compute the similarity for each element pair, resulting in a similarity cube.

The next three steps, *aggregation*, *direction*, and *selection*, determine the most plausible correspondences from the similarity cube using the given combination strategies *agg*, *dir*, and *sel* (Line 14-16 in Fig. 5a). In particular, for each element pair, the matcher-specific similarities are first aggregated to a

single value, e.g. by taking their average (*Average*). The direction then determines one schema, elements of which are ranked w.r.t. the elements of the other schema according to the aggregated similarities. For example, the *SmallLarge* strategy ranks all elements of the smaller schema S1 for each element of the larger schema S2. Finally, the best elements, e.g., the single best one (*Max1*), are selected from the ranking as match candidates and returned as the match result.

The *sim* method proceeds with the match result returned by *match* and derives a single value to indicate the similarity between the input elements *s1* and *s2*. In particular, it applies the specified strategy *combSim* to the identified correspondences (Line 22 in Fig. 5a). One possible approach is to compute the average of the similarities exhibited by all correspondences. Note that Fig. 5a shows the *sim* method of combined matchers and that simple and hybrid matchers also provide a *sim* method whose result is directly computed and not from the result of another match operation.

When a high number of elements are to be considered, we observe long execution time due to the mutual calls of the *match* and *sim* methods between the combined matchers and their default matchers. Therefore, we have optimized the implementation of CombineMatcher in different ways. First, we extend both methods to be able to compare sets of elements at a time (i.e. set-valued

s1 and *s2*). By determining the related elements/constituents for all relevant elements, we can use the (much smaller) set of unique elements/constituents for similarity computation. Second, we introduced a central cache of similarity values to avoid repeated computation. Note that these are technical optimizations for the implementation of CombineMatcher. The extension of CombineMatcher with the mapping refinement capability represents the major algorithmic improvement, which eventually enable scalable strategies for context-dependent and fragment-based matching (see Section 5).

4.2. Element/constituent identification

This initial step determines the objects to be compared in the *match* method. CombineMatcher can be configured to perform matching at different granularities, in particular for entire schemas (by considering all descendants of the root elements), restricted neighborhoods of elements, and constituents of individual elements to compute element similarity. Fig. 6 illustrates the most common types of related elements and constituents, which are briefly described in the following:

- *Constituents*. Constituents are derived directly from the properties of the input elements, such as name tokens, data type, comment, previously determined structural statistics or instance characteristics, etc.
- *Elements*. We use schema structure to identify related elements of the input elements. Common types of neighbor elements include the children, leaves, parents, siblings, ascendants, and descen-

dants of the input elements. *Self* simply passes the input elements to the default matchers in a combined matcher. Finally, *AscPaths* and *DescPaths* determine the paths from the schema roots to the input elements and the paths from the input elements to all their descendants, respectively.

4.3. Similarity combination

Similarity combination aims at deriving the correspondences or a combined similarity from a similarity cube previously computed by multiple matchers for two sets of elements. For this purpose, we use the same combination scheme with four steps, aggregation, direction, selection, and computing combined similarity, as our previous prototype COMA [6]. While the *match* method involves the first three steps, the *sim* method takes over the result of *match* and performs the last step. Fig. 7 illustrates the steps with their alternative strategies, which are described in the following:

- *Aggregation*: This step aggregates the similarity cube along the matcher dimension to obtain a similarity matrix between the two element sets. Possible strategies are *Max*, *Min*, *Average*, and *Weighted*. *Max* optimistically returns the highest similarity predicted by any matcher, while *Min* pessimistically takes the lowest one. *Average* aims at compensating between the matchers and returns the average of their predicted similarities. It is a special case of *Weighted*, which computes a weighted sum of the similarities given a weighting scheme indicating different importance of the matchers.

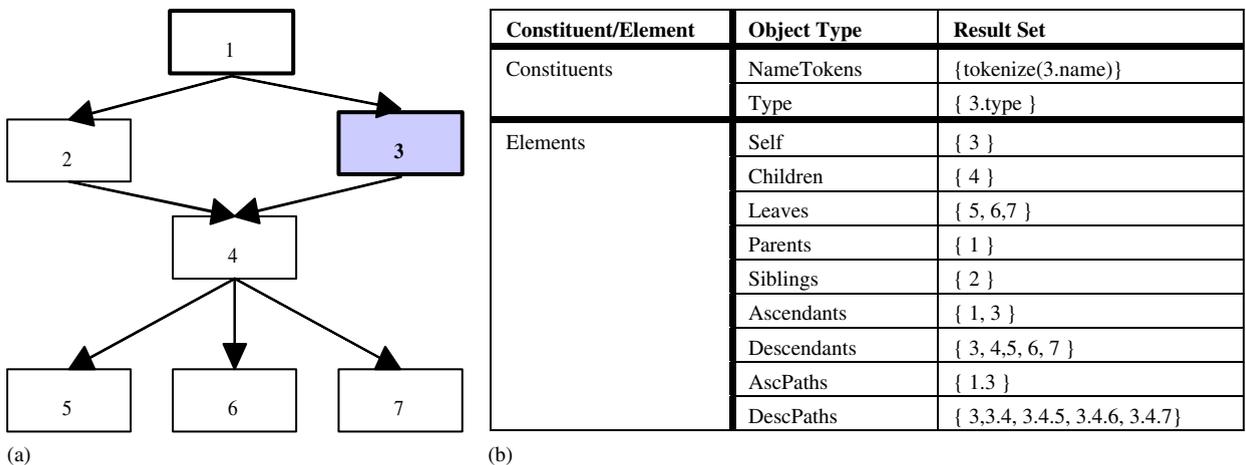


Fig. 6. Examples for related elements and constituents. (a) Sample schema graph and (b) related elements/constituents for node 3.

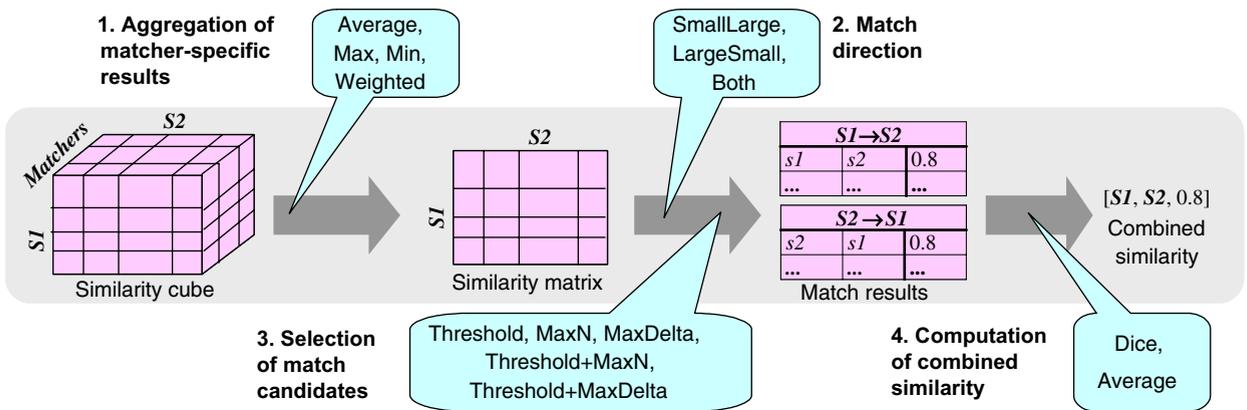


Fig. 7. Scheme for similarity combination.

- **Direction:** This step ranks the schema elements according to their similarity to each other. Two directional strategies are possible, *LargeSmall* and *SmallLarge*, differentiating the size of the input schemas. In *LargeSmall*, the elements of the larger schema are ranked and selected for each element of the smaller schema, while *SmallLarge* proceeds in the converse way. Furthermore, we support the unidirectional strategy *Both*, combining both directions to achieve independence from schema size. In particular, two elements are only accepted as a matching pair if it is identified as such in both directions.
- **Selection:** This step selects the best match candidates from a ranked list. The basic strategies are (a) *Threshold* returning all candidates showing a similarity above a particular threshold, (b) *MaxN* constantly returning the top N candidates, and (c) *MaxDelta* taking the candidates with the highest similarity Max as well as those with similarity within a relative tolerance range specified by a Delta value: $[\text{Max} - \text{Max} * \text{Delta}, \text{Max}]$. For more restrictive selection, multiple criteria can also be considered at the same time, such as *Threshold + MaxN* and *Threshold + MaxDelta*.
- **Combined similarity:** From the correspondences returned by selection, this step derives a combined similarity indicating the similarity between the sets of the related elements/constituents used to compute the similarity cube. We support two strategies, *Average* and *Dice*. The former returns the average of the similarities between the matching elements, while the latter returns the ratio of the matching elements over all elements.

Average is pessimistic and typically returns a smaller value than *Dice*. Only with all element similarities set to 1.0, as in manual match results, both predict the same combined similarity.

4.4. Matcher library

The Matcher Library of COMA++ currently contains more than 15 matchers exploiting different kinds of schema and auxiliary information. We first describe the simple and hybrid matchers, and then the combined matchers.

4.4.1. Simple and hybrid matchers

Table 3 gives an overview of the simple and hybrid matchers and characterizes the exploited schema and auxiliary information. In particular, they include four simple string matchers, *Affix*, *Trigram*, *EditDistance*, and *SoundEx*, three reuse-oriented matchers, *Synonym*, *Type*, and *Reuse*, and a structure matcher, *Statistics*, which are briefly introduced in the following:

- **String matchers:** These matchers are based on common approximate string matching techniques [11,12]. *Affix* looks for common affixes, i.e. both prefixes and suffixes, between two strings. *Trigram*, a special version of *n*-gram, compares strings according to their set of 3-grams, i.e. sequences of three characters. *EditDistance* computes the similarity from the number of edit operations necessary to transform one string to another one (the Levenshtein metric). *Soundex* estimates the phonetic similarity of two given names.

