Matching Large XML Schemas

Erhard Rahm, Hong-Hai Do, Sabine Maßmann
University of Leipzig, Germany
rahm@informatik.uni-leipzig.de

Abstract

Current schema matching approaches still have to improve for very large and complex schemas. Such schemas are increasingly written in the standard language W3C XML schema, especially in E-business applications. The high expressive power and versatility of this schema language, in particular its type system and support for distributed schemas and namespaces, introduce new issues. In this paper, we study some of the important problems in matching such large XML schemas. We propose a fragment-oriented match approach to decompose a large match problem into several smaller ones and to reuse previous match results at the level of schema fragments.

1 Introduction

Schema matching aims at identifying semantic correspondences between two schemas, e.g. database schemas, ontologies, XML message formats, etc. The need for schema matching in numerous applications and the inherent difficulty of the task have led to the development of many techniques and prototypes to semi-automatically solve the match problem [10, 4, 5, 7, 3, 8, 15]. The proposed approaches typically exploit various types of schema information (e.g. element names, data types and structural properties), characteristics of data instances, as well as background knowledge from dictionaries and thesauri. The reuse of previously determined match results proposed in [15] has also been a recent research focus promising a significant reduction in manual match work [5, 9].

The approaches developed so far were typically applied to various test schemas for which they could automatically determine most correspondences. However, as surveyed in [6] most test schemas were structurally rather simple and of small size of less than 50-100 components (elements, attributes). Unfortunately, the effectiveness of automatic match techniques studied so far may significantly decrease for larger input schemas [5, 7] because larger schemas increase the likelihood of false matches.

Advanced modeling capabilities such as complex types, aggregation and generalization, which are supported by W3C XML Schema and the object-relational SQL extensions (SQL:1999, SQL:2003), also lead to a significant complication for schema matching. For instance, complex types or substructures (e.g., for address, customer, etc.) may occur many times in a schema possibly with a context-dependent semantics. Such shared schema components require special treatment to avoid an explosion of the search space and to effectively deal with n:m match cardinalities. The support of distributed schemas and namespaces in W3C XML Schema also has not been considered in current schema match systems.

There has been a modest amount of previous work on some of the issues raised. For instance, several studies considered *is-a* hierarchies in determining the similarity of schema elements [2, 14]. Cupid [10] and COMA [5] can deal with shared components to some extent (see Section 2). [12] studies a large match problem to align two medical taxonomies with tens of thousands of concepts. To reduce match complexity, structural similarities between elements of the two input schemas are computed by considering only direct children and grandchildren.

In this paper we discuss problems and possible solution strategies for matching large schemas written in the W3C XML Schema Definition language (XSD)[1]. XSD was approved as a W3C recommendation in 2001 and since then has been increasingly adopted especially in web-based applications, e.g. for e-business. In the next section we discuss some of the new aspects to be dealt with when matching XSD schemas, in particular the high modeling flexibility enabled by the XSD type system, component reuse/sharing, and distributed schemas. We propose a fragment-oriented match approach to decompose a large match problem into several smaller match problems on schema fragments, e.g. specific message types, shared components or complex types. Moreover, the idea to reuse previous match results can be generalized to the level of fragments. While we focus on XSD in this paper, after a language-specific preprocessing phase the fragment-oriented match approach is generically applicable. The approach has been integrated into the COMA match prototype.

2 Issues in Matching Large XSD Schemas

Current match systems not only focus on small schemas but also on structurally simple schemas w.r.t. the number of nesting levels, data types, constraints, and support of shared schema components. The traditional database notion of a schema is typically assumed where all instances can be described by a single monolithic schema. However, many web and XML-based applications require a powerful and flexible schema support that goes beyond the capabilities of XML DTD and traditional database schema languages. These requirements have thus been

[1] www.w3.org/XML/Schema

incorporated into the W3C XML schema definition (XSD) language. As a result, matching schemas taking advantage of the advanced capabilities of XSD becomes much more challenging than matching DTDs or simple relational (e.g. SQL-92) schemas. To illustrate some of the new challenges we focus on three key features of XSD:

1. Type system
2. Shared schema components (reuse of schema components)
3. Distributed schemas / namespace support.

The relevance of these issues can be illustrated by examining large real-life XSD schemas. Table 1 shows some statistics for several standardized E-Business catalog and message schemas, namely BMECat, OpenTrans, and two sub-standards of XCBL, OrderManagement and Catalog.[2] These schemas are distributed across many files and have a size of several hundreds to almost 1500 schema components.

Schema	Name spaces	Files	Size	All / Global Elements	All / Global Types	Shared Comp
BMECat	3	10	429	403 / 170	25 / 14	30
OpenTrans	1	15	614	589 / 194	25 / 11	61
XCBL Order	1	63	1451	1088 / 8	358 / 358	91
XCBL Catalog	1	50	310	225 / 1	71 / 71	12

Table 1. Statistics of some E-Business XSD schemas

2.1 Type System

In contrast to DTD, XSD is based on a versatile type system distinguishing between simple and complex types. There are 43 built-in simple types (e.g., string, integer, float, boolean, time, date), which can be used for element and attribute declarations. Complex types are user-defined and can be used for element (but not attribute) declarations. In contrast to simple types, complex types can (and typically do) have elements in their content and may carry attributes.

Existing simple and complex types can both be referenced in other type definitions and be extended or restricted within a new type. These two general ways to define new types are also referred to as *composition* (aggregation) and *sub-classing* (specialization), respectively. Figure 1a illustrates the composition approach where new (complex) types use existing types as building blocks: the complex type *Supplier* is composed of elements of existing types *int* and *Contact*. With sub-classing, on the other hand, a new (simple or complex) type is derived from an existing type using either the restriction or extension mechanism. In the example of Figure 1b, type *Supplier* extends type *Contact* and thus inherits its elements *Name* and *Phone*. Composition and sub-classing can be recursively applied so that arbitrarily nested type hierarchies are possible.

The exploitation of type information is of key importance for effective schema matching and entails estimating the degree of similarity between different types. As a consequence, approaches to determine the similarity between simple and complex XSD types of different schemas have to be provided. The similarity between built-in simple types can be determined analogously to previous type matchers, e.g. by providing a static compatibility table. User-defined simple types are also relatively easy to deal with as they can always be associated with a built-in simple type. Integrity constraints (facets) such as *maxLength, minLength*, *pattern*, etc., should also be exploited for type matching.

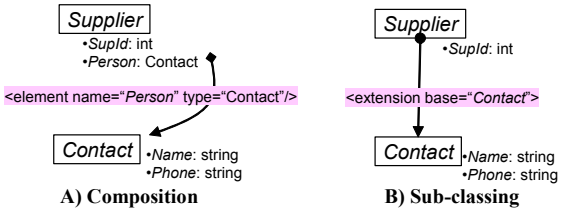

Figure 1. Type design patterns

Complex types, on the other hand, may exhibit almost unlimited complexity. In fact, determining the similarity between complex types and matching them can be as difficult as matching two complete schemas since a schema may just contain a single element of a particular complex type. This indicates that matching complex types requires a large spectrum of techniques including structural match approaches, which determine the similarity between the types' components and consider the different ways these components are used within the type definition. In particular, the composition and sub-classing alternatives need to be dealt with so that the similarity between alternative type definitions such as in Figure 1 can be determined. Furthermore, complex type matching should consider the used XSD compositors (*sequence*, *choice*, *all*), cardinality restrictions (*minOccurs*, *maxOccurs*) and other integrity constraints of their components.

2.2 Shared Schema Components

While XML instance documents are always tree-structured, in general XML schemas are graph-structured. In particular, there may be shared schema components (elements, attributes, types, groups), which are referenced in several places. In XSD, only so-called *global* components can be referenced, i.e. direct children of the <*schema*> root element of an XSD file. The key advantage of such a referencing is the reuse of schema components, which avoids redundant or unnecessarily diverse specifications. This is especially important for large schemas.