Table 3
Simple and hybrid matchers and their characteristics

Technique	Matcher	Schema info	Auxiliary info
String matching	Affix	Element names	—
	Trigram	Element names	—
	EditDistance	Element names	—
	Soundex	Element names	—
Reuse-oriented matching	Synonym	Element names	Name synonym tables
	Type	Data types	Type compatibility table
	Reuse	—	Previous match results
Structural matching	Statistics	Structural statistics	—

Table 4
Combined matchers and their default configuration

Matcher	Object type	Default matchers	Combination scheme (Agg, Dir, Sel, CombSim)
Name	Name tokens	Synonym, Trigram	Max, Both, Max1, Avg
NameType	Self	Name, Type	Wgt(0.7,0.3), -, -, -
NameStat	Self	Name, Statistics	Wgt(0.7,0.3), -, -, -
NamePath	Ascendants	Name	-, Both, Max1, Avg
Children	Children	NameType	-, Both, Max1, Avg
Leaves	Leaves	NameType	-, Both, Max1, Avg
Parents	Parents	Leaves	-, Both, Max1, Avg
Siblings	Siblings	Leaves	-, Both, Max1, Avg

- *Reuse-oriented matchers*: These matchers utilize auxiliary sources in addition to schema information. Synonym estimates the similarity between element names by looking up the terminological relationships in a *name synonym table*.³ Currently, it simply uses relationship-specific similarity values, e.g., 1.0 for a synonymy and 0.8 for a hypernymy relationship. Following the same lookup approach as Synonym, Type uses a *type synonym table* specifying the degree of compatibility between data types. Reuse exploits previously determined match results [6,13] and performs a join-like operation on a mapping path consisting of two or more mappings, such as $S1 \leftrightarrow S2$, $S2 \leftrightarrow S3$ successively sharing a common schema, to derive a new mapping between schemas S1 and S3.
- *Statistics* uses the Euclidean distance function to compute the similarity between structural statistics, which have previously been determined for

single nodes using a feature vector (capturing the number of children, parents, leaves, etc.).

4.4.2. Combined matchers

Based on the simple and hybrid matchers, eight combined matchers are currently defined, aiming at a more accurate computation of element similarity. They are mainly used as default matchers for match strategies, which compare and match sets of elements between schemas (see Section 5.2). Table 4 gives an overview of the default configuration of the combined matchers. In particular, they exploit different kinds of neighborhoods or constituents of individual elements to compute element similarity. We use the insights from the COMA evaluation [6] to set their default matchers and combination strategies. Max, Both, Max1, and Average are typically used for aggregation, direction, selection, and combined similarity, respectively. The following combined matchers are supported:

- *Name* estimates the similarity between element names. The names are tokenized, e.g. *ShipAddress* \rightarrow {Ship, Address}, and possibly expanded *PO* \rightarrow {Purchase, Order} according to the available abbreviations. The obtained tokens are

³The graphical user interface of COMA++ allows the user to manually specify or modify name synonyms, which are automatically stored in the repository and exploited by the Synonym matcher.

compared using multiple string matchers, such as Trigram and Synonym, and their similarities are combined to a single value indicating the similarity between the names.

- *NameType* and *NameStat* apply two matchers, Name and Type or Name and Statistics, to the input elements (Self). The two similarity values are then aggregated using the Weighted strategy. In both *NameType* and *NameStat*, the name similarity is assigned a larger weight, 0.7, to emphasize its importance over the type and statistics similarity. Further combination steps are not necessary as they do not affect the aggregated similarity.
- *NamePath* aims at distinguishing between different contexts of a shared element, e.g., *ShipTo.Street* and *BillTo.Street*. Given two paths, it uses Name to compute name similarities between the nodes on the paths and derives a combined value to indicate the path similarity. As only one default matcher is involved, aggregation is not necessary. The obtained similarity matrix is combined using the remaining steps of the combination scheme to derive a combined similarity.
- *Children* and *Leaves* follow a bottom-up match approach and derive the similarity between elements from the similarities between their children, i.e. direct descendants, and the leaves subsumed by them, respectively. Compared to Children, Leaves can better deal with structural conflicts, when the similar elements are modeled

at different level of detail, e.g. *PO.Contact* and *PO.Header.Contact*. Both matchers employ *NameType* to compute the similarity between the leaves.

- *Parents* and *Siblings* derive the similarity between elements from the similarities between their parents and siblings, respectively. Unlike Children and Leaves, they allow the similarity to be propagated from the ascendants (top-down matching) and from the neighbors of the same level. Both matchers employ Leaves to compute the similarity between the corresponding related elements.

5. Mapping refinement and match strategies

Besides the combination of individual matcher results, the ability to refine a previously identified match result is a further prerequisite for building workflows of match processing. We thus extend *CombineMatcher* in such a way that it can utilize a combination of matchers to derive a new mapping from a preliminary one. Based on this infrastructure, we are able to define scalable match strategies comprising multiple steps of successive mapping combination and refinement. In the following, we first describe our approach to mapping refinement. We then discuss the strategies for context-dependent and fragment-based matching, both aiming at an improved support of large schemas.

```

1 CombineMatcher(oType, defMatchers, ...)
2 match(s1, s2) { ... }
3 sim(s1, s2) { ... }
4 refine(prevMapping) {
5   //Allocate match result
6   newMapping = {}
7   //Partition and group related correspondences
8   { gc1, ..., gcK } = groupCorrespondences(prevMapping)
9   //Iterate over all grouped correspondences
10  for each gc in { gc1, ..., gcK } {
11    s = domain(gc)
12    t = range(gc)
13    map = match(s, t)
14    newMapping = merge(newMapping, map)
15  }
16  return newMapping
17 }

```

(a)

```

1 SimpleStrategy(PreMatching, RefMatching)
2 match(s1, s2) {
3   //Generate preliminary result
4   preMapping = PreMatching.match(s1, s2)
5   //Refine preliminary result
6   refMapping = RefMatching.refine(preMapping)
7   return refMapping
8 }
9 sim(s1, s2) { ... } //As in CombineMatcher
10 refine(mapping) { ... } //As in CombineMatcher

```

(b)

Fig. 8. The extended *CombineMatcher* and example of a simple match strategy. (a) *CombineMatcher* extended with *refine* and (b) *SimpleStrategy*.

5.1. Refinement of match result

Refinement focuses on matching between elements, which were previously identified as potential match candidates. Such elements may be selected by the user or determined automatically by a previous match step. Fig. 8a shows the pseudo-code of the extended CombineMatcher, i.e. it uses the same parameters and supports the same *match* and *sim* methods. It offers a new method, *refine* (Line 4), which takes as input an existing mapping, *prevMapping*, and produces a new mapping, *newMapping*, using the given configuration, i.e. the type of related elements, the default matchers, etc.

The *refine* operation does not match two complete input schemas but only considers the elements represented in the input mapping. To further improve performance we preprocess the 1:1 correspondences in the input mapping. In particular, to avoid repeatedly processing elements involved in multiple correspondences, we first group the original correspondences in *prevMapping* into a set of distinct grouped correspondences *gc1*, ..., *gck* (Line 8). Each grouped correspondence starts from an 1:1 correspondence $s1 \leftrightarrow s2$ and includes all further correspondences for either *s1* or *s2* from *prevMapping*, so that n:m relationships are represented. For instance, $Address \leftrightarrow ShipAddr$ and $Address \leftrightarrow BillAddr$ would be grouped together.

Each such grouped correspondence is individually refined. *refine* first determines the unique source and target elements of the grouped correspondence, *s* and *t*, using the *domain* and *range* operation, respectively (Lines 11 and 12). For the example above, *s* only contains *Address*, while *t* is the set $\{ShipAddr, BillAddr\}$. The *match* method is then applied to match *s* and *t* (Line 13). The result, *map*, contains the correspondences between the related elements of *s* and *t* as specified by the configuration parameter *oType*. They are merged to *newMapping* using the *merge* operation (Line 14). Both *s* and *t* are unique over all grouped correspondences, so that *map* usually adds new correspondences to *newMapping*. However, *merge* is also able to detect and ignore duplicate correspondences, i.e., between the same source and target elements. After all grouped correspondences are processed, *newMapping* is returned as the final result. Note that the

employed functions *domain*, *range*, *merge*, as well as *groupCorrespondences*, are general mapping operators implemented in the Mapping Pool.

Fig. 8b illustrates the use of refinement in a simple match strategy, *SimpleStrategy*. Like *CombineMatcher*, a match strategy supports the same interface with three methods, *match*, *sim*, and *refine*. While *sim* and *refine* are the same as in *CombineMatcher*, the *match* method executes the intended workflow with each step supported by a previously defined *CombineMatcher* instance or match strategy. In *SimpleStrategy*, two steps are performed, the first one to generate a preliminary mapping using a given match strategy, *PreMatching* (Line 4), and the second one to refine it, using another match strategy, *RefMatching* (Line 6). With the same interface methods, *CombineMatcher* instances and match strategies can be used interchangeably, making it easy to construct new match strategies from existing ones.

5.2. Context-dependent match strategies

In addition to the *NoContext* strategy, which yields context-independent correspondences, we support two match strategies to obtain correspondences between element contexts, namely *AllContext* and the new *FilteredContext* strategy. In the following, we describe the single strategies and then briefly compare their complexity.

5.2.1. NoContext

Previous match approaches typically assume tree-like schemas without shared elements. In such cases, node-level matching, which we denote as the *NoContext* strategy, is sufficient due to the unique context of all elements. However, in schemas with shared elements, this approach typically returns many false matches. For example, a match between the nodes *shipToStreet* of S1 and *Street* of S2 in Fig. 1b would indicate that the single context of *shipToStreet*, *ShipTo.shipToStreet*, matches both contexts of *Street*, *DeliverTo.Address.Street* and *BillTo.Address.Street*. However, only the former represents the correct match candidate.

In our system, *NoContext* can be implemented as a *CombineMatcher* instance as follows:

```
NoContext = new CombineMatcher(Descendants, {Name, Leaves}, ...)
matchResult = NoContext.match(S1.roots, S2.roots)
```

In particular, it is configured with the object type *Descendants*, which determines all nodes subsumed by the input nodes as elements for matching (see Section 4.2). One or several combined matchers, e.g. Name and Leaves, can be specified as the default matchers for computing node similarity. To match two schemas, NoContext is applied by calling its *match* method with the schema roots as input. Note that all roots are considered as a schema may contain multiple Subschemas, each resulting in a different root in the schema graph.