Referencing global elements is a simple form of reuse, already supported in DTDs. All names of such elements must be unique in a schema and referencing elements have the same name (and type) as the referenced element. In XSD, nested element references are not supported since global elements must not reference other global elements. On the other hand, XSD supports a more versatile reuse for global (named) types. These types can be referenced within element or attribute declarations as well as (recur-

[2] BMECat: www.bmecat.org, OpenTrans: www.opentrans.org, XCBL: www.xcbl.org

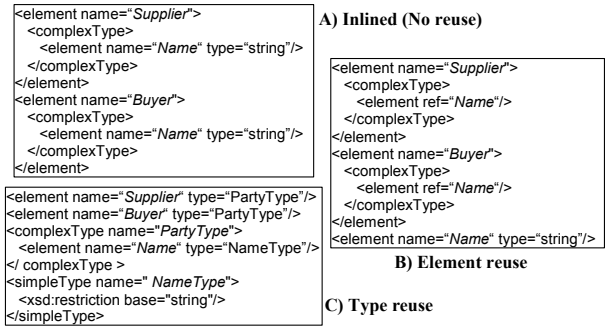

Figure 2. Component design patterns for reuse

sively) within other type definitions. Elements and attributes typically have different names than a referenced named type and these names can carry additional semantic information.

In addition to element reuse and type reuse, it is also possible to avoid shared components (no reuse) and to anonymously specify types inline (locally) when elements and attributes are declared. This inline approach results in tree-like schemas and may be sufficient for smaller schemas with few elements. The three alternatives are illustrated in Figure 2. While they may be mixed within a schema, three design philosophies each focusing on one of the approaches have been proposed.[3] The high flexibility of type reuse, which is specific to XSD, makes it a well-suited approach for large business applications.

XCBL (Table 1) follows the type reuse approach and only has few global elements as possible roots for instance documents. BMECat and OpenTrans mainly utilize the element reuse approach, resulting in a large number of global elements. There are a substantial number of shared components in all schemas.

XSD schema matching must be able to deal with alternate design approaches. For instance, it must be possible to determine the similarity of the schema fragments of Figure 2 and their match correspondences. Match processing thus requires a uniform schema representation, which is not biased to any design style and can cope with shared components and nested type references.

Shared components are especially important for schema matching, but also difficult to deal with. Within a schema, a shared component *c* indicates a similarity between *c*'s ancestors. Hence, the correspondences of *c* in a second schema may match to all these ancestors (n:m match cardinalities). On the other hand, it may be important to clearly differentiate between the contexts where a shared component is used, e.g. to distinguish the names or addresses of buyers vs. suppliers.

Most previous match systems focused on schemas with no or only few shared components. COMA [5] and Cupid [10] can deal with shared components to some extent. COMA applies a *path-based* approach differentiating all possible paths from the schema root to a shared component *c*, thereby capturing all possible contexts of *c*. In the example of Figure 3a, we obtain two paths for the shared component *Name*. Cupid follows a *materialized* approach by maintaining multiple copies of shared components in a (voluminous) tree-like schema representation. In Figure 3b we thus have two *Name* nodes each associated with a single parent (*Supplier* and *Buyer*, respectively).

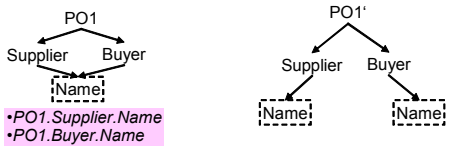

Figure 3. Resolving shared components

Unfortunately, both approaches do not scale to a higher number of shared components. They both consider all possible contexts of shared components often leading to an explosion in the number of nodes or paths per schema. For example, the XCBL Order schema contains 1,451 components including 91 shared types (Table 1), but more than 26,000 different nodes/paths after resolving the shared components. Matching two schemas of such a size at the node/path level results in unacceptable execution times in the order of hours.

[4] confirms the scalability problem for large match tasks. To improve performance it was proposed to apply a hash-join like match approach and to cache intermediate match results. These enhancements proved to be very effective for an experiment on matching two versions of the same schema with 340 and 500 elements, but with only few shared components. It seems important to investigate the applicability of such optimizations for more schemas and to also evaluate match quality.

2.3 Distributed Schemas

The conventional way to construct a schema is to put all components in a single schema document, which is quite handy for simple applications. To better deal with large schemas, especially for web applications, XSD allows a schema to be distributed over several schema documents (files) and namespaces, which is used by the e-business schemas of Table 1. Each schema document can be assigned to a so-called target namespace and XSD provides different directives (*include*, *redefine* and *import*) to in-

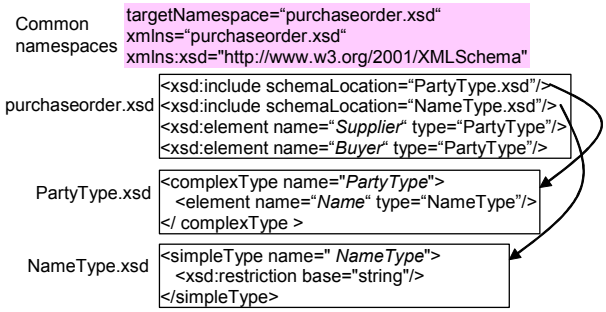

Figure 4. Distributed schema (1 target namespace)

[3] The approaches have been named "Russian doll" (inline typing), "Salami slice" (element reuse) and "Venetian blind" (type reuse), e.g. in [16]

corporate component definitions from existing documents into a new document. In the distributed schema of Figure 4, all documents declare the same target namespace *purchaseorder.xsd*. The main document purchaseorder.xsd, references type *PartyType* defined in document PartyType.xsd, which in turn references type *NameType* in document NameType.xsd.

There are many options how to organize distributed schemas and namespaces [16] and an XSD-capable match system should be able to deal with them. A distributed XSD schema is often a collection of several smaller schemas or *sub-schemas* sharing common types and elements for reuse. For instance, each message format in an e-business schema is a sub-schema that can - and should - be matched separately. For example, the XCBL Order schema consists of 8 sub-schemas representing different message formats (Order, ChangeOrder, OrderRequest, OrderResponse, etc.).

Schemas and sub-schemas directly describing instance documents (e.g., e-business messages) should further be separated from supporting *reference schemas* (e.g., type libraries), which only contain named types, global elements, etc. for reference in other schema documents. Such reference schemas are typically in separate files and may even form separate namespaces (e.g., if they are of company-wide or global relevance). Taxonomies and other ontologies are a variation of such reference schemas providing a controlled and categorized vocabulary, e.g. for use within schemas.

Therefore, a matching system should be able to identify sub-schemas and reference schemas from a collection of schema documents. While matching between reference schemas is not directly required, it is promising to match their components in advance in order to reuse the result for matching the referencing (sub-)schemas. The ideal case are (sub-)schemas sharing the same reference schema or ontology, since schema components referring to the same entry in the reference schema are often good match candidates.

3 Fragment-based Matching

For a match task with large schemas it is likely that large portions of one or both input schemas have no matching counterparts. The standard approach trying to match the complete input schemas will often lead not only to performance problems (long execution times), but also poor match quality with many false match candidates. Furthermore, it is very difficult to present the match result to a human engineer in a way that she can easily validate and correct it.

We thus advocate fragment-based schema matching, i.e. a divide-and-conquer strategy which decomposes a large match problem into smaller sub-problems by matching at the level of schema fragments (e.g., sub-schemas or shared types). As illustrated in Figure 5, the strategy encompasses four steps: (1) a decomposition step to determine suitable fragments, (2) identification of the most

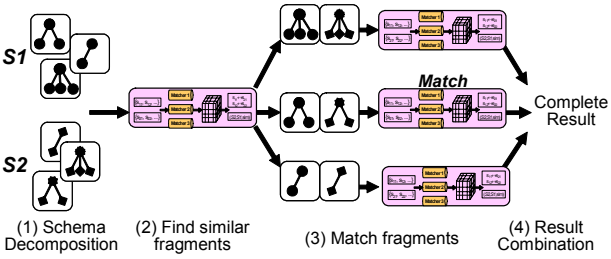

Figure 5. Fragment-based match strategy

similar fragments between schemas to match, (3) matching similar fragments, and (4) combining the fragment match results (if a result for the complete schema is to be determined).