5.2.2. AllContext

Currently, only few systems, such as COMA [6], Cupid [14], Protoplasm [15], and [16], address the problem of context-dependent matching. Although based on different implementations, they resolve all shared elements and match between all their unique contexts, which we denote as the *AllContext* strategy.

```

NodeMatching = new CombineMatcher(Descendants, {Name, Leaves}, ...)
PathMatching = new CombineMatcher(AscPaths, {NamePath}, ...)
FilteredContext = new SimpleStrategy(NodeMatching, PathMatching)
matchResult = FilteredContext.match(S1.roots, S2.roots)
                = PathMatching.refine(
                    NodeMatching.match(S1.roots, S2.roots)
                )

```

COMA captures all contexts by differentiating all paths from the schema root to a shared element. On the other hand, Cupid, Protoplasm, and [16] maintain multiple copies of shared elements in a (voluminous) tree-like schema representation and determine node correspondences. Both implementations consider the same number of components (paths or copied nodes), which may be extremely high in large schemas with many shared elements as encountered in our evaluation (see Section 6.1).

To keep the input schemas unchanged, we support the path-based implementation of AllContext, which can be specified as a CombineMatcher instance as follows:

```

AllContext = new CombineMatcher(DescPaths, {NamePath, Leaves}, ...)
matchResult = AllContext.match(S1.roots, S2.roots)

```

In particular, AllContext is configured with the object type *DescPaths*, which returns the paths from the input nodes to all their descendants (see Section 4.2). To accurately compute the similarity between paths, we use a combination of NamePath with

other combined matchers as default matchers. Like other match strategies, AllContext is utilized to match two schemas by invoking its *match* method with the schema roots as input.

5.2.3. FilteredContext

Intuitively, if two nodes do not match, they will unlikely have corresponding contexts. Furthermore, in schemas with many shared elements, the number of paths is typically much higher than that of nodes as each shared node results in multiple paths. Based on these observations, we have developed the *FilteredContext* strategy aiming at a more efficient approach for context-dependent matching. It performs two matching steps, node matching to filter the similar nodes between two schemas and path matching to match the paths, i.e. contexts, of those similar nodes. The pseudo-code of FilteredContext is as follows:

Both steps, *NodeMatching* and *PathMatching*, can be configured individually. NodeMatching utilizes the object type *Descendants* to identify the nodes for matching, while PathMatching determines all paths from the schema root to a given node using *AscPaths*. Each step is also configured with different default matchers. Currently, we use NamePath in PathMatching due to its context-sensitiveness, while NodeMatching involves other combined matchers. Both steps are combined within a SimpleStrategy instance, which applies the *match* method with the schema roots as input to match two schemas.

In the example of Fig. 9 using the schemas from Fig. 1b, NodeMatching predicts among others the

correspondence between nodes *shipToStreet* and *Street*, which is unique and refined separately from other correspondences. In PathMatching, the single path of *shipToStreet* in S1 needs only to be compared with the two paths of *Street*, but not

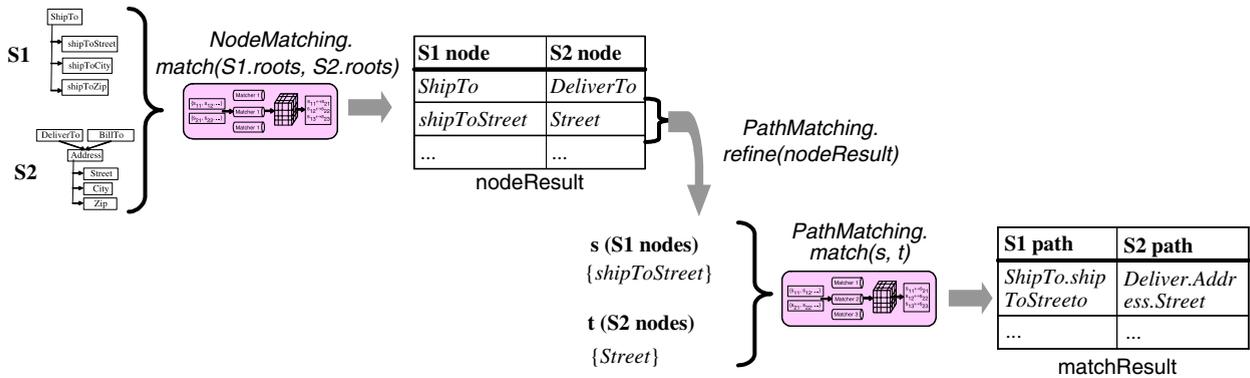


Fig. 9. Combining node and path matching in FilteredContext

with all paths of S2 as performed by AllContext. We obtain the correct correspondence between *ShipTo.shipToStreet* and *Deliver.Address.Street*, which is added to the final result. As we can also see in the example, it is not meaningful to compare the paths of dissimilar nodes, such as *shipToStreet* and *City*.

5.2.4. Complexity

The complexity of NoContext and AllContext is determined by the number of nodes and paths, respectively, in the input schemas. Although depending on the obtained node result, the path-matching step of FilteredContext typically causes only minor additional effort due to the restricted search space of single groups of similar nodes. Hence, the complexity of FilteredContext is largely comparable to that of NoContext. If the number of paths is much higher than the number of nodes due to shared elements, FilteredContext can achieve a significant reduction of complexity compared to AllContext (see Section 6.3). In schemas without shared elements, all three strategies yield the same complexity.

5.3. Fragment-based match strategies

To effectively deal with large schemas, we have implemented the fragment-based matching idea proposed in [8]. Following the divide-and-conquer philosophy, we decompose a large match problem into smaller sub-problems by matching at the level of schema fragments. With the reduced problem size, we aim not only at better execution time but also at better match quality compared to schema-level matching. Furthermore, it is easier to present the match result to a human engineer in a way that she can easily validate and correct it.

5.3.1. Fragment identification

By fragment we denote a rooted sub-graph down to the leaf level in the schema graph. In general, fragments should have little or no overlap to avoid repeated similarity computations and overlapping match results. Besides user-selected fragments for interactive use, we currently support the three strategies for automatic fragment identification, Schema, Subschema, and Shared, which are shortly motivated in the following (Note that new fragment types can be added like the object types for element/constituent identification discussed in Section 4.2.):

- **Schema:** The complete schema is considered as one fragment. Matching complete schemas is thus supported as a special case of fragment-based matching.
- **Subschema:** Subschemas represent parts of a schema which can be separately instantiated, such as XML message formats or relational table definitions. Match results for such fragments are thus often needed, e.g. for transforming messages. Each subschema is identified by a schema root.
- **Shared:** Each shared fragment is identified by a node with multiple parents. Due to the similar usage, such fragments exhibit high potential for match candidates. Furthermore, their match results may be reused many times thereby improving performance.

5.3.2. The fragment-based match approach

Based on the same refinement idea as FilteredContext, our fragment-based match approach also encompasses two steps as illustrated in Fig. 10:

1. **Find similar fragments.** The goal of this step is to identify fragments of the two schemas that are

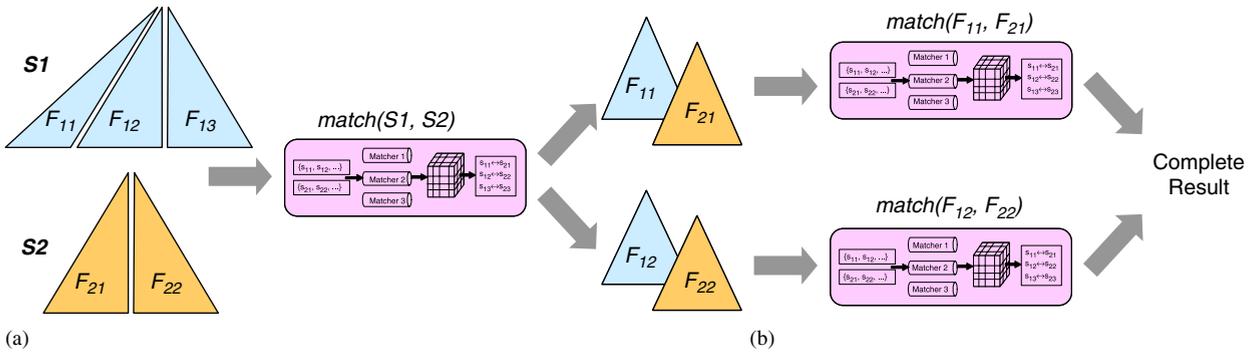


Fig. 10. Fragment-based match approach. (a) Find similar fragments and (b) match similar fragments.

```

//Definition of FindSimFragments and MatchSimFragments
FindSimFragments = new CombineMatcher(Subschema, {Name, Leaves}, ...)
MatchSimFragments = AllContext

//Usage
FragmentMatch = new SimpleStrategy(FindSimFragments, MatchSimFragments)
matchResult = FragmentMatch.match(S1.roots, S2.roots)
               = MatchSimFragments.refine(
                   FindSimFragments.match(S1.roots, S2.roots)
               )

```

Fig. 11. Fragment-based match approach using Subschema and AllContext.

sufficiently similar to be worth matching in detail. This aims at reducing match overhead by not trying to find correspondences for a fragment in irrelevant fragments from the second schema. According to the specified fragment type, the corresponding fragments are identified from the input schemas. In the example of Fig. 10, we obtain the fragment sets $\{F_{11}, F_{12}, F_{13}\}$ and $\{F_{21}, F_{22}\}$, for S1 and S2, respectively. As a fragment is uniquely identified by its root, similarity between fragments can be determined by comparing their roots and/or contexts, i.e. the paths from the Schema roots to the fragment roots.

2. *Match similar fragments.* This step performs refinement of the result from the first step. In particular, the similar fragments identified are fully matched to obtain the correspondences between their elements. Each group of the similar fragments represents an individual match problem, which is solved independently. For example, F_{11} and F_{12} need only to be matched against their similar fragments F_{21} and F_{22} , respectively, thereby reducing match complexity. The match results for single groups of similar fragments are

then merged to a single mapping, which is returned as the final result.

Each step can be configured individually, e.g. using an existing CombineMatcher instance or match strategy. As illustrated in Fig. 8.5, *FindSimFragments* implements the first step and compares the fragment roots as specified by the Subschema fragment type. *MatchSimFragments* implements the second step and simply applies AllContext to match the paths of the similar fragments. Both steps are combined within a SimpleStrategy instance, *FragmentMatch*, which represents the fragment-based match strategy. Like other match strategies, this can be utilized to match two schemas by calling the *match* method with the schema roots as input (Fig. 11).

5.3.3. Complexity

The complexity of the fragment-based match approach consists of the complexity required by each phase. For Phase 1, it is defined by the number of fragment roots or fragment contexts depending on the employed match strategy, i.e., NoContext,

Table 5
Characteristics of the test Schemas

No.	Schema	Type	Nodes/paths	Root/inner/leaf/shared nodes	Max/avg path Length
1	CIDX	XDR	27/34	1/7/20/7	4/2.9
2	Excel	XDR	32/48	1/9/23/11	4/3.5
3	Noris	XDR	46/65	1/8/38/18	4/3.2
4	Paragon	XDR	59/77	1/11/48/13	6/3.6
5	Apertum	XDR	74/136	1/22/52/24	5/3.6
6	OpenTrans	XSD	195/2,500	8/85/110/129	11/7.0
7	XcblOrder	XSD	843/26,228	10/382/461/702	18/8.8

Table 6
Statistics of the test series

Series	Source/target schemas	Tasks	Avg source nodes/paths	Avg target nodes/paths	Avg corresp
Small	PO-PO	10	36/49	60/95	48
Medium	PO-OP	5	48/72	195/2,500	55
Large	OP-XC	1	195/2,500	843/26,228	331

FilteredContext, or AllContext. Likewise, the complexity of Phase 2 is defined by the number of the elements, i.e., nodes or paths, in the corresponding fragments. There is a trade-off between the two phases. Large fragments, such as of type Schema and Subschema, lead to long execution times of Phase 2. However, Phase 1 can perform fast comparing only few fragments. On the other side, small fragments, such as Shared, lead to a fast execution of Phase 2. However, the high number of fragments may lead to a longer execution time in Phase 1 than in Phase 2.

6. Real-world evaluation

We performed a comprehensive evaluation of the match strategies on several real-world schemas, including large e-business standard schemas. The main goal was to compare their effectiveness and time performance, and to investigate the impact of different parameters, such as fragment types, matchers, combination strategies, and schema size. In the following, we first describe the test schemas and experiment design and then discuss our findings.

6.1. Test schemas and series

Table 5 shows the characteristics of the test schemas. We use five XDR schemas for purchase order (PO) taken from the COMA evaluation (1–5) and two new e-business standards OpenTrans (6)

and XcblOrder (7) written in XSD. Unlike the PO schemas, the e-business standards contain several subschemas, indicated by the number of the schema roots. They also exhibit a more deeply nested structure, resulting in longer paths. Furthermore, the high ratio of shared elements leads to a high number of paths to be considered.

In order to examine the impact of schema size, we organize the schemas in three series with different complexity, Small, Medium, and Large, as shown in Table 6. The average problem size for each series is indicated by the average number of the source and target nodes and paths. In the Small series we match the PO schemas against each other, resulting in 10 tasks, which are the same tasks in the COMA evaluation [6]. The Medium series matches the PO schemas against OpenTrans, respectively, resulting in 5 tasks. In the Large series, we match the two large schemas OpenTrans and XcblOrder. For all match tasks, we manually derived the real (context-dependent) match results to use as the golden standard.⁴ In the Large series, for example, only 331 1:1-correspondences are required from a huge search space of $2500 \times 26,000$ paths.

To illustrate the hardness of matching large schemas with shared elements, we now take a closer look at the largest match task OpenTrans-XcblOrder. Fig. 12a and b show the distribution of nodes

⁴The real match results for the Small series were taken from the COMA evaluation [6].

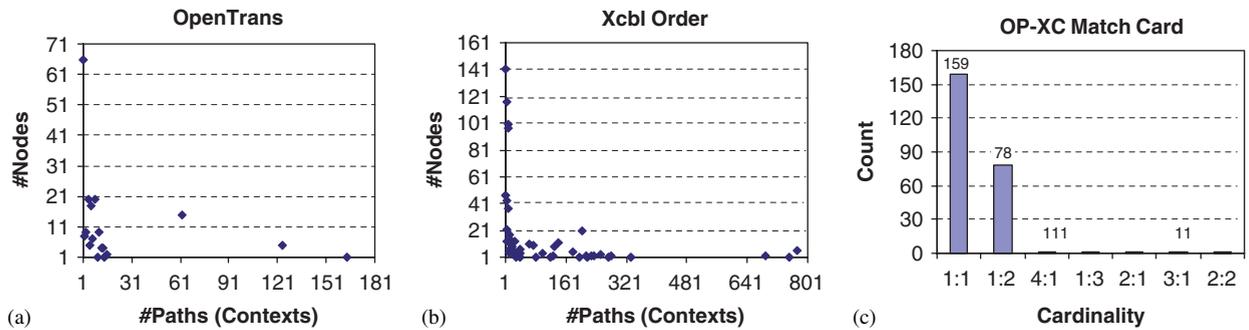


Fig. 12. Distribution of shared elements and match cardinalities for OpenTrans and XcblOrder.

Table 7
Test configuration

Series	Strategy	Fragment	Matcher combination	Combination
Small	NoC	Schema	1 single NamePath	Aggregation: Average
Medium	AllC	Subschema	127 combinations involving	Direction: Both
Large	FiltC	Shared	NamePath	Selection: Thr(0.5) + Delta(0.001-0.01)
Sum: 3	3	3	128	Combined sim: Average
				6

w.r.t. the number of their paths in the two schemas. The number of unique nodes (i.e., having only one path) is 66 in OpenTrans and 141 in XcblOrder. Comparing with Table 5, we can see that most nodes in both schemas yield multiple paths, i.e., contexts. Especially, there are several “generic” elements (like *Type* in OpenTrans and *Ident* in XcblOrder), which are widely shared and thus yield very high numbers of paths (around 160 in OpenTrans and 700 in XcblOrder). Fig. 12c shows the global cardinalities of the real match result between the two schemas. Only about half of the correspondences (159 from 331) also represent unique 1:1 matches. The remaining encode m:n matches, especially 1:2. The approach to select a constant number of match candidates (MaxN), as often employed by previous work, would lead to missing or false matches.

6.2. Experiment design

6.2.1. Test parameters

Due to the flexibility for constructing/configuring match strategies and matchers, an exhaustive evaluation to investigate the effectiveness of all parameters is virtually impossible. Hence, we refer to the insights from our evaluation of COMA and focus on the following parameters

likely to yield the best quality, which are also summarized in Table 7:

- We apply the match strategies AllContext and FilteredContext, and NoContext, to the fragment types Schema, Subschema and Shared, respectively. For short, we denote them hereafter as AllC + Schema, FiltC + Schema, NoC + Schema, etc. Each is tested with the same matcher combinations and combination strategies as follows.
- As already pointed out in [6], considering hierarchical names as implemented in NamePath is a prerequisite to achieve high quality in context-dependent matching. Therefore, we only focus on matcher combinations involving NamePath in this evaluation. In particular, with 7 other combined matchers, we obtain 127 combinations, such as NamePath + Leaves, NamePath + Children, etc., in addition to the single NamePath matcher.
- We use the best combination strategies identified in the COMA evaluation as presets for our tests, in particular, Average for aggregation, Both for direction, and Average for computing combined similarity. For selection, we use the combination of Threshold and MaxDelta for selection. In particular, MaxDelta is able to vary the number of match candidates for each element according to their similarities, and thus, can best deal with

different match cardinalities. While fixing the Threshold value to 0.5, we vary the relative tolerance range Delta between 6 data points 0.001, 0.002, 0.004, 0.006, 0.008, and 0.01.

Semantic name matching is supported by a domain dictionary containing 40 synonyms, such as {*Ship, Deliver*}, and 22 abbreviations, such as {*PO, Purchase Order*}, which are simply taken from the COMA evaluation [6]. All tests and time measurements were uniformly done using Sun Java 1.4.2 libraries on a Linux machine equipped with a 2.4GHz Intel Xeon processor and 1GB RAM.

6.2.2. Quality quantification

To determine the quality of an automatic match operation, we compare its result (*test result*) against the manually derived result (*real result*).⁵ In addition to the set of true positives B, i.e. correct correspondences automatically predicted, we also identify the set of false negatives A and of false positives C. False negatives are correspondences needed but not identified, while false positives are correspondences falsely proposed by the automatic match operation. Based on the cardinality of these sets, we determine three quality measure Precision, Recall, *F*-Measures, which are commonly used in Information Retrieval [61]. They are defined as follows:

- *Precision* = $\frac{|B|}{|B|+|C|}$ reflects the share of real correspondences among all found ones
- *Recall* = $\frac{|B|}{|A|+|B|}$ specifies the share of real correspondences that is found
- *F-Measure* = $\frac{2*|B|}{(|A|+|B|)+(|B|+|C|)} = 2 * \frac{Precision*Recall}{Precision+Recall}$ is the harmonic mean of Precision and Recall.

All measures take the maximum 1.0 when the test result is equal to the real result. The minimum of Precision, Recall, and Fmeasure is 0.0, when the test result is completely different than the real result. Neither Precision nor Recall alone can accurately assess the match quality. In particular, Recall can easily be maximized at the expense of a poor Precision by returning as many correspondences as possible, e.g., the cross product of two input schemas. Similarly, a high Precision can be achieved at the expense of a poor Recall by returning only few (correct) corre-

spondences. Therefore, we use the combined *F*-measure to rank and compare match quality. All measures were first determined for single match tasks and then averaged over all tasks in a series. Hereafter, when talking about match quality for a series, we refer to the average values of the measures over all match tasks in the series, i.e., average Precision, average Recall, and average *F*-measure.

6.3. Performance of context-dependent match strategies

We first investigate the quality of the context-dependent match strategies across the different test series. Fig. 13a shows the best average quality of NoContext,⁶ AllContext, and FilteredContext for matching complete schemas, i.e., using fragment type Schema. In general, match quality decreases with growing schema size. Among the strategies, NoContext shows the worst quality in all series. Although it achieves high average Recall (0.7–0.8) in all test series, many false matches are returned, leading to very low average Precision (~0.0 in the Large series). This effect largely depends on the degree of element reuse in the input schemas. From the Small series (with the lowest ratio of shared elements) to the Medium and Large series (with the highest ratio of shared elements), the average *F*-measure decreases from 0.6 to 0.0. NoContext is thus only feasible for schemas without shared elements.