By reducing the size of the match problem we not only aim at better performance but also at improved match quality compared to schema-level matching. Moreover, the fragment approach can be used for interactive match processing. For instance, the user may manually select a fragment of interest for which matches from the second schemas are determined automatically (by running steps 2 and 3). Then the fragment result may be manually controlled and corrected before proceeding with another fragment.

For each step there are many design options and implementation possibilities, some of which we will discuss in the following.

3.1 Schema Decomposition (Fragment Identification)

The main goal of this initial phase is to decompose the input schemas into appropriate fragments. We assume that input schemas are uniformly transformed to a directed acyclic graph representation for manipulation by the match system (e.g., during the import for new XSD schemas). By *fragment* we denote a rooted sub-graph in the schema graph. Hence, entire schemas and schema nodes (paths) are special fragment types so that the previously investigated schema-level and node-level matching approaches can be considered as variations of fragment matching. Additional *fragment types* of interest include sub-schemas and inner fragments of a schema. In general, fragments should have little or no overlap to avoid repeated similarity computations and overlapping match results.

- *Sub-schemas:* They 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 data transformations. Matching a sub-schema (e.g., one message format) is obviously much simpler than matching the complete schema (e.g., all formats) at once. In fact, the user may only be interested in a particular message format. For XSD, sub-schema selection should be confirmed by the user since a global element does not necessarily indicate a sub-schema (as in the XCBL Order example).
- *Inner fragments:* Schemas may not have sub-schemas or sub-schemas can still be very large so that fragment

matching should also be supported on finer-granularity inner fragments. One fragmentation possibility are *leaf fragments,* i.e. selected inner schema components (complex types or elements) and all their descendants down to the leaf level where only simple types are possible. Another option is to only consider *shared* sub-structures as *fragments* such as named types for address, customer, etc. The match results for such fragments may be usable many times within a schema thereby improving performance. Moreover, a suitable approach for dealing with shared components (Section 2.2) may be built on such a fragmentation.

Since fragments will be matched separately, the schema graph can be compacted to a *proxy schema graph* by replacing each inner fragment by a proxy node (e.g., the fragment root). For the example of Figure 6, the components of fragments *Contact* in schema *PO1* and *ContactPers* in *PO2* would thus be eliminated from the schema graph. The reduced graph size allows a faster match for the remaining components. The match results for the inner fragments are incorporated in step 4 (result combination).

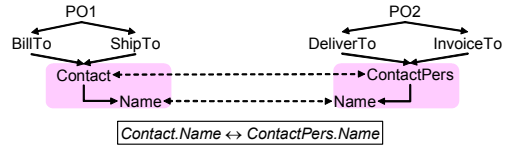

Figure 6. Fragment and context match

During the decomposition phase, fragment characteristics and statistics can also be determined as a prerequisite for computing fragment similarities in step 2. Such metadata may include the fragment name (name of the fragment root), type name of the fragment root (if available), fragment type (sub-schema, shared, or leaf), and information on the containing schema file (location, namespace, version information, change date, etc.). Relevant statistical fragment information includes size (number of nodes in fragment), local depth (maximal path length in the fragment), global depth (distance from schema root to fragment root), and number of parents (how often a fragment is used). Moreover, all *contexts* of a fragment may be determined, i.e. the paths from the schema root to the fragment root, indicating where the fragment is used within a schema.