On the other side, AllContext and FilteredContext are very competitive with each other, both showing high quality in all series. Although performing slightly worse than AllContext in the Small series, FilteredContext achieves the same average quality as AllContext in the Medium and Large series. Overall, the best average Fmeasure achieved is 0.89, 0.68, and 0.66 for the Small, Medium and Large series, respectively, which are quite promising especially if considering the high complexity of our match tasks. In the previous evaluation with the match tasks of the Small series [6], COMA yielded a best average *F*-measure of 0.85, which is now improved by 4% by COMA + +. This is possible due to the new matchers NameStat, Parents, and Siblings in COMA + +.

⁵Recall that the correspondences in our match results capture single pairs of matching elements, i.e. 1:1 cardinality. A 1:n/n:1 match will be represented and counted as *n* correspondences.

⁶To evaluate the result of NoContext, node correspondences are transformed to path correspondences by specifying any path pairs of two matching nodes as a correspondence. This also truthfully represents the context-independent nature of NoContext.

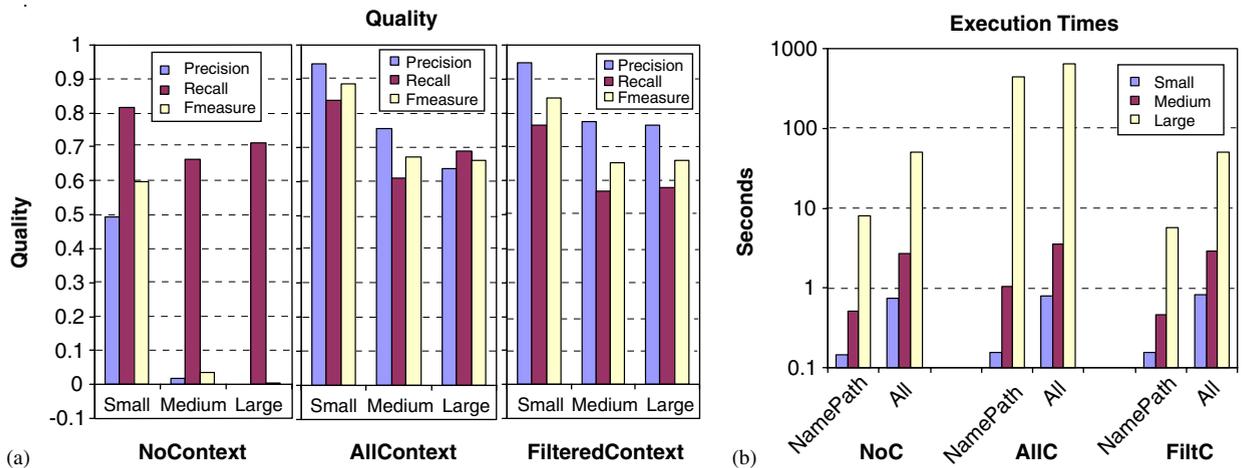


Fig. 13. Quality and execution times of context-dependent match strategies.

Besides the result quality, execution time is another critical factor for the practicability of a match strategy. We thus estimate the time range required by each strategy by testing it with two extreme matcher combinations, the least expensive approach only using the single matcher NamePath,⁷ and the most expensive approach combining all eight combined matchers, hereafter All. Fig. 13b shows the average execution times of the match strategies in the single series. In general, execution time increases with schema size and the number of the involved matchers. In the Small and Medium series, all strategies perform similarly. However, they show significantly different execution times for the Large series. In particular, AllContext with NamePath and All requires around 7 and 10 min, respectively, while NoContext and FilteredContext require even with All only about 50 s. Furthermore, we observe in general similar execution times between NoContext and FilteredContext, confirming our observation on the complexity of FilteredContext (see Section 5.2). Considering AllContext as the match approach of COMA, we can see that COMA++ also outperforms COMA in terms of execution times with the new FilteredContext strategy.

6.4. Performance of fragment-based match strategies

We now examine the performance of the context-dependent match strategies applied to different

fragment types. Due to the low quality of NoContext, we do not consider it further but only report the results for the remaining strategies, AllContext and FilteredContext. Fig. 14a shows the best average *F*-measure of the corresponding match strategies as observed in the single series. In the Small series, matching complete schemas using AIC+Schema and FiltC+Schema yields the same quality as matching subschemas using AIC+Subschema and FiltC+Subschema, respectively, as the PO schemas have only one subschema. However, in the Medium and Large series, using Subschema in both AllContext and FilteredContext yields slightly better quality than using Schema due to the reduced search space. Using Shared fragments, as in AIC+Shared and FiltC+Shared, generally performs worse than other fragment strategies due to incomplete schema coverage. However, it is still promising for schemas with a high ratio of shared elements, like OpenTrans and XcbOrder.

Fig. 14b shows the execution times of the different match strategies for the Large series. AIC+Shared does not improve execution time over AIC+Schema despite reduced fragment size. This is because of the high number of shared elements, i.e. fragments, which requires to match almost the complete schemas in order to identify similar fragments and their matching contexts. Only combined with Subschema, AllContext is feasible for large schemas. In particular, AIC+Subschema using NamePath and All requires only around 40 and 100 s, respectively, for the OpenTrans-XcbOrder match task. On the other side, FilteredContext

⁷In NoContext, NamePath is applied to match single nodes and returns the same result as Name.

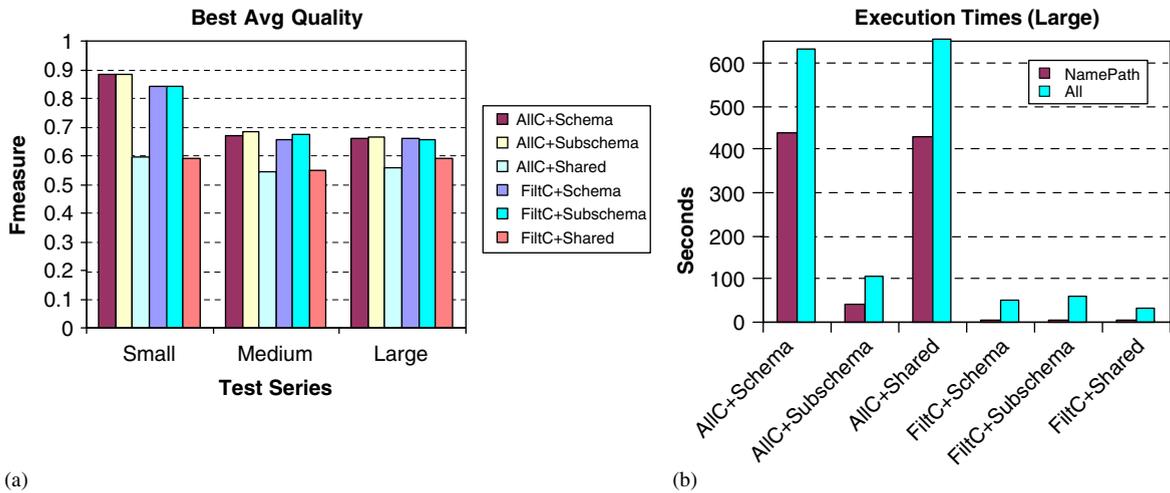


Fig. 14. Quality and execution times of fragment-based match strategies.

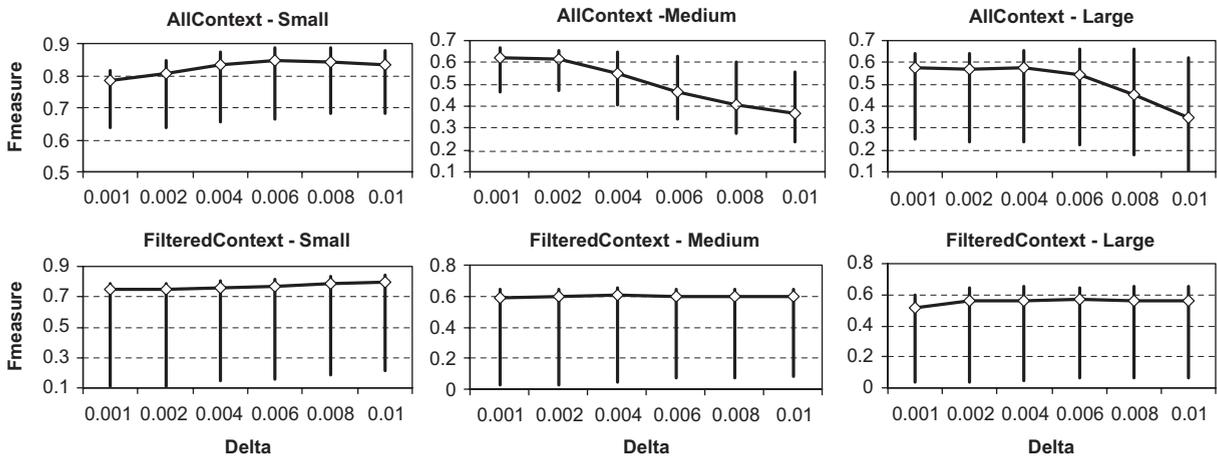


Fig. 15. Quality variation of AllContext (top) and FilteredContext (bottom).

generally shows very good time performance and, thus, represents the better strategy for large schemas. In particular, the three fragment types yield similar execution times of less than 5 s using NamePath and around 40 s using All for the same match task in the Large series.

6.5. Impact of matcher combination and Delta

To investigate the impact of the employed matcher combination and Delta value, we determine the value range, i.e. the min, max and average, of average *F*-measure achieved by all 128 matcher combinations for each Delta value in the single series. Due to the similar behavior across the tested

fragment types, we only show the result for AllContext and FilteredContext matching complete schemas, i.e. with fragment type Schema. In Fig. 15, each diagram illustrates one match strategy in one series with the *x*-axis representing the Delta values and the *y*-axis the *F*-measure values. The vertical line at a Delta value represents the value range of average *F*-measure achieved by all matcher combinations using the corresponding Delta value, while the curve in a diagram connects the average values across the different Delta values.

We first take a look at AllContext (top row in Fig. 15). In the Small series, the quality variation at each Delta value is relatively small, indicating the robustness against the choice of matchers. The

average values are close to the maximum, indicating the high quality of most matcher combinations. The best quality is achieved with Delta between 0.006 or 0.008. With increasing schema size, the variation range at each Delta value increases significantly and the quality degrades with increasing Delta. For the Medium and Large series, the best quality is only to be achieved with the smallest Delta, 0.001. This is to be explained with fact that we have to consider a very high number of element pairs, i.e., candidate correspondences, in the same similarity space [0, 1]. At the smallest Delta, the average value of *F*-measure is also closest to the maximum, indicating that most matcher combinations still perform well for large schemas with this restrictive Delta value.

As for FilteredContext (bottom row in Fig. 15), we notice a large variation range of average *F*-measure in all series. However, the average values are mostly close to the maximum, indicating that most matcher combinations achieve good quality and only few are outliers with bad quality. Furthermore, FilteredContext is more robust against the choice of Delta than AllContext, thereby limiting tuning effort. In particular, the average values of *F*-measure remain in all series largely constant despite the increase of Delta.

6.6. Choice of matcher combination

Given a library of individual matchers, one important question is the choice of a matcher combination for a given match task. We approach this first by analyzing the statistics of the best matcher combinations. We rank all matcher combinations (128) in the ascending order according to their best average *F*-measure. Focusing on the 10 best matcher combinations, we determine the

average number of involved matchers and the occurrence of single matchers, i.e. the number of matcher combinations involving a particular matcher. As NamePath is involved in all matcher combinations, we leave it out from this analysis.

Fig. 16 shows the results for AllContext and FilteredContext applied on complete schemas. The observations are largely similar for other fragment types, which are thus omitted here. We observe in Fig. 16a that with increasing schema size, less matchers should be combined. AllContext is more sensitive than FilteredContext in this aspect. In particular, the 10 best matcher combinations of AllContext involve around 4–5 matchers in the Small series, and around three matchers in the Medium and Large series. On the other side, those of FilteredContext generally involve around 4 matchers in all series.

Fig. 16b and c show the occurrence of the matchers in single series and their average occurrence over all series for AllContext and FilteredContext, respectively. Regarding AllContext, we observe a strong fluctuation of matcher occurrence w.r.t schema size. In particular, only Leaves show a constantly high occurrence in all series (involved in at least five matcher combinations). The behavior of several matchers becomes more predictable with FilteredContext. In particular, Parents, Name, Leaves, represent the best matchers showing constantly high occurrence (> 5) in all series. Siblings and Children are unstable with occurrences strongly varying between the test series. Sorting the matchers according to their average occurrence over all three series yields the same ranking as we can observe on the *x*-axis of Fig. 16b and c. Therefore, in addition to NamePath, we take three best matchers, Parents, Name and Leaves, to build our default matcher combination (hereafter Default).

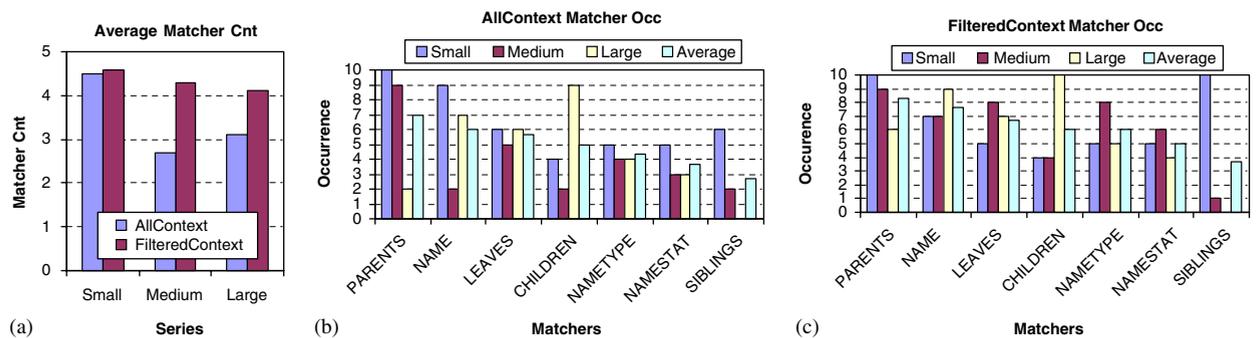


Fig. 16. Statistics of 10 best matcher combinations.

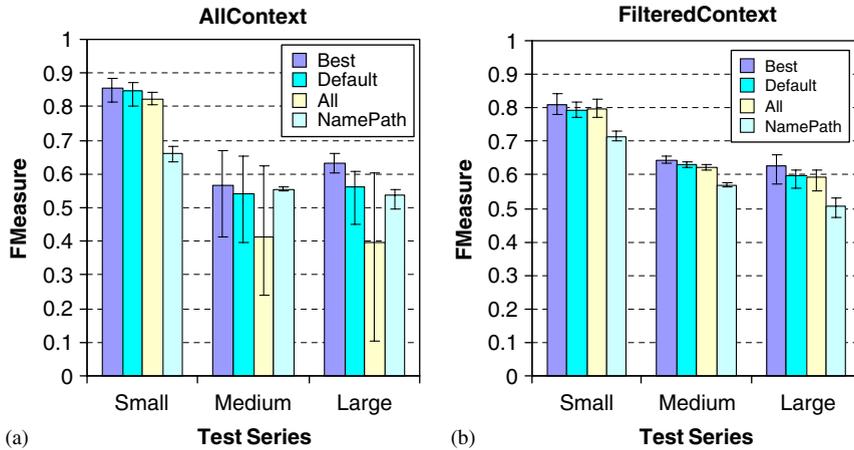


Fig. 17. Quality variation of Best, Default, All, and NamePath.

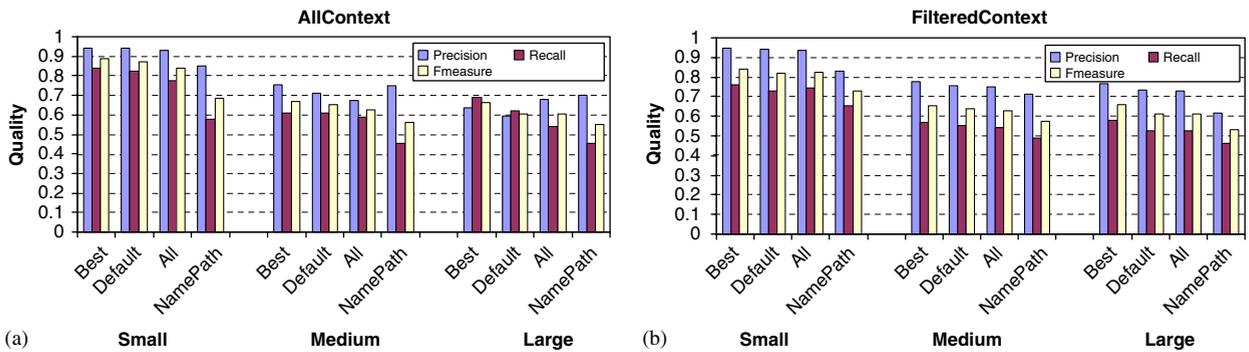


Fig. 18. Best average quality for Best, Default, All, NamePath.

In the next step, we analyze the behavior of the four most important matcher combinations: (a) Best, the best matcher combination showing the highest average Fmeasure; (b) Default, our default matcher combination; (c) All, the combination of all eight combined matchers; and (d) the single Name-Path matcher. We determine the value range, i.e., the min, max and average, of average *F*-measure achieved by the corresponding matcher combinations for the entire Delta range [0.001–0.01] in the single series. Fig. 17a and b in turn show the results for AllContext and FilteredContext, respectively, applied on complete schemas.

We observe that AllContext shows a stable behavior, i.e., low variation, in small schemas or in large schemas if only few matchers are involved. In particular, NamePath exhibits little variation in all series, while All shows the strongest variation, especially the Medium and Large series. On the other side, FilteredContext is more stable against different Delta values, showing a negligible

variation of ± 0.03 of average Fmeasure for all matcher combinations and all series. For both AllContext and FilteredContext and in all series, we observe that the quality behavior of Default is very close to that of Best, indicating the high quality and stability of our default matchers.

Fig. 18 shows all quality measures for the best average qualities from Fig. 17. In general, all matcher combinations outperform NamePath, motivating the use of multiple matchers instead of single matchers. The best quality of Default is close to that of the Best combination. With All, we can also achieve very good quality. However, its use with AllContext is not recommended for large schemas due to long execution times. In general, we achieve high average Precision (>0.9) and average Recall (>0.8) in the Small series. The quality degrades in the Medium and Large series, which yield average Precision and Recall of around 0.7 and 0.5, respectively.

6.7. Evaluation summary

We have comprehensively investigated the quality of the match strategies applied to different fragment types, matcher combinations and combination parameters, and to schemas of varying size. With a small amount of manual effort to prepare only few domain-specific synonyms and abbreviations, we were able to achieve high quality in all test series. The best average Fmeasure achieved was around 0.9 for small match tasks, and between 0.6 and 0.7 for the large match tasks of the Medium and Large series. Using the new strategy FilteredContext, we were also able to solve very large match tasks involving schemas with tens of thousands of elements in acceptable time (less than one minute on our test machine).

NoContext yields unacceptable quality for our test schemas and is therefore only feasible for schemas without shared components. For small schemas, AllContext shows a slightly better quality than FilteredContext. Both, however, achieve almost the same quality in large match tasks. AllContext, when applied to complete schemas or shared fragments, is extremely computation-intensive. Furthermore, it is sensitive to Delta, especially in large schemas, showing a fast degrading quality with increasing Delta. On the other hand, FilteredContext performs in general very fast. It is robust against the choice of Delta, thereby limiting tuning effort. Hence, while AllContext is the strategy of choice for small schemas, FilteredContext suits best for large schemas.

For fragment-based matching, complete coverage of input schemas is an important aspect for the development of new methods for fragment identification and strategies to deal with the remaining schema part. The strategies matching complete schemas and subschemas outperform the strategies considering shared fragments, which typically do not cover the complete schemas. However, in large schemas with a high degree of component reuse, matching shared fragments shows a quality close to that of other fragment strategies. FilteredContext+Shared is thus promising in such cases due to its fast execution time.

The choice of the matchers represents an important impact factor on match quality. In our evaluation, NamePath has proved to be a powerful mean in context-dependent matching. Due to its presence, all match strategies exhibit a relatively low quality variation against the choice of other

matchers to be combined. For FilteredContext, most of the tested matcher combinations yield high quality, close to the maximum achievable. The same holds for AllContext in small match tasks or in large match tasks using a restrictive Delta, such as 0.001. In general, combination of matchers yields better quality than single matchers. We were able to identify a default matcher combination with the matchers NamePath, Parents, Name, and Leaves. The default matcher combination exhibits high quality and stability across the test series. All, the combination of all combined matchers, achieves good quality in both AllContext and FilteredContext strategies and also represents a good candidate for the default matcher combination. However, because of the high complexity and long execution times, it is not recommended to be used with AllContext for large schemas.