3.2 Identifying Fragment-Pair Candidates

The goal of this step is to identify fragments of the two schemas that are sufficiently similar to be worth matching in more detail. This aims at reducing match overhead by not trying to find correspondences for a fragment in irrelevant fragments from the second schema. Hence, the similarity between two fragments should be determined cheaper than fully matching the fragments with all their components. For example, the comparisons can be performed on fragment metadata, such as fragment names, fragment contexts and statistical data collected in step 1. For instance, numerical metadata can be used in distance functions or in feature analysis techniques [8, 13] to determine the structural similarity of fragments. The comparison between two fragments is assumed to result in a normalized similarity value, which can be used to determine the most similar fragments.

For fragments, which have been fully matched in a previous match task, the detailed match result can be used to determine the fragment similarity more precisely. There are several possibilities to aggregate similarity values between components to determine fragment similarities, e.g. as supported by the combination framework of COMA [5].

3.3 Fragment Matching

In this step, the identified pairs of similar fragments are fully matched to obtain the correspondences between their components. One open question is whether or not the contexts of a fragment should be considered for this task. We favor the simpler *context-insensitive* alternative treating fragments as independent schemas, so that fragment matching is basically the same as matching schemas of reduced size. This is the method of choice for matching sub-schemas. However, it may also be sufficient for inner fragments by resolving context dependencies in step 4. Fragment matching can utilize the known schema matching techniques, such as name matching, structural matching, instance-based matching, etc. For our simple example, we could obtain the match between the *Name* components of fragments *Contact* and *ContactPers* (Figure 6a).

Particularly promising is the *reuse* of previous match results, which may be more often and more effectively applicable at the level of fragments compared to entire schemas. As pointed out in [5], reuse can be implemented by combining two or more match results (mappings) by means of a *MatchCompose* operation (which is similar to a join). For a match task F1-F2 we check in a repository of previous match results whether there are already results for F1, F2 or similar fragments. This selection can use the approaches of step 2 for finding similar fragments. For instance, if F1 was already matched to an older version of F2, F2', we can combine this existing result with the match result for F2'-F2 to solve our task. This is a useful approach if it is easier to newly match F2'-F2 than F1-F2, e.g., because only few F2 components are changed compared to F2'. Note that reusing previous match results has similarities to case-based reasoning [1]; the cases are match tasks and we try to solve a new case by searching for similar previous cases and adapting their solutions.

3.4 Result Combination

If the task is to determine the match result for two complete schemas, the match correspondences for inner fragments need to be combined with the match result for the proxy schema graphs (step 1). This assumes that the compacted schemas are separately matched to cover the components not represented in the fragments, including the

fragment contexts. For the example of Figure 6, it may have been determined that *PO1.BillTo* matches *PO2.InvoiceTo* (but not *PO2.DeliverTo*). This fragment context match needs to be combined with the local fragment correspondences so that the correspondence of Figure 6b can be derived.

For leaf fragments, the contexts may also be matched by a bottom-up propagation of the similarity values for fragment components to the ancestors in the schema graph. This generalizes an idea used in previous structural matchers to determine the similarity of inner nodes based on the similarity of leaf nodes [10, 5].

4 Prototype

We have implemented advanced XSD support and fragment matching in a heavily extended version of the COMA prototype described in [5]. Figure 7 shows the gross architecture of the prototype.

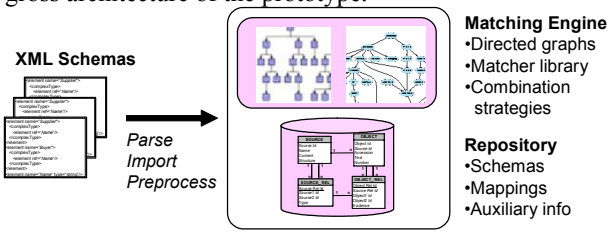

Figure 7. System gross architecture

The import of an XSD schema is a complex operation in which the schema is parsed and transformed to a uniform directed graph representation. Our parser supports schemas stored in a single file or a collection of files. The different designs, in particular, element reuse, type reuse, and type sub-classing, are resolved using a uniform structure. We further perform the preprocessing steps discussed in the last section, such as identification of sub-schemas, determination of structural statistics, and detection and removal of graph cycles.