We observe that the evaluation results generally remain consistent with those obtained from the evaluation of COMA [6]. In particular, we were able to achieve high quality using the best combination strategies previously identified, in particular, Average for aggregation, Both for direction, Threshold+Delta for selection, and Average for combined similarity. The composite match approach again proved to be a powerful method to combine diverse match algorithms. With the flexibility to construct combined matchers and match strategies, we have been able to quickly implement and compare different match algorithms.

7. Related work

Schema matching has attracted much research activity and several surveys have recently appeared on the proposed approaches [1–5,13,17]. The first extensive survey appeared in 2001 [5] and proposed a solution taxonomy differentiating between schema- and instance-level, element- and structure-level, and linguistic and constraint-based matching, as well as non-reuse and reuse-based approaches. Furthermore, it distinguishes hybrid and composite approaches to combine multiple matchers and reviews several match prototypes, including Cupid [14], SemInt [18], LSD [19], DIKE [20], Similarity-Flooding [21], TranScm [59], and MOMIS [22,41,56]. [13] also surveys newer approaches and prototypes. In the following, we first discuss previous prototypes and approaches and how they differ from COMA++. Afterwards, we review

previous evaluations of Schema matching and how they differ from our evaluation.

7.1. Previous solutions vs. COMA++

Most previous match prototypes are not generic but focus on a specific application domain or only consider a particular type of schemas, such as DTDs [23], relational schemas with instance data [24–26,58], web search interfaces [27,28,62], or ontologies [29–33,60]. Previous generic tools include Cupid, SimilarityFlooding, and Clío [34–36]. As discussed, COMA++ is also generic and supports several schema languages, including XSD, OWL, and relational schemas, and it is able to deal with complex distributed XML schemas. Only few systems support a comprehensive graphical user interface, in particular Clío, Prompt [33] and COMA++.

In order to consider multiple match criteria and properties, most prototypes use *hybrid* algorithms which are difficult to extend and improve. By contrast, COMA/COMA++, LSD, Glue [29], iMap [37], and Protoplasm [15], and [57] follow a *composite* approach to combine the results of individually executed matchers. The presented COMA++ approach for constructing new matchers and match strategies from existing ones is new and especially flexible. Our concept of match strategies is similar to that of *scripts* implemented in Protoplasm [15]. However, we group the basic steps (element identification, matcher execution, and similarity combination) in a uniform configurable unit, allowing for easy definition and combination of match strategies. The multi-matcher architecture originally introduced by COMA and extended by COMA++ has influenced a number of subsequent investigations, e.g., [15,16,38,39,40].

Previous prototypes typically assume tree-like or flat schemas, in which all elements have unique contexts so that matching at the node level is sufficient. Only few systems, in particular, COMA/COMA++, Cupid, Protoplasm, and [16], have addressed *context-dependent matching* to deal with shared elements. However, the simple approach to identify and match all contexts is impractical for large schemas with many shared elements. The new strategy FilteredContext in COMA++ was shown to perform much faster while offering the quality of the expensive approach considering all contexts. The idea of path-level matching, i.e. considering the ascendants of schema elements, can also help

instance-based matchers to distinguish between schema elements with similar instances, as done in LSD [19].

We observe various forms of *reuse of auxiliary information*, including domain-specific synonyms or match/mismatch rules (e.g., in COMA/COMA++, Cupid, DIKE, Glue, LSD, iMAP, XClust [23]). Further variations include the use of lexical dictionaries like WordNet for semantic relationships (e.g., in MOMIS [22,41], S-Match [42]), vocabularies for instance classification (e.g., county name recognition in LSD), schema corpora for additional match information (e.g., in [43]), and manually specified correspondences for training instance-based learners (e.g., in Autoplex [24], Automatch [25], LSD, Glue, iMap). COMA++ supports a new reuse approach focusing on existing mappings, which can be generalized for different reuse granularities, e.g., single element correspondences, or fragment- and schema-level match results.

Like many other prototypes, COMA++ returns a structural mapping consisting of correspondences with a similarity value to indicate their plausibility. This schema matching is the first step in creating an executable (semantic) mapping between two schemas which is executable on instance data, e.g., to perform data transformation. The second step, sometimes called *query discovery* [34], is to enrich the detected correspondences with mapping expressions to exactly specify how the elements and their instances are related to each other. To date, only a few approaches address this task or at least try to return correspondences with some higher semantics. Among others, DIKE and S-Match aim at detecting semantic relationships, such as synonymy and hypernymy, between schema elements. iMap and [44] suggest complex matches, such as string concatenations and arithmetic operations. Starting from a set of correspondences provided by either the user or (semi-) automatic match, Clío and HepTox [45] try to infer query mappings to query and transform instances of one schema to another one.

Besides our work, the *scalability* problem in matching large schemas is only addressed in [30, 15, 46] so far. In particular, [30] focuses on matching large ontologies and employs simple techniques, such as random picking and sorting element names, to identify candidate upper-level elements which are worth to descend and match in detail. Ref. [15] discusses scalability issues in matching two large schemas with several hundreds

of elements. The authors propose to apply a hash-join like match approach and to cache intermediate match results to improve execution time. Ref. [46] studies a large match problem to align two medical taxonomies with tens of thousands of concepts. To reduce match complexity, structural similarities between elements are computed by considering only their direct children and grandchildren. By comparison, COMA++ supports iterative refinement to filter relevant elements before matching them and their neighborhood in detail. Our new FilteredContext and fragment-based match strategies build on this principle and were shown to achieve high quality and fast execution time for large schemas.

7.2. Schema matching evaluations

Most evaluations have been conducted for *individual* prototypes in many diverse ways, making it difficult to assess the relative effectiveness of systems. Refs. [1, 47] represent two efforts aiming at a uniform framework for conducting schema matching evaluations. Both works propose a similar set of criteria concerning the input and output information, and the required manual effort, the measures to quantify the effectiveness and execution performance of a match approach. While [47] used the criteria to evaluate the ontology matching tool Prompt, Ref. [1] uses the criteria to compare several schema matching evaluations, including those of Autoplex [24], Automatch [25], COMA [6], Cupid [14], LSD [19], Glue [29], SemInt [18,48,49], and SimilarityFlooding [21]. The COMA++ evaluation was conducted and documented according to the criteria proposed in [1].

Several studies [14,16,38,40,42,43,50,51] performed a *comparative* evaluation of different approaches on the same match problems. However, the evaluation results are still influenced very much by the way of selecting the match tasks, configuring the single prototypes, and designing a test methodology. Especially, the lack of tuning knowledge of others' prototypes may lead to their suboptimal results. As shown by the EON Ontology Alignment Context 2004 [52,53], these effects can be reduced to a large extent by requiring the tool authors themselves to uniformly perform the evaluation on an independently developed set of match problems. The cumbersome evaluation task, either for an individual system or for the comparison of multiple systems, may benefit from an automatic approach as proposed by the recent e-Tuner system [39],

which systematically tests different configuration of a match system to identify the best one on a synthetic workload of schema and mappings obtained by systematically perturbing an initial schema. Automatic tuning of schema matching tools is a new and promising research direction.

As already pointed out in [1,8], most previous approaches were only tested with *small schemas*, with 50–100 elements. In such small evaluations, the primary focus was on match accuracy, but not on execution time, which is however an important factor for the practicability of an approach. In the evaluation of COMA++, we considered schemas of varying size from hundreds to tens of thousands of elements. With systematic tests, we were able to analyze the correlation of quality and execution time to various aspects, in particular match strategies, matcher combinations, combination strategies, and schema size.

The evaluation results described in this paper are a subset of those from [13]. The complete evaluation of COMA++ described in [13] also includes the ontology matching tests as defined by the EON Ontology Alignment Contest 2004. In comparison with other ontology matching tools, in particular, QOM [54], OLA [31], SCM [32], Prompt [55], COMA++ yields a comparable quality to the best performing participants of the contest. Moreover, Ref. [13] provides a detailed comparison of the COMA++ evaluation and previous schema matching studies, covering both individual and comparative evaluations. As reported there, the old COMA system of 2002 already achieved comparable quality to other prototypes in several independently performed comparative evaluations, i.e., [16,38,40,42,50,51]. As shown in this paper, COMA++ outperforms COMA in terms of both quality and execution time and is thus considered a highly effective schema matching system.

8. Conclusions

We described a new generic schema matching tool, COMA++, which offers a comprehensive infrastructure to solve large real-world match problems. Following the flexible composite approach of our previous prototype COMA, it provides an extensible library of simple and combined matchers and supports various alternatives to combine and refine their match results. For large schemas, we have developed new scalable

strategies for context-dependent and fragment-based matching, respectively. We conducted a comprehensive evaluation of the match strategies using large e-Business standard schemas. The evaluation has demonstrated the practicability of our system for matching large schemas and provided insights for future match implementations and evaluations.

There are several areas where we plan to extend COMA++. In the short term, we will provide a public web interface to COMA++ to make the technology widely available. We want to add new kinds of matchers, especially instance-based approaches and large-scale (re) use of external dictionaries and thesauri. Fragment-based matching represents a promising approach to deal with large schemas. Hence, we want to develop and test with more sophisticated strategies for schema decomposition, i.e. beyond the static fragment types currently supported. Furthermore, we will go beyond schema matching and derive executable mappings from the structural mappings, e.g. to support data transformation.

Acknowledgments

We thank the anonymous reviewers for many helpful comments. This work was supported by DFG Grant BIZ 6/1-1.