Imported schemas can be saved in a central repository, from which complete schemas or sub-schemas can be loaded for matching. The repository also stores approved match results and other auxiliary information, such as synonyms, for reuse purposes. Match processing is performed within a matching engine, which provides a library of individual matchers and supports various strategies to combine their results. A match operation can involve multiple matchers, which can be flexibly selected from the matcher library. Existing matchers can also be easily combined to build more powerful matchers. In both cases, we use the combination scheme already implemented in COMA to derive the best match result from the individual results predicted by the single matchers.

We added several new matchers to the original COMA matcher library, especially for implementing the fragment-based approach discussed in the last section. Currently, sub-schemas and shared fragments are supported. The approaches still need to be evaluated and will be described in a future paper.

5 Conclusions

Large schemas and advanced features of the W3C XML schema description language are still not well supported by current schema matching prototypes, thereby limiting the practical applicability of such systems. We studied several XSD features used in large e-business schemas that need to be considered for schema matching, in particular distributed schemas and namespaces, heavy use of shared schema components, and different design styles based on XSD's flexible type system. We proposed a fragment-oriented match approach to decompose large match problems into several smaller ones. Our approach includes sub-steps for schema decomposition, finding similar fragments, fragment matching and result combination. There are many design options, which need further investigation, e.g. how to best consider context dependencies for fragments.

We have implemented a first version of the approach within an extended version of COMA. To keep the system generic, we encapsulate XSD-specific aspects in the import operation and within tailored matchers. We believe that fragment-oriented matching, various forms of reusing previous match results, and the flexible combination of different matchers are cornerstones of a successful and scalable match system. Our future work will focus on completing such a system and performing comprehensive evaluations of different implementation alternatives.

Acknowledgements. We thank Phil Bernstein and Sergey Melnik for helpful comments. The second author is supported by DFG grant BIZ 6/1-1.

References

1. Aamodt, A., Plaza, E.: Case-based Reasoning – Foundational Issues, Methodological Variations, and System Approaches. AI Communications 7: 1, 1994
2. Bergamaschi, S. et al.: Semantic Integration of Semistructured and Structured Data Sources. ACM SIGMOD Record 28: 1, 1999
3. Berlin, J., A. Motro: Autoplex: Automated Discovery of Content for Virtual Databases. CoopIS 2001
4. Bernstein, P.A. et al: Industrial-strength Schema Matching. ACM SIGMOD Record, 2004 (this issue)
5. Do, H.H., E. Rahm: COMA – A System for Flexible Combination of Match Algorithms. VLDB 2002
6. Do, H.H., S. Melnik, E. Rahm: Comparison of Schema Matching Evaluations. GI-Workshop Web and Databases, LNCS 2593, 2003
7. Doan, A.H. et al.: Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach. SIGMOD 2001
8. Li, W., C. Clifton: SemInt - A Tool for Identifying Attribute Correspondences in Heterogeneous Databases Using Neural Network. Data and Knowledge Engineering 33: 1, 2000
9. Madhavan, J. et al.: Corpus-based Schema Matching. Workshop on Information Integration on the Web (IIWeb), 2003
10. Madhavan, J., P.A. Bernstein, E. Rahm: Generic Schema Matching with Cupid. VLDB 2001
11. Melnik, S., H. Garcia-Molina, E. Rahm: Similarity Flooding – A Versatile Graph Matching Algorithm. ICDE 2002
12. Mork, P., P.A. Bernstein: Adapting a Generic Match Algorithm to Align Ontologies of Human Anatomy. ICDE 2004
13. Naumann, F. et al.: Attribute Classification Using Feature Analysis. ICDE 2002
14. Palopoli, L. et al.: Uniform Techniques for Deriving Similarities of Objects and Subschemes in Heterogeneous Databases. IEEE Trans. Knowl. Data Eng. 15: 2, 2003
15. Rahm, E., P.A. Bernstein: A Survey of Approaches to Automatic Schema Matching. VLDB Journal 10: 4, 2001
16. XML Schemas - Best Practices. www.xfront.com/BestPracticesHomepage.html