References

- [1] H.H. Do, S. Melnik, E. Rahm, Comparison of schema matching evaluations, in: Proceedings of the International Workshop Web and Databases, Lecture Notes in Computer Science, vol. 2593, Springer, Berlin, 2003.
- [2] A.H. Doan, A. Halevy, Semantic integration research in the database community: a brief survey. *AI Magazine*, Special Issue on Semantic Integration, 2005.
- [3] Y. Kalfoglou, M. Schorlemmer, Ontology mapping—the state of the art, *Knowl. Eng. Rev.* 18(1) (2003).
- [4] N.F. Noy, Semantic integration—a survey of ontology-based approaches, *SIGMOD Rec.* 33(4) (2004).
- [5] E. Rahm, P.A. Bernstein, A survey of approaches to automatic schema matching, *VLDB J.* 10(4) (2001).
- [6] H.H. Do, E. Rahm, COMA—a system for flexible combination of match algorithms, in: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), 2002.
- [7] D. Aumüller, H.H. Do, S. Massmann, E. Rahm, Schema and ontology matching with COMA++ (software demonstration), in: Proceedings of the 24th ACM SIGMOD International Conference on Management of Data (SIGMOD), 2005.
- [8] E. Rahm, H.H. Do, S. Massmann, Matching large XML schemas, *SIGMOD Rec.* 33(4) (2004).
- [9] J. Euzenat, An API for ontology alignment, in: Proceedings of the third International Semantic Web Conference (ISWC), 2004.
- [10] D. Lee, W. Chu, Comparative analysis of six XML schema languages, *ACM SIGMOD Rec.* 29(3) (2000).
- [11] P. Hall, G. Dowling, Approximate string matching, *ACM Comput. Survey* 12 (4) (1980) 381–402.
- [12] G. Navarro, A guided tour to approximate string matching, *ACM Comput. Surveys* 33 (1) (2001) 31–88.
- [13] H.H. Do, Schema matching and mapping-based data integration, Dissertation, University of Leipzig, Germany, 2005. <http://lips.informatik.uni-leipzig.de:80/pub/2006-4>.
- [14] J. Madhavan, P.A. Bernstein, E. Rahm, Generic schema matching with Cupid, in: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), 2001.
- [15] P.A. Bernstein, S. Melnik, M. Petropoulos, C. Quix, Industrial-strength schema matching, *ACM SIGMOD Rec.* 33(4) (2004).
- [16] J. Lu, J. Wang, S. Wang, An experiment on the matching and reuse of XML schemas, in: Proceedings of the International Conference on Web Engineering (ICWE), Lecture Notes on Computer Science, vol. 3579, Springer, Berlin, 2005.
- [17] P. Shvaiko, J. Euzenat, A classification of schema-based matching approaches, in: Proceedings of the Meaning Coordination and Negotiation Workshop at ISWC'04, 2004.
- [18] W.S. Li, C. Clifton, SemInt—a tool for identifying attribute correspondences in heterogeneous databases using neural network, *Data Knowl. Eng.* 33 (1) (2000) 49–84.
- [19] A.H. Doan, P. Domingos, A. Halevy, Reconciling schemas of disparate data sources—a machine-learning approach, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2001.
- [20] L. Palopoli, G. Terracina, D. Ursino, The system DIKE—towards the semi-automatic synthesis of cooperative information systems and data warehouses, *ADBIS-DASFAA*, 2000.
- [21] S. Melnik, H. Garcia-Molina, E. Rahm, Similarity flooding—a versatile graph matching algorithm, in: Proceedings of the 18th International Conference on Data Engineering (ICDE), 2002.
- [22] S. Bergamaschi, S. Castano, M. Vincini, D. Beneventano, Semantic integration of heterogeneous information sources, *Data Knowl. Eng.* 36 (3) (2001) 215–249.
- [23] M. Lee, L. Yang, W. Hsu, X. Yang, Xclust—clustering XML schemas for effective integration, in: Proceedings of the International Conference on Information and Knowledge Management (CIKM), 2002.
- [24] J. Berlin, A. Motro, Autoplex, Automated discovery of content for virtual databases, in: Proceedings of the ninth International Conference on Cooperative Information Systems (CoopIS), 2001.
- [25] J. Berlin, A. Motro, Database schema matching using machine learning with feature selection, in: Proceedings of the 14th International Conference on Advanced Information Systems Engineering (CAiSE), 2002.
- [26] A. Bilke, F. Naumann, Schema matching using duplicates, in: Proceedings of the 21st International Conference on Data Engineering (ICDE), 2005.

- [27] B. He, K. Chang, Statistical schema matching across Web query interfaces, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD) 2003.
- [28] H. He, W. Meng, C. Yu, Z. Wu, WISE-Integrator—an automatic integrator of Web search interfaces for E-commerce, in: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), 2003.
- [29] A.H. Doan, J. Madhavan, P. Domingos, A. Halevy, Learning to map between ontologies on the semantic web, in: Proceedings of the 11th International World Wide Web Conference (WWW), 2002.
- [30] M. Ehrig, S. Staab: QOM—quick ontology mapping, in: Proceedings of the Third International Semantic Web Conference (ISWC), 2004.
- [31] J. Euzenat, D. Loup, M. Touzani, P. Valtchev, Ontology alignment with OLA, in: Proceedings of the Third International Workshop on Evaluation of Ontology-based Tools (EON), 2004.
- [32] T. Hoshiai, Y. Yamane, D. Nakamura, H. Tsuda. A semantic category matching approach to ontology alignment, in: Proceedings of the Third International Workshop on Evaluation of Ontology-based Tools (EON), 2004.
- [33] N.F. Noy, M.A. Musen, The PROMPT suite: interactive tools for ontology merging and mapping, *Int. J. Human-Comput. Stud.* 59 (6) (2003) 983–1024.
- [34] L.M. Haas, M.A. Hernández, H. Ho, L. Popa, M. Roth, Clio grows up: from research prototype to industrial tool, in: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2005, pp. 805–810.
- [35] F. Naumann, C.T. Ho, X. Tian, L.M. Haas, N. Megiddo, Attribute classification using feature analysis (poster), in: Proceedings of the 18th International Conference on Data Engineering (ICDE), 2002.
- [36] L. Popa, M. Hernández, Y. Velegrakis, R. Miller, Mapping XML and relational schemas with Clio (software demonstration), in: Proceedings of the 18th International Conference on Data Engineering (ICDE), 2002.
- [37] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, P. Domingos, iMAP—discovering complex semantic matches between database schemas, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD) 2004.
- [38] E. Dragut, R. Lawrence, Composing mappings between schemas using a reference ontology, in: Proceedings of the International Conference on Ontologies, Databases, and Applications of Semantics (ODBASE), 2004.
- [39] M. Sayyadian, Y. Lee, A. Doan, A. Rosenthal, eTuner—tuning schema matching software using synthetic scenarios, in: Proceedings of the 31st International Conference on Very Large Databases (VLDB), 2005.
- [40] K.W. Tu, Y. Yu, CMC—combining multiple schema-matching strategies based on credibility prediction, in: Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA), 2005.
- [41] D. Beneventano, S. Bergamaschi, F. Guerra, M. Vincini, Synthesizing an integrated ontology. *IEEE Internet Computing Magazine*, September–October, 2003.
- [42] F. Giunchiglia, P. Shvaiko, M. Yatskevich: S-Match, an algorithm and an implementation of semantic matching, in: Proceedings of the First European Semantic Web Symposium (ESWS), 2004.
- [43] J. Madhavan, P.A. Bernstein, A.H. Doan, A. Halevy, Corpus-based schema matching, in: Proceedings of the 21st International Conference on Data Engineering (ICDE), 2005.
- [44] L. Xu, D. Embley, Discovering direct and indirect matches for schema elements, in: Proceedings of the Eighth International Conference on Database Systems for Advanced Applications (DASFAA), 2003.
- [45] A. Bonifati, E.Q. Chang, T. Ho, L.V.S. Lakshmanan, R. Pottinger, HePToX—marrying XML and heterogeneity in Your P2P databases (software demonstration), in: Proceedings of the 31st International Conference on Very Large Databases (VLDB), 2005.
- [46] P. Mork, P.A. Bernstein, Adapting a generic match algorithm to align ontologies of human anatomy, in: Proceedings of the 20th International Conference on Data Engineering (ICDE), 2004.
- [47] N.F. Noy, M.A. Musen, Evaluating ontology-mapping tools: requirements and experience, in: Proceedings of the International Workshop on Evaluation of Ontology-based Tools (EON), 2002.
- [48] W.S. Li, C. Clifton, S.Y. Liu, Database Integration Using Neural Networks: Implementation and Experiences. *Knowl. Inform. Systems* 2(1) (2000).
- [49] W.S. Li, C. Clifton, Semantic integration in heterogeneous databases using neural networks, in: Proceedings of the 20th International Conference on Very Large Databases (VLDB), 1994.
- [50] P. Avesani, F. Giunchiglia, M. Yatskevich, A large scale taxonomy mapping evaluation, in: Proceedings of the fourth Intl. Semantic Web Conference (ISWC), 2005.
- [51] F. Giunchiglia, M. Yatskevich, E. Giunchiglia, Efficient semantic matching, in: Proceedings of the Second European Semantic Web Conference (ESWC), 2005.
- [52] EON Ontology Alignment Contest, <http://co4.inrialpes.fr/align/Contest/>.
- [53] J. Euzenat, Introduction to the EON experiment, in: Proceedings of the third International Workshop Evaluation of Ontology-based Tools (EON), 2004.
- [54] M. Ehrig, Y. Sure, Ontology alignment—Karlsruhe, in: Proceedings of the third International Workshop Evaluation of Ontology-based Tools (EON), 2004.
- [55] N.F. Noy, M.A. Musen, Using prompt ontology—comparison tools in the EON ontology alignment contest, in: Proceedings of the Third International Workshop Evaluation of Ontology-based Tools (EON), 2004.
- [56] S. Castano, V.D. Antonellis A Schema analysis and reconciliation tool environment, in: Proceedings of the International Database Engineering and Applications Symposium (IDEAS), 1999.
- [57] D.W. Embley, D. Jackman, L. Xu, Multifaceted exploitation of metadata for attribute match discovery in information integration, in: Proceedings of the International Workshop on Information Integration on the Web (IIW), 2001.
- [58] J. Kang, J. Naughton, On schema matching with opaque column names and data values, in: Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD), 2003.
- [59] T. Milo, S. Zohar: Using schema matching to simplify heterogeneous data translation, in: Proceedings of the 24th

- International Conference on Very Large Data Bases (VLDB) 1998.
- [60] P. Mitra, G. Wiederhold, Resolving terminological heterogeneity in ontologies, in: Proceedings of the ECAI'02 Workshop on Ontologies and Semantic Interoperability, 2002.
- [61] C.J. Van Rijsbergen, Information Retrieval, second ed, Butterworths, London, 1979.
- [62] J. Wang, J. Wen, F. Lochovsky, W. Ma, Instance-based schema matching for web databases by domain-specific query probing, in: Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), 2004.